# Condor® Version 6.8.6 Manual

Condor Team, University of Wisconsin–Madison

September 13, 2007

# CONTENTS


**1 Overview** **1**

1.1 High-Throughput Computing (HTC) and its Requirements . . . . . 1

1.2 Condor's Power . . . . . 2

1.3 Exceptional Features . . . . . 3

1.4 Current Limitations . . . . . 4

1.5 Availability . . . . . 5

1.6 Contributions to Condor . . . . . 7

1.7 Contact Information . . . . . 9

1.8 Privacy Notice . . . . . 10

**2 Users' Manual** **12**

2.1 Welcome to Condor . . . . . 12

2.2 Introduction . . . . . 12

2.3 Matchmaking with ClassAds . . . . . 13

2.3.1 Inspecting Machine ClassAds with condor_status . . . . . 14

2.4 Road-map for Running Jobs . . . . . 15

2.4.1 Choosing a Condor Universe . . . . . 16

2.5 Submitting a Job . . . . . 20

2.5.1 Sample submit description files . . . . . 21

2.5.2 About Requirements and Rank . . . . . 23
2.5.3 Submitting Jobs Using a Shared File System . . . . . 35
2.5.4 Submitting Jobs Without a Shared File System: Condor's File Transfer Mechanism 36
2.5.5 Environment Variables . . . . . 45
2.5.6 Heterogeneous Submit: Execution on Differing Architectures . . . . . 45
2.6 Managing a Job . . . . . 49
2.6.1 Checking on the progress of jobs . . . . . 50
2.6.2 Removing a job from the queue . . . . . 51
2.6.3 Placing a job on hold . . . . . 52
2.6.4 Changing the priority of jobs . . . . . 52
2.6.5 Why does the job not run? . . . . . 53
2.6.6 In the log file . . . . . 54
2.6.7 Job Completion . . . . . 57
2.7 Priorities and Preemption . . . . . 58
2.7.1 Job Priority . . . . . 58
2.7.2 User priority . . . . . 58
2.7.3 Details About How Condor Jobs Vacate Machines . . . . . 59
2.8 Java Applications . . . . . 60
2.8.1 Example Java Program . . . . . 60
2.8.2 Chirp I/O . . . . . 63
2.9 PVM Applications . . . . . 65
2.9.1 Effective Usage: the Master-Worker Paradigm . . . . . 66
2.9.2 Binary Compatibility and Runtime Differences . . . . . 66
2.9.3 Sample PVM submit file . . . . . 68
2.10 Parallel Applications (Including MPI Applications) . . . . . 70
2.10.1 Prerequisites to Running Parallel Jobs . . . . . 70
2.10.2 Parallel Job Submission . . . . . 71
2.10.3 Parallel Jobs with Separate Requirements . . . . . 72

2.10.4 MPI Applications Within Condor's Parallel Universe . . . . . 73
2.10.5 Outdated Documentation of the MPI Universe . . . . . 74
2.11 DAGMan Applications . . . . . 79
2.11.1 DAGMan Terminology . . . . . 79
2.11.2 Input File Describing the DAG . . . . . 80
2.11.3 Submit Description File . . . . . 87
2.11.4 Job Submission . . . . . 89
2.11.5 Job Monitoring, Job Failure, and Job Removal . . . . . 90
2.11.6 Job Recovery: The Rescue DAG . . . . . 91
2.11.7 Visualizing DAGs with *dot* . . . . . 91
2.11.8 Advanced Usage: A DAG within a DAG . . . . . 92
2.11.9 Single Submission of Multiple, Independent DAGs . . . . . 93
2.11.10 File Paths in DAGs . . . . . 94
2.11.11 Configuration . . . . . 95
2.12 Stork Applications . . . . . 95
2.12.1 Submitting Stork Jobs . . . . . 96
2.12.2 Managing Stork Jobs . . . . . 97
2.12.3 Fault Tolerance . . . . . 97
2.12.4 Running Stork Jobs Under DAGMan . . . . . 99
2.13 Job Monitor . . . . . 99
2.13.1 Transition States . . . . . 100
2.13.2 Events . . . . . 100
2.13.3 Selecting Jobs . . . . . 100
2.13.4 Zooming . . . . . 100
2.13.5 Keyboard and Mouse Shortcuts . . . . . 101
2.14 Special Environment Considerations . . . . . 101
2.14.1 AFS . . . . . 101
2.14.2 NFS Automounter . . . . . 102

2.14.3 Condor Daemons That Do Not Run as root . . . . . 102
2.14.4 Job Leases . . . . . 103
2.15 Potential Problems . . . . . 103
2.15.1 Renaming of argv[0] . . . . . 103

**3 Administrators' Manual 104**
3.1 Introduction . . . . . 104
3.1.1 The Different Roles a Machine Can Play . . . . . 105
3.1.2 The Condor Daemons . . . . . 106
3.2 Installation . . . . . 108
3.2.1 Obtaining Condor . . . . . 108
3.2.2 Preparation . . . . . 109
3.2.3 Newer Unix Installation Procedure . . . . . 115
3.2.4 Older Unix Installation Procedure . . . . . 116
3.2.5 Condor is installed Under Unix ... now what? . . . . . 121
3.2.6 Installation on Windows . . . . . 123
3.2.7 RPMs . . . . . 132
3.2.8 Upgrading - Installing a Newer Version of Condor . . . . . 133
3.2.9 Installing the CondorView Client Contrib Module . . . . . 133
3.2.10 Dynamic Deployment . . . . . 136
3.3 Configuration . . . . . 138
3.3.1 Introduction to Configuration Files . . . . . 138
3.3.2 The Special Configuration Macros $ENV(), $RANDOM_CHOICE(), and $RANDOM_INTEGER() 144
3.3.3 Condor-wide Configuration File Entries . . . . . 145
3.3.4 Daemon Logging Configuration File Entries . . . . . 151
3.3.5 DaemonCore Configuration File Entries . . . . . 155
3.3.6 Network-Related Configuration File Entries . . . . . 157
3.3.7 Shared File System Configuration File Macros . . . . . 159

3.3.8 Checkpoint Server Configuration File Macros . . . . . 162
3.3.9 condor_master Configuration File Macros . . . . . 163
3.3.10 condor_startd Configuration File Macros . . . . . 168
3.3.11 condor_schedd Configuration File Entries . . . . . 182
3.3.12 condor_shadow Configuration File Entries . . . . . 187
3.3.13 condor_shadow.pvm Configuration File Entries . . . . . 188
3.3.14 condor_starter Configuration File Entries . . . . . 188
3.3.15 condor_submit Configuration File Entries . . . . . 190
3.3.16 condor_preen Configuration File Entries . . . . . 192
3.3.17 condor_collector Configuration File Entries . . . . . 192
3.3.18 condor_negotiator Configuration File Entries . . . . . 196
3.3.19 condor_credd Configuration File Macros . . . . . 199
3.3.20 condor_gridmanager Configuration File Entries . . . . . 199
3.3.21 grid_monitor Configuration File Entries . . . . . 202
3.3.22 Configuration File Entries Relating to Grid Usage and Glidein . . . . . 203
3.3.23 Configuration File Entries for DAGMan . . . . . 203
3.3.24 Configuration File Entries Relating to Security . . . . . 206
3.3.25 Configuration File Entries Relating to High Availability . . . . . 208
3.3.26 Configuration File Entries Relating to Quill . . . . . 212
3.3.27 MyProxy Configuration File Macros . . . . . 213
3.3.28 Configuration File Macros Affecting APIs . . . . . 213
3.3.29 Stork Configuration File Macros . . . . . 214
3.4 User Priorities and Negotiation . . . . . 215
3.4.1 Real User Priority (RUP) . . . . . 216
3.4.2 Effective User Priority (EUP) . . . . . 216
3.4.3 Priorities and Preemption . . . . . 217
3.4.4 Priority Calculation . . . . . 217
3.4.5 Negotiation . . . . . 217

3.4.6 Group Accounting . . . . . 219
3.4.7 Group Quotas . . . . . 220
3.5 Startd Policy Configuration . . . . . 221
3.5.1 Startd ClassAd Attributes . . . . . 222
3.5.2 Job ClassAd Attributes . . . . . 226
3.5.3 The START expression . . . . . 232
3.5.4 The IS_VALID_CHECKPOINT_PLATFORM expression . . . . . 233
3.5.5 The RANK expression . . . . . 234
3.5.6 Machine States . . . . . 235
3.5.7 Machine Activities . . . . . 238
3.5.8 State and Activity Transitions . . . . . 239
3.5.9 State/Activity Transition Expression Summary . . . . . 248
3.5.10 Policy Settings . . . . . 250
3.5.11 Differences from the Version 6.0 Policy Settings . . . . . 259
3.6 Security . . . . . 260
3.6.1 Condor's Security Model . . . . . 261
3.6.2 Security Negotiation . . . . . 262
3.6.3 Authentication . . . . . 265
3.6.4 The Unified Map File for Authentication . . . . . 274
3.6.5 Encryption . . . . . 275
3.6.6 Integrity . . . . . 276
3.6.7 Example of Daemon-Side Security Configuration . . . . . 277
3.6.8 Authorization . . . . . 278
3.6.9 Security Sessions . . . . . 281
3.6.10 Host-Based Security in Condor . . . . . 282
3.6.11 Using Condor w/ Firewalls, Private Networks, and NATs . . . . . 289
3.6.12 User Accounts in Condor . . . . . 289
3.7 Networking . . . . . 294

3.7.1 Port Usage in Condor . . . . . 295
3.7.2 Configuring Condor for Machines With Multiple Network Interfaces . . . 298
3.7.3 Generic Connection Brokering (GCB) . . . . . 301
3.7.4 Using TCP to Send Updates to the *condor_collector* . . . . . 314
3.8 The Checkpoint Server . . . . . 315
3.8.1 Preparing to Install a Checkpoint Server . . . . . 316
3.8.2 Installing the Checkpoint Server Module . . . . . 316
3.8.3 Configuring your Pool to Use Multiple Checkpoint Servers . . . . . 318
3.8.4 Checkpoint Server Domains . . . . . 319
3.9 DaemonCore . . . . . 320
3.9.1 DaemonCore and Unix signals . . . . . 321
3.9.2 DaemonCore and Command-line Arguments . . . . . 321
3.10 The High Availability of Daemons . . . . . 323
3.10.1 High Availability of the Job Queue . . . . . 323
3.10.2 High Availability of the Central Manager . . . . . 324
3.11 Quill . . . . . 330
3.11.1 Installation and Configuration . . . . . 330
3.11.2 Four Usage Examples . . . . . 334
3.11.3 Quill and Its RDBMS Schema . . . . . 335
3.11.4 Quill and Security . . . . . 337
3.12 Setting Up for Special Environments . . . . . 338
3.12.1 Using Condor with AFS . . . . . 338
3.12.2 Configuring Condor for Multiple Platforms . . . . . 340
3.12.3 Full Installation of condor_compile . . . . . 342
3.12.4 Installing the *condor_kbdd* . . . . . 343
3.12.5 Configuring The CondorView Server . . . . . 344
3.12.6 Running Condor Jobs within a VMware or Xen Virtual Machine Environment346
3.12.7 Configuring The Startd for SMP Machines . . . . . 347

3.12.8 Condor's Dedicated Scheduling . . . . . 355
3.12.9 Configuring Condor for Running Backfill Jobs . . . . . 359
3.13 Java Support Installation . . . . . 365

**4 Miscellaneous Concepts 368**
4.1 Condor's ClassAd Mechanism . . . . . 368
4.1.1 Syntax . . . . . 369
4.1.2 Evaluation Semantics . . . . . 376
4.1.3 ClassAds in the Condor System . . . . . 378
4.2 Condor's Checkpoint Mechanism . . . . . 380
4.2.1 Standalone Checkpointing . . . . . 381
4.2.2 Checkpoint Safety . . . . . 382
4.2.3 Checkpoint Warnings . . . . . 382
4.2.4 Checkpoint Library Interface . . . . . 383
4.3 Computing On Demand (COD) . . . . . 384
4.3.1 Overview of How COD Works . . . . . 385
4.3.2 Authorizing Users to Create and Manage COD Claims . . . . . 385
4.3.3 Defining a COD Application . . . . . 385
4.3.4 Managing COD Resource Claims . . . . . 389
4.3.5 Limitations of COD Support in Condor . . . . . 395
4.4 Application Program Interfaces . . . . . 396
4.4.1 Web Service . . . . . 396
4.4.2 The DRMAA API . . . . . 402
4.4.3 The Command Line Interface . . . . . 403
4.4.4 The Condor GAHP . . . . . 404
4.4.5 The Condor Perl Module . . . . . 404

**5 Grid Computing 411**
5.1 Introduction . . . . . 411

5.2 Connecting Condor Pools with Flocking . . . . . 412
5.2.1 Flocking Configuration . . . . . 412
5.2.2 Job Considerations . . . . . 414
5.3 The Grid Universe . . . . . 414
5.3.1 Condor-C, The condor Grid Type . . . . . 414
5.3.2 Condor-G, the gt2, gt3, and gt4 Grid Types . . . . . 418
5.3.3 The nordugrid Grid Type . . . . . 432
5.3.4 The unicore Grid Type . . . . . 433
5.3.5 The pbs Grid Type . . . . . 433
5.3.6 The lsf Grid Type . . . . . 434
5.3.7 Matchmaking in the Grid Universe . . . . . 434
5.4 Glidein . . . . . 440
5.4.1 What *condor_glidein* Does . . . . . 440
5.4.2 Configuration Requirements in the Local Pool . . . . . 441
5.4.3 Running Jobs on the Remote Grid Resource After Glidein . . . . . 441
5.5 Dynamic Deployment . . . . . 441

**6 Platform-Specific Information 443**

6.1 Linux . . . . . 443
6.1.1 Linux Kernel-specific Information . . . . . 444
6.1.2 Red Hat Version 6.x . . . . . 444
6.1.3 Red Hat Version 7.x . . . . . 444
6.1.4 Red Hat Version 8.x . . . . . 444
6.1.5 Red Hat Version 9.x . . . . . 445
6.1.6 Red Hat Fedora 1, 2, and 3 . . . . . 445
6.2 Microsoft Windows . . . . . 445
6.2.1 Limitations under Windows . . . . . 446
6.2.2 Supported Features under Windows . . . . . 446

6.2.3 Secure Password Storage . . . . . 447
6.2.4 Executing Jobs as the Submitting User . . . . . 448
6.2.5 Details on how Condor for Windows starts/stops a job . . . . . 448
6.2.6 Security Considerations in Condor for Windows . . . . . 450
6.2.7 Network files and Condor . . . . . 451
6.2.8 Interoperability between Condor for Unix and Condor for Windows . . . . 457
6.2.9 Some differences between Condor for Unix -vs- Condor for Windows . . . 458
6.3 Macintosh OS X . . . . . 458
6.3.1 Macintosh OS X Panther (10.3) . . . . . 459
6.3.2 Macintosh OS X Tiger (10.4) . . . . . 459
6.3.3 Macintosh OS X Tiger (10.4) for Intel CPUs . . . . . 459
6.4 AIX . . . . . 459
6.4.1 AIX 5.2L . . . . . 459
6.4.2 AIX 5.1L . . . . . 459

**7 Frequently Asked Questions (FAQ) 461**
7.1 Obtaining & Installing Condor . . . . . 461
7.2 Setting up Condor . . . . . 467
7.3 Running Condor Jobs . . . . . 470
7.4 Condor on Windows . . . . . 479
7.5 Grid Computing . . . . . 484
7.6 Troubleshooting . . . . . 486
7.7 Other questions . . . . . 489

**8 Version History and Release Notes 490**
8.1 Introduction to Condor Versions . . . . . 490
8.1.1 Condor Version Number Scheme . . . . . 490
8.1.2 The Stable Release Series . . . . . 491
8.1.3 The Development Release Series . . . . . 491

8.2 Upgrade Surprises . . . . . 491
8.3 Stable Release Series 6.8 . . . . . 492
8.4 Development Release Series 6.7 . . . . . 513
8.5 Stable Release Series 6.6 . . . . . 570
8.5.1 Version 6.6.12 . . . . . 570

**9 Command Reference Manual (man pages) 595**

*cleanup_release* . . . . . 596
*condor_advertise* . . . . . 598
*condor_check_userlogs* . . . . . 602
*condor_checkpoint* . . . . . 604
*condor_chirp* . . . . . 607
*condor_cod* . . . . . 610
*condor_cold_start* . . . . . 613
*condor_cold_stop* . . . . . 617
*condor_compile* . . . . . 620
*condor_config_bind* . . . . . 623
*condor_config_val* . . . . . 625
*condor_configure* . . . . . 629
*condor_convert_history* . . . . . 633
*condor_dagman* . . . . . 635
*condor_fetchlog* . . . . . 639
*condor_findhost* . . . . . 642
*condor_glidein* . . . . . 645
*condor_history* . . . . . 653
*condor_hold* . . . . . 656
*condor_master* . . . . . 659
*condor_master_off* . . . . . 661

*condor_off* . . . . . 663
*condor_on* . . . . . 666
*condor_preen* . . . . . 669
*condor_prio* . . . . . 671
*condor_q* . . . . . 673
*condor_qedit* . . . . . 681
*condor_reconfig* . . . . . 683
*condor_reconfig_schedd* . . . . . 686
*condor_release* . . . . . 688
*condor_reschedule* . . . . . 691
*condor_restart* . . . . . 694
*condor_rm* . . . . . 697
*condor_run* . . . . . 700
*condor_stats* . . . . . 704
*condor_status* . . . . . 708
*condor_store_cred* . . . . . 714
*condor_submit* . . . . . 716
*condor_submit_dag* . . . . . 742
*condor_transfer_data* . . . . . 746
*condor_updates_stats* . . . . . 748
*condor_userlog* . . . . . 751
*condor_userprio* . . . . . 755
*condor_vacate* . . . . . 759
*condor_vacate_job* . . . . . 762
*condor_version* . . . . . 765
*condor_wait* . . . . . 767
*filelock_midwife* . . . . . 770
*filelock_undertaker* . . . . . 772

*install_release* . . . . . 774
*stork_q* . . . . . 776
*stork_list_cred* . . . . . 778
*stork_rm* . . . . . 780
*stork_rm_cred* . . . . . 782
*stork_store_cred* . . . . . 784
*stork_status* . . . . . 786
*stork_submit* . . . . . 788
*uniq_pid_midwife* . . . . . 792
*uniq_pid_undertaker* . . . . . 794

## CONDOR PUBLIC LICENSE

Condor® Public License
Version 1.1, October 30, 2003

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. For more information contact: Condor Team, Attention: Professor Miron Livny, Dept of Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

This software referred to as the Condor® Version 6.x software ("Software") was developed by the Condor Project, Condor Team, Computer Sciences Department, University of Wisconsin-Madison, under the authority of the Board of Regents of the University of Wisconsin System and includes voluntary contributions made to the Condor Project ("Copyright Holders and Contributors and the University"). For more information on the Condor Project, please see http://www.condorproject.org/.

Installation, use, reproduction, display, modification and redistribution of this Software, with or without modification, in source and binary forms, are permitted. Any exercise of rights under this license including sublicenses by you is subject to the following conditions:

1. Redistributions of this Software, with or without modification, must reproduce this Condor Public License in: (1) the Software, and (2) any user documentation or other similar material which is provided with the Software.
2. Any user documentation included with a redistribution must include the following notice:

   "This product includes software from the Condor® Project (http://www.condorproject.org/)"

   Alternatively, if that is where third-party acknowledgments normally appear, this acknowledgment must be reproduced in the Software itself.
3. Any academic report, publication, or other academic disclosure of results obtained with this Software will acknowledge this Software's use by an appropriate citation.

4. The name Condor® is a registered trademark of the University of Wisconsin-Madison. The trademark may not be used to endorse or promote software, or products derived therefrom, and, other than as required by section 2 and 3 above, it may not be affixed to modified redistributions of this Software without the prior written approval, obtainable via email to condor-admin@cs.wisc.edu.
5. To the extent that patent claims licensable by the University of Wisconsin-Madison are necessarily infringed by the use or sale of the Software, you are granted a non-exclusive, worldwide, royalty- free perpetual license under such patent claims, with the rights for you to make, use, sell, offer to sell, import and otherwise transfer the Software in source code and object code form and derivative works. This patent license shall apply to the combination of the Software with other software if, at the time the Software is added by you, such addition of the Software causes such combination to be covered by such patent claims. This patent license shall not apply to any other combinations which include the Software. No hardware per se is licensed hereunder.

   If you or any subsequent sub-licensee (a "Recipient") institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Software infringes such Recipient's patent(s), then such Recipient's rights granted (directly or indirectly) under the patent license above shall terminate as of the date such litigation is filed. All sublicenses to the Software which have been properly granted prior to termination shall survive any termination of said patent license, if not otherwise terminated pursuant to this section.
6. DISCLAIMER

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS AND THE UNIVERSITY "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, OF SATISFACTORY QUALITY, AND FITNESS FOR A PARTICULAR PURPOSE OR USE ARE DISCLAIMED. THE COPYRIGHT HOLDERS AND CONTRIBUTORS AND THE UNIVERSITY MAKE NO REPRESENTATION THAT THE SOFTWARE, MODIFICATIONS, ENHANCEMENTS OR DERIVATIVE WORKS THEREOF, WILL NOT INFRINGE ANY PATENT, COPYRIGHT, TRADEMARK, TRADE SECRET OR OTHER PROPRIETARY RIGHT.
7. LIMITATION OF LIABILITY

   THE COPYRIGHT HOLDERS AND CONTRIBUTORS AND ANY OTHER OFFICER, AGENT, OR EMPLOYEE OF THE UNIVERSITY SHALL HAVE NO LIABILITY TO LICENSEE OR OTHER PERSONS FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL, EXEMPLARY, OR PUNITIVE DAMAGES OF ANY CHARACTER INCLUDING, WITHOUT LIMITATION, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, LOSS OF USE, DATA OR PROFITS, OR BUSINESS INTERRUPTION, HOWEVER CAUSED AND ON ANY THEORY OF CONTRACT, WARRANTY, TORT (INCLUDING NEGLIGENCE), PRODUCT LIABILITY OR OTHERWISE, ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
8. Certain uses and transfers of the Software or documentation, and/or items or software incorporating the Condor Software or documentation, may require a license under U.S. Export Control laws. Licensee represents and warrants that all uses and transfers of the Condor Software or documentation and/or any items or software incorporating Condor shall be in compliance with U.S. Export Control laws, and Licensee further understands that failure to comply with such export control laws may result in criminal liability to Licensee under U.S. laws.
9. The Condor Team may publish revised and/or new versions of this Condor Public License ("this License") from time to time. Each version will be given a distinguishing version number. Once Software has been published under a particular version of this License, you may always continue to use it under the terms of that version. You may also choose to use such Software under the terms of any subsequent version of this License published by the Condor Team. No one other than the Condor Team has the right to modify the terms of this License.

For more information:
Condor Team
Attention: Professor Miron Livny
7367 Computer Sciences
1210 W. Dayton St.
Madison, WI 53706-1685
miron@cs.wisc.edu
http://www.cs.wisc.edu/~miron/miron.html

**NOTICES**

- This product includes software developed by and/or derived from the Globus Project (http://www.globus.org/) to which the U.S. Government retains certain rights. Copyright (c) 1999 University of Chicago and The University of Southern California. All Rights Reserved.
- Some distributions of Condor include a compiled, unmodified version of the GNU C library. The complete source code to GNU glibc can be found at http://www.gnu.org/software/libc/.

CHAPTER

# ONE

# Overview

## 1.1 High-Throughput Computing (HTC) and its Requirements

For many research and engineering projects, the quality of the research or the product is heavily dependent upon the quantity of computing cycles available. It is not uncommon to find problems that require weeks or months of computation to solve. Scientists and engineers engaged in this sort of work need a computing environment that delivers large amounts of computational power over a long period of time. Such an environment is called a High-Throughput Computing (HTC) environment. In contrast, High Performance Computing (HPC) environments deliver a tremendous amount of compute power over a short period of time. HPC environments are often measured in terms of FLoating point Operations Per Second (FLOPS). A growing community is not concerned about operations per second, but operations per month or per year. Their problems are of a much larger scale. They are more interested in how many jobs they can complete over a long period of time instead of how fast an individual job can complete.

The key to HTC is to efficiently harness the use of all available resources. Years ago, the engineering and scientific community relied on a large, centralized mainframe or a supercomputer to do computational work. A large number of individuals and groups needed to pool their financial resources to afford such a machine. Users had to wait for their turn on the mainframe, and they had a limited amount of time allocated. While this environment was inconvenient for users, the utilization of the mainframe was high; it was busy nearly all the time.

As computers became smaller, faster, and cheaper, users moved away from centralized mainframes and purchased personal desktop workstations and PCs. An individual or small group could afford a computing resource that was available whenever they wanted it. The personal computer is slower than the large centralized machine, but it provides exclusive access. Now, instead of one giant computer for a large institution, there may be hundreds or thousands of personal computers. This

is an environment of distributed ownership, where individuals throughout an organization own their own resources. The total computational power of the institution as a whole may rise dramatically as the result of such a change, but because of distributed ownership, individuals have not been able to capitalize on the institutional growth of computing power. And, while distributed ownership is more convenient for the users, the utilization of the computing power is lower. Many personal desktop machines sit idle for very long periods of time while their owners are busy doing other things (such as being away at lunch, in meetings, or at home sleeping).

## 1.2 Condor's Power

Condor is a software system that creates a High-Throughput Computing (HTC) environment. It effectively utilizes the computing power of workstations that communicate over a network. Condor can manage a dedicated cluster of workstations. Its power comes from the ability to effectively harness non-dedicated, preexisting resources under distributed ownership.

A user submits the job to Condor. Condor finds an available machine on the network and begins running the job on that machine. Condor has the capability to detect that a machine running a Condor job is no longer available (perhaps because the owner of the machine came back from lunch and started typing on the keyboard). It can checkpoint the job and move (migrate) the jobs to a different machine which would otherwise be idle. Condor continues job on the new machine from precisely where it left off.

In those cases where Condor can checkpoint and migrate a job, Condor makes it easy to maximize the number of machines which can run a job. In this case, there is no requirement for machines to share file systems (for example, with NFS or AFS), so that machines across an entire enterprise can run a job, including machines in different administrative domains.

Condor can be a real time saver when a job must be run many (hundreds of) different times, perhaps with hundreds of different data sets. With one command, all of the hundreds of jobs are submitted to Condor. Depending upon the number of machines in the Condor pool, dozens or even hundreds of otherwise idle machines can be running the job at any given moment.

Condor does not require an account (login) on machines where it runs a job. Condor can do this because of its *remote system call* technology, which traps library calls for such operations as reading or writing from disk files. The calls are transmitted over the network to be performed on the machine where the job was submitted.

Condor provides powerful resource management by match-making resource owners with resource consumers. This is the cornerstone of a successful HTC environment. Other compute cluster resource management systems attach properties to the job queues themselves, resulting in user confusion over which queue to use as well as administrative hassle in constantly adding and editing queue properties to satisfy user demands. Condor implements *ClassAds*, a clean design that simplifies the user's submission of jobs.

ClassAds work in a fashion similar to the newspaper classified advertising want-ads. All machines in the Condor pool advertise their resource properties, both static and dynamic, such as

available RAM memory, CPU type, CPU speed, virtual memory size, physical location, and current load average, in a *resource offer* ad. A user specifies a *resource request* ad when submitting a job. The request defines both the required and a desired set of properties of the resource to run the job. Condor acts as a broker by matching and ranking resource offer ads with resource request ads, making certain that all requirements in both ads are satisfied. During this match-making process, Condor also considers several layers of priority values: the priority the user assigned to the resource request ad, the priority of the user which submitted the ad, and desire of machines in the pool to accept certain types of ads over others.

## 1.3 Exceptional Features

**Checkpoint and Migration.** Where programs can be linked with Condor libraries, users of Condor may be assured that their jobs will eventually complete, even in the ever changing environment that Condor utilizes. As a machine running a job submitted to Condor becomes unavailable, the job can be check pointed. The job may continue after migrating to another machine. Condor's periodic checkpoint feature periodically checkpoints a job even in lieu of migration in order to safeguard the accumulated computation time on a job from being lost in the event of a system failure such as the machine being shutdown or a crash.

**Remote System Calls.** Despite running jobs on remote machines, the Condor standard universe execution mode preserves the local execution environment via remote system calls. Users do not have to worry about making data files available to remote workstations or even obtaining a login account on remote workstations before Condor executes their programs there. The program behaves under Condor as if it were running as the user that submitted the job on the workstation where it was originally submitted, no matter on which machine it really ends up executing on.

**No Changes Necessary to User's Source Code.** No special programming is required to use Condor. Condor is able to run non-interactive programs. The checkpoint and migration of programs by Condor is transparent and automatic, as is the use of remote system calls. If these facilities are desired, the user only re-links the program. The code is neither recompiled nor changed.

**Pools of Machines can be Hooked Together.** Flocking is a feature of Condor that allows jobs submitted within a first pool of Condor machines to execute on a second pool. The mechanism is flexible, following requests from the job submission, while allowing the second pool, or a subset of machines within the second pool to set policies over the conditions under which jobs are executed.

**Jobs can be Ordered.** The ordering of job execution required by dependencies among jobs in a set is easily handled. The set of jobs is specified using a directed acyclic graph, where each job is a node in the graph. Jobs are submitted to Condor following the dependencies given by the graph.

**Condor Enables Grid Computing.** As grid computing becomes a reality, Condor is already there. The technique of glidein allows jobs submitted to Condor to be executed on grid machines

in various locations worldwide. As the details of grid computing evolve, so does Condor's ability, starting with Globus-controlled resources.

**Sensitive to the Desires of Machine Owners.** The owner of a machine has complete priority over the use of the machine. An owner is generally happy to let others compute on the machine while it is idle, but wants it back promptly upon returning. The owner does not want to take special action to regain control. Condor handles this automatically.

**ClassAds.** The ClassAd mechanism in Condor provides an extremely flexible, expressive framework for matchmaking resource requests with resource offers. Users can easily request both job requirements and job desires. For example, a user can require that a job run on a machine with 64 Mbytes of RAM, but state a preference for 128 Mbytes, if available. A workstation owner can state a preference that the workstation runs jobs from a specified set of users. The owner can also require that there be no interactive workstation activity detectable at certain hours before Condor could start a job. Job requirements/preferences and resource availability constraints can be described in terms of powerful expressions, resulting in Condor's adaptation to nearly any desired policy.

## 1.4 Current Limitations

**Limitations on Jobs which can Checkpointed** Although Condor can schedule and run any type of process, Condor does have some limitations on jobs that it can transparently checkpoint and migrate:

1. Multi-process jobs are not allowed. This includes system calls such as `fork()`, `exec()`, and `system()`.
2. Interprocess communication is not allowed. This includes pipes, semaphores, and shared memory.
3. Network communication must be brief. A job *may* make network connections using system calls such as `socket()`, but a network connection left open for long periods will delay checkpointing and migration.
4. Sending or receiving the SIGUSR2 or SIGTSTP signals is not allowed. Condor reserves these signals for its own use. Sending or receiving all other signals *is* allowed.
5. Alarms, timers, and sleeping are not allowed. This includes system calls such as `alarm()`, `getitimer()`, and `sleep()`.
6. Multiple kernel-level threads are not allowed. However, multiple user-level threads *are* allowed.
7. Memory mapped files are not allowed. This includes system calls such as `mmap()` and `munmap()`.
8. File locks are allowed, but not retained between checkpoints.

9. All files must be opened read-only or write-only. A file opened for both reading and writing will cause trouble if a job must be rolled back to an old checkpoint image. For compatibility reasons, a file opened for both reading and writing will result in a warning but not an error.
10. A fair amount of disk space must be available on the submitting machine for storing a job's checkpoint images. A checkpoint image is approximately equal to the virtual memory consumed by a job while it runs. If disk space is short, a special *checkpoint server* can be designated for storing all the checkpoint images for a pool.
11. On Digital Unix (OSF/1), HP-UX, and Linux, your job must be statically linked. Dynamic linking is allowed on all other platforms.
12. Reading to or writing from files larger than 2 GB is not supported.

Note: these limitations *only* apply to jobs which Condor has been asked to transparently checkpoint. If job checkpointing is not desired, the limitations above do not apply.

**Security Implications.** Condor does a significant amount of work to prevent security hazards, but loopholes are known to exist. Condor can be instructed to run user programs only as the UNIX user nobody, a user login which traditionally has very restricted access. But even with access solely as user nobody, a sufficiently malicious individual could do such things as fill up `/tmp` (which is world writable) and/or gain read access to world readable files. Furthermore, where the security of machines in the pool is a high concern, only machines where the UNIX user root on that machine can be trusted should be admitted into the pool. Condor provides the administrator with extensive security mechanisms to enforce desired policies.

**Jobs Need to be Re-linked to get Checkpointing and Remote System Calls** Although typically no source code changes are required, Condor requires that the jobs be re-linked with the Condor libraries to take advantage of checkpointing and remote system calls. This often precludes commercial software binaries from taking advantage of these services because commercial packages rarely make their object code available. Condor's other services are still available for these commercial packages.

## 1.5 Availability

Condor is currently available as a free download from the Internet via the World Wide Web at URL http://www.cs.wisc.edu/condor/downloads. Binary distributions of Condor version 6.x are available for the platforms detailed in Table 8.1. A platform is an architecture/operating system combination. Condor binaries are available for most major versions of UNIX, as well as Windows NT.

In the table, *clipped* means that Condor does not support checkpointing or remote system calls on the given platform. This means that *standard* jobs are not supported, only *vanilla* jobs. See section 2.4.1 on page 16 for more details on job universes within Condor and their abilities and limitations.

The Condor source code is not currently available for public download. If you desire the Condor source code, please contact the Condor Team in order to discuss it further (see Section 1.7, on

page 9).

| *Architecture* | *Operating System* |
|---|---|
| Hewlett Packard PA-RISC (both PA7000 and PA8000 series) | - HPUX 11.00 (clipped) |
| Sun SPARC Sun4m, Sun4c, Sun UltraSPARC | - Solaris 8, 9<br>- Solaris 10 (clipped) (Using the Solaris 9 binaries) |
| Intel x86 | - Red Hat Linux 7.1, 7.2, 7.3, 8.0, 9<br>- Debian 4.0 (clipped)<br>- RedHat Enterprise Linux 3<br>- RedHat Enterprise Linux 4 (Using RHEL3 binaries)<br>- Debian Linux 3.1 (sarge) (Using RHEL3 binaries)<br>- Fedora Core 1, 2, 3, 4, 5 (Using RHEL3 binaries)<br>- Macintosh OS X 10.4 (clipped)<br>- Windows 2000 Professional and Server (clipped)<br>- Windows 2003 Server (Win NT 5.0) (clipped)<br>- Windows Vista (Win NT 6.0) (clipped)<br>- Windows XP Professional (Win NT 5.1) (clipped) |
| ALPHA | - Tru64 5.1 (clipped) |
| PowerPC | - Macintosh OS X 10.3, 10.4 (clipped)<br>- AIX 5.2, 5.3 (clipped)<br>- Yellowdog Linux 3.0 (clipped)<br>- SuSE Linux Enterprise Server 9 (clipped) |
| Itanium IA64 | - Red Hat Enterprise Linux 3 (clipped)<br>- SuSE Linux Enterprise Server 8.1 (clipped) |
| Opteron x86_64 | - Red Hat Enterprise Linux 3 |

Table 1.1: Condor Version 6.8.6 supported platforms

NOTE: Other Linux distributions likely work, but are not tested or supported.

For more platform-specific information about Condor's support for various operating systems, see Chapter 6 on page 443.

Jobs submitted to the standard universe utilize *condor_compile* to relink programs with libraries provided by Condor. Table 1.2 lists supported compilers by platform. Other compilers may work, but are not supported.

| Platform | Compiler | Notes |
|---|---|---|
| Solaris (all versions) on SPARC | | |
| | gcc, g++, and g77 | The entire GNU compiler suite must be versions 2.95.3 or 2.95.4 |
| | cc, CC | use the standard, native compiler |
| | f77, f90 | use the standard, native compiler |
| Red Hat Linux, versions 7.2, and 7.3 on Intel x86 | gcc, g++, and g77 | as shipped |
| | pgf77 and pgf90, version 5.0 | Portland Group compilers |
| | pghpf, version 5.0 | Portland Group high performance Fortran compiler |
| | pgcc, version 5.0 | |
| Red Hat Linux, version 8 on x86 | | |
| | gcc, g++, and g77 | version 3.2 |
| | pgf77 and pgf90, version 5.0 | Portland Group compilers |
| | pghpf, version 5.0 | Portland Group high performance Fortran compiler |
| | pgcc, version 5.0 | |
| Red Hat Linux, version 9 on x86 | gcc, g++, and g77 | as shipped |
| Red Hat Enterprise Linux 3, 4 on x86 | gcc, g++, and g77 | as shipped |
| Red Hat Debian Linux 3.1 (sarge) on x86 | gcc up to version 3.4.1 | |
| Fedora Core 1, 2, 3, 4, 5 on x86 | gcc, g++, and g77 | as shipped |

Table 1.2: Supported compilers under Condor Version 6.8.6

The following table, Table 1.3, identifies which platforms support the transfer of large files (greater than 2 Gbyte in length). For vanilla universe jobs and those platforms where large file transfer is supported, the support is automatic.

## 1.6 Contributions to Condor

The quality of the Condor project is enhanced by the contributions of external organizations. We gratefully acknowledge the following contributions.

- The Globus Alliance (http://www.globus.org), for code and assistance in developing Condor-G and the Grid Security Infrastructure (GSI) for authentication and authorization.
- The GOZAL Project from the Computer Science Department of the Technion Israel Institute of Technology (http://www.technion.ac.il/), for their enhancements for Condor's High Avail-

| *Platform* | *Large File Transfer Supported?* |
|---|---|
| Hewlett Packard PA-RISC with HPUX 11.00 | Yes |
| Sun SPARC Sun4m,Sun4c, Sun UltraSPARC with Solaris 8, 9 | Yes |
| Intel x86 with Red Hat Linux 7.1, 7.2, 7,3, 8.0, 9 | Yes |
| Intel x86 with Debian 4.0 | Yes |
| Intel x86 with Enterprise Linux 3, Debian Linux 3.1 (sarge) | Yes |
| Intel x86 with Fedora Core 1, 2, 3, 4, 5 | Yes |
| Intel x86 with Macintosh OS X | No |
| Intel x86 with Windows 2000 Professional and Server | Yes |
| Intel x86 with 2003 Server (Win NT 5.0) | Yes |
| Intel x86 with Windows Vista (Win NT 6.0) | Yes |
| Intel x86 with Windows XP Professional (Win NT 5.1) | Yes |
| ALPHA with Tru64 5.1 | No |
| PowerPC with Macintosh OS X | No |
| PowerPC with AIX 5.2 | Yes |
| PowerPC with Yellowdog Linux 3.0 | Yes |
| Itanium with Red Hat Enterprise Linux 3 | Yes |
| Itanium with SuSE Linux Enterprise 8.1, 9 | Yes |
| Opteron x86_64 with Enterprise Linux 3 | Yes |

Table 1.3: Supported platforms for large file transfer of vanilla universe job files

ability. The *condor_had* daemon allows one of multiple machines to function as the central manager for a Condor pool. Therefore, if an acting central manager fails, another can take its place.

- Micron Corporation (http://www.micron.com/) for the MSI-based installer for Condor on Windows.
- Paradyn Project (http://www.paradyn.org/) and the Universitat Autònoma de Barcelona (http://www.caos.uab.es/) for work on the Tool Daemon Protocol (TDP).

Our Web Services API acknowledges the use of gSOAP with their requested wording:

- Part of the software embedded in this product is gSOAP software. Portions created by gSOAP are Copyright (C) 2001-2004 Robert A. van Engelen, Genivia inc. All Rights Reserved.

  THE SOFTWARE IN THIS PRODUCT WAS IN PART PROVIDED BY GENIVIA INC AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING

NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Some distributions of Condor include the Google Coredumper library (http://goog-coredumper.sourceforge.net/). The Google Coredumper library is released under these terms:

  Copyright (c) 2005, Google Inc.
  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  – Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  – Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  – Neither the name of Google Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ”AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## 1.7 Contact Information

The latest software releases, publications/papers regarding Condor and other High-Throughput Computing research can be found at the official web site for Condor at http://www.cs.wisc.edu/condor.

In addition, there is an e-mail list at condor-world@cs.wisc.edu. The Condor Team uses this e-mail list to announce new releases of Condor and other major Condor-related news items. To subscribe or unsubscribe from the the list, follow the instructions at http://www.cs.wisc.edu/condor/mail-lists/. Because many of us receive too much e-mail as it is, you will be happy to know that the Condor World e-mail list group is moderated, and only major announcements of wide interest are distributed.

Our users support each other by belonging to an unmoderated mailing list targeted at solving problems with Condor. Condor team members attempt to monitor traffic to Condor Users, responding as they can. Follow the instructions at http://www.cs.wisc.edu/condor/mail-lists/.

Finally, you can reach the Condor Team directly. The Condor Team is comprised of the developers and administrators of Condor at the University of Wisconsin-Madison. Condor questions, comments, pleas for help, and requests for commercial contract consultation or support are all welcome; send Internet e-mail to mailto:condor-admin@cs.wisc.edu. Please include your name, organization, and telephone number in your message. If you are having trouble with Condor, please help us troubleshoot by including as much pertinent information as you can, including snippets of Condor log files.

## 1.8 Privacy Notice

The Condor software periodically sends short messages to the Condor Project developers at the University of Wisconsin, reporting totals of machines and jobs in each running Condor system. An example of such a message is given below.

The Condor Project uses these collected reports to publish summary figures and tables, such as the total of Condor systems worldwide, or the geographic distribution of Condor systems. This information helps the Condor Project to understand the scale and composition of Condor in the real world and improve the software accordingly.

The Condor Project will not use these reports to publicly identify any Condor system or user without permission. The Condor software does not collect or report any personal information about individual users.

We hope that you will contribute to the development of Condor through this reporting feature. However, you are free to disable it at any time by changing the configuration variables `CONDOR_DEVELOPERS` and `CONDOR_DEVELOPERS_COLLECTOR`, both described in section 3.3.17 of this manual.

Example of data reported:

```
This is an automated email from the Condor system
on machine "your.condor.pool.com".  Do not reply.

This Collector has the following IDs:
    CondorVersion: 6.6.0 Nov 12 2003
    CondorPlatform: INTEL-LINUX-GLIBC22

                     Machines Owner Claimed Unclaimed Matched Preempting

          INTEL/LINUX      810    52     716        37       0          5
        INTEL/WINNT50      120     5     115         0       0          0
```

```
SUN4u/SOLARIS28      114    12      92         9       0          1
SUN4x/SOLARIS28        5     1       0         4       0          0
          Total     1049    70     923        50       0          6

    RunningJobs             IdleJobs
            920                 3868
```

CHAPTER

# TWO

Users' Manual

## 2.1 Welcome to Condor

Presenting Condor Version 6.8.6! Condor is developed by the Condor Team at the University of Wisconsin-Madison (UW-Madison), and was first installed as a production system in the UW-Madison Computer Sciences department more than 10 years ago. This Condor pool has since served as a major source of computing cycles to UW faculty and students. For many, it has revolutionized the role computing plays in their research. An increase of one, and sometimes even two, orders of magnitude in the computing throughput of a research organization can have a profound impact on its size, complexity, and scope. Over the years, the Condor Team has established collaborations with scientists from around the world, and it has provided them with access to surplus cycles (one scientist has consumed 100 CPU years!). Today, our department's pool consists of more than 700 desktop Unix workstations and more than 100 Windows 2000 machines. On a typical day, our pool delivers more than 500 CPU days to UW researchers. Additional Condor pools have been established over the years across our campus and the world. Groups of researchers, engineers, and scientists have used Condor to establish compute pools ranging in size from a handful to hundreds of workstations. We hope that Condor will help revolutionize your compute environment as well.

## 2.2 Introduction

In a nutshell, Condor is a specialized batch system for managing compute-intensive jobs. Like most batch systems, Condor provides a queuing mechanism, scheduling policy, priority scheme, and resource classifications. Users submit their compute jobs to Condor, Condor puts the jobs in a queue, runs them, and then informs the user as to the result.

Batch systems normally operate only with dedicated machines. Often termed compute servers, these dedicated machines are typically owned by one organization and dedicated to the sole purpose of running compute jobs. Condor can schedule jobs on dedicated machines. But unlike traditional batch systems, Condor is also designed to effectively utilize non-dedicated machines to run jobs. By being told to only run compute jobs on machines which are currently not being used (no keyboard activity, no load average, no active telnet users, etc), Condor can effectively harness otherwise idle machines throughout a pool of machines. This is important because often times the amount of compute power represented by the aggregate total of all the non-dedicated desktop workstations sitting on people's desks throughout the organization is far greater than the compute power of a dedicated central resource.

Condor has several unique capabilities at its disposal which are geared toward effectively utilizing non-dedicated resources that are not owned or managed by a centralized resource. These include transparent process checkpoint and migration, remote system calls, and ClassAds. Read section 1.2 for a general discussion of these features before reading any further.

## 2.3 Matchmaking with ClassAds

Before you learn about how to submit a job, it is important to understand how Condor allocates resources. Understanding the unique framework by which Condor matches submitted jobs with machines is the key to getting the most from Condor's scheduling algorithm.

Condor simplifies job submission by acting as a matchmaker of ClassAds. Condor's ClassAds are analogous to the classified advertising section of the newspaper. Sellers advertise specifics about what they have to sell, hoping to attract a buyer. Buyers may advertise specifics about what they wish to purchase. Both buyers and sellers list constraints that need to be satisfied. For instance, a buyer has a maximum spending limit, and a seller requires a minimum purchase price. Furthermore, both want to rank requests to their own advantage. Certainly a seller would rank one offer of $50 dollars higher than a different offer of $25. In Condor, users submitting jobs can be thought of as buyers of compute resources and machine owners are sellers.

All machines in a Condor pool advertise their attributes, such as available RAM memory, CPU type and speed, virtual memory size, current load average, along with other static and dynamic properties. This machine ClassAd also advertises under what conditions it is willing to run a Condor job and what type of job it would prefer. These policy attributes can reflect the individual terms and preferences by which all the different owners have graciously allowed their machine to be part of the Condor pool. You may advertise that your machine is only willing to run jobs at night and when there is no keyboard activity on your machine. In addition, you may advertise a preference (rank) for running jobs submitted by you or one of your co-workers.

Likewise, when submitting a job, you specify a ClassAd with your requirements and preferences. The ClassAd includes the type of machine you wish to use. For instance, perhaps you are looking for the fastest floating point performance available. You want Condor to rank available machines based upon floating point performance. Or, perhaps you care only that the machine has a minimum of 128 Mbytes of RAM. Or, perhaps you will take any machine you can get! These job attributes

and requirements are bundled up into a job ClassAd.

Condor plays the role of a matchmaker by continuously reading all the job ClassAds and all the machine ClassAds, matching and ranking job ads with machine ads. Condor makes certain that all requirements in both ClassAds are satisfied.

### 2.3.1 Inspecting Machine ClassAds with condor_status

Once Condor is installed, you will get a feel for what a machine ClassAd does by trying the *condor_status* command. Try the *condor_status* command to get a summary of information from ClassAds about the resources available in your pool. Type *condor_status* and hit enter to see a summary similar to the following:

```
Name       Arch     OpSys        State      Activity   LoadAv Mem  ActvtyTime

adriana.cs INTEL    SOLARIS251   Claimed    Busy       1.000  64    0+01:10:00
alfred.cs. INTEL    SOLARIS251   Claimed    Busy       1.000  64    0+00:40:00
amul.cs.wi SUN4u    SOLARIS251   Owner      Idle       1.000  128   0+06:20:04
anfrom.cs. SUN4x    SOLARIS251   Claimed    Busy       1.000  32    0+05:16:22
anthrax.cs INTEL    SOLARIS251   Claimed    Busy       0.285  64    0+00:00:00
astro.cs.w INTEL    SOLARIS251   Claimed    Busy       0.949  64    0+05:30:00
aura.cs.wi SUN4u    SOLARIS251   Owner      Idle       1.043  128   0+14:40:15

                                      ...
```

The *condor_status* command has options that summarize machine ads in a variety of ways. For example,

***condor_status -available*** shows only machines which are willing to run jobs now.

***condor_status -run*** shows only machines which are currently running jobs.

***condor_status -l*** lists the machine ClassAds for all machines in the pool.

Refer to the *condor_status* command reference page located on page 708 for a complete description of the *condor_status* command.

Figure 2.1 shows the complete machine ClassAd for a single workstation: alfred.cs.wisc.edu. Some of the listed attributes are used by Condor for scheduling. Other attributes are for information purposes. An important point is that *any* of the attributes in a machine ad can be utilized at job submission time as part of a request or preference on what machine to use. Additional attributes can be easily added. For example, your site administrator can add a physical location attribute to your machine ClassAds.

```
MyType = "Machine"
TargetType = "Job"
Name = "alfred.cs.wisc.edu"
Machine = "alfred.cs.wisc.edu"
StartdIpAddr = "<128.105.83.11:32780>"
Arch = "INTEL"
OpSys = "SOLARIS251"
UidDomain = "cs.wisc.edu"
FileSystemDomain = "cs.wisc.edu"
State = "Unclaimed"
EnteredCurrentState = 892191963
Activity = "Idle"
EnteredCurrentActivity = 892191062
VirtualMemory = 185264
Disk = 35259
KFlops = 19992
Mips = 201
LoadAvg = 0.019531
CondorLoadAvg = 0.000000
KeyboardIdle = 5124
ConsoleIdle = 27592
Cpus = 1
Memory = 64
AFSCell = "cs.wisc.edu"
START = LoadAvg - CondorLoadAvg <= 0.300000 && KeyboardIdle > 15 * 60
Requirements = TRUE
Rank = Owner == "johndoe" || Owner == "friendofjohn"
CurrentRank =  - 1.000000
LastHeardFrom = 892191963
```

Figure 2.1: Sample output from *condor_status -l alfred*

## 2.4 Road-map for Running Jobs

The road to using Condor effectively is a short one. The basics are quickly and easily learned.

Here are all the steps needed to run a job using Condor.

**Code Preparation.** A job run under Condor must be able to run as a background batch job. Condor runs the program unattended and in the background. A program that runs in the background will not be able to do interactive input and output. Condor can redirect console output (stdout and stderr) and keyboard input (stdin) to and from files for you. Create any needed files that contain the proper keystrokes needed for program input. Make certain the program will run correctly with the files.

**The Condor Universe.** Condor has several runtime environments (called a *universe*) from which to choose. Of the universes, two are likely choices when learning to submit a job to Condor: the standard universe and the vanilla universe. The standard universe allows a job running

under Condor to handle system calls by returning them to the machine where the job was submitted. The standard universe also provides the mechanisms necessary to take a checkpoint and migrate a partially completed job, should the machine on which the job is executing become unavailable. To use the standard universe, it is necessary to relink the program with the Condor library using the *condor_compile* command. The manual page for *condor_compile* on page 620 has details.

The vanilla universe provides a way to run jobs that cannot be relinked. There is no way to take a checkpoint or migrate a job executed under the vanilla universe. For access to input and output files, jobs must either use a shared file system, or use Condor's File Transfer mechanism.

Choose a universe under which to run the Condor program, and re-link the program if necessary.

**Submit description file.** Controlling the details of a job submission is a submit description file. The file contains information about the job such as what executable to run, the files to use for keyboard and screen data, the platform type required to run the program, and where to send e-mail when the job completes. You can also tell Condor how many times to run a program; it is simple to run the same program multiple times with multiple data sets.

Write a submit description file to go with the job, using the examples provided in section 2.5.1 for guidance.

**Submit the Job.** Submit the program to Condor with the *condor_submit* command.

Once submitted, Condor does the rest toward running the job. Monitor the job's progress with the *condor_q* and *condor_status* commands. You may modify the order in which Condor will run your jobs with *condor_prio*. If desired, Condor can even inform you in a log file every time your job is checkpointed and/or migrated to a different machine.

When your program completes, Condor will tell you (by e-mail, if preferred) the exit status of your program and various statistics about its performances, including time used and I/O performed. If you are using a log file for the job(which is recommended) the exit status will be recorded in the log file. You can remove a job from the queue prematurely with *condor_rm*.

### 2.4.1 Choosing a Condor Universe

A *universe* in Condor defines an execution environment. Condor Version 6.8.6 supports several different universes for user jobs:

- Standard
- Vanilla
- PVM
- MPI

- Grid
- Java
- Scheduler
- Local
- Parallel

The `universe` attribute is specified in the submit description file. If a universe is not specified, the default is standard.

The standard universe provides migration and reliability, but has some restrictions on the programs that can be run. The vanilla universe provides fewer services, but has very few restrictions. The PVM universe is for programs written to the Parallel Virtual Machine interface. See section 2.9 for more about PVM and Condor. The MPI universe is for programs written to the MPICH interface. See section 2.10.5 for more about MPI and Condor. The MPI Universe has been superseded by the parallel universe. The grid universe allows users to submit jobs using Condor's interface. These jobs are submitted for execution on grid resources. The java universe allows users to run jobs written for the Java Virtual Machine (JVM). The scheduler universe allows users to submit lightweight jobs to be spawned by the *condor_schedd* daemon on the submit host itself. The parallel universe is for programs that require multiple machines for one job. See section 2.10 for more about the Parallel universe

**Standard Universe**

In the standard universe, Condor provides *checkpointing* and *remote system calls*. These features make a job more reliable and allow it uniform access to resources from anywhere in the pool. To prepare a program as a standard universe job, it must be relinked with *condor_compile*. Most programs can be prepared as a standard universe job, but there are a few restrictions.

Condor checkpoints a job at regular intervals. A *checkpoint image* is essentially a snapshot of the current state of a job. If a job must be migrated from one machine to another, Condor makes a checkpoint image, copies the image to the new machine, and restarts the job continuing the job from where it left off. If a machine should crash or fail while it is running a job, Condor can restart the job on a new machine using the most recent checkpoint image. In this way, jobs can run for months or years even in the face of occasional computer failures.

Remote system calls make a job perceive that it is executing on its home machine, even though the job may execute on many different machines over its lifetime. When a job runs on a remote machine, a second process, called a *condor_shadow* runs on the machine where the job was submitted. When the job attempts a system call, the *condor_shadow* performs the system call instead and sends the results to the remote machine. For example, if a job attempts to open a file that is stored on the submitting machine, the *condor_shadow* will find the file, and send the data to the machine where the job is running.

To convert your program into a standard universe job, you must use *condor_compile* to relink it with the Condor libraries. Put *condor_compile* in front of your usual link command. You do not need to modify the program's source code, but you do need access to the unlinked object files. A commercial program that is packaged as a single executable file cannot be converted into a standard universe job.

For example, if you would have linked the job by executing:

```
% cc main.o tools.o -o program
```

Then, relink the job for Condor with:

```
% condor_compile cc main.o tools.o -o program
```

There are a few restrictions on standard universe jobs:

1. Multi-process jobs are not allowed. This includes system calls such as `fork()`, `exec()`, and `system()`.
2. Interprocess communication is not allowed. This includes pipes, semaphores, and shared memory.
3. Network communication must be brief. A job *may* make network connections using system calls such as `socket()`, but a network connection left open for long periods will delay checkpointing and migration.
4. Sending or receiving the SIGUSR2 or SIGTSTP signals is not allowed. Condor reserves these signals for its own use. Sending or receiving all other signals *is* allowed.
5. Alarms, timers, and sleeping are not allowed. This includes system calls such as `alarm()`, `getitimer()`, and `sleep()`.
6. Multiple kernel-level threads are not allowed. However, multiple user-level threads *are* allowed.
7. Memory mapped files are not allowed. This includes system calls such as `mmap()` and `munmap()`.
8. File locks are allowed, but not retained between checkpoints.
9. All files must be opened read-only or write-only. A file opened for both reading and writing will cause trouble if a job must be rolled back to an old checkpoint image. For compatibility reasons, a file opened for both reading and writing will result in a warning but not an error.
10. A fair amount of disk space must be available on the submitting machine for storing a job's checkpoint images. A checkpoint image is approximately equal to the virtual memory consumed by a job while it runs. If disk space is short, a special *checkpoint server* can be designated for storing all the checkpoint images for a pool.

11. On Digital Unix (OSF/1), HP-UX, and Linux, your job must be statically linked. Dynamic linking is allowed on all other platforms.
12. Reading to or writing from files larger than 2 GB is not supported.

#### Vanilla Universe

The vanilla universe in Condor is intended for programs which cannot be successfully re-linked. Shell scripts are another case where the vanilla universe is useful. Unfortunately, jobs run under the vanilla universe cannot checkpoint or use remote system calls. This has unfortunate consequences for a job that is partially completed when the remote machine running a job must be returned to its owner. Condor has only two choices. It can suspend the job, hoping to complete it at a later time, or it can give up and restart the job *from the beginning* on another machine in the pool.

Since Condor's remote system call features cannot be used with the vanilla universe, access to the job's input and output files becomes a concern. One option is for Condor to rely on a shared file system, such as NFS or AFS. Alternatively, Condor has a mechanism for transferring files on behalf of the user. In this case, Condor will transfer any files needed by a job to the execution site, run the job, and transfer the output back to the submitting machine.

Under Unix, the Condor presumes a shared file system for vanilla jobs. However, if a shared file system is unavailable, a user can enable the Condor File Transfer mechanism. On Windows platforms, the default is to use the File Transfer mechanism. For details on running a job with a shared file system, see section 2.5.3 on page 35. For details on using the Condor File Transfer mechanism, see section 2.5.4 on page 36.

#### PVM Universe

The PVM universe allows programs written for the Parallel Virtual Machine interface to be used within the opportunistic Condor environment. Please see section 2.9 for more details.

#### Grid Universe

The Grid universe in Condor is intended to provide the standard Condor interface to users who wish to start jobs intended for remote management systems. Section 5.3 on page 414 has details on using the Grid universe. The manual page for *condor_submit* on page 716 has detailed descriptions of the grid-related attributes.

#### Java Universe

A program submitted to the Java universe may run on any sort of machine with a JVM regardless of its location, owner, or JVM version. Condor will take care of all the details such as finding the JVM binary and setting the classpath.

#### Scheduler Universe

The scheduler universe allows users to submit lightweight jobs to be run immediately, alongside the *condor_schedd* daemon on the submit host itself. Scheduler universe jobs are not matched with a remote machine, and will never be preempted. They do not obey the machine's `requirements` expression.

Originally intended for meta-schedulers such as *condor_dagman*, the scheduler universe can also be used to manage jobs of any sort that must run on the submit host.

However, unlike the local universe, the scheduler universe does not use a *condor_starter* daemon to manage the job, and thus offers limited features and policy support. The local universe is a better choice for most jobs which must run on the submit host, as it offers a richer set of job management features, and is more consistent with other universes such as the vanilla universe. The scheduler universe may be retired in the future, in favor of the newer local universe.

#### Parallel Universe

The parallel universe allows parallel programs, such as MPI jobs, to be run within the opportunistic Condor environment. Please see section 2.10 for more details.

#### Local Universe

The local universe allows a Condor job to be submitted and executed with different assumptions for the execution conditions of the job. The job does not wait to be matched with a machine. It instead executes right away, on the machine where the job is submitted. The job will never be preempted. The machine requirements are not considered for local universe jobs.

## 2.5 Submitting a Job

A job is submitted for execution to Condor using the *condor_submit* command. *condor_submit* takes as an argument the name of a file called a submit description file. This file contains commands and keywords to direct the queuing of jobs. In the submit description file, Condor finds everything it needs to know about the job. Items such as the name of the executable to run, the initial working directory, and command-line arguments to the program all go into the submit description file. *condor_submit* creates a job ClassAd based upon the information, and Condor works toward running the job.

The contents of a submit file can save time for Condor users. It is easy to submit multiple runs of a program to Condor. To run the same program 500 times on 500 different input data sets, arrange your data files accordingly so that each run reads its own input, and each run writes its own output. Each individual run may have its own initial working directory, stdin, stdout, stderr, command-line

arguments, and shell environment. A program that directly opens its own files will read the file names to use either from stdin or from the command line. A program that opens a static filename every time will need to use a separate subdirectory for the output of each run.

The *condor_submit* manual page is on page 716 and contains a complete and full description of how to use *condor_submit*.

### 2.5.1 Sample submit description files

In addition to the examples of submit description files given in the *condor_submit* manual page, here are a few more.

#### Example 1

Example 1 is the simplest submit description file possible. It queues up one copy of the program *foo*(which had been created by *condor_compile*) for execution by Condor. Since no platform is specified, Condor will use its default, which is to run the job on a machine which has the same architecture and operating system as the machine from which it was submitted. No `input`, `output`, and `error` commands are given in the submit description file, so the files `stdin`, `stdout`, and `stderr` will all refer to `/dev/null`. The program may produce output by explicitly opening a file and writing to it. A log file, `foo.log`, will also be produced that contains events the job had during its lifetime inside of Condor. When the job finishes, its exit conditions will be noted in the log file. It is recommended that you always have a log file so you know what happened to your jobs.

```
####################
#
# Example 1
# Simple condor job description file
#
####################

Executable     = foo
Log            = foo.log
Queue
```

#### Example 2

Example 2 queues two copies of the program *mathematica*. The first copy will run in directory `run_1`, and the second will run in directory `run_2`. For both queued copies, `stdin` will be `test.data`, `stdout` will be `loop.out`, and `stderr` will be `loop.error`. There will be two sets of files written, as the files are each written to their own directories. This is a convenient way to organize data if you have a large group of Condor jobs to run. The example file shows

program submission of *mathematica* as a vanilla universe job. This may be necessary if the source and/or object code to program *mathematica* is not available.

```
####################
#
# Example 2: demonstrate use of multiple
# directories for data organization.
#
####################

Executable     = mathematica
Universe = vanilla
input   = test.data
output  = loop.out
error   = loop.error
Log     = loop.log

Initialdir     = run_1
Queue

Initialdir     = run_2
Queue
```

**Example 3**

The submit description file for Example 3 queues 150 runs of program *foo* which has been compiled and linked for Sun workstations running Solaris 8. This job requires Condor to run the program on machines which have greater than 32 megabytes of physical memory, and expresses a preference to run the program on machines with more than 64 megabytes, if such machines are available. It also advises Condor that it will use up to 28 megabytes of memory when running. Each of the 150 runs of the program is given its own process number, starting with process number 0. So, files `stdin`, `stdout`, and `stderr` will refer to `in.0`, `out.0`, and `err.0` for the first run of the program, `in.1`, `out.1`, and `err.1` for the second run of the program, and so forth. A log file containing entries about when and where Condor runs, checkpoints, and migrates processes for the 150 queued programs will be written into file `foo.log`.

```
####################
#
# Example 3: Show off some fancy features including
# use of pre-defined macros and logging.
#
####################

Executable     = foo
```

```
Requirements   = Memory >= 32 && OpSys == "SOLARIS28" && Arch =="SUN4u"
Rank  = Memory >= 64
Image_Size     = 28 Meg

Error   = err.$(Process)
Input   = in.$(Process)
Output  = out.$(Process)
Log = foo.log

Queue 150
```

### 2.5.2 About Requirements and Rank

The `requirements` and `rank` commands in the submit description file are powerful and flexible. Using them effectively requires care, and this section presents those details.

Both `requirements` and `rank` need to be specified as valid Condor ClassAd expressions, however, default values are set by the *condor_submit* program if these are not defined in the submit description file. From the *condor_submit* manual page and the above examples, you see that writing ClassAd expressions is intuitive, especially if you are familiar with the programming language C. There are some pretty nifty expressions you can write with ClassAds. A complete description of ClassAds and their expressions can be found in section 4.1 on page 368.

All of the commands in the submit description file are case insensitive, *except* for the ClassAd attribute string values. ClassAds attribute names are case insensitive, but ClassAd string values are *case preserving*.

Note that the comparison operators (<, >, <=, >=, and ==) compare strings case insensitively. The special comparison operators =?= and =!= compare strings case sensitively.

The allowed ClassAd attributes are those that appear in a machine or a job ClassAd. To see all of the machine ClassAd attributes for all machines in the Condor pool, run *condor_status -l*. The *-l* argument to *condor_status* means to display all the complete machine ClassAds. The job ClassAds, if there jobs in the queue, can be seen with the *condor_q -l* command. This will show you all the available attributes you can play with.

To help you out with what these attributes all signify, descriptions follow for the attributes which will be common to every machine ClassAd. Remember that because ClassAds are flexible, the machine ads in your pool may include additional attributes specific to your site's installation and policies.

#### ClassAd Machine Attributes

**`Activity`:** String which describes Condor job activity on the machine. Can have one of the following values:

**"Idle":** There is no job activity

**"Busy":** A job is busy running

**"Suspended":** A job is currently suspended

**"Vacating":** A job is currently checkpointing

**"Killing":** A job is currently being killed

**"Benchmarking":** The startd is running benchmarks

**Arch:** String with the architecture of the machine. Typically one of the following:

**"INTEL":** Intel x86 CPU (Pentium, Xeon, etc).

**"IA64":** Intel 64-bit CPU

**"ALPHA":** Digital Alpha CPU

**"SGI":** Silicon Graphics MIPS CPU

**"SUN4u":** Sun UltraSparc CPU

**"SUN4x":** A Sun Sparc CPU other than an UltraSparc, i.e. sun4m or sun4c CPU found in older Sparc workstations such as the Sparc 10, Sparc 20, IPC, IPX, etc.

**"PPC":** Power Macintosh

**"HPPA1":** Hewlett Packard PA-RISC 1.x CPU (i.e. PA-RISC 7000 series CPU) based workstation

**"HPPA2":** Hewlett Packard PA-RISC 2.x CPU (i.e. PA-RISC 8000 series CPU) based workstation

**CheckpointPlatform:** A string which opaquely encodes various aspects about a machine's operating system, hardware, and kernel attributes. It is used to identify systems where previously taken checkpoints for the standard universe may resume.

**ClockDay:** The day of the week, where 0 = Sunday, 1 = Monday, . . ., 6 = Saturday.

**ClockMin:** The number of minutes passed since midnight.

**CondorLoadAvg:** The portion of the load average generated by Condor (either from remote jobs or running benchmarks).

**ConsoleIdle:** The number of seconds since activity on the system console keyboard or console mouse has last been detected.

**Cpus:** Number of CPUs in this machine, i.e. 1 = single CPU machine, 2 = dual CPUs, etc.

**CurrentRank:** A float which represents this machine owner's affinity for running the Condor job which it is currently hosting. If not currently hosting a Condor job, `CurrentRank` is 0.0. When a machine is claimed, the attribute's value is computed by evaluating the machine's `Rank` expression with respect to the current job's ClassAd.

**Disk:** The amount of disk space on this machine available for the job in Kbytes ( e.g. 23000 = 23 megabytes ). Specifically, this is the amount of disk space available in the directory specified in the Condor configuration files by the `EXECUTE` macro, minus any space reserved with the `RESERVED_DISK` macro.

**EnteredCurrentActivity:** Time at which the machine entered the current Activity (see `Activity` entry above). On all platforms (including NT), this is measured in the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

**FileSystemDomain:** A "domain" name configured by the Condor administrator which describes a cluster of machines which all access the same, uniformly-mounted, networked file systems usually via NFS or AFS. This is useful for Vanilla universe jobs which require remote file access.

**KeyboardIdle:** The number of seconds since activity on any keyboard or mouse associated with this machine has last been detected. Unlike `ConsoleIdle`, `KeyboardIdle` also takes activity on pseudo-terminals into account (i.e. virtual "keyboard" activity from telnet and rlogin sessions as well). Note that `KeyboardIdle` will always be equal to or less than `ConsoleIdle`.

**KFlops:** Relative floating point performance as determined via a Linpack benchmark.

**LastHeardFrom:** Time when the Condor central manager last received a status update from this machine. Expressed as the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970). Note: This attribute is only inserted by the central manager once it receives the ClassAd. It is not present in the *condor_startd* copy of the ClassAd. Therefore, you could not use this attribute in defining *condor_startd* expressions (and you would not want to).

**LoadAvg:** A floating point number with the machine's current load average.

**Machine:** A string with the machine's fully qualified hostname.

**Memory:** The amount of RAM in megabytes.

**Mips:** Relative integer performance as determined via a Dhrystone benchmark.

**MyType:** The ClassAd type; always set to the literal string `"Machine"`.

**Name:** The name of this resource; typically the same value as the `Machine` attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate virtual machines, each with with a unique name. These names will be of the form "vm#@full.hostname", for example, "vm1@vulture.cs.wisc.edu", which signifies virtual machine 1 from vulture.cs.wisc.edu.

**OpSys:** String describing the operating system running on this machine. For Condor Version 6.8.6 typically one of the following:

**"HPUX10":** for HPUX 10.20

**"HPUX11":** for HPUX B.11.00

**"LINUX":** for LINUX 2.0.x, LINUX 2.2.x, LINUX 2.4.x, or LINUX 2.6.x kernel systems

**"OSF1":** for Digital Unix 4.x

**"SOLARIS25":** for Solaris 2.4 or 5.5

**"SOLARIS251":** for Solaris 2.5.1 or 5.5.1

**"SOLARIS26":** for Solaris 2.6 or 5.6

**"SOLARIS27":** for Solaris 2.7 or 5.7

**"SOLARIS28":** for Solaris 2.8 or 5.8

**"SOLARIS29":** for Solaris 2.9 or 5.9

**"WINNT50":** for Windows 2000

**"WINNT51":** for Windows XP

**"WINNT52":** for Windows Server 2003

**"OSX":** for Darwin

**"OSX10_2":** for Darwin 6.4

**Requirements:** A boolean, which when evaluated within the context of the machine ClassAd and a job ClassAd, must evaluate to TRUE before Condor will allow the job to use this machine.

**MaxJobRetirementTime:** An expression giving the maximum time in seconds that the startd will wait for the job to finish before kicking it off if it needs to do so. This is evaluated in the context of the job ClassAd, so it may refer to job attributes as well as machine attributes.

**StartdIpAddr:** String with the IP and port address of the *condor_startd* daemon which is publishing this machine ClassAd.

**State:** String which publishes the machine's Condor state. Can be:

**"Owner":** The machine owner is using the machine, and it is unavailable to Condor.

**"Unclaimed":** The machine is available to run Condor jobs, but a good match is either not available or not yet found.

**"Matched":** The Condor central manager has found a good match for this resource, but a Condor scheduler has not yet claimed it.

**"Claimed":** The machine is claimed by a remote *condor_schedd* and is probably running a job.

**"Preempting":** A Condor job is being preempted (possibly via checkpointing) in order to clear the machine for either a higher priority job or because the machine owner wants the machine back.

**TargetType:** Describes what type of ClassAd to match with. Always set to the string literal "Job", because machine ClassAds always want to be matched with jobs, and vice-versa.

**UidDomain:** a domain name configured by the Condor administrator which describes a cluster of machines which all have the same passwd file entries, and therefore all have the same logins.

**VirtualMachineID:** For SMP machines, the integer that identifies the VM. The value will be X for the VM with

```
name="vmX@full.hostname"
```

For non-SMP machines with one virtual machine, the value will be 1.

**VirtualMemory:** The amount of currently available virtual memory (swap space) expressed in Kbytes.

In addition, there are a few attributes that are automatically inserted into the machine ClassAd whenever a resource is in the Claimed state:

**ClientMachine:** The hostname of the machine that has claimed this resource

**RemoteOwner:** The name of the user who originally claimed this resource.

**RemoteUser:** The name of the user who is currently using this resource. In general, this will always be the same as the RemoteOwner, but in some cases, a resource can be claimed by one entity that hands off the resource to another entity which uses it. In that case, RemoteUser would hold the name of the entity currently using the resource, while RemoteOwner would hold the name of the entity that claimed the resource.

**PreemptingOwner:** The name of the user who is preempting the job that is currently running on this resource.

**PreemptingUser:** The name of the user who is preempting the job that is currently running on this resource. The relationship between PreemptingUser and PreemptingOwner is the same as the relationship between RemoteUser and RemoteOwner.

**PreemptingRank:** A float which represents this machine owner's affinity for running the Condor job which is waiting for the current job to finish or be preempted. If not currently hosting a Condor job, PreemptingRank is undefined. When a machine is claimed and there is already a job running, the attribute's value is computed by evaluating the machine's Rank expression with respect to the preempting job's ClassAd.

**TotalClaimRunTime:** A running total of the amount of time (in seconds) that all jobs (under the same claim) ran (have spent in the Claimed/Busy state).

**TotalClaimSuspendTime:** A running total of the amount of time (in seconds) that all jobs (under the same claim) have been suspended (in the Claimed/Suspended state).

**TotalJobRunTime:** A running total of the amount of time (in seconds) that a single job ran (has spent in the Claimed/Busy state).

**TotalJobSuspendTime:** A running total of the amount of time (in seconds) that a single job has been suspended (in the Claimed/Suspended state).

There are a few attributes that are only inserted into the machine ClassAd if a job is currently executing. If the resource is claimed but no job are running, none of these attributes will be defined.

**JobId:** The job's identifier (for example, `152.3`), as seen from *condor_q* on the submitting machine.

**JobStart:** The time stamp in integer seconds of when the job began executing, since the Unix epoch (00:00:00 UTC, Jan 1, 1970). For idle machines, the value is UNDEFINED.

**LastPeriodicCheckpoint:** If the job has performed a periodic checkpoint, this attribute will be defined and will hold the time stamp of when the last periodic checkpoint was begun. If the job has yet to perform a periodic checkpoint, or cannot checkpoint at all, the `LastPeriodicCheckpoint` attribute will not be defined.

Finally, the single attribute, `CurrentTime`, is defined by the ClassAd environment.

**CurrentTime:** Evaluates to the the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

#### ClassAd Job Attributes

**Args:** String representing the arguments passed to the job.

**CkptArch:** String describing the architecture of the machine this job executed on at the time it last produced a checkpoint. If the job has never produced a checkpoint, this attribute is `undefined`.

**CkptOpSys:** String describing the operating system of the machine this job executed on at the time it last produced a checkpoint. If the job has never produced a checkpoint, this attribute is `undefined`.

**ClusterId:** Integer cluster identifier for this job. A cluster is a group of jobs that were submitted together. Each job has its own unique job identifier within the cluster, but shares a common cluster identifier. The value changes each time a job or set of jobs are queued for execution under Condor.

**Cmd:** The path to and the file name of the job to be executed.

**CompletionDate:** The time when the job completed, or the value 0 if the job has not yet completed. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

**CumulativeSuspensionTime:** A running total of the number of seconds the job has spent in suspension for the life of the job.

**CurrentHosts:** The number of hosts in the claimed state, due to this job.

**EnteredCurrentStatus:** An integer containing the epoch time of when the job entered into its current status So for example, if the job is on hold, the ClassAd expression

```
CurrentTime - EnteredCurrentStatus
```

will equal the number of seconds that the job has been on hold.

**ExecutableSize:** Size of the executable in Kbytes.

**ExitBySignal:** An attribute that is `True` when a user job exits via a signal and `False` otherwise. For some grid universe jobs, how the job exited is unavailable. In this case, `ExitBySignal` is set to `False`.

**ExitCode:** When a user job exits by means other than a signal, this is the exit return code of the user job. For some grid universe jobs, how the job exited is unavailable. In this case, `ExitCode` is set to 0.

**ExitSignal:** When a user job exits by means of an unhandled signal, this attribute takes on the numeric value of the signal. For some grid universe jobs, how the job exited is unavailable. In this case, `ExitSignal` will be undefined.

**ExitStatus:** The way that Condor previously dealt with a job's exit status. This attribute should no longer be used. It is not always accurate in heterogeneous pools, or if the job exited with a signal. Instead, see the attributes: `ExitBySignal`, `ExitCode`, and `ExitSignal`.

**HoldReasonCode:** An integer value that represents the reason that a job was put on hold.

| *Integer Code* | *Reason for Hold* |
|---|---|
| 1 | The user put the job on hold with *condor_hold*. |
| 2 | Globus middleware reported an error. `HoldReasonSubCode` is the GRAM error number. |
| 3 | The `PERIODIC_HOLD` expression evaluated to `True`. |
| 4 | The credentials for the job are invalid. |
| 5 | A job policy expression evaluated to `Undefined`. |
| 6 | The *condor_starter* failed to start the executable. `HoldReasonSubCode` is the Unix error number. |
| 7 | The standard output file for the job could not be opened. `HoldReasonSubCode` is the Unix error number |
| 8 | The standard input file for the job could not be opened. `HoldReasonSubCode` is the Unix error number. |
| 9 | The standard output stream for the job could not be opened. `HoldReasonSubCode` is the Unix error num |
| 10 | The standard input stream for the job could not be opened. `HoldReasonSubCode` is the Unix error numb |
| 11 | An internal Condor protocol error was encountered when transferring files. |
| 12 | The *condor_starter* failed to download input files. `HoldReasonSubCode` is the Unix error number. |
| 13 | The *condor_starter* failed to upload output files. `HoldReasonSubCode` is the Unix error number. |
| 14 | The initial working directory of the job cannot be accessed. `HoldReasonSubCode` is the Unix error num |

**HoldReasonSubCode:** An integer value that represents further information to go along with the `HoldReasonCode`, for some values of `HoldReasonCode`. See `HoldReasonCode` for the values.

**HoldKillSig:** Currently only for scheduler and local universe jobs, a string containing a name of a signal to be sent to the job if the job is put on hold.

**HoldReason:** A string containing a human-readable message about why a job is on hold. This is the message that will be displayed in response to the command `condor_q -hold`. It can be used to determine if a job should be released or not.

**ImageSize:** Estimate of the memory image size of the job in Kbytes. The initial estimate may be specified in the job submit file. Otherwise, the initial value is equal to the size of the executable. When the job checkpoints, the `ImageSize` attribute is set to the size of the checkpoint file (since the checkpoint file contains the job's memory image). A vanilla universe job's `ImageSize` is recomputed internally every 15 seconds.

**JobLeaseDuration:** The number of seconds set for a job lease, the amount of time that a job may continue running on a remote resource, despite its submitting machine's lack of response. See section 2.14.4 for details on job leases.

**JobPrio:** Integer priority for this job, set by *condor_submit* or *condor_prio*. The default value is 0. The higher the number, the greater (better) the priority.

**JobStartDate:** Time at which the job first began running. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

**JobStatus:** Integer which indicates the current status of the job.

| *Value* | *Status* |
|---|---|
| 0 | Unexpanded (the job has never run) |
| 1 | Idle |
| 2 | Running |
| 3 | Removed |
| 4 | Completed |
| 5 | Held |

**JobUniverse:** Integer which indicates the job universe.

**LastCheckpointPlatform:** An opaque string which is the `CheckpointPlatform` identifier from the last machine where this standard universe job had successfully produced a checkpoint.

**LastCkptServer:** Host name of the last checkpoint server used by this job. When a pool is using multiple checkpoint servers, this tells the job where to find its checkpoint file.

**LastCkptTime:** Time at which the job last performed a successful checkpoint. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

| *Value* | *Universe* |
|---|---|
| 1 | standard |
| 4 | PVM |
| 5 | vanilla |
| 7 | scheduler |
| 8 | MPI |
| 9 | grid |
| 10 | java |
| 11 | parallel |
| 12 | local |

**LastMatchTime:** An integer containing the epoch time when the job was last successfully matched with a resource (gatekeeper) Ad.

**LastRejMatchReason:** If, at any point in the past, this job failed to match with a resource ad, this attribute will contain a string with a human-readable message about why the match failed.

**LastRejMatchTime:** An integer containing the epoch time when Condor-G last tried to find a match for the job, but failed to do so.

**LastSuspensionTime:** Time at which the job last performed a successful suspension. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

**LastVacateTime:** Time at which the job was last evicted from a remote workstation. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

**LocalSysCpu:** An accumulated number of seconds of system CPU time that the job caused to the machine upon which the job was submitted.

**LocalUserCpu:** An accumulated number of seconds of user CPU time that the job caused to the machine upon which the job was submitted.

**MaxHosts:** The maximum number of hosts that this job would like to claim. As long as `CurrentHosts` is the same as `MaxHosts`, no more hosts are negotiated for.

**MaxJobRetirementTime:** Maximum time in seconds to let this job run uninterrupted before kicking it off when it is being preempted. This can only decrease the amount of time from what the corresponding startd expression allows.

**MinHosts:** The minimum number of hosts that must be in the claimed state for this job, before the job may enter the running state.

**NiceUser:** Boolean value which indicates whether this is a nice-user job.

**NumCkpts:** A count of the number of checkpoints written by this job during its lifetime.

**NumGlobusSubmits:** An integer that is incremented each time the *condor_gridmanager* receives confirmation of a successful job submission into Globus.

**NumJobMatches:** An integer that is incremented by the *condor_schedd* each time the job is matched with a resource ad by the negotiator.

**NumRestarts:** A count of the number of restarts from a checkpoint attempted by this job during its lifetime.

**NumSystemHolds:** An integer that is incremented each time Condor-G places a job on hold due to some sort of error condition. This counter is useful, since Condor-G will always place a job on hold when it gives up on some error condition. Note that if the user places the job on hold using the *condor_hold* command, this attribute is not incremented.

**Owner:** String describing the user who submitted this job.

**ProcId:** Integer process identifier for this job. Within a cluster of many jobs, each job has the same `ClusterId`, but will have a unique `ProcId`. Within a cluster, assignment of a `ProcId` value will start with the value 0. The job (process) identifier described here is unrelated to operating system PIDs.

**QDate:** Time at which the job was submitted to the job queue. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

**ReleaseReason:** A string containing a human-readable message about why the job was released from hold.

**RemoteIwd:** The path to the directory in which a job is to be executed on a remote machine.

**RemoteSysCpu:** The total number of seconds of system CPU time (the time spent at system calls) the job used on remote machines.

**RemoteUserCpu:** The total number of seconds of user CPU time the job used on remote machines.

**RemoteWallClockTime:** Cumulative number of seconds the job has been allocated a machine. This also includes time spent in suspension (if any), so the total real time spent running is

```
RemoteWallClockTime - CumulativeSuspensionTime
```

Note that this number does not get reset to zero when a job is forced to migrate from one machine to another.

**RemoveKillSig:** Currently only for scheduler universe jobs, a string containing a name of a signal to be sent to the job if the job is removed.

**StreamErr:** An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, and `TransferErr` is `True`, then standard error is streamed back to the submit machine, instead of doing the transfer (as a whole) after the job completes. If `False`, then standard error is transfered back to the submit machine (as a whole) after the job completes. If `TransferErr` is `False`, then this job attribute is ignored.

**StreamOut:** An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, and `TransferOut` is `True`, then job output is streamed back to the submit machine, instead of doing the transfer (as a whole) after the job completes. If `False`, then job output is transferred back to the submit machine (as a whole) after the job completes. If `TransferOut` is `False`, then this job attribute is ignored.

**TotalSuspensions:** A count of the number of times this job has been suspended during its lifetime.

**TransferErr:** An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the error output from the job is transferred from the remote machine back to the submit machine. The name of the file after transfer is the file referred to by job attribute `Err`. If `False`, no transfer takes place (remote to submit machine), and the name of the file is the file referred to by job attribute `Err`.

**TransferExecutable:** An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the job executable is transferred from the submit machine to the remote machine. The name of the file (on the submit machine) that is transferred is given by the job attribute `Cmd`. If `False`, no transfer takes place, and the name of the file used (on the remote machine) will be as given in the job attribute `Cmd`.

**TransferIn:** An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the job input is transferred from the submit machine to the remote machine. The name of the file that is transferred is given by the job attribute `In`. If `False`, then the job's input is taken from a file on the remote machine (pre-staged), and the name of the file is given by the job attribute `In`.

**TransferOut:** An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the output from the job is transferred from the remote machine back to the submit machine. The name of the file after transfer is the file referred to by job attribute `Out`. If `False`, no transfer takes place (remote to submit machine), and the name of the file is the file referred to by job attribute `Out`.

#### Rank Expression Examples

When considering the match between a job and a machine, rank is used to choose a match from among all machines that satisfy the job's requirements and are available to the user, after accounting for the user's priority and the machine's rank of the job. The rank expressions, simple or complex, define a numerical value that expresses preferences.

The job's `rank` expression evaluates to one of three values. It can be UNDEFINED, ERROR, or a floating point value. If `rank` evaluates to a floating point value, the best match will be the one with the largest, positive value. If no `rank` is given in the submit description file, then Condor substitutes a default value of 0.0 when considering machines to match. If the job's `rank` of a given machine evaluates to UNDEFINED or ERROR, this same value of 0.0 is used. Therefore, the machine is still considered for a match, but has no rank above any other.

A boolean expression evaluates to the numerical value of 1.0 if true, and 0.0 if false.

The following `rank` expressions provide examples to follow.

For a job that desires the machine with the most available memory:

```
Rank = memory
```

For a job that prefers to run on a friend's machine on Saturdays and Sundays:

```
Rank = ( (clockday == 0) || (clockday == 6) )
       && (machine == "friend.cs.wisc.edu")
```

For a job that prefers to run on one of three specific machines:

```
Rank = (machine == "friend1.cs.wisc.edu") ||
       (machine == "friend2.cs.wisc.edu") ||
       (machine == "friend3.cs.wisc.edu")
```

For a job that wants the machine with the best floating point performance (on Linpack benchmarks):

```
Rank = kflops
```

This particular example highlights a difficulty with rank expression evaluation as currently defined. While all machines have floating point processing ability, not all machines will have the `kflops` attribute defined. For machines where this attribute is not defined, `Rank` will evaluate to the value UNDEFINED, and Condor will use a default rank of the machine of 0.0. The `rank` attribute will only rank machines where the attribute is defined. Therefore, the machine with the highest floating point performance may not be the one given the highest rank.

So, it is wise when writing a `rank` expression to check if the expression's evaluation will lead to the expected resulting ranking of machines. This can be accomplished using the *condor_status* command with the *-constraint* argument. This allows the user to see a list of machines that fit a constraint. To see which machines in the pool have `kflops` defined, use

```
condor_status -constraint kflops
```

Alternatively, to see a list of machines where `kflops` is not defined, use

```
condor_status -constraint "kflops=?=undefined"
```

For a job that prefers specific machines in a specific order:

```
Rank = ((machine == "friend1.cs.wisc.edu")*3) +
       ((machine == "friend2.cs.wisc.edu")*2) +
        (machine == "friend3.cs.wisc.edu")
```

If the machine being ranked is `"friend1.cs.wisc.edu"`, then the expression

```
(machine == "friend1.cs.wisc.edu")
```

is true, and gives the value 1.0. The expressions

```
(machine == "friend2.cs.wisc.edu")
```

and

```
(machine == "friend3.cs.wisc.edu")
```

are false, and give the value 0.0. Therefore, `rank` evaluates to the value 3.0. In this way, machine `"friend1.cs.wisc.edu"` is ranked higher than machine `"friend2.cs.wisc.edu"`, machine `"friend2.cs.wisc.edu"` is ranked higher than machine `"friend3.cs.wisc.edu"`, and all three of these machines are ranked higher than others.

### 2.5.3 Submitting Jobs Using a Shared File System

If vanilla, java, parallel (or MPI) universe jobs are submitted without using the File Transfer mechanism, Condor must use a shared file system to access input and output files. In this case, the job *must* be able to access the data files from any machine on which it could potentially run.

As an example, suppose a job is submitted from blackbird.cs.wisc.edu, and the job requires a particular data file called `/u/p/s/psilord/data.txt`. If the job were to run on cardinal.cs.wisc.edu, the file `/u/p/s/psilord/data.txt` must be available through either NFS or AFS for the job to run correctly.

Condor allows users to ensure their jobs have access to the right shared files by using the `FileSystemDomain` and `UidDomain` machine ClassAd attributes. These attributes specify which machines have access to the same shared file systems. All machines that mount the same shared directories in the same locations are considered to belong to the same file system domain. Similarly, all machines that share the same user information (in particular, the same UID, which is important for file systems like NFS) are considered part of the same UID domain.

The default configuration for Condor places each machine in its own UID domain and file system domain, using the full hostname of the machine as the name of the domains. So, if a pool *does* have access to a shared file system, the pool administrator *must* correctly configure Condor such that all the machines mounting the same files have the same `FileSystemDomain` configuration.

Similarly, all machines that share common user information must be configured to have the same `UidDomain` configuration.

When a job relies on a shared file system, Condor uses the `requirements` expression to ensure that the job runs on a machine in the correct `UidDomain` and `FileSystemDomain`. In this case, the default `requirements` expression specifies that the job must run on a machine with the same `UidDomain` and `FileSystemDomain` as the machine from which the job is submitted. This default is almost always correct. However, in a pool spanning multiple `UidDomain`s and/or `FileSystemDomain`s, the user may need to specify a different `requirements` expression to have the job run on the correct machines.

For example, imagine a pool made up of both desktop workstations and a dedicated compute cluster. Most of the pool, including the compute cluster, has access to a shared file system, but some of the desktop machines do not. In this case, the administrators would probably define the `FileSystemDomain` to be `cs.wisc.edu` for all the machines that mounted the shared files, and to the full hostname for each machine that did not. An example is `jimi.cs.wisc.edu`.

In this example, a user wants to submit vanilla universe jobs from her own desktop machine (jimi.cs.wisc.edu) which does not mount the shared file system (and is therefore in its own file system domain, in its own world). But, she wants the jobs to be able to run on more than just her own machine (in particular, the compute cluster), so she puts the program and input files onto the shared file system. When she submits the jobs, she needs to tell Condor to send them to machines that have access to that shared data, so she specifies a different `requirements` expression than the default:

```
Requirements = TARGET.UidDomain == "cs.wisc.edu" && \
               TARGET.FileSystemDomain == "cs.wisc.edu"
```

WARNING: If there is *no* shared file system, or the Condor pool administrator does not configure the `FileSystemDomain` setting correctly (the default is that each machine in a pool is in its own file system and UID domain), a user submits a job that cannot use remote system calls (for example, a vanilla universe job), and the user does not enable Condor's File Transfer mechanism, the job will *only* run on the machine from which it was submitted.

### 2.5.4 Submitting Jobs Without a Shared File System: Condor's File Transfer Mechanism

Condor works well without a shared file system. The Condor file transfer mechanism is utilized by the user when the user submits jobs. Condor will transfer any files needed by a job from the machine where the job was submitted into a temporary working directory on the machine where the job is to be executed. Condor executes the job and transfers output back to the submitting machine. The user specifies which files to transfer, and at what point the output files should be copied back to the submitting machine. This specification is done within the job's submit description file.

The default behavior of the file transfer mechanism varies across the different Condor universes, and it differs between UNIX and Windows machines.

#### Default Behavior across Condor Universes and Platforms

For jobs submitted under the standard universe, the existence of a shared file system is not relevant. Access to files (input and output) is handled through Condor's remote system call mechanism. The executable and checkpoint files are transfered automatically, when needed. Therefore, the user does not need to change the submit description file if there is no shared file system.

For the vanilla, java, MPI, and parallel universes, access to files (including the executable) through a shared file system is presumed as a default on UNIX machines. If there is no shared file system, then Condor's file transfer mechanism must be explicitly enabled. When submitting a job from a Windows machine, Condor presumes the opposite: no access to a shared file system. It instead enables the file transfer mechanism by default. Submission of a job might need to specify which files to transfer, and/or when to transfer the output files back.

For the grid universe, jobs are to be executed on remote machines, so there would never be a shared file system between machines. See section 5.3.2 for more details.

For the PVM universe, file transfer other than the master's executable and files given in **input**, **output**, and **error** commands is not supported. This is not usually an impediment (shared file system or not), since PVM jobs are set up to have the master direct the workers, and I/O from the workers is usually passed back to the master via PVM messages, not files.

For the scheduler universe, Condor is only using the machine from which the job is submitted. Therefore, the existence of a shared file system is not relevant.

#### Specifying If and When to Transfer Files

To enable the file transfer mechanism, two commands are placed in the job's submit description file: **should_transfer_files** and **when_to_transfer_output**. An example is:

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
```

The **should_transfer_files** command specifies whether Condor should transfer input files from the submit machine to the remote machine where the job executes. It also specifies whether the output files are transferred back to the submit machine. The command takes on one of three possible values:

1. `YES`: Condor always transfers both input and output files.
2. `IF_NEEDED`: Condor transfers files if the job is matched with (and to be executed on) a machine in a different `FileSystemDomain` than the one the submit machine belongs to. If the job is matched with a machine in the local `FileSystemDomain`, Condor will not transfer files and relies on a shared file system.
3. `NO`: Condor's file transfer mechanism is disabled.

The **when_to_transfer_output** command tells Condor when output files are to be transferred back to the submit machine after the job has executed on a remote machine. The command takes on one of two possible values:

1. `ON_EXIT`: Condor transfers output files back to the submit machine only when the job exits on its own.
2. `ON_EXIT_OR_EVICT`: Condor will always do the transfer, whether the job completes on its own, is preempted by another job, vacates the machine, or is killed. As the job completes on its own, files are transferred back to the directory where the job was submitted, as expected. For the other cases, *files are transferred back at eviction time*. These files are placed in the directory defined by the configuration variable `SPOOL`, not the directory from which the job was submitted. The transferred files are named using the `ClusterId` and `ProcId` job ClassAd attributes. The file name takes the form:

   ```
   cluster<X>.proc<Y>.subproc0
   ```

   where `<X>` is the value of `ClusterId`, and `<Y>` is the value of `ProcId`. As an example, job 735.0 may produce the file

   ```
   $(SPOOL)/cluster735.proc0.subproc0
   ```

   This is only useful if partial runs of the job are valuable. An example of valuable partial runs is when the application produces its own checkpoints.

There is no default value for **when_to_transfer_output**. If using the file transfer mechanism, this command must be defined. If **when_to_transfer_output** is specified in the submit description file, but **should_transfer_files** is not, Condor assumes a value of `YES` for **should_transfer_files**.

NOTE: The combination of:

```
should_transfer_files = IF_NEEDED
when_to_transfer_output = ON_EXIT_OR_EVICT
```

would produce undefined file access semantics. Therefore, this combination is prohibited by *condor_submit*.

When submitting from a Unix platform, the file transfer mechanism is unused by default. If neither **when_to_transfer_output** or **should_transfer_files** are defined, Condor assumes `should_transfer_files = NO`.

When submitting from a Windows platform, Condor does not provide any way to use a shared file system for jobs. Therefore, if neither **when_to_transfer_output** or **should_transfer_files** are defined, the file transfer mechanism is enabled by default with the following values:

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
```

NOTE: Prior to Condor version 6.5.2, different attributes were used to control when and if files should be transferred. Previously, a single attribute was used to control both things, and the `IF_NEEDED` value was not supported. This older attribute is still allowed in newer versions of Condor but it is now deprecated. **when_to_transfer_output** and **should_transfer_files** should be used instead. However, beware that these settings will not work with Condor versions older than 6.5.2.

#### Specifying What Files to Transfer

If the file transfer mechanism is enabled, Condor will transfer the following files before the job is run on a remote machine.

1. the executable
2. the input, as defined with the **input** command
3. any jar files (for the Java universe)

If the job requires any other input files, the submit description file should utilize the **transfer_input_files** command. This comma-separated list specifies any other files that Condor is to transfer to a remote site to set up the execution environment for the job before it is run. These files are placed in the same temporary working directory as the job's executable. At this time, directories can not be transferred in this way. For example:

```
transfer_input_files = file1,file2
```

As a default, for jobs other than those submitted to the grid universe, any files that are modified or created by the job in the temporary directory at the remote site are transferred back to the machine from which the job was submitted. Most of the time, this is the best option. To restrict the files that are transferred, specify the exact list of files with **transfer_output_files**. Delimite these file names with a comma. When this list is defined, and any of the files do not exist as the job exits, Condor considers this an error, and re-runs the job.

WARNING: Do not specify **transfer_output_files** (for other than grid universe jobs) unless there is a really good reason – it is best to let Condor figure things out by itself based upon what output the job produces.

For grid universe jobs, files to be transferred (other than standard output and standard error) must be specified using **transfer_output_files** in the submit description file.

#### File Paths for File Transfer

The file transfer mechanism specifies file names and/or paths on both the file system of the submit machine and on the file system of the execute machine. Care must be taken to know which machine (submit or execute) is utilizing the file name and/or path.

Files in the **transfer_input_files** command are specified as they are accessed on the submit machine. The program (as it executes) accesses files as they are found on the execute machine.

There are three ways to specify files and paths for **transfer_input_files**:

1. Relative to the submit directory, if the submit command **initialdir** is not specified.
2. Relative to the initial directory, if the submit command **initialdir** is specified.
3. Absolute.

Before executing the program, Condor copies the executable, an input file as specified by the submit command **input**, along with any input files specified by **transfer_input_files**. All these files are placed into a temporary directory (on the execute machine) in which the program runs. Therefore, the executing program must access input files *without* paths. Because all transferred files are placed into a single, flat directory, input files must be uniquely named to avoid collision when transferred. A collision causes the last file in the list to overwrite the earlier one.

If the program creates output files during execution, it must create them within the temporary working directory. Condor transfers back all files within the temporary working directory that have been modified or created. To transfer back only a subset of these files, the submit command **transfer_output_files** is defined. Transfer of files that exist, but are not within the temporary working directory is not supported. Condor's behavior in this instance is undefined.

It is okay to create files outside the temporary working directory on the file system of the execute machine, (in a directory such as `/tmp`) if this directory is guaranteed to exist and be accessible on all possible execute machines. However, transferring such a file back after execution completes may not be done.

Here are several examples to illustrate the use of file transfer. The program executable is called *my_program*, and it uses three command-line arguments as it executes: two input file names and an output file name. The program executable and the submit description file for this job are located in directory `/scratch/test`.

The directory tree for all these examples:

```
/scratch/test (directory)
      my_program.condor (the submit description file)
      my_program (the executable)
      files (directory)
          logs2 (directory)
          in1 (file)
          in2 (file)
      logs (directory)
```

**Example 1** This simple example explicitly transfers input files. These input files to be transferred are specified relative to the directory where the job is submitted. The single out-

put file, `out1`, created when the job is executed will be transferred back into the directory `/scratch/test`, *not* the `files` directory.

```
# file name:  my_program.condor
# Condor submit description file for my_program
Executable      = my_program
Universe        = vanilla
Error           = logs/err.$(cluster)
Output          = logs/out.$(cluster)
Log             = logs/log.$(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = files/in1, files/in2

Arguments       = in1 in2 out1
Queue
```

**Example 2** This second example is identical to Example 1, except that absolute paths to the input files are specified, instead of relative paths to the input files.

```
# file name:  my_program.condor
# Condor submit description file for my_program
Executable      = my_program
Universe        = vanilla
Error           = logs/err.$(cluster)
Output          = logs/out.$(cluster)
Log             = logs/log.$(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = /scratch/test/files/in1, /scratch/test/files/in2

Arguments       = in1 in2 out1
Queue
```

**Example 3** This third example illustrates the use of the submit command **initialdir**, and its effect on the paths used for the various files. The expected location of the executable is not affected by the **initialdir** command. All other files (specified by **input**, **output**, **transfer_input_files**, as well as files modified or created by the job and automatically transferred back) are located relative to the specified **initialdir**. Therefore, the output file, `out1`, will be placed in the `files` directory. Note that the `logs2` directory exists to make this example work correctly.

```
# file name:  my_program.condor
# Condor submit description file for my_program
Executable      = my_program
Universe        = vanilla
Error           = logs2/err.$(cluster)
Output          = logs2/out.$(cluster)
Log             = logs2/log.$(cluster)

initialdir      = files
```

```
should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = in1, in2

Arguments       = in1 in2 out1
Queue
```

**Example 4 – Illustrates an Error** This example illustrates a job that will fail. The files specified using the **transfer_input_files** command work correctly (see Example 1). However, relative paths to files in the **arguments** command cause the executing program to fail. The file system on the submission side may utilize relative paths to files, however those files are placed into a single, flat, temporary directory on the execute machine.

Note that this specification and submission will cause the job to fail and reexecute.

```
# file name:  my_program.condor
# Condor submit description file for my_program
Executable      = my_program
Universe        = vanilla
Error           = logs/err.$(cluster)
Output          = logs/out.$(cluster)
Log             = logs/log.$(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = files/in1, files/in2

Arguments       = files/in1 files/in2 files/out1
Queue
```

This example fails with the following error:

```
err: files/out1: No such file or directory.
```

**Example 5 – Illustrates an Error** As with Example 4, this example illustrates a job that will fail. The executing program's use of absolute paths cannot work.

```
# file name:  my_program.condor
# Condor submit description file for my_program
Executable      = my_program
Universe        = vanilla
Error           = logs/err.$(cluster)
Output          = logs/out.$(cluster)
Log             = logs/log.$(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = /scratch/test/files/in1, /scratch/test/files/in2

Arguments = /scratch/test/files/in1 /scratch/test/files/in2 /scratch/test/files/out1
Queue
```

The job fails with the following error:

```
err: /scratch/test/files/out1: No such file or directory.
```

**Example 6 – Illustrates an Error** This example illustrates a failure case where the executing program creates an output file in a directory other than within the single, flat, temporary directory that the program executes within. The file creation may or may not cause an error, depending on the existence and permissions of the directories on the remote file system.

Further incorrect usage is seen during the attempt to transfer the output file back using the **transfer_output_files** command. The behavior of Condor for this case is undefined.

```
# file name:  my_program.condor
# Condor submit description file for my_program
Executable      = my_program
Universe        = vanilla
Error           = logs/err.$(cluster)
Output          = logs/out.$(cluster)
Log             = logs/log.$(cluster)

should_transfer_files = YES
when_to_transfer_output = ON_EXIT
transfer_input_files = files/in1, files/in2
transfer_output_files = /tmp/out1

Arguments       = in1 in2 /tmp/out1
Queue
```

**Requirements and Rank for File Transfer**

The `requirements` expression for a job must depend on the `should_transfer_files` command. The job must specify the correct logic to ensure that the job is matched with a resource that meets the file transfer needs. If no `requirements` expression is in the submit description file, or if the expression specified does not refer to the attributes listed below, *condor_submit* adds an appropriate clause to the `requirements` expression for the job. *condor_submit* appends these clauses with a logical AND, `&&`, to ensure that the proper conditions are met. Here are the default clauses corresponding to the different values of `should_transfer_files`:

1. `should_transfer_files = YES` results in the addition of the clause `(HasFileTransfer)`. If the job is always going to transfer files, it is required to match with a machine that has the capability to transfer files. This is a backward compatibility issue, since all versions of Condor since version 6.3.3 support file transfer and have `HasFileTransfer` defined to `TRUE`.
2. `should_transfer_files = NO` results in the addition of `(TARGET.FileSystemDomain == MY.FileSystemDomain)`. In addition, Condor automatically adds the `FileSystemDomain` attribute to the job ad, with whatever string is defined for the *condor_schedd* to which the job is submitted. If the job is not using the file transfer mechanism, Condor assumes it will need a shared file system, and therefore, a machine in the same `FileSystemDomain` as the submit machine.

3. `should_transfer_files = IF_NEEDED` results in the addition of

```
(HasFileTransfer || (TARGET.FileSystemDomain == MY.FileSystemDomain))
```

   If Condor will optionally transfer files, it must require that the machine is *either* capable of transferring files *or* in the same file system domain.

To ensure that the job is matched to a machine with enough local disk space to hold all the transfered files, Condor automatically adds the `DiskUsage` job attribute. This attribute includes the total size of the job's executable and all input files to be transferred. Condor then adds an additional clause to the `Requirements` expression that states that the remote machine must have at least enough available disk space to hold all these files:

```
&& (Disk >= DiskUsage)
```

If `should_transfer_files = IF_NEEDED` and the job prefers to run on a machine in the local file system domain over transferring files, (but are still willing to allow the job to run remotely and transfer files), the `rank` expression works well. Use:

```
rank = (TARGET.FileSystemDomain == MY.FileSystemDomain)
```

The `rank` expression is a floating point number, so if other items are considered in ranking the possible machines this job may run on, add the items:

```
rank = kflops + (TARGET.FileSystemDomain == MY.FileSystemDomain)
```

The value of `kflops` can vary widely among machines, so this `rank` expression will likely not do as it intends. To place emphasis on the job running in the same file system domain, but still consider kflops among the machines in the file system domain, weight the part of the rank expression that is matching the file system domains. For example:

```
rank = kflops + (10000 * (TARGET.FileSystemDomain == MY.FileSystemDomain))
```

**Old Attributes for File Transfer**

The `should_transfer_files` and `when_to_transfer_output` commands in the submit description file result in two corresponding string attributes in the job ClassAd: `ShouldTransferFiles` and `WhenToTransferOutput`. These attributes are only defined when the job is matched with an execute machine running Condor version 6.5.3 or a more recent version. So, for backward compatibility, *condor_submit* also includes the old attribute used to control this feature: `TransferFiles`. If you examine a job with the **-long** option to *condor_q*, and you see `TransferFiles`, that attribute is only there for backward compatibility, and it is ignored

if matched with a machine running version 6.5.3 or greater. There were problems with this old attribute, since it was not flexible enough to handle the new `IF_NEEDED` functionality, and it was confusing for users. Therefore, `TransferFiles` is deprecated, and we will no longer document its use. If your submit file refers to `transfer_files`, consider switching it to use the settings described here.

### 2.5.5 Environment Variables

The environment under which a job executes often contains information that is potentially useful to the job. Condor allows a user to both set and reference environment variables for a job or job cluster.

Within a submit description file, the user may define environment variables for the job's environment by using the **environment** command. See the *condor_submit* manual page at section 9 for more details about this command.

The submittor's entire environment can be copied into the job ClassAd for the job at job submission. The **getenv** command within the submit description file does this. See the *condor_submit* manual page at section 9 for more details about this command.

Commands within the submit description file may reference the environment variables of the submitter as a job is submitted. Submit description file commands use `$ENV(EnvironmentVariableName)` to reference the value of an environment variable. Again, see the *condor_submit* manual page at section 9 for more details about this usage.

Condor sets several additional environment variables for each executing job that may be useful for the job to reference.

- `_CONDOR_SCRATCH_DIR` gives the directory where the job may place temporary data files. This directory is unique for every job that is run, and it's contents are deleted by Condor when the job stops running on a machine, no matter how the job completes.
- `CONDOR_VM` gives the name of the virtual machine (for SMP machines), on which the job is run. This setting is only available in the standard universe. See section 3.12.7 for more details about SMP machines and their configuration.
- `X509_USER_PROXY` gives the full path to the X509 user proxy file if one is associated with the job. (Typically a user will specify **x509userproxy** in the submit file.) This setting is currently available in the local, java, and vanilla universes.

### 2.5.6 Heterogeneous Submit: Execution on Differing Architectures

If executables are available for the different platforms of machines in the Condor pool, Condor can be allowed the choice of a larger number of machines when allocating a machine for a job. Modifications to the submit description file allow this choice of platforms.

A simplified example is a cross submission. An executable is available for one platform, but the submission is done from a different platform. Given the correct executable, the `requirements` command in the submit description file specifies the target architecture. For example, an executable compiled for a Sun 4, submitted from an Intel architecture running Linux would add the `requirement`

```
requirements = Arch == "SUN4x" && OpSys == "SOLARIS251"
```

Without this `requirement`, *condor_submit* will assume that the program is to be executed on a machine with the same platform as the machine where the job is submitted.

Cross submission works for both `standard` and `vanilla` universes. The burden is on the user to both obtain and specify the correct executable for the target architecture. To list the architecture and operating systems of the machines in a pool, run *condor_status*.

**Vanilla Universe Example for Execution on Differing Architectures**

A more complex example of a heterogeneous submission occurs when a job may be executed on many different architectures to gain full use of a diverse architecture and operating system pool. If the executables are available for the different architectures, then a modification to the submit description file will allow Condor to choose an executable after an available machine is chosen.

A special-purpose Machine Ad substitution macro can be used in the `executable`, `environment`, and `arguments` attributes in the submit description file. The macro has the form

```
$$(MachineAdAttribute)
```

Note that this macro is ignored in all other submit description attributes. The $$() informs Condor to substitute the requested `MachineAdAttribute` from the machine where the job will be executed.

An example of the heterogeneous job submission has executables available for three platforms: LINUX Intel, Solaris26 Intel, and Solaris 8 Sun. This example uses *povray* to render images using a popular free rendering engine.

The substitution macro chooses a specific executable after a platform for running the job is chosen. These executables must therefore be named based on the machine attributes that describe a platform. The executables named

```
povray.LINUX.INTEL
povray.SOLARIS26.INTEL
povray.SOLARIS28.SUN4u
```

will work correctly for the macro

```
povray.$$(OpSys).$$(Arch)
```

The executables or links to executables with this name are placed into the initial working directory so that they may be found by Condor. A submit description file that queues three jobs for this example:

```
####################
#
# Example of heterogeneous submission
#
####################

universe     = vanilla
Executable   = povray.$$(OpSys).$$(Arch)
Log          = povray.log
Output       = povray.out.$(Process)
Error        = povray.err.$(Process)

Requirements = (Arch == "INTEL" && OpSys == "LINUX") || \
               (Arch == "INTEL" && OpSys =="SOLARIS26") || \
               (Arch == "SUN4u" && OpSys == "SOLARIS28")

Arguments    = +W1024 +H768 +Iimage1.pov
Queue

Arguments    = +W1024 +H768 +Iimage2.pov
Queue

Arguments    = +W1024 +H768 +Iimage3.pov
Queue
```

These jobs are submitted to the vanilla universe to assure that once a job is started on a specific platform, it will finish running on that platform. Switching platforms in the middle of job execution cannot work correctly.

There are two common errors made with the substitution macro. The first is the use of a non-existent `MachineAdAttribute`. If the specified `MachineAdAttribute` does not exist in the machine's ClassAd, then Condor will place the job in the machine state of hold until the problem is resolved.

The second common error occurs due to an incomplete job set up. For example, the submit description file given above specifies three available executables. If one is missing, Condor report back that an executable is missing when it happens to match the job with a resource that requires the missing binary.

**Standard Universe Example for Execution on Differing Architectures**

Jobs submitted to the standard universe may produce checkpoints. A checkpoint can then be used to start up and continue execution of a partially completed job. For a partially completed job, the checkpoint and the job are specific to a platform. If migrated to a different machine, correct execution requires that the platform must remain the same.

In previous versions of Condor, the author of the heterogeneous submission file would need to write extra policy expressions in the `requirements` expression to force Condor to choose the same type of platform when continuing a checkpointed job. However, since it is needed in the common case, this additional policy is now automatically added to the `requirements` expression. The additional expression is added provided the user does not use `CkptArch` in the `requirements` expression. Condor will remain backward compatible for those users who have explicitly specified `CkptRequirements`–implying use of `CkptArch`, in their `requirements` expression.

The expression added when the attribute `CkptArch` is not specified will default to

```
# Added by Condor
CkptRequirements = ((CkptArch == Arch) || (CkptArch =?= UNDEFINED)) && \
                   ((CkptOpSys == OpSys) || (CkptOpSys =?= UNDEFINED))

Requirements = (<user specified policy>) && $(CkptRequirements)
```

The behavior of the `CkptRequirements` expressions and its addition to `requirements` is as follows. The `CkptRequirements` expression guarantees correct operation in the two possible cases for a job. In the first case, the job has not produced a checkpoint. The ClassAd attributes `CkptArch` and `CkptOpSys` will be undefined, and therefore the meta operator (`=?=`) evaluates to true. In the second case, the job has produced a checkpoint. The Machine ClassAd is restricted to require further execution only on a machine of the same platform. The attributes `CkptArch` and `CkptOpSys` will be defined, ensuring that the platform chosen for further execution will be the same as the one used just before the checkpoint.

Note that this restriction of platforms also applies to platforms where the executables are binary compatible.

The complete submit description file for this example:

```
####################
#
# Example of heterogeneous submission
#
####################

universe     = standard
Executable   = povray.$$(OpSys).$$(Arch)
Log          = povray.log
Output       = povray.out.$(Process)
```

```
Error        = povray.err.$(Process)

# Condor automatically adds the correct expressions to insure that the
# checkpointed jobs will restart on the correct platform types.
Requirements = ( (Arch == "INTEL" && OpSys == "LINUX") || \
               (Arch == "INTEL" && OpSys =="SOLARIS26") || \
               (Arch == "SUN4u" && OpSys == "SOLARIS28") )

Arguments    = +W1024 +H768 +Iimage1.pov
Queue

Arguments    = +W1024 +H768 +Iimage2.pov
Queue

Arguments    = +W1024 +H768 +Iimage3.pov
Queue
```

## 2.6 Managing a Job

This section provides a brief summary of what can be done once jobs are submitted. The basic mechanisms for monitoring a job are introduced, but the commands are discussed briefly. You are encouraged to look at the man pages of the commands referred to (located in Chapter 9 beginning on page 595) for more information.

When jobs are submitted, Condor will attempt to find resources to run the jobs. A list of all those with jobs submitted may be obtained through *condor_status* with the *-submitters* option. An example of this would yield output similar to:

```
%  condor_status -submitters

Name                Machine      Running IdleJobs HeldJobs

ballard@cs.wisc.edu  bluebird.c         0       11        0
nice-user.condor@cs. cardinal.c         6      504        0
wright@cs.wisc.edu   finch.cs.w         1        1        0
jbasney@cs.wisc.edu  perdita.cs         0        0        5

                          RunningJobs           IdleJobs           HeldJobs

 ballard@cs.wisc.edu                0                 11                  0
 jbasney@cs.wisc.edu                0                  0                  5
nice-user.condor@cs.                6                504                  0
  wright@cs.wisc.edu                1                  1                  0

               Total                7                516                  5
```

### 2.6.1 Checking on the progress of jobs

At any time, you can check on the status of your jobs with the *condor_q* command. This command displays the status of all queued jobs. An example of the output from *condor_q* is

```
%  condor_q

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 125.0   jbasney         4/10 15:35   0+00:00:00 I  -10 1.2  hello.remote
 127.0   raman           4/11 15:35   0+00:00:00 R  0   1.4  hello
 128.0   raman           4/11 15:35   0+00:02:33 I  0   1.4  hello

3 jobs; 2 idle, 1 running, 0 held
```

This output contains many columns of information about the queued jobs. The ST column (for status) shows the status of current jobs in the queue. An R in the status column means the the job is currently running. An I stands for idle. The job is not running right now, because it is waiting for a machine to become available. The status H is the hold state. In the hold state, the job will not be scheduled to run until it is released (see the *condor_hold* reference page located on page 656 and the *condor_release* reference page located on page 688). Older versions of Condor used a U in the status column to stand for unexpanded. In this state, a job has never produced a checkpoint, and when the job starts running, it will start running from the beginning. Newer versions of Condor do not use the U state.

The CPU_USAGE time reported for a job is the time that has been committed to the job. It is not updated for a job until the job checkpoints. At that time, the job has made guaranteed forward progress. Depending upon how the site administrator configured the pool, several hours may pass between checkpoints, so do not worry if you do not observe the CPU_USAGE entry changing by the hour. Also note that this is actual CPU time as reported by the operating system; it is not time as measured by a wall clock.

Another useful method of tracking the progress of jobs is through the user log. If you have specified a log command in your submit file, the progress of the job may be followed by viewing the log file. Various events such as execution commencement, checkpoint, eviction and termination are logged in the file. Also logged is the time at which the event occurred.

When your job begins to run, Condor starts up a *condor_shadow* process on the submit machine. The shadow process is the mechanism by which the remotely executing jobs can access the environment from which it was submitted, such as input and output files.

It is normal for a machine which has submitted hundreds of jobs to have hundreds of shadows running on the machine. Since the text segments of all these processes is the same, the load on the submit machine is usually not significant. If, however, you notice degraded performance, you can limit the number of jobs that can run simultaneously through the MAX_JOBS_RUNNING configuration parameter. Please talk to your system administrator for the necessary configuration change.

You can also find all the machines that are running your job through the *condor_status* command. For example, to find all the machines that are running jobs submitted by "breach@cs.wisc.edu," type:

```
%  condor_status -constraint 'RemoteUser == "breach@cs.wisc.edu"'

Name       Arch     OpSys        State      Activity   LoadAv Mem  ActvtyTime

alfred.cs. INTEL    SOLARIS251   Claimed    Busy       0.980  64    0+07:10:02
biron.cs.w INTEL    SOLARIS251   Claimed    Busy       1.000  128   0+01:10:00
cambridge. INTEL    SOLARIS251   Claimed    Busy       0.988  64    0+00:15:00
falcons.cs INTEL    SOLARIS251   Claimed    Busy       0.996  32    0+02:05:03
happy.cs.w INTEL    SOLARIS251   Claimed    Busy       0.988  128   0+03:05:00
istat03.st INTEL    SOLARIS251   Claimed    Busy       0.883  64    0+06:45:01
istat04.st INTEL    SOLARIS251   Claimed    Busy       0.988  64    0+00:10:00
istat09.st INTEL    SOLARIS251   Claimed    Busy       0.301  64    0+03:45:00
...
```

To find all the machines that are running any job at all, type:

```
%  condor_status -run

Name       Arch     OpSys        LoadAv RemoteUser           ClientMachine

adriana.cs INTEL    SOLARIS251   0.980  hepcon@cs.wisc.edu   chevre.cs.wisc.
alfred.cs. INTEL    SOLARIS251   0.980  breach@cs.wisc.edu   neufchatel.cs.w
amul.cs.wi SUN4u    SOLARIS251   1.000  nice-user.condor@cs. chevre.cs.wisc.
anfrom.cs. SUN4x    SOLARIS251   1.023  ashoks@jules.ncsa.ui jules.ncsa.uiuc
anthrax.cs INTEL    SOLARIS251   0.285  hepcon@cs.wisc.edu   chevre.cs.wisc.
astro.cs.w INTEL    SOLARIS251   1.000  nice-user.condor@cs. chevre.cs.wisc.
aura.cs.wi SUN4u    SOLARIS251   0.996  nice-user.condor@cs. chevre.cs.wisc.
balder.cs. INTEL    SOLARIS251   1.000  nice-user.condor@cs. chevre.cs.wisc.
bamba.cs.w INTEL    SOLARIS251   1.574  dmarino@cs.wisc.edu  riola.cs.wisc.e
bardolph.c INTEL    SOLARIS251   1.000  nice-user.condor@cs. chevre.cs.wisc.
...
```

### 2.6.2 Removing a job from the queue

A job can be removed from the queue at any time by using the *condor_rm* command. If the job that is being removed is currently running, the job is killed without a checkpoint, and its queue entry is removed. The following example shows the queue of jobs before and after a job is removed.

```
%  condor_q

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 125.0   jbasney         4/10 15:35   0+00:00:00 I  -10 1.2  hello.remote
 132.0   raman           4/11 16:57   0+00:00:00 R  0   1.4  hello

2 jobs; 1 idle, 1 running, 0 held

%  condor_rm 132.0
Job 132.0 removed.

%  condor_q

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
```

```
 125.0   jbasney         4/10 15:35   0+00:00:00 I  -10 1.2  hello.remote

1 jobs; 1 idle, 0 running, 0 held
```

### 2.6.3 Placing a job on hold

A job in the queue may be placed on hold by running the command *condor_hold*. A job in the hold state remains in the hold state until later released for execution by the command *condor_release*.

Use of the *condor_hold* command causes a hard kill signal to be sent to a currently running job (one in the running state). For a standard universe job, this means that no checkpoint is generated before the job stops running and enters the hold state. When released, this standard universe job continues its execution using the most recent checkpoint available.

Jobs in universes other than the standard universe that are running when placed on hold will start over from the beginning when released.

The manual page for *condor_hold* on page 656 and the manual page for *condor_release* on page 688 contain usage details.

### 2.6.4 Changing the priority of jobs

In addition to the priorities assigned to each user, Condor also provides each user with the capability of assigning priorities to each submitted job. These job priorities are local to each queue and can be any integer value, with higher values meaning better priority.

The default priority of a job is 0, but can be changed using the *condor_prio* command. For example, to change the priority of a job to -15,

```
%  condor_q raman

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 126.0   raman           4/11 15:06   0+00:00:00 I  0   0.3  hello

1 jobs; 1 idle, 0 running, 0 held

%  condor_prio -p -15 126.0

%  condor_q raman

-- Submitter: froth.cs.wisc.edu : <128.105.73.44:33847> : froth.cs.wisc.edu
 ID      OWNER            SUBMITTED    CPU_USAGE ST PRI SIZE CMD
 126.0   raman           4/11 15:06   0+00:00:00 I  -15 0.3  hello

1 jobs; 1 idle, 0 running, 0 held
```

It is important to note that these *job* priorities are completely different from the *user* priorities assigned by Condor. Job priorities do not impact user priorities. They are only a mechanism for

the user to identify the relative importance of jobs among all the jobs submitted by the user to that specific queue.

### 2.6.5 Why does the job not run?

Users sometimes find that their jobs do not run. There are several reasons why a specific job does not run. These reasons include failed job or machine constraints, bias due to preferences, insufficient priority, and the preemption throttle that is implemented by the *condor_negotiator* to prevent thrashing. Many of these reasons can be diagnosed by using the *-analyze* option of *condor_q*. For example, a job (assigned the cluster.process value of 331228.2359) submitted to the local pool at UW-Madison is not running. Running *condor_q*'s analyzer provided the following information:

```
% condor_q -pool condor -name beak -analyze 331228.2359

Warning:  No PREEMPTION_REQUIREMENTS expression in config file --- assuming FALSE


-- Schedd: beak.cs.wisc.edu : <128.105.146.14:30918>
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
 ---
331228.2359:  Run analysis summary.  Of 819 machines,
    159 are rejected by your job's requirements
    137 reject your job because of their own requirements
    488 match, but are serving users with a better priority in the pool
    11 match, but prefer another specific job despite its worse user-priority
    24 match, but cannot currently preempt their existing job
     0 are available to run your job
```

A second example shows a job that does not run because the job does not have a high enough priority to cause other running jobs to be preempted.

```
% condor_q -pool condor -name beak -analyze 207525.0

Warning:  No PREEMPTION_REQUIREMENTS expression in config file --- assuming FALSE

-- Schedd: beak.cs.wisc.edu : <128.105.146.14:30918>
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
 ---
 207525.000:  Run analysis summary.  Of 818 machines,
     317 are rejected by your job's requirements
     419 reject your job because of their own requirements
     79 match, but are serving users with a better priority in the pool
      3 match, but prefer another specific job despite its worse user-priority
      0 match, but cannot currently preempt their existing job
      0 are available to run your job
        Last successful match: Wed Jan  8 14:57:42 2003
        Last failed match: Fri Jan 10 15:46:45 2003
        Reason for last match failure: insufficient priority
```

While the analyzer can diagnose most common problems, there are some situations that it cannot reliably detect due to the instantaneous and local nature of the information it uses to detect the problem. Thus, it may be that the analyzer reports that resources are available to service the request,

but the job still does not run. In most of these situations, the delay is transient, and the job will run during the next negotiation cycle.

If the problem persists and the analyzer is unable to detect the situation, it may be that the job begins to run but immediately terminates due to some problem. Viewing the job's error and log files (specified in the submit command file) and Condor's `SHADOW_LOG` file may assist in tracking down the problem. If the cause is still unclear, please contact your system administrator.

### 2.6.6 In the log file

In a job's log file are a log of events (a listing of events in chronological order) that occurred during the life of the job. The formatting of the events is always the same, so that they may be machine readable. Four fields are always present, and they will most often be followed by other fields that give further information that is specific to the type of event.

The first field in an event is the numeric value assigned as the event type in a 3-digit format. The second field identifies the job which generated the event. Within parentheses are the ClassAd job attributes of `ClusterId` value, `ProcId` value, and the MPI-specific rank for MPI universe jobs or a set of zeros (for jobs run under universes other than MPI), separated by periods. The third field is the date and time of the event logging. The fourth field is a string that briefly describes the event. Fields that follow the fourth field give further information for the specific event type.

These are all of the events that can show up in a job log file:

**Event Number:** 000
**Event Name:** Job submitted
**Event Description:** This event occurs when a user submits a job. It is the first event you will see for a job, and it should only occur once.

**Event Number:** 001
**Event Name:** Job executing
**Event Description:** This shows up when a job is running. It might occur more than once.

**Event Number:** 002
**Event Name:** Error in executable
**Event Description:** The job couldn't be run because the executable was bad.

**Event Number:** 003
**Event Name:** Job was checkpointed
**Event Description:** The job's complete state was written to a checkpoint file. This might happen without the job being removed from a machine, because the checkpointing can happen periodically.

**Event Number:** 004
**Event Name:** Job evicted from machine
**Event Description:** A job was removed from a machine before it finished, usually for a policy reason: perhaps an interactive user has claimed the computer, or perhaps another job is higher priority.

**Event Number:** 005
**Event Name:** Job terminated
**Event Description:** The job has completed.

**Event Number:** 006
**Event Name:** Image size of job updated
**Event Description:** This is informational. It is referring to the memory that the job is using while running. It does not reflect the state of the job.

**Event Number:** 007
**Event Name:** Shadow exception
**Event Description:** The *condor_shadow*, a program on the submit computer that watches over the job and performs some services for the job, failed for some catastrophic reason. The job will leave the machine and go back into the queue.

**Event Number:** 008
**Event Name:** Generic log event
**Event Description:** Not used.

**Event Number:** 009
**Event Name:** Job aborted
**Event Description:** The user cancelled the job.

**Event Number:** 010
**Event Name:** Job was suspended
**Event Description:** The job is still on the computer, but it is no longer executing. This is usually for a policy reason, like an interactive user using the computer.

**Event Number:** 011
**Event Name:** Job was unsuspended
**Event Description:** The job has resumed execution, after being suspended earlier.

**Event Number:** 012
**Event Name:** Job was held
**Event Description:** The user has paused the job, perhaps with the *condor_hold* command. It was stopped, and will go back into the queue again until it is aborted or released.

**Event Number:** 013
**Event Name:** Job was released
**Event Description:** The user is requesting that a job on hold be re-run.

**Event Number:** 014
**Event Name:** Parallel node executed
**Event Description:** A parallel (MPI) program is running on a node.

**Event Number:** 015
**Event Name:** Parallel node terminated
**Event Description:** A parallel (MPI) program has completed on a node.

**Event Number:** 016
**Event Name:** POST script terminated
**Event Description:** A node in a DAGMan workflow has a script that should be run after a job. The script is run on the submit host. This event signals that the post script has completed.

**Event Number:** 017
**Event Name:** Job submitted to Globus
**Event Description:** A grid job has been delegated to Globus (version 2, 3, or 4).

**Event Number:** 018
**Event Name:** Globus submit failed
**Event Description:** The attempt to delegate a job to Globus failed.

**Event Number:** 019
**Event Name:** Globus resource up
**Event Description:** The Globus resource that a job wants to run on was unavailable, but is now available.

**Event Number:** 020
**Event Name:** Detected Down Globus Resource
**Event Description:** The Globus resource that a job wants to run on has become unavailable.

**Event Number:** 021
**Event Name:** Remote error
**Event Description:** The *condor_starter* (which monitors the job on the execution machine) has failed.

**Event Number:** 022
**Event Name:** Remote system call socket lost
**Event Description:** The *condor_shadow* and *condor_starter* (which communicate while the job runs) have lost contact.

**Event Number:** 023
**Event Name:** Remote system call socket reestablished
**Event Description:** The *condor_shadow* and *condor_starter* (which communicate while the job runs) have been able to resume contact before the job lease expired.

**Event Number:** 024
**Event Name:** Remote system call reconnect failure
**Event Description:** The *condor_shadow* and *condor_starter* (which communicate while the job runs) were unable to resume contact before the job lease expired.

**Event Number:** 025
**Event Name:** Grid Resource Back Up
**Event Description:** A grid resource that was previously unavailable is now available.

**Event Number:** 026
**Event Name:** Detected Down Grid Resource
**Event Description:** The grid resource that a job is to run on is unavailable.

**Event Number:** 027
**Event Name:** Job submitted to grid resource
**Event Description:** A job has been submitted, and is under the auspices of the grid resource.

### 2.6.7 Job Completion

When your Condor job completes(either through normal means or abnormal termination by signal), Condor will remove it from the job queue (i.e., it will no longer appear in the output of *condor_q*) and insert it into the job history file. You can examine the job history file with the *condor_history* command. If you specified a log file in your submit description file, then the job exit status will be recorded there as well.

By default, Condor will send you an email message when your job completes. You can modify this behavior with the *condor_submit* "notification" command. The message will include the exit status of your job (i.e., the argument your job passed to the exit system call when it completed) or notification that your job was killed by a signal. It will also include the following statistics (as appropriate) about your job:

**Submitted at:** when the job was submitted with *condor_submit*

**Completed at:** when the job completed

**Real Time:** elapsed time between when the job was submitted and when it completed (days hours:minutes:seconds)

**Run Time:** total time the job was running (i.e., real time minus queuing time)

**Committed Time:** total run time that contributed to job completion (i.e., run time minus the run time that was lost because the job was evicted without performing a checkpoint)

**Remote User Time:** total amount of committed time the job spent executing in user mode

**Remote System Time:** total amount of committed time the job spent executing in system mode

**Total Remote Time:** total committed CPU time for the job

**Local User Time:** total amount of time this job's *condor_shadow* (remote system call server) spent executing in user mode

**Local System Time:** total amount of time this job's *condor_shadow* spent executing in system mode

**Total Local Time:** total CPU usage for this job's *condor_shadow*

**Leveraging Factor:** the ratio of total remote time to total system time (a factor below 1.0 indicates that the job ran inefficiently, spending more CPU time performing remote system calls than actually executing on the remote machine)

**Virtual Image Size:** memory size of the job, computed when the job checkpoints

**Checkpoints written:** number of successful checkpoints performed by the job

**Checkpoint restarts:** number of times the job successfully restarted from a checkpoint

**Network:** total network usage by the job for checkpointing and remote system calls

**Buffer Configuration:** configuration of remote system call I/O buffers

**Total I/O:** total file I/O detected by the remote system call library

**I/O by File:** I/O statistics per file produced by the remote system call library

**Remote System Calls:** listing of all remote system calls performed (both Condor-specific and Unix system calls) with a count of the number of times each was performed

## 2.7 Priorities and Preemption

Condor has two independent priority controls: *job* priorities and *user* priorities.

### 2.7.1 Job Priority

Job priorities allow the assignment of a priority level to each submitted Condor job in order to control order of execution. To set a job priority, use the *condor_prio* command — see the example in section 2.6.4, or the command reference page on page 671. Job priorities do not impact user priorities in any fashion. A job priority can be any integer, and higher values are "better".

### 2.7.2 User priority

Machines are allocated to users based upon a user's priority. A lower numerical value for user priority means higher priority, so a user with priority 5 will get more resources than a user with priority 50. User priorities in Condor can be examined with the *condor_userprio* command (see page 755). Condor administrators can set and change individual user priorities with the same utility.

Condor continuously calculates the share of available machines that each user should be allocated. This share is inversely related to the ratio between user priorities. For example, a user with a priority of 10 will get twice as many machines as a user with a priority of 20. The priority of each individual user changes according to the number of resources the individual is using. Each user starts out with the best possible priority: 0.5. If the number of machines a user currently has is greater than the user priority, the user priority will worsen by numerically increasing over time. If the number of machines is less then the priority, the priority will improve by numerically decreasing over time. The long-term result is fair-share access across all users. The speed at which Condor adjusts the priorities is controlled with the configuration macro `PRIORITY_HALFLIFE` , an exponential half-life value. The default is one day. If a user that has user priority of 100 and is utilizing

100 machines removes all his/her jobs, one day later that user's priority will be 50, and two days later the priority will be 25.

Condor enforces that each user gets his/her fair share of machines according to user priority both when allocating machines which become available and by priority preemption of currently allocated machines. For instance, if a low priority user is utilizing all available machines and suddenly a higher priority user submits jobs, Condor will immediately checkpoint and vacate jobs belonging to the lower priority user. This will free up machines that Condor will then give over to the higher priority user. Condor will not starve the lower priority user; it will preempt only enough jobs so that the higher priority user's fair share can be realized (based upon the ratio between user priorities). To prevent thrashing of the system due to priority preemption, the Condor site administrator can define a `PREEMPTION_REQUIREMENTS` expression in Condor's configuration. The default expression that ships with Condor is configured to only preempt lower priority jobs that have run for at least one hour. So in the previous example, in the worse case it could take up to a maximum of one hour until the higher priority user receives his fair share of machines. For a general discussion of limiting preemption, please see section 3.5.10 of the Administrator's manual.

User priorities are keyed on "username@domain", for example "johndoe@cs.wisc.edu". The domain name to use, if any, is configured by the Condor site administrator. Thus, user priority and therefore resource allocation is not impacted by which machine the user submits from or even if the user submits jobs from multiple machines.

An extra feature is the ability to submit a job as a *nice* job (see page 735). Nice jobs artificially boost the user priority by one million just for the nice job. This effectively means that nice jobs will only run on machines that no other Condor job (that is, non-niced job) wants. In a similar fashion, a Condor administrator could set the user priority of any specific Condor user very high. If done, for example, with a guest account, the guest could only use cycles not wanted by other users of the system.

### 2.7.3 Details About How Condor Jobs Vacate Machines

When Condor needs a job to vacate a machine for whatever reason, it sends the job an asynchronous signal specified in the `KillSig` attribute of the job's ClassAd. The value of this attribute can be specified by the user at submit time by placing the **kill_sig** option in the Condor submit description file.

If a program wanted to do some special work when required to vacate a machine, the program may set up a signal handler to use a trappable signal as an indication to clean up. When submitting this job, this clean up signal is specified to be used with **kill_sig**. Note that the clean up work needs to be quick. If the job takes too long to go away, Condor follows up with a SIGKILL signal which immediately terminates the process.

A job that is linked using *condor_compile* and is subsequently submitted into the standard universe, will checkpoint and exit upon receipt of a SIGTSTP signal. Thus, SIGTSTP is the default value for `KillSig` when submitting to the standard universe. The user's code may still checkpoint itself at any time by calling one of the following functions exported by the Condor libraries:

`ckpt()()` Performs a checkpoint and then returns.

`ckpt_and_exit()()` Checkpoints and exits; Condor will then restart the process again later, potentially on a different machine.

For jobs submitted into the vanilla universe, the default value for `KillSig` is SIGTERM, the usual method to nicely terminate a Unix program.

## 2.8 Java Applications

Condor allows users to access a wide variety of machines distributed around the world. The Java Virtual Machine (JVM) provides a uniform platform on any machine, regardless of the machine's architecture or operating system. The Condor Java universe brings together these two features to create a distributed, homogeneous computing environment.

Compiled Java programs can be submitted to Condor, and Condor can execute the programs on any machine in the pool that will run the Java Virtual Machine.

The *condor_status* command can be used to see a list of machines in the pool for which Condor can use the Java Virtual Machine.

```
% condor_status -java

Name          JavaVendor  Ver    State      Activity   LoadAv Mem   ActvtyTime

coral.cs.wisc Sun Microsy 1.2.2  Unclaimed  Idle       0.000   511  0+02:28:04
doc.cs.wisc.e Sun Microsy 1.2.2  Unclaimed  Idle       0.000   511  0+01:05:04
dsonokwa.cs.w Sun Microsy 1.2.2  Unclaimed  Idle       0.000   511  0+01:05:04
...
```

If there is no output from the *condor_status* command, then Condor does not know the location details of the Java Virtual Machine on machines in the pool, or no machines have Java correctly installed. In this case, contact your system administrator or see section 3.13 for more information on getting Condor to work together with Java.

### 2.8.1 Example Java Program

Here is a complete, if simple, example. Start with a simple Java program, `Hello.java`:

```
public class Hello {
        public static void main( String [] args ) {
                System.out.println("Hello, world!\n");
        }
}
```

Build this program using your Java compiler. On most platforms, this is accomplished with the command

```
javac Hello.java
```

Submission to Condor requires a submit description file. If submitting where files are accessible using a shared file system, this simple submit description file works:

```
####################
#
# Example 1
# Execute a single Java class
#
####################

universe       = java
executable     = Hello.class
arguments      = Hello
output         = Hello.output
error          = Hello.error
queue
```

The Java universe must be explicitly selected.

The main class of the program is given in the `executable` statement. This is a file name which contains the entry point of the program. The name of the main class (not a file name) must be specified as the first argument to the program.

If submitting the job where a shared file system is *not* accessible, the submit description file becomes:

```
####################
#
# Example 1
# Execute a single Java class,
# not on a shared file system
#
####################

universe       = java
executable     = Hello.class
arguments      = Hello
output         = Hello.output
error          = Hello.error
should_transfer_files = YES
```

```
when_to_transfer_output = ON_EXIT
queue
```

For more information about using Condor's file transfer mechanisms, see section 2.5.4.

To submit the job, where the submit description file is named `Hello.cmd`, execute

```
condor_submit Hello.cmd
```

To monitor the job, the commands *condor_q* and *condor_rm* are used as with all jobs.

For programs that consist of more than one `.class` file, one option that tells Condor of the additional files adds this line to the submit description file:

```
transfer_input_files = Larry.class,Curly.class,Moe.class
```

If the program consists of a large number of class files, it may be easier to collect them all together into a single Java Archive (JAR) file. A JAR can be created with:

```
% jar cvf Library.jar Larry.class Curly.class Moe.class
```

Condor must then be told where to find the JAR by adding the following to the submit description file:

```
jar_files = Library.jar
```

Note that the JVM must know whether it is receiving JAR files or class files. Therefore, Condor must also be informed, in order to pass the information on to the JVM. That is why there is a difference in submit description file commands for the two ways of specifying files (`transfer_input_files` and `jar_files`).

If the program uses Java features found only in certain JVMs, then inform Condor by adding a `requirements` statement to the submit description file. For example, to require version 3.2, add to the submit description file:

```
requirements = (JavaVersion=="3.2")
```

Each machine with Java capability in a Condor pool will execute a benchmark to determine its speed. The benchmark is taken when Condor is started on the machine, and it uses the SciMark2 (http://math.nist.gov/scimark2) benchmark. The result of the benchmark is held as an attribute within the machine ClassAd. The attribute is called `JavaMFlops`. Jobs that are run under the Java universe (as all other Condor jobs) may prefer or require a machine of a specific speed by setting `rank` or `requirements` in the submit description file. As an example, to execute only on machines of a minimum speed:

```
requirements = (JavaMFlops>4.5)
```

Options to the Java VM itself can be set in the submit description file:

```
java_vm_args = -DMyProperty=Value -verbose:gc
```

These options are those which go after the java command, but before the user's main class. Do not use this to set the classpath, as Condor handles that itself. Setting these options is useful for setting system properties, system assertions and debugging certain kinds of problems.

### 2.8.2 Chirp I/O

If a job has more sophisticated I/O requirements that cannot be met by Condor's file transfer mechanism, then the Chirp facility may provide a solution. Chirp has two advantages over simple, whole-file transfers. First, it permits the input files to be decided upon at run-time rather than submit time, and second, it permits partial-file I/O with results than can be seen as the program executes. However, small changes to the program are required in order to take advantage of Chirp. Depending on the style of the program, use either Chirp I/O streams or UNIX-like I/O functions.

Chirp I/O streams are the easiest way to get started. Modify the program to use the objects `ChirpInputStream` and `ChirpOutputStream` instead of `FileInputStream` and `FileOutputStream`. These classes are completely documented in the Condor Software Developer's Kit (SDK). Here is a simple code example:

```
import java.io.*;
import edu.wisc.cs.condor.chirp.*;

public class TestChirp {

   public static void main( String args[] ) {

      try {
         BufferedReader in = new BufferedReader(
            new InputStreamReader(
               new ChirpInputStream("input")));

         PrintWriter out = new PrintWriter(
            new OutputStreamWriter(
               new ChirpOutputStream("output")));

         while(true) {
            String line = in.readLine();
            if(line==null) break;
```

```
            out.println(line);
         }
         out.close();
      } catch( IOException e ) {
         System.out.println(e);
      }
   }
}
```

To perform UNIX-like I/O with Chirp, create a `ChirpClient` object. This object supports familiar operations such as `open`, `read`, `write`, and `close`. Exhaustive detail of the methods may be found in the Condor SDK, but here is a brief example:

```
import java.io.*;
import edu.wisc.cs.condor.chirp.*;

public class TestChirp {

   public static void main( String args[] ) {

      try {
         ChirpClient client = new ChirpClient();
         String message = "Hello, world!\n";
         byte [] buffer = message.getBytes();

         // Note that we should check that actual==length.
         // However, skip it for clarity.

         int fd = client.open("output","wct",0777);
         int actual = client.write(fd,buffer,0,buffer.length);
         client.close(fd);

         client.rename("output","output.new");
         client.unlink("output.new");

      } catch( IOException e ) {
         System.out.println(e);
      }
   }
}
```

Regardless of which I/O style, the Chirp library must be specified and included with the job. The Chirp JAR (`Chirp.jar`) is found in the `lib` directory of the Condor installation. Copy it into your working directory in order to compile the program after modification to use Chirp I/O.

```
% condor_config_val LIB
/usr/local/condor/lib
% cp /usr/local/condor/lib/Chirp.jar .
```

Rebuild the program with the Chirp JAR file in the class path.

```
% javac -classpath Chirp.jar:. TestChirp.java
```

The Chirp JAR file must be specified in the submit description file. Here is an example submit description file that works for both of the given test programs:

```
universe = java
executable = TestChirp.class
arguments = TestChirp
jar_files = Chirp.jar
queue
```

## 2.9 PVM Applications

Applications that use PVM (Parallel Virtual Machine) may use Condor. PVM offers a set of message passing primitives for use in C and C++ language programs. The primitives, together with the PVM environment allow parallelism at the program level. Multiple processes may run on multiple machines, while communicating with each other. More information about PVM is available at http://www.epm.ornl.gov/pvm/.

Condor-PVM provides a framework to run PVM applications in Condor's opportunistic environment. Where PVM needs dedicated machines to run PVM applications, Condor does not. Condor can be used to dynamically construct PVM virtual machines from a Condor pool of machines.

In Condor-PVM, Condor acts as the resource manager for the PVM daemon. Whenever a PVM program asks for nodes (machines), the request is forwarded to Condor. Condor finds a machine in the Condor pool using usual mechanisms, and adds it to the virtual machine. If a machine needs to leave the pool, the PVM program is notified by normal PVM mechanisms.

NOTE: Condor-PVM is an optional Condor module. It is not automatically installed with Condor. To check and see if it has been installed at your site, enter the command:

```
ls -l `condor_config_val PVMD`
```

Please note the use of back ticks in the above command. They specify to run the *condor_config_val* program. If the result of this program shows the file `condor_pvmd` on your system, then the Condor-PVM module is installed. If not, ask your site administrator to download and install Condor-PVM from http://www.cs.wisc.edu/condor/downloads/.

### 2.9.1 Effective Usage: the Master-Worker Paradigm

There are several different parallel programming paradigms. One of the more common is the *master-worker* (or *pool of tasks*) arrangement. In a master-worker program model, one node acts as the controlling master for the parallel application and sends pieces of work out to worker nodes. The worker node does some computation, and it sends the result back to the master node. The master has a pool of work that needs to be done, so it assigns the next piece of work out to the next worker that becomes available.

Condor-PVM is designed to run PVM applications which follow the master-worker paradigm. Condor runs the master application on the machine where the job was submitted and will not preempt it. Workers are pulled in from the Condor pool as they become available.

Not all parallel programming paradigms lend themselves to Condor's opportunistic environment. In such an environment, any of the nodes could be preempted and disappear at any moment. The master-worker model does work well in this environment. The master keeps track of which piece of work it sends to each worker. The master node is informed of the addition and disappearance of nodes. If the master node is informed that a worker node has disappeared, the master places the unfinished work it had assigned to the disappearing node back into the pool of tasks. This work is sent again to the next available worker node. If the master notices that the number of workers has dropped below an acceptable level, it requests more workers (using `pvm_addhosts()`). Alternatively, the master requests a replacement node every time it is notified that a worker has gone away. The benefit of this paradigm is that the number of workers is not important and changes in the size of the virtual machine are easily handled.

A tool called *MW* has been developed to assist in the development of master-worker style applications for distributed, opportunistic environments like Condor. MW provides a C++ API which hides the complexities of managing a master-worker Condor-PVM application. We suggest that you consider modifying your PVM application to use MW instead of developing your own dynamic PVM master from scratch. Additional information about MW is available at http://www.cs.wisc.edu/condor/mw/.

### 2.9.2 Binary Compatibility and Runtime Differences

Condor-PVM does not define a new API (application program interface); programs use the existing resource management PVM calls such as `pvm_addhosts()` and `pvm_notify()`. Because of this, some master-worker PVM applications are ready to run under Condor-PVM with no changes at all. Regardless of using Condor-PVM or not, it is good master-worker design to handle the case of a disappearing worker node, and therefore many programmers have already constructed their master program with all the necessary fault tolerant logic.

Regular PVM and Condor-PVM are *binary compatible*. The same binary which runs under regular PVM will run under Condor, and vice-versa. There is no need to re-link for Condor-PVM. This permits easy application development (develop your PVM application interactively with the regular PVM console, XPVM, etc.) as well as binary sharing between Condor and some dedicated MPP systems.

This release of Condor-PVM is based on PVM 3.4.2. PVM versions 3.4.0 through 3.4.2 are all supported. The vast majority of the PVM library functions under Condor maintain the same semantics as in PVM 3.4.2, including messaging operations, group operations, and `pvm_catchout()`.

The following list is a summary of the changes and new features of PVM running within the Condor environment:

- Condor introduces the concept of machine class. A pool of machines is likely to contain machines of more than one platform. Under Condor-PVM, machines of different architectures belong to different machine classes. With the concept machine class, Condor can be told what type of machine to allocate. Machine classes are assigned integer values, starting with 0. A machine class is specified in a submit description file when the job is submitted to Condor.
- `pvm_addhosts()`. When an application adds a host machine, it calls `pvm_addhosts()`. The first argument to `pvm_addhosts()` is a string that specifies the machine class. For example, to specify class 0, a pointer to the string "0" is the first argument. Condor finds a machine that satisfies the requirements of class 0 and adds it to the PVM virtual machine.

  The function `pvm_addhosts()` does not block. It returns immediately, before hosts are added to the virtual machine. In a non-dedicated environment the amount of time it takes until a machine becomes available is not bound. A program should call `pvm_notify()` before calling `pvm_addhosts()`. When a host is added later, the program will be notified in the usual PVM fashion (with a `PvmHostAdd` notification message).

  After receiving a `PvmHostAdd` notification, the PVM master can unpack the following information about the added host: an integer specifying the TID of the new host, a string specifying the name of the new host, followed by a string specifying the machine class of the new host. The PVM master can then call `pvm_spawn()` to start a worker process on the new host, specifying this machine class as the architecture and using the appropriate executable path for this machine class. Note that the name of the host is given by the startd and may be of the form "vmN@hostname" on SMP machines.
- `pvm_notify()`. Under Condor, there are two additional possible notification types to the function `pvm_notify()`. They are `PvmHostSuspend` and `PvmHostResume`. The program calls `pvm_notify()` with a host tid and `PvmHostSuspend` (or `PvmHostResume`) as arguments, and the program will receive a notification for the event of a host being suspended. Note that a notification occurs only once for each request. As an example, a `PvmHostSuspend` notification request for tid 4 results in a single `PvmHostSuspend` message for tid 4. There will not be another `PvmHostSuspend` message for that tid without another notification request.

  The easiest way to handle this is the following: When a worker node starts up, set up a notification for `PvmHostSuspend` on its tid. When that node gets suspended, set up a `PvmHostResume` notification. When it resumes, set up a `PvmHostSuspend` notification.

  If your application uses the `PvmHostSuspend` and `PvmHostResume` notification types, you will need to modify your PVM distribution to support them as follows. First, go to your $(PVM_ROOT). In `include/pvm3.h`, add

```
#define PvmHostSuspend  6   /* condor suspension */
```

```
#define PvmHostResume   7   /* condor resumption */
```

to the list of "pvm_notify kinds". In `src/lpvmgen.c`, in `pvm_notify()`, change

```
} else {
        switch (what) {
        case PvmHostDelete:
        ....
```

to

```
} else {
        switch (what) {
        case PvmHostSuspend:  /* for condor */
        case PvmHostResume:   /* for condor */
        case PvmHostDelete:
        ....
```

And that's it. Re-compile, and you're done.

- `pvm_spawn()`. If the flag in `pvm_spawn()` is `PvmTaskArch`, then a machine class string should be used. If there is only one machine class in a virtual machine, "0" is the string for the desired architecture.

  Under Condor, only one PVM task spawned per node is currently allowed, due to Condor's machine load checks. Most Condor sites will suspend or vacate a job if the load on its machine is higher than a specified threshold. Having more than one PVM task per node pushes the load higher than the threshold.

  Also, Condor only supports starting one copy of the executable with each call to `pvm_spawn()` (i.e., the fifth argument must always be equal to one). To spawn multiple copies of an executable in Condor, you must call `pvm_spawn()` once for each copy.

  A good fault tolerant program will be able to deal with `pvm_spawn()` failing. It happens more often in opportunistic environments like Condor than in dedicated ones.

- `pvm_exit()`. If a PVM task calls `pvm_catchout()` during its run to catch the output of child tasks, `pvm_exit()` will attempt to gather the output of all child tasks before returning. Due to the dynamic nature of the virtual machine in Condor, this cleanup procedure (in the PVM library and daemon) is error-prone and should be avoided. So, any PVM tasks which call `pvm_catchout()` should be sure to call it again with a NULL argument to disable output collection before calling `pvm_exit()`.

### 2.9.3 Sample PVM submit file

PVM jobs are submitted to the PVM universe. The following is an example of a submit description file for a PVM job. This job has a master PVM program called `master.exe`.

```
##############################################################
# sample_submit
# Sample submit file for PVM jobs.
##############################################################

# The job is a PVM universe job.
universe = PVM

# The executable of the master PVM program is ``master.exe''.
executable = master.exe

input = "in.dat"
output = "out.dat"
error = "err.dat"

###################  Machine class 0  ##################

Requirements = (Arch == "INTEL") && (OpSys == "LINUX")

# We want at least 2 machines in class 0 before starting the
# program.  We can use up to 4 machines.
machine_count = 2..4
queue

###################  Machine class 1  ##################

Requirements = (Arch == "SUN4x") && (OpSys == "SOLARIS26")

# We need at least 1 machine in class 1 before starting the
# executable.  We can use up to 3 to start with.
machine_count = 1..3
queue

###################  Machine class 2  ##################

Requirements = (Arch == "INTEL") && (OpSys == "SOLARIS26")

# We don't need any machines in this class at startup, but we can use
# up to 3.
machine_count = 0..3
queue

##############################################################
# note: the program will not be started until the least
#       requirements in all classes are satisfied.
##############################################################
```

In this sample submit file, the command `universe = PVM` specifies that the jobs should be submitted into PVM universe.

The command `executable = master.exe` tells Condor that the PVM master program is *master.exe*. This program will be started on the submitting machine. The workers should be spawned by this master program during execution.

The `input`, `output`, and `error` commands specify files that should be redirected to the standard in, out, and error of the PVM master program. Note that these files will not include output

from worker processes unless the master calls `pvm_catchout()`.

This submit file also tells Condor that the virtual machine consists of three different classes of machine. Class 0 contains machines with INTEL processors running LINUX; class 1 contains machines with SUN4x (SPARC) processors running SOLARIS26; class 2 contains machines with INTEL processors running SOLARIS26.

By using `machine_count = <min>..<max>`, the submit file tells Condor that before the PVM master is started, there should be at least `<min>` number of machines of the current class. It also asks Condor to give it as many as `<max>` machines. During the execution of the program, the application may request more machines of each of the class by calling `pvm_addhosts()` with a string specifying the machine class. It is often useful to specify `<min>` of 0 for each class, so the PVM master will be started immediately when the first host from any machine class is allocated.

The `queue` command should be inserted after the specifications of each class.

## 2.10 Parallel Applications (Including MPI Applications)

Condor's Parallel universe supports a wide variety of parallel programming environments, and it encompasses the execution of MPI jobs. It supports jobs which need to be co-scheduled. A co-scheduled job has more than one process that must be running at the same time on different machines to work correctly. The parallel universe supersedes the mpi universe. The mpi universe eventually will be removed from Condor.

### 2.10.1 Prerequisites to Running Parallel Jobs

Condor must be configured such that resources (machines) running parallel jobs are dedicated. Note that *dedicated* has a very specific meaning in Condor: dedicated machines never vacate their executing Condor jobs, should the machine's interactive owner return. This is implemented by running a single dedicated scheduler process on a machine in the pool, which becomes the single machine from which parallel universe jobs are submitted. Once the dedicated scheduler claims a dedicated machine for use, the dedicated scheduler will try to use that machine to satisfy the requirements of the queue of parallel universe or MPI universe jobs. If the dedicated scheduler cannot use a machine for a configurable amount of time, it will release its claim on the machine, making it available again for the opportunistic scheduler.

Since Condor does not ordinarily run this way, (Condor usually uses opportunistic scheduling), dedicated machines must be specially configured. Section 3.12.8 of the Administrator's Manual describes the necessary configuration and provides detailed examples.

To simplify the scheduling of dedicated resources, a single machine becomes the scheduler of dedicated resources. This leads to a further restriction that jobs submitted to execute under the parallel universe must be submitted from the machine acting as the dedicated scheduler.

### 2.10.2 Parallel Job Submission

Given correct configuration, parallel universe jobs may be submitted from the machine running the dedicated scheduler. The dedicated scheduler claims machines for the parallel universe job, and invokes the job when the correct number of machines of the correct platform (architecture and operating system) are claimed. Note that the job likely consists of more than one process, each to be executed on a separate machine. The first process (machine) invoked is treated different than the others. When this first process exits, Condor shuts down all the others, even if they have not yet completed their execution.

An overly simplified submit description file for a parallel universe job appears as

```
##############################################
##   submit description file for a parallel program
##############################################
universe = parallel
executable = /bin/sleep
arguments = 30
machine_count = 8
queue
```

This job specifies the **universe** as **parallel**, letting Condor know that dedicated resources are required. The **machine_count** command identifies the number of machines required by the job.

When submitted, the dedicated scheduler allocates eight machines with the same architecture and operating system as the submit machine. It waits until all eight machines are available before starting the job. When all the machines are ready, it invokes the */bin/sleep* command, with a command line argument of 30 on all eight machines more or less simultaneously.

A more realistic example of a parallel job utilizes other features.

```
######################################
## Parallel example submit description file
######################################
universe = parallel
executable = /bin/cat
log = logfile
input = infile.$(NODE)
output = outfile.$(NODE)
error = errfile.$(NODE)
machine_count = 4
queue
```

The specification of the **input**, **output**, and **error** files utilize the predefined macro `$(NODE)`. See the *condor_submit* manual page on page 716 for further description of predefined macros. The

`$(NODE)` macro is given a unique value as processes are assigned to machines. The `$(NODE)` value is fixed for the entire length of the job. It can therefore be used to identify individual aspects of the computation. In this example, it is used to utilize and assign unique names to input and output files.

This example presumes a shared file system across all the machines claimed for the parallel universe job. Where no shared file system is either available or guaranteed, use Condor's file transfer mechanism, as described in section 2.5.4 on page 36. This example uses the file transfer mechanism.

```
######################################
## Parallel example submit description file
## without using a shared file system
######################################
universe = parallel
executable = /bin/cat
log = logfile
input = infile.$(NODE)
output = outfile.$(NODE)
error = errfile.$(NODE)
machine_count = 4
should_transfer_files = yes
when_to_transfer_output = on_exit
queue
```

The job requires exactly four machines, and queues four processes. Each of these processes requires a correctly named input file, and produces an output file.

### 2.10.3 Parallel Jobs with Separate Requirements

The different machines executing for a parallel universe job may specify different machine requirements. A common example requires that the head node execute on a specific machine. It may be also useful for debugging purposes.

Consider the following example.

```
######################################
## Example submit description file
## with multiple procs
######################################
universe = parallel
executable = example
machine_count = 1
requirements = ( machine == "machine1")
queue
```

```
requirements = ( machine =!= "machine1")
machine_count = 3
queue
```

The dedicated scheduler allocates four machines. All four executing jobs have the same value for `$(Cluster)` macro. The `$(Process)` macro takes on two values; the value 0 will be assigned for the single executable that must be executed on machine1, and the value 1 will be assigned for the other three that must be executed anywhere but on machine1.

Carefully consider the ordering and nature of multiple sets of requirements in the same submit description file. The scheduler matches jobs to machines based on the ordering within the submit description file. Mutually exclusive requirements eliminate the dependence on ordering within the submit description file. Without mutually exclusive requirements, the scheduler may unable to schedule the job. The ordering within the submit description file may preclude the scheduler considering the specific allocation that could satisfy the requirements.

### 2.10.4 MPI Applications Within Condor's Parallel Universe

MPI applications utilize a single executable that is invoked in order to execute in parallel on one or more machines. Condor's parallel universe provides the environment within which this executable is executed in parallel. However, the various implementations of MPI (for example, LAM or MPICH) require further framework items within a system-wide environment. Condor supports this necessary framework through user visible and modifiable scripts. An MPI implementation-dependent script becomes the Condor job. The script sets up the extra, necessary framework, and then invokes the MPI application's executable.

Condor provides these scripts in the `$(RELEASE_DIR)/etc/examples` directory. The script for the LAM implementation is `lamscript`. The script for the MPICH implementation is `mp1script`. Therefore, a Condor submit description file for these implementations would appear similar to:

```
######################################
## Example submit description file
## for MPICH 1 MPI
## works with MPICH 1.2.4, 1.2.5 and 1.2.6
######################################
universe = parallel
executable = mp1script
arguments = my_mpich_linked_executable arg1 arg2
machine_count = 4
should_transfer_files = yes
when_to_transfer_output = on_exit
transfer_input_files = my_mpich_linked_executable
queue
```

or

```
######################################
## Example submit description file
## for LAM MPI
######################################
universe = parallel
executable = lamscript
arguments = my_lam_linked_executable arg1 arg2
machine_count = 4
should_transfer_files = yes
when_to_transfer_output = on_exit
transfer_input_files = my_lam_linked_executable
queue
```

The **executable** is the MPI implementation-dependent script. The first argument to the script is the MPI application's executable. Further arguments to the script are the MPI application's arguments. Condor must transfer this executable; do this with the **transfer_input_files** command.

For other implementations of MPI, copy and modify one of the given scripts. Most MPI implementations require two system-wide prerequisites. The first prerequisite is the ability to run a command on a remote machine without being prompted for a password. *ssh* is commonly used, but other command may be used. The second prerequisite is an ASCII file containing the list of machines that may utilize *ssh*. These common prerequisites are implemented in a further script called `sshd.sh`. `sshd.sh` generates ssh keys (to enable password-less remote execution), and starts an *sshd* daemon. The machine name and MPI rank are given to the submit machine.

The *sshd.sh* script requires the definition of two Condor configuration variables. Configuration variable `CONDOR_SSHD` is an absolute path to an implementation of *sshd*. *sshd.sh* has been tested with *openssh* version 3.9, but should work with more recent versions. Configuration variable `CONDOR_SSH_KEYGEN` points to the corresponding *ssh-keygen* executable.

Scripts *lamscript* and *mp1script* each have their own idiosyncrasies. In *mp1script*, the `PATH` to the MPICH installation must be set. The shell variable MPDIR indicates its proper value. This directory contains the MPICH *mpirun* executable. For LAM, there is a similar path setting, but it is called `LAMDIR` in the *lamscript* script. In addition, this path must be part of the path set in the user's `.cshrc` script. As of this writing, the LAM implementation does not work if the user's login shell is the Bourne or compatible shell.

### 2.10.5 Outdated Documentation of the MPI Universe

The following sections on implementing MPI applications utilizing the MPI universe are superseded by the sections describing MPI applications utilizing the parallel universe. These sections are included in the manual as reference, until the time when the MPI universe is no longer supported within Condor.

MPI stands for Message Passing Interface. It provides an environment under which parallel programs may synchronize, by providing communication support. Running the MPI-based parallel programs within Condor eases the programmer's effort. Condor dedicates machines for running the programs, and it does so using the same interface used when submitting non-MPI jobs.

The MPI universe in Condor currently supports MPICH versions 1.2.2, 1.2.3, and 1.2.4 using the ch_p4 device. The MPI universe does not support MPICH version 1.2.5. These supported implementations are offered by Argonne National Labs without charge by download. See the web page at http://www-unix.mcs.anl.gov/mpi/mpich/ for details and availability. Programs to be submitted for execution under Condor will have been compiled using *mpicc*. No further compilation or linking is necessary to run jobs under Condor.

The Parallel universe 2.10 is now the preferred way to run MPI jobs. Support for the MPI universe will be removed from Condor at a future date.

**MPI Details of Set Up**

Administratively, Condor must be configured such that resources (machines) running MPI jobs are dedicated. Dedicated machines never vacate their running condor jobs should the machine's interactive owner return. Once the dedicated scheduler claims a dedicated machine for use, it will try to use that machine to satisfy the requirements of the queue of MPI jobs.

Since Condor is not ordinarily used in this manner (Condor uses opportunistic scheduling), machines that are to be used as dedicated resources must be configured as such. Section 3.12.8 of Administrator's Manual describes the necessary configuration and provides detailed examples.

To simplify the dedicated scheduling of resources, a single machine becomes the scheduler of dedicated resources. This leads to a further restriction that jobs submitted to execute under the MPI universe (with dedicated machines) must be submitted from the machine running as the dedicated scheduler.

**MPI Job Submission**

Once the programs are written and compiled, and Condor resources are correctly configured, jobs may be submitted. Each Condor job requires a submit description file. The simplest submit description file for an MPI job:

```
############################################
##   submit description file for mpi_program
############################################
universe = MPI
executable = mpi_program
machine_count = 4
queue
```

This job specifies the `universe` as `mpi`, letting Condor know that dedicated resources will be required. The `machine_count` command identifies the number of machines required by the job. The four machines that run the program will default to be of the same architecture and operating system as the machine on which the job is submitted, since a platform is not specified as a requirement.

The simplest example does not specify an input or output, meaning that the computation completed is useless, since both input comes from and the output goes to `/dev/null`. A more complex example of a submit description file utilizes other features.

```
######################################
## MPI example submit description file
######################################
universe = MPI
executable = simplempi
log = logfile
input = infile.$(NODE)
output = outfile.$(NODE)
error = errfile.$(NODE)
machine_count = 4
queue
```

The specification of the input, output, and error files utilize a predefined macro that is only relevant to mpi universe jobs. See the *condor_submit* manual page on page 716 for further description of predefined macros. The `$(NODE)` macro is given a unique value as programs are assigned to machines. This value is what the MPICH version ch_p4 implementation terms the rank of a program. Note that this term is unrelated and independent of the Condor term rank. The `$(NODE)` value is fixed for the entire length of the job. It can therefore be used to identify individual aspects of the computation. In this example, it is used to give unique names to input and output files.

If your site does NOT have a shared file system across all the nodes where your MPI computation will execute, you can use Condor's file transfer mechanism. You can find out more details about these settings by reading the *condor_submit* man page or section 2.5.4 on page 36. Assuming your job only reads input from STDIN, here is an example submit file for a site without a shared file system:

```
######################################
## MPI example submit description file
## without using a shared file system
######################################
universe = MPI
executable = simplempi
log = logfile
input = infile.$(NODE)
output = outfile.$(NODE)
error = errfile.$(NODE)
```

```
machine_count = 4
should_transfer_files = yes
when_to_transfer_output = on_exit
queue
```

Consider the following C program that uses this example submit description file.

```
/**************
 * simplempi.c
 **************/
#include <stdio.h>
#include "mpi.h"

int main(argc,argv)
    int argc;
    char *argv[];
{
    int myid;
    char line[128];

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    fprintf ( stdout, "Printing to stdout...%d\n", myid );
    fprintf ( stderr, "Printing to stderr...%d\n", myid );
    fgets ( line, 128, stdin );
    fprintf ( stdout, "From stdin: %s", line );

    MPI_Finalize();
    return 0;
}
```

Here is a makefile that works with the example. It would build the MPI executable, using the MPICH version ch_p4 implementation.

```
######################################################################
## This is a very basic Makefile                                    ##
######################################################################

# the location of the MPICH compiler
CC          = /usr/local/bin/mpicc
CLINKER     = $(CC)

CFLAGS    = -g
EXECS     = simplempi
```

```
all: $(EXECS)

simplempi: simplempi.o
        $(CLINKER) -o simplempi simplempi.o -lm

.c.o:
        $(CC) $(CFLAGS) -c $*.c
```

The submission to Condor requires exactly four machines, and queues four programs. Each of these programs requires an input file (correctly named) and produces an output file.

If input file for `$(NODE)` = 0 (called `infile.0`) contains

```
Hello number zero.
```

and the input file for `$(NODE)` = 1 (called `infile.1`) contains

```
Hello number one.
```

then after the job is submitted to Condor, there will be eight files created: `errfile.[0-3]` and `outfile.[0-3]`. `outfile.0` will contain

```
Printing to stdout...0
From stdin: Hello number zero.
```

and `errfile.0` will contain

```
Printing to stderr...0
```

Different nodes for an MPI job can have different machine requirements. For example, often the first node, sometimes called the head node, needs to run on a specific machine. This can be also useful for debugging. Condor accomodates this by supporting multiple `queue` statements in the submit file, much like with the other universes. For example:

```
######################################
## MPI example submit description file
## with multiple procs
######################################
universe = MPI
executable = simplempi
log = logfile
input = infile.$(NODE)
output = outfile.$(NODE)
```

```
error = errfile.$(NODE)
machine_count = 1
should_transfer_files = yes
when_to_transfer_output = on_exit
requirements = ( machine == "machine1")
queue

requirements = ( machine =!= "machine1")
machine_count = 3
queue
```

The dedicated scheduler will allocate four machines (nodes) total in two procs for this job. The first proc has one node, (rank 0 in MPI terms) and will run on the machine named machine1. The other three nodes, in the second proc, will run on other machines. Like in the other condor universes, the second requirements command overwrites the first, but the other commands are inherited from the first proc.

When submitting jobs with multiple requirements, it is best to write the requirements to be mutually exclusive, or to have the most selective requirement first in the submit file. This is because the scheduler tries to match jobs to machine in submit file order. If the requirements are not mutually exclusive, it can happen that the scheduler may unable to schedule the job, even if all needed resources are available.

## 2.11 DAGMan Applications

A directed acyclic graph (DAG) can be used to represent a set of computations where the input, output, or execution of one or more computations is dependent on one or more other computations. The computations are nodes (vertices) in the graph, and the edges (arcs) identify the dependencies. Condor finds machines for the execution of programs, but it does not schedule programs based on dependencies. The Directed Acyclic Graph Manager (DAGMan) is a meta-scheduler for the execution of programs (computations). DAGMan submits the programs to Condor in an order represented by a DAG and processes the results. A DAG input file describes the DAG, and further submit description file(s) are used by DAGMan when submitting programs to run under Condor.

DAGMan is itself executed as a scheduler universe job within Condor. As DAGMan submits programs, it monitors log file(s) to to enforce the ordering required within the DAG. DAGMan is also responsible for scheduling, recovery, and reporting on the set of programs submitted to Condor.

### 2.11.1 DAGMan Terminology

To DAGMan, a node in a DAG may encompass more than a single program submitted to run under Condor. Figure 2.2 illustrates the elements of a node.

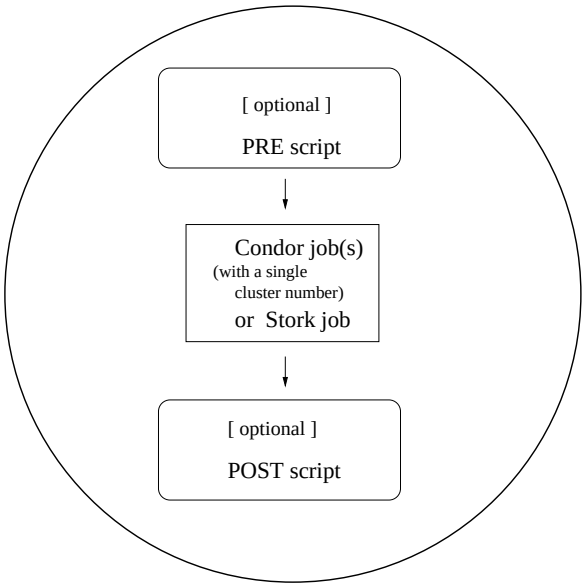



Figure 2.2: One Node within a DAG

Before Condor version 6.7.17, the number of Condor jobs per node was restricted to one. This restriction is now relaxed such that all Condor jobs within a node must share a single cluster number. See the *condor_submit* manual page for a further definition of a cluster. A limitation exists such that all jobs within the single cluster must use the same log file.

As DAGMan schedules and submits jobs within nodes to Condor, these jobs are defined to succeed or fail based on their return values. This success or failure is propagated in well-defined ways to the level of a node within a DAG. Further progression of computation (towards completing the DAG) may be defined based upon the success or failure of one or more nodes.

The failure of a single job within a cluster of multiple jobs (within a single node) causes the entire cluster of jobs to fail. Any other jobs within the failed cluster of jobs are immediately removed. Each node within a DAG is further defined to succeed or fail, based upon the return values of a PRE script, the job(s) within the cluster, and/or a POST script.

### 2.11.2 Input File Describing the DAG

The input file used by DAGMan is called a DAG input file. It may specify seven types of items:

1. A list of the nodes in the DAG which cause the submission of one or more Condor jobs. Each entry serves to name a node and specify a Condor submit description file.
2. A list of the nodes in the DAG which cause the submission of a data placement job. Each entry serves to name a node and specify the Stork submit description file.

3. Any processing required to take place before submission of a node's Condor or Stork job, or after a node's Condor or Stork job has completed execution.
4. A description of the dependencies within the DAG.
5. The number of times to retry a node's execution, if a node within the DAG fails.
6. Any definition of macros associated with a node.
7. A node's exit value that causes the entire DAG to abort.

Comments may be placed in the DAG input file. The pound character (#) as the first character on a line identifies the line as a comment. Comments do not span lines.

A simple diamond-shaped DAG, as shown in Figure 2.3 is presented as a starting point for examples. This DAG contains 4 nodes.

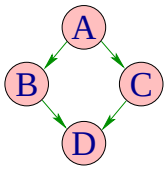



Figure 2.3: Diamond DAG

A very simple DAG input file for this diamond-shaped DAG is

```
# Filename: diamond.dag
#
JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
JOB  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
```

Each DAG input file key word is described below.

**JOB**

The first set of lines in the simple DAG input file list each of the jobs that appear in the DAG. Each job to be managed by Condor is described by a single line that begins with the key word *JOB*. The syntax used for each *JOB* entry is

**JOB** *JobName SubmitDescriptionFileName* [**DIR** *directory*] [**DONE**]

A *JOB* entry maps a *JobName* to a Condor submit description file. The *JobName* uniquely identifies nodes within the DAGMan input file and in output messages. Note that the name for each node within the DAG must be unique.

The key words *JOB* and *DONE* are not case sensitive. Therefore, *DONE*, *Done*, and *done* are all equivalent. The values defined for *JobName* and *SubmitDescriptionFileName* are case sensitive, as file names in the Unix file system are case sensitive. The *JobName* can be any string that contains no white space.

The *DIR* option specifies a working directory for this node, from which the Condor job will be submitted, and from which a *PRE* and/or *POST* script will be run. Note that a DAG containing *DIR* specifications cannot be run in conjunction with the *-usedagdir* command-line argument to *condor_submit_dag*. A rescue DAG generated by a DAG run with the *-usedagdir* argument will contain DIR specifications, so the rescue DAG must be run *without* the *-usedagdir* argument.

The optional *DONE* identifies a job as being already completed. This is useful in situations where the user wishes to verify results, but does not need all programs within the dependency graph to be executed. The *DONE* feature is also utilized when an error occurs causing the DAG to be aborted without completion. DAGMan generates a Rescue DAG, a DAG input file that can be used to restart and complete a DAG without re-executing completed nodes.

### DATA

The *DATA* key word specifies a job to be managed by the Stork data placement server. The syntax used for each *DATA* entry is

**DATA** *JobName SubmitDescriptionFileName* [**DIR** *directory*] [**DONE**]

A *DATA* entry maps a *JobName* to a Stork submit description file. In all other respects, the *DATA* key word is identical to the *JOB* key word.

Here is an example of a simple DAG that stages in data using Stork, processes the data using Condor, and stages the processed data out using Stork. Depending upon the implementation, multiple data jobs to stage in data or to stage out data may be run in parallel.

```
DATA    STAGE_IN1  stage_in1.stork
DATA    STAGE_IN2  stage_in2.stork
JOB     PROCESS    process.condor
DATA    STAGE_OUT1 stage_out1.stork
DATA    STAGE_OUT2 stage_out2.stork
PARENT  STAGE_IN1 STAGE_IN2 CHILD PROCESS
PARENT  PROCESS CHILD STAGE_OUT1 STAGE_OUT2
```

### SCRIPT

The third type of item in a DAG input file identifies processing that is done either before a job within the DAG is submitted to Condor or Stork for execution or after a job within the DAG completes its execution. Processing done before a job is submitted to Condor or Stork is called a *PRE* script. Processing done after a job completes its execution under Condor or Stork is called a *POST* script. A node in the DAG is comprised of the job together with *PRE* and/or *POST* scripts.

*PRE* and *POST* script lines within the DAG input file use the syntax:

**SCRIPT PRE** *JobName ExecutableName* [*arguments*]

**SCRIPT POST** *JobName ExecutableName* [*arguments*]

The *SCRIPT* key word identifies the type of line within the DAG input file. The *PRE* or *POST* key word specifies the relative timing of when the script is to be run. The *JobName* specifies the node to which the script is attached. The *ExecutableName* specifies the script to be executed, and it may be followed by any command line arguments to that script. The *ExecutableName* and optional *arguments* are case sensitive; they have their case preserved.

Scripts are optional for each job, and any scripts are executed on the machine from which the DAG is submitted; this is not necessarily the same machine upon which the node's Condor or Stork job is run. Further, a single cluster of Condor jobs may be spread across several machines.

A PRE script is commonly used to place files in a staging area for the cluster of jobs to use. A POST script is commonly used to clean up or remove files once the cluster of jobs is finished running. An example uses PRE and POST scripts to stage files that are stored on tape. The PRE script reads compressed input files from the tape drive, and it uncompresses them, placing the input files in the current directory. The cluster of Condor jobs reads these input files. and produces output files. The POST script compresses the output files, writes them out to the tape, and then removes both the staged input files and the output files.

DAGMan takes note of the exit value of the scripts as well as the job. A script with an exit value not equal to 0 fails. If the PRE script fails, then neither the job nor the POST script runs, and the node fails.

If the PRE script succeeds, the Condor or Stork job is submitted. If the job fails and there is no POST script, the DAG node is marked as failed. An exit value not equal to 0 indicates program failure. It is therefore important that a successful program return the exit value 0.

If the job fails and there is a POST script, node failure is determined by the exit value of the POST script. A failing value from the POST script marks the node as failed. A succeeding value from the POST script (even with a failed job) marks the node as successful. Therefore, the POST script may need to consider the return value from the job.

By default, the POST script is run regardless of the job's return value.

A node not marked as failed at any point is successful. Table 2.1 summarizes the success or failure of an entire node for all possibilities. An *S* stands for success, an *F* stands for failure, and the dash character (-) identifies that there is no script.

Two variables may be used within the DAG input file, and may ease script writing. The variables are often utilized in the arguments passed to a PRE or POST script. The variable `$JOB` evaluates to the (case sensitive) string defined for *JobName*. For use as an argument to POST scripts, the `$RETURN` variable evaluates to the return value of the Condor or Stork job. A job that dies due to

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PRE | - | - | F | S | S | - | - | - | - | S | S | S | S |
| JOB | S | F | not run | S | F | S | S | F | F | S | F | F | S |
| POST | - | - | not run | - | - | S | F | S | F | S | S | F | F |
| node | S | F | F | S | F | S | F | S | F | S | S | F | F |

Table 2.1: Node success or failure definition

a signal is reported with a `$RETURN` value representing the negative signal number. For example, SIGKILL (signal 9) is reported as -9. A job whose batch system submission fails is reported as -1001. A job that is externally removed from the batch system queue (by something other than *condor_dagman*) is reported as -1002.

As an example, consider the diamond-shaped DAG example. Suppose the PRE script expands a compressed file needed as input to nodes B and C. The file is named of the form *JobName*`.gz`. The DAG input file becomes

```
# Filename: diamond.dag
#
JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
JOB  D  D.condor
SCRIPT PRE  B  pre.csh $JOB .gz
SCRIPT PRE  C  pre.csh $JOB .gz
PARENT A CHILD B C
PARENT B C CHILD D
```

The script `pre.csh` uses the arguments to form the file name of the compressed file:

```
#!/bin/csh
gunzip $argv[1]$argv[2]
```

**PARENT..CHILD**

The fourth type of item in the DAG input file describes the dependencies within the DAG. Nodes are parents and/or children within the DAG. A parent node must be completed successfully before any of its children may be started. A child node may only be started once all its parents have successfully completed.

The syntax of a dependency line within the DAG input file:

**PARENT** *ParentJobName*. . . **CHILD** *ChildJobName*. . .

The *PARENT* key word is followed by one or more *ParentJobName*s. The *CHILD* key word is followed by one or more *ChildJobName*s. Each child job depends on every parent job within the line. A single line in the input file can specify the dependencies from one or more parents to one or more children. As an example, the line

```
PARENT p1 p2 CHILD c1 c2
```

produces four dependencies:

1. `p1` to `c1`
2. `p1` to `c2`
3. `p2` to `c1`
4. `p2` to `c2`

**RETRY**

The fifth type of item in the DAG input file provides a way to retry failed nodes. Use of retry is optional. The syntax for retry is

**RETRY** *JobName NumberOfRetries* [**UNLESS-EXIT** *value*]

where *JobName* identifies the node. *NumberOfRetries* is an integer number of times to retry the node after failure. The implied number of retries for any node is 0, the same as not having a retry line in the file. Retry is implemented on nodes, not parts of a node.

The diamond-shaped DAG example may be modified to retry node C:

```
# Filename: diamond.dag
#
JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
JOB  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
Retry  C 3
```

If node C is marked as failed (for any reason), then it is started over as a first retry. The node will be tried a second and third time, if it continues to fail. If the node is marked as successful, then further retries do not occur.

Retry of a node may be short circuited using the optional key word *UNLESS-EXIT* (followed by an integer exit value). If the node exits with the specified integer exit value, then no further processing will be done on the node.

**VARS**

The sixth type of item in the DAG input file provides a method for defining a macro. This macro may then be used in a submit description file. These macros are defined on a per-node basis, using the following format.

**VARS** *JobName* **macroname=** *"string"* [**macroname=** *"string"*...]

The macro may be used within the submit description file. A *macroname* consists of alphanumeric characters (a..Z and 0..9), as well as the underscore character. The space character delimits macros, when there is more than one macro defined for a node.

Correct syntax requires that the *string* must be enclosed in double quotes. To use a double quote inside *string*, escape it with the backslash character (\). To add the backslash character itself, use two backslashes (\\).

Note that macro names cannot begin with the string "queue" (in any combination of upper and lower case).

If the DAG input file contains

```
# Filename: diamond.dag
#
JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
JOB  D  D.condor
VARS A state="Wisconsin"
```

then file `A.condor` may use the macro *state*. This example submit description file for the Condor job in node A passes the value of the macro as a command-line argument to the job.

```
# file name: A.condor
executable = A.exe
log        = A.log
error      = A.err
arguments  = $(state)
queue
```

This Condor job's command line will be

```
A.exe Wisconsin
```

The use of macros may allow a reduction in the necessary number of unique submit description files.

### ABORT-DAG-ON

The seventh type of item in the DAG input file provides a way to abort the entire DAG if a given node returns a specific exit code. The syntax for ABORT-DAG-ON:

**ABORT-DAG-ON** *JobName AbortExitValue* [**RETURN** *DAGReturnValue*]

If the node specified by *JobName* returns the specified *AbortExitValue*, the DAG is immediately aborted. A DAG abort differs from a node failure, in that a DAG abort causes all nodes within

the DAG to be stopped immediately. This includes removing the jobs in nodes that are currently running. A node failure allows the DAG to continue running, until no more progress can be made due to dependencies.

An abort overrides node retries. If a node returns the abort exit value, the DAG is aborted, even if the node has retry specified.

When a DAG aborts, by default it exits with the node return value that caused the abort. This can be changed by using the optional *RETURN* key word along with specifying the desired *DAGReturnValue.* The DAG abort return value can be used for DAGs within DAGs, allowing an inner DAG to cause an abort of an outer DAG.

Adding *ABORT-DAG-ON* for node C in the diamond-shaped DAG

```
# Filename: diamond.dag
#
JOB  A  A.condor
JOB  B  B.condor
JOB  C  C.condor
JOB  D  D.condor
PARENT A CHILD B C
PARENT B C CHILD D
Retry  C 3
ABORT-DAG-ON C 10 RETURN 1
```

causes the DAG to be aborted, if node C exits with a return value of 10. Any other currently running nodes (only node B is a possibility for this particular example) are stopped and removed. If this abort occurs, the return value for the DAG is 1.

### 2.11.3 Submit Description File

Each node in a DAG may use a unique submit description file. One key limitation is that each Condor submit description file must submit jobs described by a single cluster number. At the present time DAGMan cannot deal with a submit file producing multiple job clusters.

At one time, DAGMan required that all jobs within all nodes specify the same, single log file. This is no longer the case. However, if the DAG utilizes a large number of separate log files, performance may suffer. Therefore, it is better to have fewer, or even only a single log file. Unfortunately, each Stork job currently requires a separate log file. DAGMan enforces the dependencies within a DAG using the events recorded in the log file(s) produced by job submission to Condor.

Here is a modified version of the DAG input file for the diamond-shaped DAG. The modification has each node use the same submit description file.

```
# Filename: diamond.dag
#
JOB  A  diamond_job.condor
JOB  B  diamond_job.condor
```

```
JOB  C  diamond_job.condor
JOB  D  diamond_job.condor
PARENT A CHILD B C
PARENT B C CHILD D
```

Here is the single Condor submit description file for this DAG:

```
# Filename: diamond_job.condor
#
executable   = /path/diamond.exe
output       = diamond.out.$(cluster)
error        = diamond.err.$(cluster)
log          = diamond_condor.log
universe     = vanilla
notification = NEVER
queue
```

This example uses the same Condor submit description file for all the jobs in the DAG. This implies that each node within the DAG runs the same job. The `$(cluster)` macro produces unique file names for each job's output. As the Condor job within each node causes a separate job submission, each has a unique cluster number.

Notification is set to `NEVER` in this example. This tells Condor not to send e-mail about the completion of a job submitted to Condor. For DAGs with many nodes, this reduces or eliminates excessive numbers of e-mails.

A separate example shows an intended use of a *VARS* entry in the DAG input file. This use may dramatically reduce the number of Condor submit description files needed for a DAG. In the case where the submit description file for each node varies only in file naming, the use of a substitution macro within the submit description file reduces the need to a single submit description file. Note that the user log file for a job currently cannot be specified using a macro passed from the DAG.

The example uses a single submit description file in the DAG input file, and uses the *Vars* entry to name output files.

The relevant portion of the DAG input file appears as

```
JOB A theonefile.sub
JOB B theonefile.sub
JOB C theonefile.sub

VARS A outfilename="A"
VARS B outfilename="B"
VARS C outfilename="C"
```

The submit description file appears as

```
# submit description file called:  theonefile.sub
executable   = progX
universe     = standard
output       = $(outfilename)
error        = error.$(outfilename)
log          = progX.log
queue
```

For a DAG like this one with thousands of nodes, being able to write and maintain a single submit description file and a single, yet more complex, DAG input file is preferable.

### 2.11.4 Job Submission

A DAG is submitted using the program *condor_submit_dag*. See the manual page 742 for complete details. A simple submission has the syntax

*condor_submit_dag DAGInputFileName*

The diamond-shaped DAG example may be submitted with

```
condor_submit_dag diamond.dag
```

In order to guarantee recoverability, the DAGMan program itself is run as a Condor job. As such, it needs a submit description file. *condor_submit_dag* produces this needed submit description file, naming it by appending `.condor.sub` to the *DAGInputFileName*. This submit description file may be edited if the DAG is submitted with

```
condor_submit_dag -no_submit diamond.dag
```

causing *condor_submit_dag* to generate the submit description file, but not submit DAGMan to Condor. To submit the DAG, once the submit description file is edited, use

```
condor_submit diamond.dag.condor.sub
```

An optional argument to *condor_submit_dag*, *-maxjobs*, is used to specify the maximum number of batch jobs that DAGMan may submit at one time. It is commonly used when there is a limited amount of input file staging capacity. As a specific example, consider a case where each job will require 4 Mbytes of input files, and the jobs will run in a directory with a volume of 100 Mbytes of free space. Using the argument *-maxjobs 25* guarantees that a maximum of 25 jobs, using a maximum of 100 Mbytes of space, will be submitted to Condor and/or Stork at one time.

While the *-maxjobs* argument is used to limit the number of batch system jobs submitted at one time, it may be desirable to limit the number of scripts running at one time. The optional *-maxpre* argument limits the number of PRE scripts that may be running at one time, while the optional *-maxpost* argument limits the number of POST scripts that may be running at one time.

An optional argument to *condor_submit_dag*, *-maxidle*, is used to limit the number of idle jobs within a given DAG. When the number of idle node jobs in the DAG reaches the specified value, *condor_dagman* will stop submitting jobs, even if there are ready nodes in the DAG. Once some of the idle jobs start to run, *condor_dagman* will resume submitting jobs. Note that this parameter only limits the number of idle jobs submitted by a given instance of *condor_dagman*. Idle jobs submitted by other sources (including other *condor_dagman* runs) are ignored.

DAGs that submit jobs to Stork using the *DATA* key word must also specify the Stork user log file, using the *-storklog* argument.

### 2.11.5 Job Monitoring, Job Failure, and Job Removal

After submission, the progress of the DAG can be monitored by looking at the log file(s), observing the e-mail that job submission to Condor causes, or by using *condor_q -dag*. There is a large amount of information in an extra file. The name of this extra file is produced by appending `.dagman.out` to *DAGInputFileName*; for example, if the DAG file is `diamond.dag`, this extra file is `diamond.dag.dagman.out`. If this extra file grows too large, limit its size with the `MAX_DAGMAN_LOG` configuration macro (see section 3.3.4).

If you have some kind of problem in your DAGMan run, please save the corresponding `dagman.out` file; it is the most important debugging tool for DAGMan. As of version 6.8.2, the `dagman.out` is appended to, rather than overwritten, with each new DAGMan run.

*condor_submit_dag* attempts to check the DAG input file. If a problem is detected, *condor_submit_dag* prints out an error message and aborts.

To remove an entire DAG, consisting of DAGMan plus any jobs submitted to Condor or Stork, remove the DAGMan job running under Condor. *condor_q* will list the job number. Use the job number to remove the job, for example

```
% condor_q
-- Submitter: turunmaa.cs.wisc.edu : <128.105.175.125:36165> : turunmaa.cs.wisc.edu
 ID      OWNER          SUBMITTED     RUN_TIME ST PRI SIZE CMD
  9.0   smoler         10/12 11:47   0+00:01:32 R  0   8.7  condor_dagman -f -
 11.0   smoler         10/12 11:48   0+00:00:00 I  0   3.6  B.out
 12.0   smoler         10/12 11:48   0+00:00:00 I  0   3.6  C.out

    3 jobs; 2 idle, 1 running, 0 held

% condor_rm 9.0
```

Before the DAGMan job stops running, it uses *condor_rm* and/or *stork_rm* to remove any jobs within the DAG that are running.

In the case where a machine is scheduled to go down, DAGMan will clean up memory and exit. However, it will leave any submitted jobs in Condor's queue.

### 2.11.6 Job Recovery: The Rescue DAG

DAGMan can help with the resubmission of uncompleted portions of a DAG, when one or more nodes results in failure. If any node in the DAG fails, the remainder of the DAG is continued until no more forward progress can be made based on the DAG's dependencies. At this point, DAGMan produces a file called a Rescue DAG.

The Rescue DAG is a DAG input file, functionally the same as the original DAG file. It additionally contains an indication of successfully completed nodes by appending the *DONE* key word to the node's *JOB* or *DATA* lines. If the DAG is resubmitted using this Rescue DAG input file, the nodes marked as completed will not be re-executed.

The Rescue DAG is automatically generated by DAGMan when a node within the DAG fails. The file name assigned is *DAGInputFileName*, appended with the suffix `.rescue`. Statistics about the failed DAG execution are presented as comments at the beginning of the Rescue DAG input file.

If the Rescue DAG file is generated before all retries of a node are completed, then the Rescue DAG file will also contain Retry entries. The number of retries will be set to the appropriate remaining number of retries.

The granularity defining success or failure in the Rescue DAG input file is given for nodes. The Condor job within a node may result in the submission of multiple Condor jobs under a single cluster. If one of the multiple jobs fails, the node fails. Therefore, a resubmission of the Rescue DAG will again result in the submission of the entire cluster of jobs.

### 2.11.7 Visualizing DAGs with *dot*

It can be helpful to see a picture of a DAG. DAGMan can assist you in visualizing a DAG by creating the input files used by the AT&T Research Labs *graphviz* package. *dot* is a program within this package, available from http://www.graphviz.org/, and it is used to draw pictures of DAGs.

DAGMan produces one or more dot files as the result of an extra line in a DAGMan input file. The line appears as

```
DOT dag.dot
```

This creates a file called `dag.dot`. which contains a specification of the DAG before any jobs within the DAG are submitted to Condor. The `dag.dot` file is used to create a visualization of the DAG by using this file as input to *dot*. This example creates a Postscript file, with a visualization of the DAG:

```
dot -Tps dag.dot -o dag.ps
```

Within the DAGMan input file, the DOT command can take several optional parameters:

- **UPDATE** This will update the dot file every time a significant update happens.
- **DONT-UPDATE** Creates a single dot file, when the DAGMan begins executing. This is the default if the parameter **UPDATE** is not used.
- **OVERWRITE** Overwrites the dot file each time it is created. This is the default, unless **DONT-OVERWRITE** is specified.
- **DONT-OVERWRITE** Used to create multiple dot files, instead of overwriting the single one specified. To create file names, DAGMan uses the name of the file concatenated a period and an integer. For example, the DAGMan input file line

  ```
  DOT dag.dot DONT-OVERWRITE
  ```

  causes files `dag.dot.0`, `dag.dot.1`, `dag.dot.2`, etc. to be created. This option is most useful combined with the **UPDATE** option to visualize the history of the DAG after it has finished executing.
- **INCLUDE** *path-to-filename* Includes the contents of a file given by `path-to-filename` in the file produced by the **DOT** command. The include file contents are always placed after the line of the form `label=`. This may be useful if further editing of the created files would be necessary, perhaps because you are automatically visualizing the DAG as it progresses.

If conflicting parameters are used in a DOT command, the last one listed is used.

### 2.11.8 Advanced Usage: A DAG within a DAG

The organization and dependencies of the jobs within a DAG are the keys to its utility. There are cases when a DAG is easier to visualize and construct hierarchically, as when a node within a DAG is also a DAG. Condor DAGMan handles this situation with grace. Since more than one DAG is being discussed, terminology is introduced to clarify which DAG is which. Reuse the example diamond-shaped DAG as given in Figure 2.3. Assume that node B of this diamond-shaped DAG will itself be a DAG. The DAG of node B is called the inner DAG, and the diamond-shaped DAG is called the outer DAG.

To make DAGs within DAGs, the essential element is getting the name of the submit description file for the inner DAG correct within the outer DAG's input file.

Work on the inner DAG first. The goal is to generate a Condor submit description file for this inner DAG. Here is a very simple linear DAG input file used as an example of the inner DAG.

```
# Filename: inner.dag
#
JOB  X  X.submit
JOB  Y  Y.submit
JOB  Z  Z.submit
PARENT X CHILD Y
PARENT Y CHILD Z
```

Use *condor_submit_dag* to create a submit description file for this inner dag:

```
condor_submit_dag -no_submit inner.dag
```

The resulting file will be named `inner.dag.condor.sub`. This file will be needed in the DAG input file of the outer DAG. The naming of the file is the name of the DAG input file (`inner.dag`) with the suffix `.condor.sub`.

A simple example of a DAG input file for the outer DAG is

```
# Filename: diamond.dag
#
    JOB  A  A.submit
    JOB  B  inner.dag.condor.sub
    JOB  C  C.submit
    JOB  D  D.submit
    PARENT A CHILD B C
    PARENT B C CHILD D
```

The outer DAG is then submitted as before, with

```
condor_submit_dag diamond.dag
```

More than one level of nested DAGs is supported.

One item to get right: to locate the log files used in ordering the DAG, DAGMan either needs a completely flat directory structure (*all* files for outer and inner DAGs within the same directory), or it needs full path names to all log files.

### 2.11.9 Single Submission of Multiple, Independent DAGs

A single use of *condor_submit_dag* may execute multiple, independent DAGs. Each independent DAG has its own DAG input file. These DAG input files are command-line arguments to *condor_submit_dag* (see the *condor_submit_dag* manual page at 9).

Internally, all of the independent DAGs are combined into a single, larger DAG, with no dependencies between the original independent DAGs. As a result, any generated rescue DAG file represents all of the input DAGs as a single DAG. The file name of this rescue DAG is based on the DAG input file listed first within the command-line arguments to *condor_submit_dag* (unlike a single-DAG rescue DAG file, however, the file name will be <`whatever`>`.dag_multi.rescue`, as opposed to just <`whatever`>`.dag.rescue`). Other files such as `dagman.out` and the lock file also have names based on this first DAG input file.

The success or failure of the independent DAGs is well defined. When multiple, independent DAGs are submitted with a single command, the success of the composite DAG is defined as the

logical AND of the success of each independent DAG. This implies that failure is defined as the logical OR of the failure of any of the independent DAGs.

By default, DAGMan internally renames the nodes to avoid node name collisions. If all node names are unique, the renaming of nodes may be disabled by setting the configuration variable `DAGMAN_MUNGE_NODE_NAMES` to `False` (see 3.3.23).

### 2.11.10 File Paths in DAGs

By default, *condor_dagman* assumes that all relative paths in a DAG input file and the associated Condor submit description files are relative to the current working directory when *condor_submit_dag* is run. Note that relative paths in submit description files can be modified by the submit command **initialdir**; see the *condor_submit* manual page at 9 for more details. The rest of this discussion ignores **initialdir**.

In most cases, path names relative to the current working directory is the desired behavior. However, if running multiple DAGs with a single *condor_dagman*, and each DAG is in its own directory, this will cause problems. In this case, use the *-usedagdir* command-line argument to *condor_submit_dag* (see the *condor_submit_dag* manual page at 9 for more details). This tells *condor_dagman* to run each DAG as if *condor_submit_dag* had been run in the directory in which the relevant DAG file exists.

For example, assume that a directory called `parent` contains two subdirectories called `dag1` and `dag2`, and that `dag1` contains the DAG input file `one.dag` and `dag2` contains the DAG input file `two.dag`. Further, assume that each DAG is set up to be run from its own directory with the following command:

```
cd dag1; condor_submit_dag one.dag
```

This will correctly run `one.dag`.

The goal is to run the two, independent DAGs located within `dag1` and `dag2` while the current working directory is `parent`. To do so, run the following command:

```
condor_submit_dag -usedagdir dag1/one.dag dag2/two.dag
```

Of course, if all paths in the DAG input file(s) and the relevant submit description files are absolute, the *-usedagdir* argument is not needed; however, using absolute paths is NOT generally a good idea.

If you *do not* use *-usedagdir*, relative paths can still work for multiple DAGs, if all file paths are given relative to the current working directory as *condor_submit_dag* is executed. However, this means that, if the DAGs are in separate directories, they cannot be submitted from their own directories, only from the parent directory the paths are set up for.

Note that if you use the *-usedagdir* argument, and your run results in a rescue DAG, the rescue

DAG file will be written to the current working directory, and should be run from that directory. The rescue DAG includes all the path information necessary to run each node job in the proper directory.

### 2.11.11 Configuration

Configuration variables relating to DAGMan may be found in section 3.3.23.

## 2.12 Stork Applications

Today's scientific applications have huge data requirements, which continue to increase drastically every year. These data are generally accessed by many users from all across the the globe. This requires moving huge amounts of data around wide area networks to complete the computation cycle, which brings with it the problem of efficient and reliable data placement.

Stork is a scheduler for data placement. With Stork, *data placement jobs* have been elevated to the same level as Condor's computational jobs; data placements are queued, managed, queried and autonomously restarted upon error. Stork understands the semantics and protocols of data placement.

The underlying data placement jobs are performed by Stork *modules*, typically installed in the Condor `libexec` directory. The module name is encoded from the data placement type and functions. For example, the `stork.transfer.file-file` module transfers data from the `file:/` (local filesystem) to the `file:/` protocol. The `stork.transfer.file-file` module is the only module bundled with Condor/Stork. Additionally, contributed modules may be downloaded for these data transfer protocols:

| | |
|---|---|
| ftp:// | FTP File Transfer Protocol |
| http:// | HTTP Hypertext Transfer Protocol |
| gsiftp:// | Globus Grid FTP |
| nest:// | Condor NeST network storage appliance (see http://www.cs.wisc.edu/condor/nest/) |
| srb:// | SDSC Storage Resource Broker (SRB) (see http://www.sdsc.edu/srb/) |
| srm:// | Storage Resource Manager (SRM) (see http://sdm.lbl.gov/srm-wg/) |
| csrm:// | Castor Storage Resource Manager (Castor SRM) (see http://castor.web.cern.ch/castor/) |
| unitree:// | NCSA UniTree (see http://www.ncsa.uiuc.edu/Divisions/CC/HPDM/unitree/) |

The Stork module API is simple and extensible, enabling users to create and use their own modules.

Stork includes high level features for managing data transfers. By configuration, the number of active jobs running from a Stork server may be limited. Stork includes built in fault tolerance, with capabilities for retrying failed jobs, together with the specification of alternate protocols. Stork users also have access to a higher level job manager, Condor DAGMan (section 2.11), which can manage both Stork data placement jobs and traditional Condor jobs at the same time.

### 2.12.1 Submitting Stork Jobs

As with Condor jobs, Stork jobs are specified with a submit description file. It is important to note the syntax of the submit description file for a Stork job is different than that used by Condor jobs. Specifically, Stork submit description files are written in the ClassAd language. See the ClassAd Language Reference Manual for complete details. Please note that while most of Condor uses ClassAds, Stork utilizes the most recent version of this language, which has evolved over time. Stork defines keywords. When present in the job submit file, keywords define the function of the job.

Here is sample Stork job submit description file, showing file syntax and keywords. A job specifies a 1-to-1 mapping of a data source URL to destination URL.

```
// This is a comment line.
[
    dap_type = transfer;
    src_url = "file:/etc/termcap";
    dest_url = "file:/tmp/stork/file-termcap";
]
```

This example shows the ClassAd pairs that form the heart of a Stork job specification. The minimum keywords required to specify a Stork job are:

**dap_type** Currently, the data type is constrained to **transfer**.

**src_url** Specify the data protocol and URL of the source.

**dest_url** Specify the data protocol and URL of the destination.

Additionally, the following keywords may be used in a Stork submit description file:

**x509proxy** Specifies the location of the X.509 proxy file for protocols that use GSI authentication, such as **gsiftp://**. The special value of *"default"* (quotes are required) invokes GSI libraries to search for the user credential in the standard locations.

**alt_protocols** A comma separated list of alternative protocol pairs (for source and destination protocols), used in a round robin fashion when transfers fail. See section 2.12.3 for a further discussion and examples.

Stork places no restriction on the submit file name or extension, and will accept any valid file name for a Stork submit description file.

Submit data placement jobs to Stork using the *stork_submit* tool. For example, after creating the submit description file `sample.stork` with an editor, submit the data transfer job with the command:

```
stork_submit sample.stork
```

Stork then returns the associated job id, which is used by other Stork job control tools.

Only the first ClassAd (a record expression within brackets) within a Stork submit description file becomes a data placement job upon submission. Other ClassAds within the file are ignored.

### 2.12.2 Managing Stork Jobs

Stork provides a set of command-line user tools for job management, including submitting, querying, and removing data placement jobs.

#### Querying Stork Jobs

Use *stork_status* to check the status of any active or completed Stork job. *stork_status* takes a single argument: the job id. For example, to check the status of the Stork job with job id 3:

```
stork_status 3
```

Use *stork_q* to query all active Stork jobs. *stork_q* does not report on completed Stork jobs.

For example, to check the status all active Stork jobs:

```
stork_q
```

#### Removing Stork Jobs

Active jobs may be removed from the job queue with the *stork_rm* tool. *stork_rm* takes a single argument: the job id of the job to remove. All jobs may be removed, provided they have not completed.

For example, to remove the queued job with job id 4:

```
stork_rm 4
```

### 2.12.3 Fault Tolerance

In an ideal world, all data transfers succeed on the first attempt. However, data transfers do fail for various reasons. Stork is designed with data transfer fault tolerance. Based on configuration, Stork retries failed data transfer jobs using specified protocols.

If a transfer fails, Stork attempts the transfer again, until the number of attempts reaches the limit, as defined by the configuration variable STORK_MAX_RETRY (section 3.3.29).

For each attempt at transfer, the transfer protocols to be used at both source and destination are defined. These transfer protocols may vary, when defined by an **alt_protocols** entry in the submit description file. The location of the data at the source and destination is unchanged by the **alt_protocols** entry. **alt_protocols** defines an ordered list of alternative translation protocols to be used. Each entry in the list is a pair. The first of the pair defines the protocol to be used at the source of the transfer. The second of the pair defines the protocol to be used at the destination of the transfer.

The syntax is a comma-separated list of pairs. A dash character separated the pairs. The protocol name is given in all lower case letters, without colons or slash characters. Stork uses these strings to identify the protocol translation and transfer module to be used.

The initial translation protocol (specified in the **src_url** and **dest_url** entries) together with the list defined by an **alt_protocols** entry form the ordered list of protocols to be utilized in a round robin fashion.

For example, if STORK_MAX_RETRY has the value 4, and the Stork job submit description file contains

```
[
    dap_type = transfer;
    src_url = "gsiftp://serverA/dirA/fileA";
    dest_url = "http://serverB/dirB/fileB";
]
```

then Stork will attempt up to 4 transfers, with each using the same translation protocol. **gsiftp://** is used at the source, and **http://** is used at the destination. The Stork job fails if it has not been completed after 4 attempts.

A second example shows the transfer protocols used for each attempted transfer, when **alt_protocols** is used. For this example, assume that STORK_MAX_RETRY has the value 7.

```
[
    dap_type = transfer;
    src_url = "gsiftp://no-such-server/dir/file";
    dest_url = "file:/dir/file";
    alt_protocols = "ftp-file, http-file";
]
```

Stork attempts the following transfers, in the given order, stopping when the transfer succeeds.

1. from `gsiftp://no-such-server/dir/file` to `file:/dir/file`
2. from `ftp://no-such-server/dir/file` to `file:/dir/file`
3. from `http://no-such-server/dir/file` to `file:/dir/file`
4. from `gsiftp://no-such-server/dir/file` to `file:/dir/file`

5. from `ftp://no-such-server/dir/file` to `file:/dir/file`
6. from `http://no-such-server/dir/file` to `file:/dir/file`
7. from `gsiftp://no-such-server/dir/file` to `file:/dir/file`

### 2.12.4 Running Stork Jobs Under DAGMan

Condor DAGMan (section 2.11) provides high level management of both traditional CPU jobs and Stork data placement jobs. Using DAGMan, users can specify data placement using the *DATA* keyword. DAGMan can mix Stork data transfer jobs and Condor jobs. This capability lends itself well to grid computing, as data is often staged in (transferred) before processing the data. After processing, output is often staged out (transferred).

Here is a sample DAGMan input file that stages in input files using Stork transfers, processes the data as a Condor job, and stages out the result using a Stork transfer.

```
# Transfer input files using Stork
DATA INPUT1 transfer_input_data1.stork
DATA INPUT1 transfer_input_data2.stork

DATA INPUT2 transfer_data
#
# Process the data using Condor
JOB PROCESS process.condor
#
# Transfer output file using Stork
DATA RESULT transfer_result_data.stork
#
# Specify job dependencies
PARENT INPUT1 INPUT2 CHILD PROCESS
PARENT PROCESS CHILD RESULT
```

## 2.13 Job Monitor

The Condor Job Monitor is a Java application designed to allow users to view user log files.

To view a user log file, select it using the open file command in the File menu. After the file is parsed, it will be visually represented. Each horizontal line represents an individual job. The x-axis is time. Whether a job is running at a particular time is represented by its color at that time – white for running, black for idle. For example, a job which appears predominantly white has made efficient progress, whereas a job which appears predominantly black has received an inordinately small proportion of computational time.

### 2.13.1 Transition States

A transition state is the state of a job at any time. It is called a ”transition” because it is defined by the two events which bookmark it. There are two basic transition states: running and idle. An idle job typically is a job which has just been submitted into the Condor pool and is waiting to be matched with an appropriate machine or a job which has vacated from a machine and has been returned to the pool. A running job, by contrast, is a job which is making active progress.

Advanced users may want a visual distinction between two types of running transitions: ”goodput” or ”badput”. Goodput is the transition state preceding an eventual job completion or checkpoint. Badput is the transition state preceding a non-checkpointed eviction event. Note that ”badput” is potentially a misleading nomenclature; a job which is not checkpointed by the Condor program may checkpoint itself or make progress in some other way. To view these two transition as distinct transitions, select the appropriate option from the ”View” menu.

### 2.13.2 Events

There are two basic kinds of events: checkpoint events and error events. Plus advanced users can ask to see more events.

### 2.13.3 Selecting Jobs

To view any arbitrary selection of jobs in a job file, use the job selector tool. Jobs appear visually by order of appearance within the actual text log file. For example, the log file might contain jobs 775.1, 775.2, 775.3, 775.4, and 775.5, which appear in that order. A user who wishes to see only jobs 775.2 and 775.5 can select only these two jobs in the job selector tool and click the ”Ok” or ”Apply” button. The job selector supports double clicking; double click on any single job to see it drawn in isolation.

### 2.13.4 Zooming

To view a small area of the log file, zoom in on the area which you would like to see in greater detail. You can zoom in, out and do a full zoom. A full zoom redraws the log file in its entirety. For example, if you have zoomed in very close and would like to go all the way back out, you could do so with a succession of zoom outs or with one full zoom.

There is a difference between using the menu driven zooming and the mouse driven zooming. The menu driven zooming will recenter itself around the current center, whereas mouse driven zooming will recenter itself (as much as possible) around the mouse click. To help you re-find the clicked area, a box will flash after the zoom. This is called the ”zoom finder” and it can be turned off in the zoom menu if you prefer.

### 2.13.5 Keyboard and Mouse Shortcuts

1. The Keyboard shortcuts:
    - Arrows - an approximate ten percent scrollbar movement
    - PageUp and PageDown - an approximate one hundred percent scrollbar movement
    - Control + Left or Right - approximate one hundred percent scrollbar movement
    - End and Home - scrollbar movement to the vertical extreme
    - Others - as seen beside menu items
2. The mouse shortcuts:
    - Control + Left click - zoom in
    - Control + Right click - zoom out
    - Shift + left click - re-center

## 2.14 Special Environment Considerations

### 2.14.1 AFS

The Condor daemons do not run authenticated to AFS; they do not possess AFS tokens. Therefore, no child process of Condor will be AFS authenticated. The implication of this is that you must set file permissions so that your job can access any necessary files residing on an AFS volume without relying on having your AFS permissions.

If a job you submit to Condor needs to access files residing in AFS, you have the following choices:

1. Copy the needed files from AFS to either a local hard disk where Condor can access them using remote system calls (if this is a standard universe job), or copy them to an NFS volume.
2. If you must keep the files on AFS, then set a host ACL (using the AFS *fs setacl* command) on the subdirectory to serve as the current working directory for the job. If a standard universe job, then the host ACL needs to give read/write permission to any process on the submit machine. If vanilla universe job, then you need to set the ACL such that any host in the pool can access the files without being authenticated. If you do not know how to use an AFS host ACL, ask the person at your site responsible for the AFS configuration.

The Condor Team hopes to improve upon how Condor deals with AFS authentication in a subsequent release.

Please see section 3.12.1 on page 339 in the Administrators Manual for further discussion of this problem.

### 2.14.2 NFS Automounter

If your current working directory when you run *condor_submit* is accessed via an NFS automounter, Condor may have problems if the automounter later decides to unmount the volume before your job has completed. This is because *condor_submit* likely has stored the dynamic mount point as the job's initial current working directory, and this mount point could become automatically unmounted by the automounter.

There is a simple work around: When submitting your job, use the *initialdir* command in your submit description file to point to the stable access point. For example, suppose the NFS automounter is configured to mount a volume at mount point `/a/myserver.company.com/vol1/johndoe` whenever the directory `/home/johndoe` is accessed. Adding the following line to the submit description file solves the problem.

```
initialdir = /home/johndoe
```

### 2.14.3 Condor Daemons That Do Not Run as root

Condor is normally installed such that the Condor daemons have root permission. This allows Condor to run the condor_shadow process and your job with your UID and file access rights. When Condor is started as root, your Condor jobs can access whatever files you can.

However, it is possible that whomever installed Condor did not have root access, or decided not to run the daemons as root. That is unfortunate, since Condor is designed to be run as the Unix user root. To see if Condor is running as root on a specific machine, enter the command

```
condor_status -master -l <machine-name>
```

where `machine-name` is the name of the specified machine. This command displays a condor_master ClassAd; if the attribute `RealUid` equals zero, then the Condor daemons are indeed running with root access. If the `RealUid` attribute is not zero, then the Condor daemons do not have root access.

NOTE: The Unix program *ps* is *not* an effective method of determining if Condor is running with root access. When using *ps*, it may often appear that the daemons are running as the condor user instead of root. However, note that the *ps*, command shows the current *effective* owner of the process, not the *real* owner. (See the *getuid*(2) and *geteuid*(2) Unix man pages for details.) In Unix, a process running under the real UID of root may switch its effective UID. (See the *seteuid*(2) man page.) For security reasons, the daemons only set the effective UID to root when absolutely necessary (to perform a privileged operation).

If they are not running with root access, you need to make any/all files and/or directories that your job will touch readable and/or writable by the UID (user id) specified by the RealUid attribute. Often this may mean using the Unix command `chmod 777` on the directory where you submit your Condor job.

### 2.14.4 Job Leases

A job lease specifies how long a given job will attempt to run on a remote resource, even if that resource loses contact with the submitting machine. Similarly, it is the length of time the submitting machine will spend trying to reconnect to the (now disconnected) execution host, before the submitting machine gives up and tries to claim another resource to run the job. The goal aims at run only once semantics, so that the *condor_schedd* daemon does not allow the same job to run on multiple sites simultaneously.

If the submitting machine is alive, it periodically renews the job lease, and all is well. If the submitting machine is dead, or the network goes down, the job lease will no longer be renewed. Eventually the lease expires. While the lease has not expired, the execute host continues to try to run the job, in the hope that the submit machine will come back to life and reconnect. If the job completes, the lease has not expired,yet the submitting machine is still dead, the *condor_starter* daemon will wait for a *condor_shadow* daemon to reconnect, before sending final information on the job, and its output files. Should the lease expire, the *condor_startd* daemon kills off the *condor_starter* daemon and user job.

The user must set a value for `job_lease_duration` to keep a job running in the case that the submit side no longer renews the lease. There is a tradeoff in setting the value of `job_lease_duration`. Too small a value, and the job might get killed before the submitting machine has a chance to recover. Forward progress on the job will be lost. Too large a value, and execute resource will be tied up waiting for the job lease to expire. The value should be chosen based on how long is the user willing to tie up the execute machines, how quickly submit machines come back up, and how much work would be lost if the lease expires, the job is killed, and the job must start over from its beginning.

`job_lease_duration` is only valid for vanilla and java universe jobs. Chirp I/O and streaming I/O (which uses Chirp I/O) may not be used in conjunction with a defined `job_lease_duration`.

A current limitation is that jobs with a defined `job_lease_duration` will not reconnect if the jobs flock to a remote pool.

## 2.15 Potential Problems

### 2.15.1 Renaming of argv[0]

When Condor starts up your job, it renames argv[0] (which usually contains the name of the program) to condor_exec. This is convenient when examining a machine's processes with the Unix command *ps*; the process is easily identified as a Condor job.

Unfortunately, some programs read argv[0] expecting their own program name and get confused if they find something unexpected like condor_exec.

CHAPTER

# THREE

# Administrators' Manual

## 3.1 Introduction

This is the Condor Administrator's Manual for Unix. Its purpose is to aid in the installation and administration of a Condor pool. For help on using Condor, see the Condor User's Manual.

A Condor pool is comprised of a single machine which serves as the *central manager*, and an arbitrary number of other machines that have joined the pool. Conceptually, the pool is a collection of resources (machines) and resource requests (jobs). The role of Condor is to match waiting requests with available resources. Every part of Condor sends periodic updates to the central manager, the centralized repository of information about the state of the pool. Periodically, the central manager assesses the current state of the pool and tries to match pending requests with the appropriate resources.

Each resource has an owner, the user who works at the machine. This person has absolute power over their own resource and Condor goes out of its way to minimize the impact on this owner caused by Condor. It is up to the resource owner to define a policy for when Condor requests will serviced and when they will be denied.

Each resource request has an owner as well: the user who submitted the job. These people want Condor to provide as many CPU cycles as possible for their work. Often the interests of the resource owners are in conflict with the interests of the resource requesters.

The job of the Condor administrator is to configure the Condor pool to find the happy medium that keeps both resource owners and users of resources satisfied. The purpose of this manual is to help you understand the mechanisms that Condor provides to enable you to find this happy medium for your particular set of users and resource owners.

### 3.1.1 The Different Roles a Machine Can Play

Every machine in a Condor pool can serve a variety of roles. Most machines serve more than one role simultaneously. Certain roles can only be performed by single machines in your pool. The following list describes what these roles are and what resources are required on the machine that is providing that service:

**Central Manager** There can be only one central manager for your pool. The machine is the collector of information, and the negotiator between resources and resource requests. These two halves of the central manager's responsibility are performed by separate daemons, so it would be possible to have different machines providing those two services. However, normally they both live on the same machine. This machine plays a very important part in the Condor pool and should be reliable. If this machine crashes, no further matchmaking can be performed within the Condor system (although all current matches remain in effect until they are broken by either party involved in the match). Therefore, choose for central manager a machine that is likely to be online all the time, or at least one that will be rebooted quickly if something goes wrong. The central manager will ideally have a good network connection to all the machines in your pool, since they all send updates over the network to the central manager. All queries go to the central manager.

**Execute** Any machine in your pool (including your Central Manager) can be configured for whether or not it should execute Condor jobs. Obviously, some of your machines will have to serve this function or your pool won't be very useful. Being an execute machine doesn't require many resources at all. About the only resource that might matter is disk space, since if the remote job dumps core, that file is first dumped to the local disk of the execute machine before being sent back to the submit machine for the owner of the job. However, if there isn't much disk space, Condor will simply limit the size of the core file that a remote job will drop. In general the more resources a machine has (swap space, real memory, CPU speed, etc.) the larger the resource requests it can serve. However, if there are requests that don't require many resources, any machine in your pool could serve them.

**Submit** Any machine in your pool (including your Central Manager) can be configured for whether or not it should allow Condor jobs to be submitted. The resource requirements for a submit machine are actually much greater than the resource requirements for an execute machine. First of all, every job that you submit that is currently running on a remote machine generates another process on your submit machine. So, if you have lots of jobs running, you will need a fair amount of swap space and/or real memory. In addition all the checkpoint files from your jobs are stored on the local disk of the machine you submit from. Therefore, if your jobs have a large memory image and you submit a lot of them, you will need a lot of disk space to hold these files. This disk space requirement can be somewhat alleviated with a checkpoint server (described below), however the binaries of the jobs you submit are still stored on the submit machine.

**Checkpoint Server** One machine in your pool can be configured as a checkpoint server. This is optional, and is not part of the standard Condor binary distribution. The checkpoint server is a centralized machine that stores all the checkpoint files for the jobs submitted in your pool.

This machine should have lots of disk space and a good network connection to the rest of your pool, as the traffic can be quite heavy.

Now that you know the various roles a machine can play in a Condor pool, we will describe the actual daemons within Condor that implement these functions.

### 3.1.2 The Condor Daemons

The following list describes all the daemons and programs that could be started under Condor and what they do:

***condor_master*** This daemon is responsible for keeping all the rest of the Condor daemons running on each machine in your pool. It spawns the other daemons, and periodically checks to see if there are new binaries installed for any of them. If there are, the master will restart the affected daemons. In addition, if any daemon crashes, the master will send e-mail to the Condor Administrator of your pool and restart the daemon. The *condor_master* also supports various administrative commands that let you start, stop or reconfigure daemons remotely. The *condor_master* will run on every machine in your Condor pool, regardless of what functions each machine are performing.

***condor_startd*** This daemon represents a given resource (namely, a machine capable of running jobs) to the Condor pool. It advertises certain attributes about that resource that are used to match it with pending resource requests. The startd will run on any machine in your pool that you wish to be able to execute jobs. It is responsible for enforcing the policy that resource owners configure which determines under what conditions remote jobs will be started, suspended, resumed, vacated, or killed. When the startd is ready to execute a Condor job, it spawns the *condor_starter*, described below.

***condor_starter*** This program is the entity that actually spawns the remote Condor job on a given machine. It sets up the execution environment and monitors the job once it is running. When a job completes, the starter notices this, sends back any status information to the submitting machine, and exits.

***condor_schedd*** This daemon represents resource requests to the Condor pool. Any machine that you wish to allow users to submit jobs from needs to have a *condor_schedd* running. When users submit jobs, they go to the schedd, where they are stored in the *job queue*, which the schedd manages. Various tools to view and manipulate the job queue (such as *condor_submit*, *condor_q*, or *condor_rm*) all must connect to the schedd to do their work. If the schedd is down on a given machine, none of these commands will work.

The schedd advertises the number of waiting jobs in its job queue and is responsible for claiming available resources to serve those requests. Once a schedd has been matched with a given resource, the schedd spawns a *condor_shadow* (described below) to serve that particular request.

***condor_shadow*** This program runs on the machine where a given request was submitted and acts as the resource manager for the request. Jobs that are linked for Condor's standard universe, which perform remote system calls, do so via the *condor_shadow*. Any system call performed on the remote execute machine is sent over the network, back to the *condor_shadow* which actually performs the system call (such as file I/O) on the submit machine, and the result is sent back over the network to the remote job. In addition, the shadow is responsible for making decisions about the request (such as where checkpoint files should be stored, how certain files should be accessed, etc).

***condor_collector*** This daemon is responsible for collecting all the information about the status of a Condor pool. All other daemons periodically send ClassAd updates to the collector. These ClassAds contain all the information about the state of the daemons, the resources they represent or resource requests in the pool (such as jobs that have been submitted to a given schedd). The *condor_status* command can be used to query the collector for specific information about various parts of Condor. In addition, the Condor daemons themselves query the collector for important information, such as what address to use for sending commands to a remote machine.

***condor_negotiator*** This daemon is responsible for all the match-making within the Condor system. Periodically, the negotiator begins a *negotiation cycle*, where it queries the collector for the current state of all the resources in the pool. It contacts each schedd that has waiting resource requests in priority order, and tries to match available resources with those requests. The negotiator is responsible for enforcing user priorities in the system, where the more resources a given user has claimed, the less priority they have to acquire more resources. If a user with a better priority has jobs that are waiting to run, and resources are claimed by a user with a worse priority, the negotiator can preempt that resource and match it with the user with better priority.

NOTE: A higher numerical value of the user priority in Condor translate into worse priority for that user. The best priority you can have is 0.5, the lowest numerical value, and your priority gets worse as this number grows.

***condor_kbdd*** This daemon is only needed on Digital Unix. On that platforms, the *condor_startd* cannot determine console (keyboard or mouse) activity directly from the system. The *condor_kbdd* connects to the X Server and periodically checks to see if there has been any activity. If there has, the kbdd sends a command to the startd. That way, the startd knows the machine owner is using the machine again and can perform whatever actions are necessary, given the policy it has been configured to enforce.

***condor_ckpt_server*** This is the checkpoint server. It services requests to store and retrieve checkpoint files. If your pool is configured to use a checkpoint server but that machine (or the server itself is down) Condor will revert to sending the checkpoint files for a given job back to the submit machine.

See figure 3.1 for a graphical representation of the pool architecture.

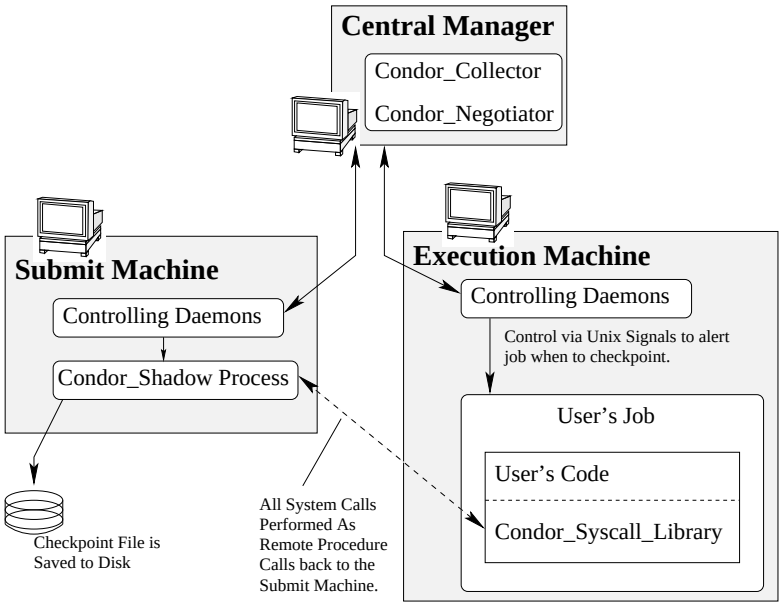



Figure 3.1: Pool Architecture

## 3.2 Installation

This section contains the instructions for installing Condor at your Unix site.

There are two sets of instructions for installing Condor. They are identified as the older and the newer installation instructions. All discussion prior to installation is valid whether using the newer installation script or the older method. The installation will have a default configuration that can be customized. Sections of the manual that follow this one explain customization.

Read this entire section before starting installation.

Please read the copyright and disclaimer information in section  on page xiii of the manual, or in the file `LICENSE.TXT`, before proceeding. Installation and use of Condor is acknowledgment that you have read and agree to the terms.

### 3.2.1 Obtaining Condor

The first step to installing Condor is to download it from the Condor web site, http://www.cs.wisc.edu/condor. The downloads are available from the downloads page, at http://www.cs.wisc.edu/condor/downloads/.

The platform-dependent Condor files are currently available from two sites. The main site is at the University of Wisconsin–Madison, Madison, Wisconsin, USA. A second site is the Istituto

Nazionale di Fisica Nucleare Sezione di Bologna, Bologna, Italy. Please choose the site nearest to you.

Make note of the location of where you download the binary into.

The Condor binary distribution is packaged in the following 5 files and 2 directories:

**DOC** directions on where to find Condor documentation

**INSTALL** these installation directions

**LICENSE.TXT** the licensing agreement. By installing Condor, you agree to the contents of this file

**README** general information

**condor_install** the Perl script used to install and configure Condor

**examples** directory containing C, Fortran and C++ example programs to run with Condor

**release.tar** tar file of the release directory, which contains the Condor binaries and libraries

Before you install, please consider joining the condor-world mailing list. Traffic on this list is kept to an absolute minimum. It is only used to announce new releases of Condor. To subscribe, send a message to majordomo@cs.wisc.edu with the body:

```
subscribe condor-world
```

### 3.2.2 Preparation

Before installation, make a few important decisions about the basic layout of your pool. The decisions answer the questions:

1. What machine will be the central manager?
2. What machines should be allowed to submit jobs?
3. Will Condor run as root or not?
4. Who will be administering Condor on the machines in your pool?
5. Will you have a Unix user named condor and will its home directory be shared?
6. Where should the machine-specific directories for Condor go?
7. Where should the parts of the Condor system be installed?
    - Configuration files

- Release directory
  - user binaries
  - system binaries
  - `lib` directory
  - `etc` directory
- Documentation

8. Am I using AFS?
9. Do I have enough disk space for Condor?

**1. What machine will be the central manager?** One machine in your pool must be the central manager. Install Condor on this machine first. This is the centralized information repository for the Condor pool, and it is also the machine that does match-making between available machines and submitted jobs. If the central manager machine crashes, any currently active matches in the system will keep running, but no new matches will be made. Moreover, most Condor tools will stop working. Because of the importance of this machine for the proper functioning of Condor, install the central manager on a machine that is likely to stay up all the time, or on one that will be rebooted quickly if it does crash.

Also consider network traffic and your network layout when choosing your central manager. All the daemons send updates (by default, every 5 minutes) to this machine. Memory requirements for the central manager differ by the number of machines in the pool. A pool with up to about 100 machines will require approximately 25 Mbytes of memory for the central manager's tasks. A pool with about 1000 machines will require approximately 100 Mbytes of memory for the central manager's tasks.

A faster CPU will improve the time to do matchmaking.

**2. Which machines should be allowed to submit jobs?** Condor can restrict the machines allowed to submit jobs. Alternatively, it can allow any machine the network allows to connect to a submit machine to submit jobs. If the Condor pool is behind a firewall, and all machines inside the firewall are trusted, the `HOSTALLOW_WRITE` configuration entry can be set to *. Otherwise, it should be set to reflect the set of machines permitted to submit jobs to this pool. Condor tries to be secure by default, so out of the box, the configuration file ships with an invalid definition for this configuration variable. This invalid value allows no machine to connect and submit jobs, so after installation, change this entry. Look for the entry defined with the value `YOU_MUST_CHANGE_THIS_INVALID_CONDOR_CONFIGURATION_VALUE`.

**3. Will Condor run as root or not?** Start up the Condor daemons as the Unix user root. Without this, Condor can do very little to enforce security and policy decisions. You can install Condor as any user, however there are both serious security and performance consequences. Please see section 3.6.12 on page 290 in the manual for the details and ramifications of running Condor as a Unix user other than root.

**4. Who will administer Condor?** Either root will be administering Condor directly, or someone else would be acting as the Condor administrator. If root has delegated the responsibility to another person but doesn't want to grant that person root access, root can specify a

`condor_config.root` file that will override settings in the other condor configuration files. This way, the global `condor_config` file can be owned and controlled by whoever is condor-admin, and the condor_config.root can be owned and controlled only by root. Settings that would compromise root security (such as which binaries are started as root) can be specified in the `condor_config.root` file while other settings that only control policy or condor-specific settings can still be controlled without root access.

5. **Will you have a Unix user named condor, and will its home directory be shared?** To simplify installation of Condor, create a Unix user named condor on all machines in the pool. The Condor daemons will create files (such as the log files) owned by this user, and the home directory can be used to specify the location of files and directories needed by Condor. The home directory of this user can either be shared among all machines in your pool, or could be a separate home directory on the local partition of each machine. Both approaches have advantages and disadvantages. Having the directories centralized can make administration easier, but also concentrates the resource usage such that you potentially need a lot of space for a single shared home directory. See the section below on machine-specific directories for more details.

   If you choose not to create a user named condor, then you must specify either via the `CONDOR_IDS` environment variable or the `CONDOR_IDS` config file setting which uid.gid pair should be used for the ownership of various Condor files. See section 3.6.12 on UIDs in Condor on page 289 in the Administrator's Manual for details.

6. **Where should the machine-specific directories for Condor go?** Condor needs a few directories that are unique on every machine in your pool. These are `spool`, `log`, and `execute`. Generally, all three are subdirectories of a single machine specific directory called the local directory (specified by the `LOCAL_DIR` macro in the configuration file). Each should be owned by the user that Condor is to be run as.

   If you have a Unix user named condor with a local home directory on each machine, the `LOCAL_DIR` could just be user condor's home directory (`LOCAL_DIR` = `$(TILDE)` in the configuration file). If this user's home directory is shared among all machines in your pool, you would want to create a directory for each host (named by host name) for the local directory (for example, `LOCAL_DIR` = `$(TILDE)`/hosts/`$(HOSTNAME)`). If you do not have a condor account on your machines, you can put these directories wherever you'd like. However, where to place them will require some thought, as each one has its own resource needs:

   **`execute`** This is the directory that acts as the current working directory for any Condor jobs that run on a given execute machine. The binary for the remote job is copied into this directory, so there must be enough space for it. (Condor will not send a job to a machine that does not have enough disk space to hold the initial binary). In addition, if the remote job dumps core for some reason, it is first dumped to the execute directory before it is sent back to the submit machine. So, put the execute directory on a partition with enough space to hold a possible core file from the jobs submitted to your pool.

   **`spool`** The `spool` directory holds the job queue and history files, and the checkpoint files for all jobs submitted from a given machine. As a result, disk space requirements for the `spool` directory can be quite large, particularly if users are submitting jobs with very large executables or image sizes. By using a checkpoint server (see section 3.8

on Installing a Checkpoint Server on page 315 for details), you can ease the disk space requirements, since all checkpoint files are stored on the server instead of the spool directories for each machine. However, the initial checkpoint files (the executables for all the clusters you submit) are still stored in the spool directory, so you will need some space, even with a checkpoint server.

**log** Each Condor daemon writes its own log file, and each log file is placed in the `log` directory. You can specify what size you want these files to grow to before they are rotated, so the disk space requirements of the directory are configurable. The larger the log files, the more historical information they will hold if there is a problem, but the more disk space they use up. If you have a network file system installed at your pool, you might want to place the log directories in a shared location (such as `/usr/local/condor/logs/$(HOSTNAME)`), so that you can view the log files from all your machines in a single location. However, if you take this approach, you will have to specify a local partition for the `lock` directory (see below).

**lock** Condor uses a small number of lock files to synchronize access to certain files that are shared between multiple daemons. Because of problems encountered with file locking and network file systems (particularly NFS), these lock files should be placed on a local partition on each machine. By default, they are placed in the `log` directory. If you place your `log` directory on a network file system partition, specify a local partition for the lock files with the `LOCK` parameter in the configuration file (such as `/var/lock/condor`).

Generally speaking, it is recommended that you do not put these directories (except `lock`) on the same partition as `/var`, since if the partition fills up, you will fill up `/var` as well. This will cause lots of problems for your machines. Ideally, you will have a separate partition for the Condor directories. Then, the only consequence of filling up the directories will be Condor's malfunction, not your whole machine.

**7. Where should the parts of the Condor system be installed?**

- Release directory
  - User Binaries
  - System Binaries
  - `lib` Directory
  - `etc` Directory
- Documentation
- Configuration Files

**Configuration Files** There are a number of configuration files that allow you different levels of control over how Condor is configured at each machine in your pool. The global configuration file is shared by all machines in the pool. For ease of administration, this file should be located on a shared file system, if possible. In addition, there is a local configuration file for each machine, where you can override settings in the global file. This allows you to have different daemons running, different policies for when to start and stop Condor jobs, and so on. You can also have configuration files specific to each platform in your pool. See section 3.12.2 on page 340 about Configuring Condor for Multiple Platforms for details.

In addition, because we recommend that you start the Condor daemons as root, we allow you to create configuration files that are owned and controlled by root that will override any other Condor settings. This way, if the Condor administrator is not root, the regular Condor configuration files can be owned and writable by condor-admin, but root does not have to grant root access to this person. See section **??** on page **??** in the manual for a detailed discussion of the root configuration files, if you should use them, and what settings should be in them.

In general, there are a number of places that Condor will look to find its configuration files. The first file it looks for is the global configuration file. These locations are searched in order until a configuration file is found. If none contain a valid configuration file, Condor will print an error message and exit:

1. File specified in the `CONDOR_CONFIG` environment variable
2. `/etc/condor/condor_config`
3. `~condor/condor_config`
4. `$(GLOBUS_LOCATION)/etc/condor_config`

If you specify a file in the `CONDOR_CONFIG` environment variable and there's a problem reading that file, Condor will print an error message and exit right away, instead of continuing to search the other options. However, if no `CONDOR_CONFIG` environment variable is set, Condor will search through the other options.

Next, Condor tries to load the local configuration file(s). The only way to specify the local configuration file(s) is in the global configuration file, with the `LOCAL_CONFIG_FILE` macro. If that macro is not set, no local configuration file is used. This macro can be a list of files or a single file.

The root configuration files come in last. The global file is searched for in the following places:

1. `/etc/condor/condor_config.root`
2. `~condor/condor_config.root`

The local root configuration file(s) are found with the `LOCAL_ROOT_CONFIG_FILE` macro. If that is not set, no local root configuration file is used. This macro can be a list of files or a single file.

**Release Directory** Every binary distribution contains a `release.tar` file that contains five subdirectories: `bin`, `etc`, `lib`, `sbin`, and `libexec`. Wherever you choose to install these five directories we call the release directory (specified by the `RELEASE_DIR` macro in the configuration file). Each release directory contains platform-dependent binaries and libraries, so you will need to install a separate one for each kind of machine in your pool. For ease of administration, these directories should be located on a shared file system, if possible.

- User Binaries:
  All of the files in the `bin` directory are programs the end Condor users should expect to have in their path. You could either put them in a well known location (such as `/usr/local/condor/bin`) which you have Condor users add to their `PATH` environment variable, or copy those files directly into a well known place already in the user's PATHs (such as `/usr/local/bin`). With the above examples, you

could also leave the binaries in `/usr/local/condor/bin` and put in soft links from `/usr/local/bin` to point to each program.

- System Binaries:
  All of the files in the `sbin` directory are Condor daemons and agents, or programs that only the Condor administrator would need to run. Therefore, add these programs only to the `PATH` of the Condor administrator.
- Private Condor Binaries:
  All of the files in the `libexec` directory are Condor programs that should never be run by hand, but are only used internally by Condor.
- `lib` Directory:
  The files in the `lib` directory are the Condor libraries that must be linked in with user jobs for all of Condor's checkpointing and migration features to be used. `lib` also contains scripts used by the *condor_compile* program to help re-link jobs with the Condor libraries. These files should be placed in a location that is world-readable, but they do not need to be placed in anyone's `PATH`. The *condor_compile* script checks the configuration file for the location of the `lib` directory.
- `etc` Directory:
  `etc` contains an `examples` subdirectory which holds various example configuration files and other files used for installing Condor. `etc` is the recommended location to keep the master copy of your configuration files. You can put in soft links from one of the places mentioned above that Condor checks automatically to find its global configuration file.

**Documentation** The documentation provided with Condor is currently available in HTML, Postscript and PDF (Adobe Acrobat). It can be locally installed wherever is customary at your site. You can also find the Condor documentation on the web at: http://www.cs.wisc.edu/condor/manual.

**7. Am I using AFS?** If you are using AFS at your site, be sure to read the section 3.12.1 on page 338 in the manual. Condor does not currently have a way to authenticate itself to AFS. A solution is not ready for Version 6.8.6. This implies that you are probably not going to want to have the `LOCAL_DIR` for Condor on AFS. However, you can (and probably should) have the Condor `RELEASE_DIR` on AFS, so that you can share one copy of those files and upgrade them in a centralized location. You will also have to do something special if you submit jobs to Condor from a directory on AFS. Again, read manual section 3.12.1 for all the details.

**8. Do I have enough disk space for Condor?** Condor takes up a fair amount of space. This is another reason why it is a good idea to have it on a shared file system. The size requirements for the downloads are given on the downloads page. They currently vary from about 20 Mbytes (statically linked HP Unix on a PA RISC) to more than 50 Mbytes (dynamically linked Irix on an SGI).

In addition, you will need a lot of disk space in the local directory of any machines that are submitting jobs to Condor. See question 6 above for details on this.

### 3.2.3 Newer Unix Installation Procedure

The Perl script *condor_configure* installs Condor. Command-line arguments specify all needed information to this script. The script can be executed multiple times, to modify or further set the configuration. *condor_configure* has been tested using Perl 5.003. Use this or a more recent version of Perl.

After download, all the files are in a compressed, tar format. They need to be untarred, as

```
tar xzf completename.tar.gz
```

After untarring, the directory will have the Perl script *condor_configure*, as well as a second tar file called `release.tar`. *condor_configure* works on `release.tar`.

*condor_configure* is completely command-line driven; it is not interactive. Several command-line arguments are always needed with *condor_configure*. The argument

```
--install=/path/to/release.tar
```

specifies the path to the Condor release tarball. The argument

```
--install-dir=directory
```

specifies the path to the install directory. The argument

```
--local-dir=directory
```

specifies the path to the local directory.

The **–type** option to *condor_configure* specifies one or more of the roles that a machine may take on within the Condor pool: central manager, submit or execute. These options are given in a comma separated list. So, if a machine is both a submit and execute machine, the proper command-line option is

```
--type=manager,execute
```

Configure Condor on the central manager machine first. If Condor will run as root in this pool (Item 3 above), run *condor_configure* as root, and it will install and set the file permissions correctly. On the central manager machine, run *condor_configure* as follows.

```
% condor_configure --install=r.tar --install-dir=~condor \
--local-dir=/scratch/condor --type=manager
```

The central manager can also be a submit point or and execute machine, but this is only recommended for very small pools. If this is the case, the **–type** option changes to `manager,execute` or `manager,submit` or `manager,submit,execute`.

After the central manager is installed, the execute and submit machines should then be configured. Decisions about whether to run Condor as root should be consistent throughout the pool. For each machine in the pool, run

```
% condor_configure --install=r.tar --install-dir=~condor \
--local-dir=/scratch/condor --type=execute,submit
```

See the *condor_configure* manual page in section 9 on page 629 for details.

Please skip to section 3.2.5 for final instructions on configuring and starting Condor.

### 3.2.4 Older Unix Installation Procedure

IF YOU HAVE DECIDED TO CREATE A condor USER AND GROUP, DO THAT ON ALL YOUR MACHINES BEFORE YOU DO ANYTHING ELSE.

After download, all the files are in a compressed, tar format. They need to be untarred, as

```
tar xzf completename.tar.gz
```

To install Condor, use one or both of the scripts provided to help you: *condor_install* and *condor_init*. Run these scripts as the user that you are going to run the Condor daemons as. First, run *condor_install* on the machine that will be a file server for shared files used by Condor, such as the release directory, and possibly the condor user's home directory. When you do, choose the "full-install" option in step #1 described below.

Once you have run *condor_install* on a file server to set up your release directory and configure Condor for your site, you should run *condor_init* on any other machines in your pool to create any locally used files that are not created by *condor_install*. In the most simple case, where nearly all of Condor is installed on a shared file system, even though *condor_install* will create nearly all the files and directories you need, you will still need to use *condor_init* to create the `LOCK` directory on the local disk of each machine. If you have a shared release directory, but the `LOCAL_DIR` is local on each machine, *condor_init* will create all the directories and files needed in `LOCAL_DIR`. In addition, *condor_init* will create any soft links on each machine that are needed so that Condor can find its global configuration file.

If you do not have a shared file system, you need to run *condor_install* on each machine in your pool to set up Condor. In this case, there is no need to run *condor_init* at all.

In addition, you will want to run *condor_install* on your central manager machine if that machine is different from your file server, using the "central-manager" option in step #1 described below. Run *condor_install* on your file server first, then on your central manager.

If this step fails for some reason (NFS permissions, etc), you can do it manually quite easily. All this does is copy the `condor_config.local.central.manager` file from `<release_dir>/etc/examples` to the proper location for the local configuration file of your central manager machine. If your central manager is an Alpha or an SGI, you might want to add `KBDD` to the `$(DAEMON_LIST)` macro. See section 3.3 Configuring Condor on page 138 of the manual for details.

*condor_install* assumes you have perl installed in `/usr/bin/perl`. If this is not the case, you can either edit the script to put in the right path, or you will have to invoke perl directly from your shell (assuming perl is in your PATH):

```
% perl condor_install
```

*condor_install* breaks down the installation procedure into various steps. Each step is clearly numbered. The following section explains what each step is for, and suggests how to answer the questions *condor_install* will ask you for each one.

***condor_install*, step-by-step**

**STEP 1: What type of Condor installation do you want?** There are three types of Condor installation you might choose: 'submit-only', 'full-install', and 'central-manager'. A submit-only machine can submit jobs to a Condor pool, but Condor jobs will not run on it. A full-install machine can both submit and run Condor jobs.

If you are planning to run Condor jobs on your machines, you should either install and run Condor as root, or as the Unix user condor.

If you are planning to set up a submit-only machine, you can either install Condor machine-wide as root or user condor, or, you can install Condor as yourself into your home directory.

The other possible installation type is setting up a machine as a central manager. If you do a full-install and you say that you want the local host to be your central manager, this step will be done automatically. You should only choose the central-manager option at step 1 if you have already run *condor_install* on your file server and you now want to run *condor_install* on a different machine that will be your central manager.

**STEP 2: How many machines are you setting up this way?** If you are installing Condor for multiple machines and you have a shared file system, then *condor_install* will prompt you for the host names of each machine you want to add to your Condor pool. If you do not have a shared file system, you will have to run *condor_install* locally on each machine, so *condor_install* does not ask for the names. If you provide a list, it will use the names to automatically create directories and files later. At the end, *condor_install* will dump out this list to a `roster` file which can be used by scripts to help maintain your Condor pool.

If you are only installing Condor on 1 machine, you would answer no to the first question, and move on.

**STEP 3: Install the Condor release directory** The release directory contains five subdirectories: `bin`, `etc`, `lib`, `libexec` and `sbin`. `bin` contains user-level executable programs. `etc`

is the recommended location for your Condor configuration files, and it also includes an `examples` directory with default configuration files and other default files used for installing Condor. `lib` contains libraries to link Condor user programs and scripts used by the Condor system. `sbin` contains all administrative executable programs and the Condor daemons. `libexec` contains programs that only Condor needs to execute.

If you have multiple machines with a shared file system that will be running Condor, put the release directory on that shared file system so you only have one copy of all the binaries, and so that when you update them, you can do so in one place. Note that the release directory is architecture dependent, so download separate binary distributions for every platform in your pool.

*condor_install* tries to find an already installed release directory. If it cannot find one, it asks if you have installed one already. If you have not installed one, it tries to do so for you by untarring the `release.tar` file from the binary distribution.

NOTE: If you are only setting up a central manager (you chose 'central manager' in STEP 1), STEP 3 is the last question you will need to answer.

**STEP 4: How and where should Condor send e-mail if things go wrong?** Various parts of Condor will send e-mail to a condor administrator if something goes wrong that needs human attention. You will need to specify the e-mail address of this administrator.

You also specify the full path to a mail program that Condor will use to send the e-mail. This program needs to understand the **-s** option, to specify a subject for the outgoing message. The default on most platforms will probably be correct. On Linux machines, since there is such variation in Linux distributions and installations, verify that the default works. If the script complains that it cannot find the mail program that was specified, try

```
% which mail
```

to see what mail program is currently in your PATH. If there is none, try

```
% which mailx
```

If you still cannot find anything, ask your system administrator. Verify that the program you use supports **-s**. The man page for that program will probably tell you.

**STEP 5: File system and UID domains.** Condor does not depend on a shared file system and common UID space for running jobs outside the standard universe. Jobs can specify file transfer within the submit description file. See section 2.5.4 on page 36 for information on properly setting up file transfer for jobs.

To utilize the convenience of a shared file system (such as NFS), a common UID space is needed. This is one in which there is a unique mapping of user names to UIDs across all machines in the common UID space. It is important to correctly configure Condor with respect to a shared file system. For complete details on what these settings do and how you should answer the questions, read section 3.3.7, Shared File System Configuration File Entries", on page 159.

You will be asked if you have a shared file system. If so, condor_install will configure your FILESYSTEM_DOMAIN setting to be set to the domain name of the machine running condor_install. If not, FILESYSTEM_DOMAIN will be set to $(FULL_HOSTNAME), indicating that each machine is in its own domain.

For the UID domain, Condor needs to know if all user names across all the machines in the pool have a unique UID. If so, UID_DOMAIN will be set to the domain name of the machine running condor_install. If not, UID_DOMAIN will be set to $(FULL_HOSTNAME), indicating that each machine is in its own domain.

If you have a common UID_DOMAIN, condor_install will ask if you have a *soft UID domain*, meaning that although you have unique UIDs, not every machine in your pool has all the users in their individual password files. Please see the description of SOFT_UID_DOMAIN in section 3.3.7 on page 160 for details.

**STEP 6: Java Universe support in Condor.** Condor has the ability to run Java programs with remote i/o, but no checkpointing. If you would like to enable this feature in Condor, then select yes for enabling of Java Universe. The installer will try to determine if you have a valid JVM and prompt you if nothing suitable can be found. If you do not care to use the Java Universe, then it is safe to say no here.

**STEP 7: Where should public programs be installed?** It is recommended that you install the user-level Condor programs in the release directory, (where they go by default). This way, when you want to install a new version of the Condor binaries, you can just replace your release directory and everything will be updated at once. So, one option is to have Condor users add <release_dir>/bin to their PATH, so that they can access the programs. However, we recommend putting in soft links from some directory already in their PATH (such as /usr/local/bin) that point back to the Condor user programs. *condor_install* will do this for you. All you do is tell it what directory to put these links into. This way, users do not have to change their PATH to use Condor, and you can still have the binaries installed in their own location.

If you are installing Condor as neither root nor condor, there is a perl script wrapper to all the Condor tools that is created which sets some appropriate environment variables and automatically passes certain options to the tools. This is all created automatically by *condor_install*. So, you need to tell *condor_install* where to put this perl script. The script itself is linked to itself with many different names, since it is the name that determines the behavior of the script. This script should go somewhere that is in your PATH already, if possible (such as ~bin).

At this point, the remaining steps differ based on the whether the installation is a full install or a submit-only. Skip to the appropriate section below, based on the kind of installation.

### Full Install

**STEP 8: What machine will be your central manager?** Type in the full host name of the machine you have chosen for your central manager. If *condor_install* cannot find information about the host you typed by querying your name server, it will print out an error message and ask you to confirm.

**STEP 9: Where will the local directory go?** This is the directory discussed in question 5 of the installation introduction. *condor_install* tries to make some educated guesses as to what directory you want to use for the purpose. Agree to the correct guess, or (when *condor_install* has run out of guesses) type in what you want. Since this directory needs to be unique, it is common to use the host name of each machine in its name. When typing in your own path, you can use '`$(HOSTNAME)`' which *condor_install* (and the Condor configuration files) will expand to the host name of the machine you are currently on. *condor_install* will try to create the corresponding directories for all the machines you told it about in STEP 2 above.

Once you have selected the local directory, *condor_install* creates all the needed subdirectories of each one with the proper permissions. They should have the following permissions and ownerships:

```
drwxr-xr-x   2 condor   root         1024 Mar  6 01:30 execute/
drwxr-xr-x   2 condor   root         1024 Mar  6 01:30 log/
drwxr-xr-x   2 condor   root         1024 Mar  6 01:30 spool/
```

If your local directory is on a shared file system, *condor_install* will prompt you for the location of your lock files, as discussed in question #5 above. In this case, when *condor_install* is finished, you will have to run *condor_init* on each machine in your pool to create the `lock` directory before you can start up Condor.

**STEP 10: Where will the local (machine-specific) configuration files go?** As discussed in question STEP 6 above, there are a few different levels of Condor configuration files. There is the global configuration file that will be installed in `<release_dir>/etc/condor_config`, and there are machine-specific, or local configuration files, that override the settings in the global file. If you are installing on multiple machines or are configuring your central manager machine, you must select a location for your local configuration files.

The two main options are to have a single directory that holds all the local configuration files, each one named `$(HOSTNAME).local`, or to have the local configuration files go into the individual local directories for each machine. Given a shared file system, we recommend the first option, since it makes it easier to configure your pool from a centralized location.

**STEP 11: How shall Condor find its configuration file?** Since there are a few known places Condor looks to find your configuration file, we recommend that you put a soft link from one of them to point to `<release_dir>/etc/condor_config`. This way, you can keep your Condor configuration in a centralized location, but all the Condor daemons and tools will be able to find their configuration files. Alternatively, you can set the CONDOR_CONFIG environment variable to contain `<release_dir>/etc/condor_config`.

*condor_install* will ask you if you want to create a soft link from either of the two fixed locations that Condor searches.

Once you have completed STEP 11, you are done. *condor_install* prints out a messages describing what to do next. Please skip to section 3.2.5.

**Submit Only**

A submit-only installation of Condor implies that the machine will be submitting jobs to one or more established Condor pools. Configuration for this installation needs to account for the other pools.

For the submit-only installation, STEP 6 continues and completes the installation.

**STEP 6: continued.** A submit-only machine has the option of submission to more than one Condor pool. The full host name of the central manager is required for each pool. The first entered becomes the default for start up and job submission.

There is a separate configuration file for each pool. The location of each file is specified.

Identification of each pool requires a unique name. A final question sets a name for each pool. The name will be the argument for **-pool** command line options.

### 3.2.5 Condor is installed Under Unix ... now what?

Now that Condor has been installed on your machine(s), there are a few things you should check before you start up Condor.

1. Read through the <`release_dir`>`/etc/condor_config` file. There are a lot of possible settings and you should at least take a look at the first two main sections to make sure everything looks okay. In particular, you might want to set up security for Condor. See the section 3.6.1 on page 261 to learn how to do this.
2. Condor can monitor the activity of your mouse and keyboard, provided that you tell it where to look. You do this with the `CONSOLE_DEVICES` entry in the condor_startd section of the configuration file. On most platforms, reasonable defaults are provided. For example, the default device for the mouse on Linux is 'mouse', since most Linux installations have a soft link from `/dev/mouse` that points to the right device (such as `tty00` if you have a serial mouse, `psaux` if you have a PS/2 bus mouse, etc). If you do not have a `/dev/mouse` link, you should either create one (you will be glad you did), or change the `CONSOLE_DEVICES` entry in Condor's configuration file. This entry is a comma separated list, so you can have any devices in `/dev` count as 'console devices' and activity will be reported in the condor_startd's ClassAd as `ConsoleIdleTime`.
3. (Linux only) Condor needs to be able to find the `utmp` file. According to the Linux File System Standard, this file should be `/var/run/utmp`. If Condor cannot find it there, it looks in `/var/adm/utmp`. If it still cannot find it, it gives up. So, if your Linux distribution places this file somewhere else, be sure to put a soft link from `/var/run/utmp` to point to the real location.

To start up the Condor daemons, execute <`release_dir`>`/sbin/condor_master`. This is the Condor master, whose only job in life is to make sure the other Condor daemons are running.

The master keeps track of the daemons, restarts them if they crash, and periodically checks to see if you have installed new binaries (and if so, restarts the affected daemons).

If you are setting up your own pool, you should start Condor on your central manager machine first. If you have done a submit-only installation and are adding machines to an existing pool, the start order does not matter.

To ensure that Condor is running, you can run either:

```
ps -ef | egrep condor_
```

or

```
ps -aux | egrep condor_
```

depending on your flavor of Unix. On a central manager machine that can submit jobs as well as execute them, there will be processes for:

- condor_master
- condor_collector
- condor_negotiator
- condor_startd
- condor_schedd

On a central manager machine that does not submit jobs nor execute them, there will be processes for:

- condor_master
- condor_collector
- condor_negotiator

For a machine that only submits jobs, there will be processes for:

- condor_master
- condor_schedd

For a machine that only executes jobs, there will be processes for:

- condor_master

- condor_startd

(NOTE: On Alphas, there will also be a condor_kbdd – see section 3.12.4 on page 343 of the manual for details.)

Once you are sure the Condor daemons are running, check to make sure that they are communicating with each other. You can run *condor_status* to get a one line summary of the status of each machine in your pool.

Once you are sure Condor is working properly, you should add *condor_master* into your startup/bootup scripts (i.e. `/etc/rc` ) so that your machine runs *condor_master* upon bootup. *condor_master* will then fire up the necessary Condor daemons whenever your machine is rebooted.

If your system uses System-V style init scripts, you can look in `<release_dir>/etc/examples/condor.boot` for a script that can be used to start and stop Condor automatically by init. Normally, you would install this script as `/etc/init.d/condor` and put in soft link from various directories (for example, `/etc/rc2.d`) that point back to `/etc/init.d/condor`. The exact location of these scripts and links will vary on different platforms.

If your system uses BSD style boot scripts, you probably have an `/etc/rc.local` file. Add a line to start up `<release_dir>/sbin/condor_master`.

Now that the Condor daemons are running, there are a few things you can and should do:

1. (Optional) Do a full install for the *condor_compile* script. condor_compile assists in linking jobs with the Condor libraries to take advantage of all of Condor's features. As it is currently installed, it will work by placing it in front of any of the following commands that you would normally use to link your code: gcc, g++, g77, cc, acc, c89, CC, f77, fort77 and ld. If you complete the full install, you will be able to use condor_compile with any command whatsoever, in particular, make. See section 3.12.3 on page 342 in the manual for directions.
2. Try building and submitting some test jobs. See `examples/README` for details.
3. If your site uses the AFS network file system, see section 3.12.1 on page 338 in the manual.
4. We strongly recommend that you start up Condor (run the *condor_master* daemon) as user root. If you must start Condor as some user other than root, see section 3.6.12 on page 290.

### 3.2.6 Installation on Windows

This section contains the instructions for installing the Microsoft Windows version of Condor at your site. The install program will set you up with a slightly customized configuration file that you can further customize after the installation has completed.

Please read the copyright and disclaimer information in section  on page xiii of the manual, or in the file `LICENSE.TXT`, before proceeding. Installation and use of Condor is acknowledgement that you have read and agreed to these terms.

Be sure that the Condor tools that get run are of the same version as the daemons installed. If they were not (such as 6.5.3 daemons, when running 6.4 *condor_submit*), then things will not work. There may be errors generated by the *condor_schedd* daemon (in the log). It is likely that a job would be correctly placed in the queue, but the job will never run.

The Condor executable for distribution is packaged in a single file such as:

```
condor-6.7.8-winnt40-x86.msi
```

This file is approximately 80 Mbytes in size, and may be removed once Condor is fully installed.

Before installing Condor, please consider joining the condor-world mailing list. Traffic on this list is kept to an absolute minimum. It is only used to announce new releases of Condor. To subscribe, follow the directions given at http://www.cs.wisc.edu/condor/mail-lists/.

**Installation Requirements**

- Condor for Windows requires Windows 2000 (or better) or Windows XP.
- 300 megabytes of free disk space is recommended. Significantly more disk space could be desired to be able to run jobs with large data files.
- Condor for Windows will operate on either an NTFS or FAT filesystem. However, for security purposes, NTFS is preferred.

**Preparing to Install Condor under Windows**

Before you install the Windows version of Condor at your site, there are two major decisions to make about the basic layout of your pool.

1. What machine will be the central manager?
2. Do I have enough disk space for Condor?

If you feel that you already know the answers to these questions, skip to the Windows Installation Procedure section below, section 3.2.6 on page 125. If you are unsure, read on.

- What machine will be the central manager?

  One machine in your pool must be the central manager. This is the centralized information repository for the Condor pool and is also the machine that matches available machines with waiting jobs. If the central manager machine crashes, any currently active matches in the system will keep running, but no new matches will be made. Moreover, most Condor tools will stop working. Because of the importance of this machine for the proper functioning of Condor, we recommend you install it on a machine that is likely to stay up all the time, or at

the very least, one that will be rebooted quickly if it does crash. Also, because all the services will send updates (by default every 5 minutes) to this machine, it is advisable to consider network traffic and your network layout when choosing the central manager.

For Personal Condor, your machine will act as your central manager.

Install Condor on the central manager before installing on the other machines within the pool.

- Do I have enough disk space for Condor?

  The Condor release directory takes up a fair amount of space. The size requirement for the release directory is approximately 200 Mbytes.

  Condor itself, however, needs space to store all of your jobs, and their input files. If you will be submitting large amounts of jobs, you should consider installing Condor on a volume with a large amount of free space.

#### Installation Procedure using the included Set Up Program

Installation of Condor must be done by a user with administrator privileges. After installation, the Condor services will be run under the local system account. When Condor is running a user job, however, it will run that User job with normal user permissions.

Download Condor, and start the installation process by running the file (or by double clicking on the file). The Condor installation is completed by answering questions and choosing options within the following steps.

**If Condor is already installed.** For upgrade purposes, you may be running the installation of Condor after it has been previously installed. In this case, a dialog box will appear before the installation of Condor proceeds. The question asks if you wish to preserve your current Condor configuration files. Answer yes or no, as appropriate.

If you answer yes, your configuration files will not be changed, and you will proceed to the point where the new binaries will be installed.

If you answer no, then there will be a second question that asks if you want to use answers given during the previous installation as default answers.

**STEP 1: License Agreement.** The first step in installing Condor is a welcome screen and license agreement. You are reminded that it is best to run the installation when no other Windows programs are running. If you need to close other Windows programs, it is safe to cancel the installation and close them. You are asked to agree to the license. Answer yes or no. If you should disagree with the License, the installation will not continue.

After agreeing to the license terms, the next Window is where fill in your name and company information, or use the defaults as given.

**STEP 2: Condor Pool Configuration.** The Condor installation will require different information depending on whether the installer will be creating a new pool, or joining an existing one.

If you are creating a new pool, the installation program requires that this machine is the central manager. For the creation of a new Condor pool, you will be asked some basic information about your new pool:

**Name of the pool**

**hostname** of this machine.

**Size of pool** Condor needs to know if this a Personal Condor installation, or if there will be more than one machine in the pool. A Personal Condor pool implies that there is only one machine in the pool. For Personal Condor, several of the following steps are omitted as noted.

If you are joining an existing pool, all the installation program requires is the hostname of the central manager for your pool.

**STEP 3: This Machine's Roles.** This step is omitted for the installation of Personal Condor.

Each machine within a Condor pool may either submit jobs or execute submitted jobs, or both submit and execute jobs. This step allows the installation on this machine to choose if the machine will only submit jobs, only execute submitted jobs, or both. The common case is both, so the default is both.

**STEP 4: Where will Condor be installed?** The next step is where the destination of the Condor files will be decided. It is recommended that Condor be installed in the location shown as the default in the dialog box: `C:\Condor`.

Installation on the local disk is chosen for several reasons.

The Condor services run as local system, and within Microsoft Windows, local system has no network privileges. Therefore, for Condor to operate, Condor should be installed on a local hard drive as opposed to a network drive (file server).

The second reason for installation on the local disk is that the Windows usage of drive letters has implications for where Condor is placed. The drive letter used must be not change, even when different users are logged in. Local drive letters do not change under normal operation of Windows.

While it is strongly discouraged, it may be possible to place Condor on a hard drive that is not local, if a dependency is added to the service control manager such that Condor starts after the required file services are available.

**STEP 5: Where is the Java Virtual Machine?** While not required, it is possible for Condor to run jobs in the Java universe. In order for Condor to have support for java, you must supply a path to `java.exe` on your system. The installer will tell you if the path is invalid before proceeding to the next step. To disable the Java universe, simply leave this field blank.

**STEP 6: Where should Condor send e-mail if things go wrong?** Various parts of Condor will send e-mail to a Condor administrator if something goes wrong and requires human attention. You specify the e-mail address and the SMTP relay host of this administrator. Please pay close attention to this email since it will indicate problems in your Condor pool.

**STEP 7: The domain.** This step is omitted for the installation of Personal Condor.

Enter the machine's accounting (or UID) domain. On this version of Condor for Windows, this setting only used for User priorities (see section 3.4 on page 215) and to form a default email address for the user.

**STEP 8: Access permissions.** This step is omitted for the installation of Personal Condor.

Machines within the Condor pool will need various types of access permission. The three categories of permission are read, write, and administrator. Enter the machines to be given access permissions.

**Read** Read access allows a machine to obtain information about Condor such as the status of machines in the pool and the job queues. All machines in the pool should be given read access. In addition, giving read access to *.cs.wisc.edu will allow the Condor team to obtain information about your Condor pool in the event that debugging is needed.

**Write** All machines in the pool should be given write access. It allows the machines you specify to send information to your local Condor daemons, for example, to start a Condor Job. Note that for a machine to join the Condor pool, it must have both read and write access to all of the machines in the pool.

**Administrator** A machine with administrator access will be allowed more extended permission to to things such as change other user's priorities, modify the job queue, turn Condor services on and off, and restart Condor. The central manager should be given administrator access and is the default listed. This setting is granted to the entire machine, so care should be taken not to make this too open.

For more details on these access permissions, and others that can be manually changed in your `condor_config` file, please see the section titled Setting Up IP/Host-Based Security in Condor in section section 3.6.10 on page 282.

**STEP 9: Job Start Policy.** Condor will execute submitted jobs on machines based on a preference given at installation. Three options are given, and the first is most commonly used by Condor pools. This specification may be changed or refined in the machine ClassAd requirements attribute.

The three choices:

**After 15 minutes of no console activity and low CPU activity.**

**Always run Condor jobs.**

**After 15 minutes of no console activity.**

Console activity is the use of the mouse or keyboard. For instance, if you are reading this document online, and are using either the mouse or the keyboard to change your position, you are generating Console activity.

Low CPU activity is defined as a load of less than 30%(and is configurable in your `condor_config` file). If you have a multiple processor machine, this is the average percentage of CPU activity for both processors.

For testing purposes, it is often helpful to use use the Always run Condor jobs option. For production mode, however, most people chose the After 15 minutes of no console activity and low CPU activity.

**STEP 10: Job Vacate Policy.** This step is omitted if Condor jobs are always run as the option chosen in STEP 9.

If Condor is executing a job and the user returns, Condor will immediately suspend the job, and after five minutes Condor will decide what to do with the partially completed job. There are currently two options for the job.

**The job is killed 5 minutes after your return.** The job is suspended immediately once there is console activity. If the console activity continues, then the job is vacated (killed) after 5 minutes. Since this version does not include check-pointing, the job will be restarted from the beginning at a later time. The job will be placed back into the queue.

**Suspend job, leaving it in memory.** The job is suspended immediately. At a later time, when the console activity has stopped for ten minutes, the execution of Condor job will be resumed (the job will be unsuspended). The drawback to this option is that since the job will remain in memory, it will occupy swap space. In many instances, however, the amount of swap space that the job will occupy is small.

So which one do you choose? Killing a job is less intrusive on the workstation owner than leaving it in memory for a later time. A suspended job left in memory will require swap space, which could possibly be a scarce resource. Leaving a job in memory, however, has the benefit that accumulated run time is not lost for a partially completed job.

**STEP 11: Review entered information.** Check that the entered information is correctly entered. You have the option to return to previous dialog boxes to fix entries.

#### Unattended Installation Procedure using the included Set Up Program

This section details how to run the Condor for Windows installer in an unattended batch mode, i.e. completely from the command prompt without the GUI interface.

The Condor for Windows installer uses the Microsoft Installer (MSI) technology, and can be configured for unattended installs just like any other ordinary MSI installer.

The following is a sample batch file that is used to set all the properties necessary for an unattended install.

```
@echo on
set ARGS=
set ARGS=%ARGS% NEWPOOL=N
set ARGS=%ARGS% POOLNAME=""
set ARGS=%ARGS% RUNJOBS=C
set ARGS=%ARGS% VACATEJOBS=Y
set ARGS=%ARGS% SUBMITJOBS=Y
```

```
set ARGS=%ARGS% CONDOREMAIL="you@yours.com"
set ARGS=%ARGS% SMTPSERVER="smtp.localhost"
set ARGS=%ARGS% HOSTALLOWREAD="*"
set ARGS=%ARGS% HOSTALLOWWRITE="*"
set ARGS=%ARGS% HOSTALLOWADMINISTATOR="$(FULL_HOSTNAME)"
set ARGS=%ARGS% INSTALLDIR="C:\Condor"
set ARGS=%ARGS% INSTALLDIR_NTS="C:\Condor"
set ARGS=%ARGS% POOLHOSTNAME="$(FULL_HOSTNAME)"
set ARGS=%ARGS% ACCOUNTINGDOMAIN="none"
set ARGS=%ARGS% JVMLOCATION="C:\Windows\system32\java.exe"
set ARGS=%ARGS% STARTSERVICE="Y"

msiexec /qb /l* condor-install-log.txt /i condor-6.7.18-winnt50-x86.msi %ARGS%
```

Each property corresponds to answers supplied in the interactive installer as described above. The following is a brief explanation of each property as it applies to unattended installations:

**NEWPOOL** = < **Y** | **N** > determines whether the installer will create a new pool with the target machine as the central manager.

**POOLNAME** sets the name of the pool if a new pool is to be created. Possible values are either the name or the empty string `""`.

**RUNJOBS** = < **N** | **A** | **I** | **C** > determines when Condor will run jobs. This can be set to:

- Never run jobs (N)
- Always run jobs (A)
- Only run jobs when the keyboard and mouse are Idle (I)
- Only run jobs when the keyboard and mouse are idle and the CPU usage is low (C)

**VACATEJOBS** = < **Y** | **N** > determines what Condor should do when it has to stop the execution of a user job. When set to Y, Condor will vacate the job and start it somewhere else if possible. When set to N, Condor will merely suspend the job in memory and wait for the machine to become available again.

**SUBMITJOBS** = < **Y** | **N** > will cause the installer to configure the machine as a submit node when set to Y.

**CONDOREMAIL** sets the e-mail address of the Condor admininistrator. Possible values are an e-mail address or the empty string `""`.

**HOSTALLOWREAD** is a list of host names that are allowed to issue READ commands to Condor daemons. This value should be set in accordance with the `HOSTALLOW_READ` setting in the configuration file, as described in section 3.6.10 on page 282.

**HOSTALLOWWRITE** is a list of host names that are allowed to issue WRITE commands to Condor daemons. This value should be set in accordance with the `HOSTALLOW_WRITE` setting in the configuration file, as described in section 3.6.10 on page 282.

**HOSTALLOWADMINISTRATOR** is a list of host names that are allowed to issue ADMINISTRATOR commands to Condor daemons. This value should be set in accordance with the `HOSTALLOW_ADMINISTRATOR` setting in the configuration file, as described in section 3.6.10 on page 282.

**INSTALLDIR** defines the path to where Condor will be installed.

**INSTALLDIR_NTS** should be set to whatever INSTALLDIR is set to, with the additional restriction that it cannot end in a backslash. The installer will be fixed in an upcoming version of Condor to not require this property.

**POOLHOSTNAME** defines the host name of the pool's central manager.

**ACCOUNTINGDOMAIN** defines the accounting (or UID) domain the target machine will be in.

**JVMLOCATION** defines the path to Java virtual machine on the target machine.

**SMTPSERVER** defines the host name of the SMTP server that the target machine is to use to send e-mail.

**STARTSERVICE =** $<$ **Y** $|$ **N** $>$ determines whether the Condor service will be started after the installation completes.

After defining each of these properties for the MSI installer, the installer can be started with the `msiexec` command. The following command starts the installer in unattended mode, and dumps a journal of the installer's progress to a log file:
`msiexec /qb /l* condor-install-log.txt /i condor-6.7.18-winnt50-x86.msi` [property=value] ...

More information on the features of `msiexec` can be found at Microsoft's website at http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/msiexec.mspx.

#### Manual Installation Condor on Windows

If you are to install Condor on many different machines, you may wish to use some other mechanism to install Condor on additional machines rather than running the Setup program described above on each machine.

WARNING: This is for advanced users only! All others should use the Setup program described above.

Here is a brief overview of how to install Condor manually without using the provided GUI-based setup program:

**The Service** The service that Condor will install is called "Condor". The Startup Type is Automatic. The service should log on as System Account, but **do not enable** "Allow Service to Interact with Desktop". The program that is run is *condor_master.exe*.

The Condor service can be installed and removed using the `sc.exe` tool, which is included in Windows XP and Windows 2003 Server. The tool is also available as part of the Windows 2000 Resource Kit.

Installation can be done as follows:

```
sc create Condor binpath= c:\condor\bin\condor_master.exe
```

To remove the service, use:

```
sc delete Condor
```

**The Registry** Condor uses a few registry entries in its operation. The key that Condor uses is HKEY_LOCAL_MACHINE/Software/Condor. The values that Condor puts in this registry key serve two purposes.

1. The values of CONDOR_CONFIG and RELEASE_DIR are used for Condor to start its service.

   CONDOR_CONFIG should point to the `condor_config` file. In this version of Condor, it **must** reside on the local disk.

   RELEASE_DIR should point to the directory where Condor is installed. This is typically `C:\Condor`, and again, this **must** reside on the local disk.
2. The other purpose is storing the entries from the last installation so that they can be used for the next one.

**The Filesystem** The files that are needed for Condor to operate are identical to the Unix version of Condor, except that executable files end in `.exe`. For example the on Unix one of the files is `condor_master` and on Condor the corresponding file is `condor_master.exe`.

These files currently must reside on the local disk for a variety of reasons. Advanced Windows users might be able to put the files on remote resources. The main concern is twofold. First, the files must be there when the service is started. Second, the files must always be in the same spot (including drive letter), no matter who is logged into the machine.

Note also that when installing manually, you will need to create the directories that Condor will expect to be present given your configuration. This normally is simply a matter of creating the `log`, `spool`, and `execute` directories.

#### Condor Is Installed Under Windows ... Now What?

After the installation of Condor is completed, the Condor service must be started. If you used the GUI-based setup program to install Condor, the Condor service should already be started. If you installed manually, Condor must be started by hand, or you can simply reboot. NOTE: The Condor service will start automatically whenever you reboot your machine.

To start Condor by hand:

1. From the Start menu, choose Settings.

2. From the Settings menu, choose Control Panel.
3. From the Control Panel, choose Services.
4. From Services, choose Condor, and Start.

Or, alternatively you can enter the following command from a command prompt:

```
net start condor
```

Run the Task Manager (Control-Shift-Escape) to check that Condor services are running. The following tasks should be running:

- *condor_master.exe*
- *condor_negotiator.exe*, if this machine is a central manager.
- *condor_collector.exe*, if this machine is a central manager.
- *condor_startd.exe*, if you indicated that this Condor node should start jobs
- *condor_schedd.exe*, if you indicated that this Condor node should submit jobs to the Condor pool.

Also, you should now be able to open up a new cmd (DOS prompt) window, and the Condor bin directory should be in your path, so you can issue the normal Condor commands, such as *condor_q* and *condor_status*.

**Condor is Running Under Windows ... Now What?**

Once Condor services are running, try building and submitting some test jobs. See the `README.TXT` file in the examples directory for details.

### 3.2.7 RPMs

RPMs are available in Version 6.8.6. This packaging method provides for installation and configuration in one easy step. It is currently available for Linux systems only.

The format of the installation command is

```
rpm -i <filename> --prefix=<installation dir>
```

The user provides the path name to the directory used for the installation. The *rpm* program calls *condor_configure* to do portions of the installation. If the condor user is present on the system, the installation script will assume that that is the effective user that Condor should run as (see

section 3.6.12 on page 289). If the condor user is not present, the daemon user will be used. This user will be present on all Linux systems. Note that the user can later be changed by running the *condor_configure* program using the **owner** option, of the format:

```
condor_configure  --owner=<user>
```

After a successful installation, the CONDOR_CONFIG configuration variable must be set to point to

```
<installation dir>/etc/condor_config
```

before starting Condor daemons or invoking Condor tools.

RPM upgrade (**-u** option) does not currently work for Condor Version 6.8.6.

### 3.2.8 Upgrading - Installing a Newer Version of Condor

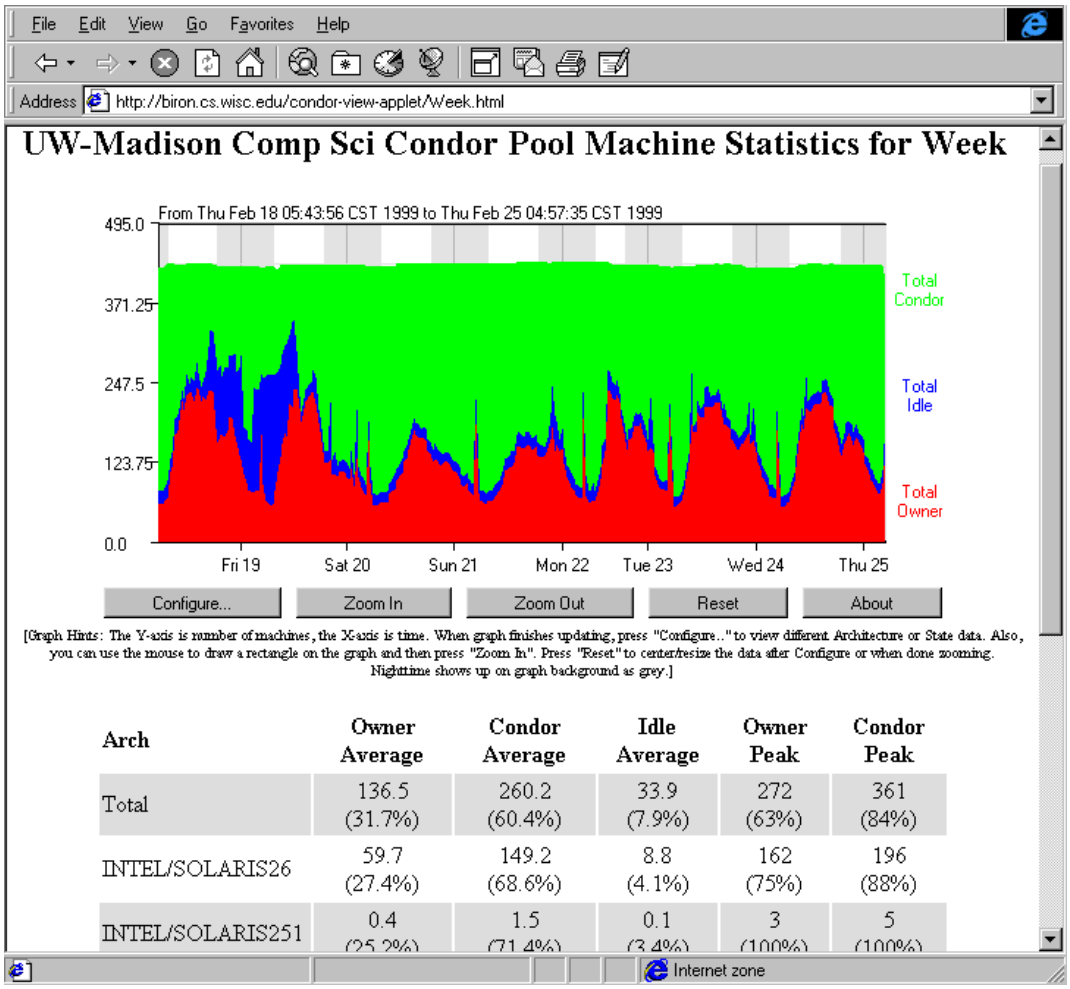
An upgrade changes the running version of Condor from the current installation to a newer version. The safe method to install and start running a newer version of Condor in essence is: shutdown the current installation of Condor, install the newer version, and then restart Condor using the newer version.

To allow for falling back to the current version, place the new version in a separate directory. Copy the existing configuration files, and modify the copy to point to and use the new version. Set the CONDOR_CONFIG environment variable to point to the new copy of the configuration, so the new version of Condor will use the new configuration when restarted.

When upgrading from an earlier version of Condor to a version of 6.8, note that the configuration settings must be modified for security reasons. Specifically, the HOSTALLOW_WRITE configuration variable must be explicitly changed, or no jobs may be submitted, and error messages will be issued by Condor tools.

### 3.2.9 Installing the CondorView Client Contrib Module

The CondorView Client contrib module is used to automatically generate World Wide Web pages to display usage statistics of a Condor pool. Included in the module is a shell script which invokes the *condor_stats* command to retrieve pool usage statistics from the CondorView server, and generate HTML pages from the results. Also included is a Java applet, which graphically visualizes Condor usage information. Users can interact with the applet to customize the visualization and to zoom in to a specific time frame. Figure 3.2 on page 134 is a screen shot of a web page created by CondorView. To get a further feel for what pages generated by CondorView look like, view the statistics for the University of Wisconsin-Madison pool by visiting the URL http://www.cs.wisc.edu/condor and clicking on Condor View.




| Arch | Owner Average | Condor Average | Idle Average | Owner Peak | Condor Peak |
|---|---|---|---|---|---|
| Total | 136.5 (31.7%) | 260.2 (60.4%) | 33.9 (7.9%) | 272 (63%) | 361 (84%) |
| INTEL/SOLARIS26 | 59.7 (27.4%) | 149.2 (68.6%) | 8.8 (4.1%) | 162 (75%) | 196 (88%) |
| INTEL/SOLARIS251 | 0.4 (25.2%) | 1.5 (71.4%) | 0.1 (3.4%) | 3 (100%) | 5 (100%) |

Figure 3.2: Screenshot of CondorView Client

After unpacking and installing the CondorView Client, a script named *make_stats* can be invoked to create HTML pages displaying Condor usage for the past hour, day, week, or month. By using the Unix *cron* facility to periodically execute *make_stats*, Condor pool usage statistics can be kept up to date automatically. This simple model allows the CondorView Client to be easily installed; no Web server CGI interface is needed.

### Step-by-Step Installation of the CondorView Client

1. Make certain that the CondorView Server is configured. Section 3.12.5 describes configuration of the server. The server logs information on disk in order to provide a persistent,

historical database of pool statistics. The CondorView Client makes queries over the network to this database. The *condor_collector* includes this database support. To activate the persistent database logging, add the following entries to the configuration file on the central manager:

```
POOL_HISTORY_DIR = /full/path/to/directory/to/store/historical/data
KEEP_POOL_HISTORY = True
```

2. Create a directory where CondorView is to place the HTML files. This directory should be one published by a web server, so that HTML files which exist in this directory can be accessed using a web browser. This directory is referred to as the `VIEWDIR` directory.

3. Download the 6.1.8 *view_client* contrib module.

4. Unpack or untar this contrib module into the directory `VIEWDIR`. This creates several files and subdirectories.

5. Edit the *make_stats* script. At the beginning of the file are six parameters to customize. The parameters are

   **`ORGNAME`** A brief name that identifies an organization. An example is "Univ of Wisconsin". Do not use any slashes in the name or other special regular-expression characters. Avoid the characters \ ^ and $.

   **`CONDORADMIN`** The e-mail address of the Condor administrator at your site. This e-mail address will appear at the bottom of the web pages.

   **`VIEWDIR`** The full path name (*not* a relative path) to the `VIEWDIR` directory set by installation step 2. It is the directory that contains the *make_stats* script.

   **`STATSDIR`** The full path name of the directory which contains the *condor_stats* binary. The *condor_stats* program is included in the <`release_dir`>`/bin` directory. The value for `STATSDIR` is added to the `PATH` parameter by default.

   **`PATH`** A list of subdirectories, separated by colons, where the *make_stats* script can find the *awk*, *bc*, *sed*, *date*, and *condor_stats* programs. If *perl* is installed, the path should also include the directory where *perl* is installed. The following default works on most systems:

   ```
   PATH=/bin:/usr/bin:$STATSDIR:/usr/local/bin
   ```

6. To create all of the initial HTML files, run

   ```
   ./make_stats setup
   ```

   Open the file `index.html` to verify that things look good.

7. Add the *make_stats* program to *cron*. Running *make_stats* in step 6 created a `cronentries` file. This `cronentries` file is ready to be processed by the Unix *crontab* command. The *crontab* manual page contains details about the *crontab* command and the *cron* daemon.

Look at the `cronentries` file; by default, it will run *make_stats hour* every 15 minutes, *make_stats day* once an hour, *make_stats week* twice per day, and *make_stats month* once per day. These are reasonable defaults. Add these commands to cron on any system that can access the `VIEWDIR` and `STATSDIR` directories, even on a system that does not have Condor installed. The commands do not need to run as root user; in fact, they should probably not run as root. These commands can run as any user that has read/write access to the `VIEWDIR` directory. To add these commands to cron, run

```
crontab cronentries
```

8. Point the web browser at the `VIEWDIR` directory to complete the installation.

### 3.2.10 Dynamic Deployment

Dynamic deployment is a mechanism that allows rapid, automated installation and start up of Condor resources on a given machine. In this way any machine can be added to a Condor pool. The dynamic deployment tool set also provides tools to remove a machine from the pool, without leaving residual effects on the machine such as leftover installations, log files, and working directories.

Installation and start up is provided by *condor_cold_start*. The *condor_cold_start* program determines the operating system and architecture of the target machine, and transfers the correct installation package from an ftp, http, or grid ftp site. After transfer, it installs Condor and creates a local working directory for Condor to run in. As a last step, *condor_cold_start* begins running Condor in a manner which allows for later easy and reliable shut down.

The program that reliably shuts down and uninstalls a previously dynamically installed Condor instance is *condor_cold_stop*. *condor_cold_stop* begins by safely and reliably shutting off the running Condor installation. It ensures that Condor has completely shut down before continuing, and optionally ensures that there are no queued jobs at the site. Next, *condor_cold_stop* removes and optionally archives the Condor working directories, including the `log` directory. These archives can be stored to a mounted file system or to a grid ftp site. As a last step, *condor_cold_stop* uninstalls the Condor executables and libraries. The end result is that the machine resources are left unchanged after a dynamic deployment of Condor leaves.

#### Configuration and Usage

Dynamic deployment is designed for the expert Condor user and administrator. Tool design choices were made for functionality, not ease-of-use.

Like every installation of Condor, a dynamically deployed installation relies on a configuration. To add a target machine to a previously created Condor pool, the global configuration file for that pool is a good starting point. Modifications to that configuration can be made in a separate, local configuration file used in the dynamic deployment. The global configuration file must be placed on an ftp, http, grid ftp, or file server accessible by *condor_cold_start*. The local configuration file is to be on a file system accessible by the target machine. There are some specific configuration

variables that may be set for dynamic deployment. A list of executables and directories which must be present for Condor to start on the target machine may be set with the configuration variables DEPLOYMENT_REQUIRED_EXECS and DEPLOYMENT_REQUIRED_DIRS . If defined and the comma-separated list of executables or directories are not present, then *condor_cold_start* exits with error. Note this does not affect what is installed, only whether start up is successful.

A list of executables and directories which are recommended to be present for Condor to start on the target machine may be set with the configuration variables DEPLOYMENT_RECOMMENDED_EXECS and DEPLOYMENT_RECOMMENDED_DIRS . If defined and the comma-separated lists of executables or directories are not present, then *condor_cold_start* prints a warning message and continues. Here is a portion of the configuration relevant to a dynamic deployment of a Condor submit node:

```
DEPLOYMENT_REQUIRED_EXECS    = MASTER, SCHEDD, PREEN, STARTER, STARTER_PVM,\
                               STARTER_STANDARD, SHADOW, SHADOW_PVM, \
                               SHADOW_STANDARD, GRIDMANAGER, GAHP, CONDOR_GAHP
DEPLOYMENT_REQUIRED_DIRS     = SPOOL, LOG, EXECUTE
DEPLOYMENT_RECOMMENDED_EXECS = CREDD
DEPLOYMENT_RECOMMENDED_DIRS  = LIB, LIBEXEC
```

Additionally, the user must specify which Condor services will be started. This is done through the DAEMON_LIST configuration variable. Another excerpt from a dynamic submit node deployment configuration:

```
DAEMON_LIST  = MASTER, SCHEDD
```

Finally, the location of the dynamically installed Condor executables is tricky to set, since the location is unknown before installation. Therefore, the variable DEPLOYMENT_RELEASE_DIR is defined in the environment. It corresponds to the location of the dynamic Condor installation. If, as is often the case, the configuration file specifies the location of Condor executables in relation to the RELEASE_DIR variable, the configuration can be made dynamically deployable by setting RELEASE_DIR to DEPLOYMENT_RELEASE_DIR as

```
RELEASE_DIR = $(DEPLOYMENT_RELEASE_DIR)
```

In addition to setting up the configuration, the user must also determine where the installation package will reside. The installation package can be in either tar or gzipped tar form, and may reside on a ftp, http, grid ftp, or file server. Create this installation package by tar'ing up the binaries and libraries needed, and place them on the appropriate server. The binaries can be tar'ed in a flat structure or within bin and sbin. Here is a list of files to give an example structure for a dynamic deployment of the *condor_schedd* daemon.

```
% tar tfz latest-i686-Linux-2.4.21-37.ELsmp.tar.gz
bin/
bin/condor_config_val
bin/condor_q
```

```
sbin/
sbin/condor_preen
sbin/condor_shadow.std
sbin/condor_starter.std
sbin/condor_schedd
sbin/condor_master
sbin/condor_starter.pvm
sbin/condor_shadow.pvm
sbin/condor_gridmanager
sbin/gt3_gahp
sbin/gt4_gahp
sbin/gahp_server
sbin/condor_starter
sbin/condor_shadow
sbin/condor_c-gahp
sbin/condor_off
```

## 3.3 Configuration

This section describes how to configure all parts of the Condor system. General information about the configuration files and their syntax is followed by a description of settings that affect all Condor daemons and tools. The settings that control the policy under which Condor will start, suspend, resume, vacate or kill jobs are described in section 3.5 on Startd Policy Configuration.

### 3.3.1 Introduction to Configuration Files

The Condor configuration files are used to customize how Condor operates at a given site. The basic configuration as shipped with Condor works well for most sites.

Each Condor program will, as part of its initialization process, configure itself by calling a library routine which parses the various configuration files that might be used including pool-wide, platform-specific, machine-specific, and root-owned configuration files. Environment variables may also contribute to the configuration.

The result of configuration is a list of key/value pairs. Each key is a configuration variable name, and each value is a string literal that may utilize macro substitution (as defined below). Note that the string literal value portion of a pair is not an expression, and therefore it is not evaluated. Those configuration variables that express the policy for starting and stopping of jobs appear as expressions in the configuration file. However, these expressions (for configuration) are string literals. At appropriate times, Condor daemons and tools use these strings as expressions, parsing them in order to do evaluation.

#### Ordered Evaluation to Set the Configuration

Multiple files, as well as a program's environment variables determine the configuration. The order in which attributes are defined is important, as later definitions override existing definitions. The

order in which the (multiple) configuration files are parsed is designed to ensure the security of the system. Attributes which must be set a specific way must appear in the last file to be parsed. This prevents both the naive and the malicious Condor user from subverting the system through its configuration. The order in which items are parsed is

1. global configuration file
2. local configuration file
3. global root-owned configuration file
4. local root-owned configuration file
5. specific environment variables prefixed with `_CONDOR_`

The locations for these files are as given in section 3.2.2 on page 112.

Some Condor tools utilize environment variables to set their configuration. These tools search for specifically-named environment variables. The variables are prefixed by the string `_CONDOR_` or `_condor_`. The tools strip off the prefix, and utilize what remains as configuration. As the use of environment variables is the last within the ordered evaluation, the environment variable definition is used. The security of the system is not compromised, as only specific variables are considered for definition in this manner, not any environment variables with the `_CONDOR_` prefix.

**Configuration File Macros**

Macro definitions are of the form:

```
<macro_name> = <macro_definition>
```

NOTE: There must be white space between the macro name, the "=" sign, and the macro definition.

Macro invocations are of the form:

```
$(macro_name)
```

Macro definitions may contain references to other macros, even ones that are not yet defined (as long as they are eventually defined in the configuration files). All macro expansion is done after all configuration files have been parsed (with the exception of macros that reference themselves, described below).

```
A = xxx
C = $(A)
```

is a legal set of macro definitions, and the resulting value of C is xxx. Note that C is actually bound to $(A), not its value.

As a further example,

```
A = xxx
C = $(A)
A = yyy
```

is also a legal set of macro definitions, and the resulting value of C is yyy.

A macro may be incrementally defined by invoking itself in its definition. For example,

```
A = xxx
B = $(A)
A = $(A)yyy
A = $(A)zzz
```

is a legal set of macro definitions, and the resulting value of A is xxxyyyzzz. Note that invocations of a macro in its own definition are immediately expanded. $(A) is immediately expanded in line 3 of the example. If it were not, then the definition would be impossible to evaluate.

Recursively defined macros such as

```
A = $(B)
B = $(A)
```

are not allowed. They create definitions that Condor refuses to parse.

NOTE: Macros should not be incrementally defined in the LOCAL_ROOT_CONFIG_FILE for security reasons.

All entries in a configuration file must have an operator, which will be an equals sign (=). Identifiers are alphanumerics combined with the underscore character, optionally with a subsystem name and a period as a prefix. As a special case, a line without an operator that begins with a left square bracket will be ignored. The following two-line example treats the first line as a comment, and correctly handles the second line.

```
[Condor Settings]
my_classad = [ foo=bar ]
```

To simplify pool administration, any configuration variable name may be prefixed by a subsystem (see the $(SUBSYSTEM) macro in section 3.3.1 for the list of subsystems) and the period (.) character. For configuration variables defined this way, the value is applied to the specific subsystem. For example, the ports that Condor may use can be restricted to a range using the HIGHPORT and LOWPORT configuration variables. If the range of intended ports is different for specific daemons, this syntax may be used.

```
MASTER.LOWPORT   = 20000
MASTER.HIGHPORT  = 20100
NEGOTIATOR.LOWPORT   =  22000
NEGOTIATOR.HIGHPORT  =  22100
```

Note that all configuration variables may utilize this syntax, but nonsense configuration variables may result. For example, it makes no sense to define

```
NEGOTIATOR.MASTER_UPDATE_INTERVAL = 60
```

since the *condor_negotiator* daemon does not use the MASTER_UPDATE_INTERVAL variable.

It makes little sense to do so, but Condor will configure correctly with a definition such as

```
MASTER.MASTER_UPDATE_INTERVAL = 60
```

The *condor_master* uses this configuration variable, and the prefix of MASTER. causes this configuration to be specific to the *condor_master* daemon.

#### Comments and Line Continuations

A Condor configuration file may contain comments and line continuations. A comment is any line beginning with a "#" character. A continuation is any entry that continues across multiples lines. Line continuation is accomplished by placing the "\" character at the end of any line to be continued onto another. Valid examples of line continuation are

```
START = (KeyboardIdle > 15 * $(MINUTE)) && \
((LoadAvg - CondorLoadAvg) <= 0.3)
```

and

```
ADMIN_MACHINES = condor.cs.wisc.edu, raven.cs.wisc.edu, \
stork.cs.wisc.edu, ostrich.cs.wisc.edu, \
bigbird.cs.wisc.edu
HOSTALLOW_ADMIN = $(ADMIN_MACHINES)
```

Note that a line continuation character may currently be used within a comment, so the following example does *not* set the configuration variable FOO:

```
# This comment includes the following line, so FOO is NOT set \
FOO = BAR
```

It is a poor idea to use this functionality, as it is likely to stop working in future Condor releases.

### Executing a Program to Produce Configuration Macros

Instead of reading from a file, Condor may run a program to obtain configuration macros. The vertical bar character (| ) as the last character defining a file name provides the syntax necessary to tell Condor to run a program. This syntax may only be used in the definition of the CONDOR_CONFIG environment variable, the LOCAL_CONFIG_FILE configuration variable, or the LOCAL_ROOT_CONFIG_FILE configuration variable.

The command line for the program is formed by the characters preceding the vertical bar character. The standard output of the program is parsed as a configuration file would be.

An example:

```
LOCAL_CONFIG_FILE = /bin/make_the_config|
```

Program */bin/make_the_config* is executed, and its output is the set of configuration macros.

Note that either a program is executed to generate the configuration macros or the configuration is read from one or more files. The syntax uses space characters to separate command line elements, if an executed program produces the configuration macros. Space characters would otherwise separate the list of files. This syntax does not permit distinguishing one from the other, so only one may be specified.

### Pre-Defined Macros

Condor provides pre-defined macros that help configure Condor. Pre-defined macros are listed as $(macro_name).

This first set are entries whose values are determined at run time and cannot be overwritten. These are inserted automatically by the library routine which parses the configuration files.

**$(FULL_HOSTNAME)** The fully qualified hostname of the local machine (hostname plus domain name).

**$(HOSTNAME)** The hostname of the local machine (no domain name).

**$(IP_ADDRESS)** The ASCII string version of the local machine's IP address.

**$(TILDE)** The full path to the home directory of the Unix user condor, if such a user exists on the local machine.

**$(SUBSYSTEM)** The subsystem name of the daemon or tool that is evaluating the macro. This is a unique string which identifies a given daemon within the Condor system. The possible subsystem names are:

- STARTD
- SCHEDD

- MASTER
- COLLECTOR
- NEGOTIATOR
- KBDD
- SHADOW
- STARTER
- CKPT_SERVER
- SUBMIT
- GRIDMANAGER
- TOOL
- HAD
- REPLICATION
- QUILL

This second set of macros are entries whose default values are determined automatically at run-time but which can be overwritten.

**$(ARCH)** Defines the string used to identify the architecture of the local machine to Condor. The *condor_startd* will advertise itself with this attribute so that users can submit binaries compiled for a given platform and force them to run on the correct machines. *condor_submit* will append a requirement to the job ClassAd that it must run on the same ARCH and OPSYS of the machine where it was submitted, unless the user specifies ARCH and/or OPSYS explicitly in their submit file. See the the *condor_submit* manual page on page 716 for details.

**$(OPSYS)** Defines the string used to identify the operating system of the local machine to Condor. If it is not defined in the configuration file, Condor will automatically insert the operating system of this machine as determined by *uname*.

**$(UNAME_ARCH)** The architecture as reported by *uname*(2)'s machine field. Always the same as ARCH on Windows.

**$(UNAME_OPSYS)** The operating system as reported by *uname*(2)'s sysname field. Always the same as OPSYS on Windows.

**$(PID)** The process ID for the daemon or tool.

**$(PPID)** The process ID of the parent process for the daemon or tool.

**$(USERNAME)** The user name of the UID of the daemon or tool. It is useful for setting GRIDMANAGER_LOG, as that needs to be done on a per-user basis. For daemons started as root, but running under another UID (typically the user condor), this will be the other UID.

**$(FILESYSTEM_DOMAIN)** Defaults to the fully qualified hostname of the machine it is evaluated on. See section 3.3.7, Shared File System Configuration File Entries for the full description of its use and under what conditions you would want to change it.

**$(UID_DOMAIN)** Defaults to the fully qualified hostname of the machine it is evaluated on. See section 3.3.7 for the full description of this configuration variable.

Since $(ARCH) and $(OPSYS) will automatically be set to the correct values, we recommend that you do not overwrite them. Only do so if you know what you are doing.

### 3.3.2 The Special Configuration Macros $ENV(), $RANDOM_CHOICE(), and $RANDOM_INTEGER()

References to the Condor process's environment are allowed in the configuration files. Environment references use the ENV macro and are of the form:

```
$ENV(environment_variable_name)
```

For example,

```
A = $ENV(HOME)
```

binds A to the value of the HOME environment variable. Environment references are not currently used in standard Condor configurations. However, they can sometimes be useful in custom configurations.

This same syntax is used in the RANDOM_CHOICE() macro to allow a random choice of a parameter within a configuration file. These references are of the form:

```
$RANDOM_CHOICE(list of parameters)
```

This allows a random choice within the parameter list to be made at configuration time. Of the list of parameters, one is chosen when encountered during configuration. For example, if one of the integers 0-8 (inclusive) should be randomly chosen, the macro usage is

```
$RANDOM_CHOICE(0,1,2,3,4,5,6,7,8)
```

The RANDOM_INTEGER() macro is similar to the RANDOM_CHOICE() macro, and is used to select a random integer within a configuration file. References are of the form:

```
$RANDOM_INTEGER(min, max [, step])
```

A random integer within the range min and max, inclusive, is selected at configuration time. The optional step parameter controls the stride within the range, and it defaults to the value 1. For example, to randomly chose an even integer in the range 0-8 (inclusive), the macro usage is

```
$RANDOM_INTEGER(0, 8, 2)
```

See section 7.2 on page 469 for an actual use of this specialized macro.

### 3.3.3 Condor-wide Configuration File Entries

This section describes settings which affect all parts of the Condor system. Other system-wide settings can be found in section 3.3.6 on "Network-Related Configuration File Entries", and section 3.3.7 on "Shared File System Configuration File Entries".

**CONDOR_HOST** This macro may be used to define the `$(NEGOTIATOR_HOST)` and is used to define the `$(COLLECTOR_HOST)` macro. Normally the *condor_collector* and *condor_negotiator* would run on the same machine. If for some reason they were not run on the same machine, `$(CONDOR_HOST)` would not be needed. Some of the host-based security macros use `$(CONDOR_HOST)` by default. See section 3.6.10, on Setting up IP/host-based security in Condor for details.

**COLLECTOR_HOST** The hostname of the machine where the *condor_collector* is running for your pool. Normally, it is defined relative to the `$(CONDOR_HOST)` macro. There is no default value for this macro; `COLLECTOR_HOST` must be defined for the pool to work properly.

In addition to defining the hostname, this setting can optionally be used to specify the network port of the *condor_collector*. The port is separated from the hostname by a colon (':'). For example,

```
COLLECTOR_HOST = $(CONDOR_HOST):1234
```

If no port is specified, the default port of 9618 is used. Using the default port is recommended for most sites. It is only changed if there is a conflict with another service listening on the same network port. For more information about specifying a non-standard port for the *condor_collector* daemon, see section 3.7.1 on page 295.

**NEGOTIATOR_HOST** This configuration variable is no longer used. The port where the *condor_negotiator* is listening is normally dynamically allocated since version 6.7.4.

For pools running 6.7.3 and older versions: The host name of the machine where the *condor_negotiator* is running for the pool. Normally, it is defined relative to the `$(CONDOR_HOST)` macro. There is no default value for this macro; `NEGOTIATOR_HOST` must be defined for the pool to work properly. This variable may also be used to optionally define a network port for the *condor_negotiator* daemon, as explained for the `COLLECTOR_HOST` variable.

**CONDOR_VIEW_HOST** The hostname of the machine where the CondorView server is running. This service is optional, and requires additional configuration if you want to enable it. There is no default value for `CONDOR_VIEW_HOST`. If `CONDOR_VIEW_HOST` is not defined, no CondorView server is used. See section 3.12.5 on page 344 for more details.

**SCHEDD_HOST** The hostname of the machine where the *condor_schedd* is running for your pool. This is the host that queues submitted jobs. Note that, in most condor installations, there is a *condor_schedd* running on each host from which jobs are submitted. The default value of `SCHEDD_HOST` is the current host. For most pools, this macro is not defined.

**RELEASE_DIR** The full path to the Condor release directory, which holds the `bin`, `etc`, `lib`, and `sbin` directories. Other macros are defined relative to this one. There is no default value for `RELEASE_DIR` .

**BIN** This directory points to the Condor directory where user-level programs are installed. It is usually defined relative to the `$(RELEASE_DIR)` macro. There is no default value for `BIN` .

**LIB** This directory points to the Condor directory where libraries used to link jobs for Condor's standard universe are stored. The *condor_compile* program uses this macro to find these libraries, so it must be defined for *condor_compile* to function. `$(LIB)` is usually defined relative to the `$(RELEASE_DIR)` macro, and has no default value.

**LIBEXEC** This directory points to the Condor directory where support commands that Condor needs will be placed. Do not add this directory to a user or system-wide path.

**INCLUDE** This directory points to the Condor directory where header files reside. `$(INCLUDE)` would usually be defined relative to the `$(RELEASE_DIR)` configuration macro. There is no default value, but if defined, it can make inclusion of necessary header files for compilation of programs (such as those programs that use `libcondorapi.a`) easier through the use of *condor_config_val*.

**SBIN** This directory points to the Condor directory where Condor's system binaries (such as the binaries for the Condor daemons) and administrative tools are installed. Whatever directory `$(SBIN)` points to ought to be in the `PATH` of users acting as Condor administrators. `SBIN` has no default value.

**LOCAL_DIR** The location of the local Condor directory on each machine in your pool. One common option is to use the condor user's home directory which may be specified with `$(TILDE)`. There is no default value for `LOCAL_DIR` . For example:

```
LOCAL_DIR = $(tilde)
```

On machines with a shared file system, where either the `$(TILDE)` directory or another directory you want to use is shared among all machines in your pool, you might use the `$(HOSTNAME)` macro and have a directory with many subdirectories, one for each machine in your pool, each named by host names. For example:

```
LOCAL_DIR = $(tilde)/hosts/$(hostname)
```

or:

```
LOCAL_DIR = $(release_dir)/hosts/$(hostname)
```

**LOG** Used to specify the directory where each Condor daemon writes its log files. The names of the log files themselves are defined with other macros, which use the `$(LOG)` macro by default.

The log directory also acts as the current working directory of the Condor daemons as the run, so if one of them should produce a core file for any reason, it would be placed in the directory defined by this macro. LOG is required to be defined. Normally, $(LOG) is defined in terms of $(LOCAL_DIR).

**SPOOL** The spool directory is where certain files used by the *condor_schedd* are stored, such as the job queue file and the initial executables of any jobs that have been submitted. In addition, for systems not using a checkpoint server, all the checkpoint files from jobs that have been submitted from a given machine will be store in that machine's spool directory. Therefore, you will want to ensure that the spool directory is located on a partition with enough disk space. If a given machine is only set up to execute Condor jobs and not submit them, it would not need a spool directory (or this macro defined). There is no default value for SPOOL , and the *condor_schedd* will not function without it SPOOL defined. Normally, $(SPOOL) is defined in terms of $(LOCAL_DIR).

**EXECUTE** This directory acts as the current working directory of any Condor job that is executing on the local machine. If a given machine is only set up to only submit jobs and not execute them, it would not need an execute directory (or this macro defined). There is no default value for EXECUTE, and the *condor_startd* will not function if EXECUTE is not defined. Normally, $(EXECUTE) is defined in terms of $(LOCAL_DIR).

**LOCAL_CONFIG_FILE** Identifies the location of the local, machine-specific configuration file for each machine in the pool. The two most common choices would be putting this file in the $(LOCAL_DIR), or putting all local configuration files for the pool in a shared directory, each one named by host name. For example,

```
LOCAL_CONFIG_FILE = $(LOCAL_DIR)/condor_config.local
```

or,

```
LOCAL_CONFIG_FILE = $(release_dir)/etc/$(hostname).local
```

or, not using the release directory

```
LOCAL_CONFIG_FILE = /full/path/to/configs/$(hostname).local
```

The value of $(LOCAL_CONFIG_FILE) is treated as a list of files, not a single file. The items in the list are delimited by either commas or space characters. This allows the specification of multiple files as the local configuration file, each one processed in the order given (with parameters set in later files overriding values from previous files). This allows the use of one global configuration file for multiple platforms in the pool, defines a platform-specific configuration file for each platform, and uses a local configuration file for each machine. If the list of files is changed in one of the later read files, the new list replaces the old list, but any files that have already been processed remain processed, and are removed from the new

list if they are present to prevent cycles. See section 3.3.1 on page 142 for directions on using a program to generate the configuration macros that would otherwise reside in one or more files as described here. If LOCAL_CONFIG_FILE is not defined, no local configuration files are processed. For more information on this, see section 3.12.2 about Configuring Condor for Multiple Platforms on page 340.

**REQUIRE_LOCAL_CONFIG_FILE** Beginning in Condor 6.5.5, it is permissible for files listed in LOCAL_CONFIG_FILE to be missing. This is most useful for sites that have large numbers of machines in the pool, and a local configuration file that uses the $(HOSTNAME) macro in its definition. Instead of having an empty file for every host in the pool, files can simply be omitted. The default value is True, causing Condor to exit with an error, if a file listed as the local configuration file cannot be read. A value of False allows local configuration files to be missing.

**LOCAL_CONFIG_DIR** Beginning in Condor 6.7.18, a directory may be used as a container for local configuration files. The files found in the directory are sorted into lexicographical order, and then each file is treated as though it was listed in LOCAL_CONFIG_FILE. LOCAL_CONFIG_DIR is processed before any files listed in LOCAL_CONFIG_FILE, and is checked again after processing the LOCAL_CONFIG_FILE list. It is a list of directories, and each directory is processed in the order it appears in the list. The process is not recursive, so any directories found inside the directory being processed are ignored.

**LOCAL_ROOT_CONFIG_FILE** A comma or space separated list of path and file names specifying the local, root configuration files.

**CONDOR_IDS** The User ID (UID) and Group ID (GID) pair that the Condor daemons should run as, if the daemons are spawned as root. This value can also be specified in the CONDOR_IDS environment variable. If the Condor daemons are not started as root, then neither this CONDOR_IDS configuration macro nor the CONDOR_IDS environment variable are used. The value is given by two integers, separated by a period. For example, CONDOR_IDS = 1234.1234. If this pair is not specified in either the configuration file or in the environment, and the Condor daemons are spawned as root, then Condor will search for a condor user on the system, and run as that user's UID and GID. See section 3.6.12 on UIDs in Condor for more details.

**CONDOR_ADMIN** The email address that Condor will send mail to if something goes wrong in your pool. For example, if a daemon crashes, the *condor_master* can send an *obituary* to this address with the last few lines of that daemon's log file and a brief message that describes what signal or exit status that daemon exited with. There is no default value for CONDOR_ADMIN .

**CONDOR_SUPPORT_EMAIL** The email address to be included at the bottom of all email Condor sends out under the label "Email address of the local Condor administrator:". This is the address where Condor users at your site should send their questions about Condor and get technical support. If this setting is not defined, Condor will use the address specified in CONDOR_ADMIN (described above).

**MAIL** The full path to a mail sending program that uses **-s** to specify a subject for the message. On all platforms, the default shipped with Condor should work. Only if you installed things

in a non-standard location on your system would you need to change this setting. There is no default value for MAIL, and the *condor_schedd* will not function unless MAIL is defined.

**RESERVED_SWAP** Determines how much swap space you want to reserve for your own machine. Condor will not start up more *condor_shadow* processes if the amount of free swap space on your machine falls below this level. RESERVED_SWAP is specified in megabytes. The default value of RESERVED_SWAP is 5 megabytes.

**RESERVED_DISK** Determines how much disk space you want to reserve for your own machine. When Condor is reporting the amount of free disk space in a given partition on your machine, it will always subtract this amount. An example is the *condor_startd*, which advertises the amount of free space in the $(EXECUTE) directory. The default value of RESERVED_DISK is zero.

**LOCK** Condor needs to create lock files to synchronize access to various log files. Because of problems with network file systems and file locking over the years, we *highly* recommend that you put these lock files on a local partition on each machine. If you do not have your $(LOCAL_DIR) on a local partition, be sure to change this entry.

Whatever user or group Condor is running as needs to have write access to this directory. If you are not running as root, this is whatever user you started up the *condor_master* as. If you are running as root, and there is a condor account, it is most likely condor. Otherwise, it is whatever you set in the CONDOR_IDS environment variable, or whatever you define in the CONDOR_IDS setting in the Condor config files. See section 3.6.12 on UIDs in Condor for details.

If no value for LOCK is provided, the value of LOG is used.

**HISTORY** Defines the location of the Condor history file, which stores information about all Condor jobs that have completed on a given machine. This macro is used by both the *condor_schedd* which appends the information and *condor_history*, the user-level program used to view the history file. This configuration macro is given the default value of $(SPOOL)/history in the default configuration. If not defined, no history file is kept.

**ENABLE_HISTORY_ROTATION** If this is defined to be true, then the history file will be rotated. If it is false, then it will not be rotated, and it will grow indefinitely, to the limits allowed by the operating system. If this is not defined, it is assumed to be true. The rotated files will be stored in the same directory as the history file.

**MAX_HISTORY_LOG** Defines the maximum size for the history file, in bytes. It defaults to 20MB. This parameter is only used if history file rotation is enabled.

**MAX_HISTORY_ROTATIONS** When history file rotation is turned on, this controls how many backup files there are. It default to 2, which means that there may be up to three history files (two backups, plus the history file that is being currently written to). When the history file is rotated, and this rotation would cause the number of backups to be too large, the oldest file is removed.

**MAX_JOB_QUEUE_LOG_ROTATIONS** The schedd periodically rotates the job queue database file in order to save disk space. This option controls how many rotated files are saved. It defaults

to 1, which means there may be up to two history files (the previous one, which was rotated out of use, and the current one that is being written to). When the job queue file is rotated, and this rotation would cause the number of backups to be larger the the maximum specified, the oldest file is removed. The primary reason to save one or more rotated job queue files is if you are using Quill, and you want to ensure that Quill keeps an accurate history of all events logged in the job queue file. Quill keeps track of where it last left off when reading logged events, so when the file is rotated, Quill will resume reading from where it last left off, provided that the rotated file still exists. If Quill finds that it needs to read events from a rotated file that has been deleted, it will be forced to skip the missing events and resume reading in the next chronological job queue file that can be found. Such an event should not lead to an inconsistency in Quill's view of the current queue contents, but it would create a inconsistency in Quill's record of the history of the job queue.

**DEFAULT_DOMAIN_NAME** If you do not use a fully qualified name in file `/etc/hosts` (or NIS, etc.) for either your official hostname or as an alias, Condor would not normally be able to use fully qualified names in places that it wants to. You can set this macro to the domain to be appended to your hostname, if changing your host information is not a good option. This macro must be set in the global configuration file (not the `$(LOCAL_CONFIG_FILE)`. The reason for this is that the special `$(FULL_HOSTNAME)` macro is used by the configuration file code in Condor needs to know the full hostname. So, for `$(DEFAULT_DOMAIN_NAME)` to take effect, Condor must already have read in its value. However, Condor must set the `$(FULL_HOSTNAME)` special macro since you might use that to define where your local configuration file is. After reading the global configuration file, Condor figures out the right values for `$(HOSTNAME)` and `$(FULL_HOSTNAME)` and inserts them into its configuration table.

**NO_DNS** A boolean value that defaults to `False`. When `True`, Condor constructs host names using the host's IP address together with the value defined for `DEFAULT_DOMAIN_NAME`.

**CM_IP_ADDR** If neither `COLLECTOR_HOST` nor `COLLECTOR_IP_ADDR` macros are defined, then this macro will be used to determine the IP address of the central manager (collector daemon). This macro is defined by an IP address.

**EMAIL_DOMAIN** By default, if a user does not specify `notify_user` in the submit description file, any email Condor sends about that job will go to "username@UID_DOMAIN". If your machines all share a common UID domain (so that you would set `UID_DOMAIN` to be the same across all machines in your pool), but email to user@UID_DOMAIN is not the right place for Condor to send email for your site, you can define the default domain to use for email. A common example would be to set `EMAIL_DOMAIN` to the fully qualified hostname of each machine in your pool, so users submitting jobs from a specific machine would get email sent to user@machine.your.domain, instead of user@your.domain. You would do this by setting `EMAIL_DOMAIN` to `$(FULL_HOSTNAME)`. In general, you should leave this setting commented out unless two things are true: 1) `UID_DOMAIN` is set to your domain, not `$(FULL_HOSTNAME)`, and 2) email to user@UID_DOMAIN will not work.

**CREATE_CORE_FILES** Defines whether or not Condor daemons are to create a core file in the `LOG` directory if something really bad happens. It is used to set the resource limit for the size of a core file. If not defined, it leaves in place whatever limit was in effect when the

Condor daemons (normally the *condor_master*) were started. This allows Condor to inherit the default system core file generation behavior at startup. For Unix operating systems, this behavior can be inherited from the parent shell, or specified in a shell script that starts Condor. If this parameter is set and `True`, the limit is increased to the maximum. If it is set to `False`, the limit is set at 0 (which means that no core files are created). Core files greatly help the Condor developers debug any problems you might be having. By using the parameter, you do not have to worry about tracking down where in your boot scripts you need to set the core limit before starting Condor. You set the parameter to whatever behavior you want Condor to enforce. This parameter defaults to undefined to allow the initial operating system default value to take precedence, and is commented out in the default configuration file.

**Q_QUERY_TIMEOUT** Defines the timeout (in seconds) that *condor_q* uses when trying to connect to the *condor_schedd*. Defaults to 20 seconds.

**PASSWD_CACHE_REFRESH** Condor can cause NIS servers to become overwhelmed by queries for uid and group information in large pools. In order to avoid this problem, Condor caches UID and group information internally. This setting allows pool administrators to specify (in seconds) how long Condor should wait until refreshes a cache entry. The default is set to 300 seconds, or 5 minutes. This means that if a pool administrator updates the user or group database (for example, `/etc/passwd` or `/etc/group`), it can take up to 5 minutes before Condor will have the updated information. This caching feature can be disabled by setting the refresh interval to 0. In addition, the cache can also be flushed explicitly by running the command

```
condor_reconfig -full
```

This configuration variable has no effect on Windows.

**SYSAPI_GET_LOADAVG** If set to False, then Condor will not attempt to compute the load average on the system, and instead will always report the system load average to be 0.0. Defaults to True.

**NETWORK_MAX_PENDING_CONNECTS** This specifies a limit to the maximum number of simultaneous network connection attempts. This is primarily relevant to *condor_schedd*, which may try to connect to large numbers of startds when claiming them. The negotiator may also connect to large numbers of startds when initiating security sessions used for sending MATCH messages. On Unix, the default for this parameter is eighty percent of the process file descriptor limit. On windows, the default is 1600.

### 3.3.4 Daemon Logging Configuration File Entries

These entries control how and where the Condor daemons write their log files. Each of the entries in this section represents multiple macros. There is one for each subsystem (listed in section 3.3.1). The macro name for each substitutes <SUBSYS> with the name of the subsystem corresponding to the daemon.

**<SUBSYS>_LOG** The name of the log file for a given subsystem. For example, $(STARTD_LOG) gives the location of the log file for *condor_startd*.

The actual names of the files are also used in the $(VALID_LOG_FILES) entry used by *condor_preen*. A change to one of the file names with this setting requires a change to the $(VALID_LOG_FILES) entry as well, or *condor_preen* will delete your newly named log files.

**MAX_<SUBSYS>_LOG** Controls the maximum length in bytes to which a log will be allowed to grow. Each log file will grow to the specified length, then be saved to a file with the suffix .old. The .old files are overwritten each time the log is saved, thus the maximum space devoted to logging for any one program will be twice the maximum length of its log file. A value of 0 specifies that the file may grow without bounds. The default is 1 Mbyte.

**TRUNC_<SUBSYS>_LOG_ON_OPEN** If this macro is defined and set to True, the affected log will be truncated and started from an empty file with each invocation of the program. Otherwise, new invocations of the program will append to the previous log file. By default this setting is False for all daemons.

**<SUBSYS>_LOCK** This macro specifies the lock file used to synchronize append operations to the log file for this subsystem. It must be a separate file from the $(<SUBSYS>_LOG) file, since the $(<SUBSYS>_LOG) file may be rotated and you want to be able to synchronize access across log file rotations. A lock file is only required for log files which are accessed by more than one process. Currently, this includes only the SHADOW subsystem. This macro is defined relative to the $(LOCK) macro. If, for some strange reason, you decide to change this setting, be sure to change the $(VALID_LOG_FILES) entry that *condor_preen* uses as well.

**FILE_LOCK_VIA_MUTEX** This macro setting only works on Win32 – it is ignored on Unix. If set to be True, then log locking is implemented via a kernel mutex instead of via file locking. On Win32, mutex access is FIFO, while obtaining a file lock is non-deterministic. Thus setting to True fixes problems on Win32 where processes (usually shadows) could starve waiting for a lock on a log file. Defaults to True on Win32, and is always False on Unix.

**ENABLE_USERLOG_LOCKING** When True (the default value), a user's job log (as specified in a submit description file) will be locked before being written to. If False, Condor will not lock the file before writing.

**TOUCH_LOG_INTERVAL** The time interval in seconds between when daemons touch their log files. The change in last modification time for the log file is useful when a daemon restarts after failure or shut down. The last modification date is printed, and it provides an upper bound on the length of time that the daemon was not running. Defaults to 60 seconds.

**<SUBSYS>_DEBUG** All of the Condor daemons can produce different levels of output depending on how much information is desired. The various levels of verbosity for a given daemon are determined by this macro. All daemons have the default level D_ALWAYS, and log messages for that level will be printed to the daemon's log, regardless of this macro's setting. Settings are a comma- or space-separated list of the following values:

**D_ALL** This flag turns on *all* debugging output by enabling all of the debug levels at once. There is no need to list any other debug levels in addition to D_ALL; doing so would be

redundant. Be warned: this will generate about a *HUGE* amount of output. To obtain a higher level of output than the default, consider using D_FULLDEBUG before using this option.

**D_FULLDEBUG** This level provides verbose output of a general nature into the log files. Frequent log messages for very specific debugging purposes would be excluded. In those cases, the messages would be viewed by having that another flag and D_FULLDEBUG both listed in the configuration file.

**D_DAEMONCORE** Provides log file entries specific to DaemonCore, such as timers the daemons have set and the commands that are registered. If both D_FULLDEBUG and D_DAEMONCORE are set, expect *very* verbose output.

**D_PRIV** This flag provides log messages about the *privilege state* switching that the daemons do. See section 3.6.12 on UIDs in Condor for details.

**D_COMMAND** With this flag set, any daemon that uses DaemonCore will print out a log message whenever a command comes in. The name and integer of the command, whether the command was sent via UDP or TCP, and where the command was sent from are all logged. Because the messages about the command used by *condor_kbdd* to communicate with the *condor_startd* whenever there is activity on the X server, and the command used for keep-alives are both only printed with D_FULLDEBUG enabled, it is best if this setting is used for all daemons.

**D_LOAD** The *condor_startd* keeps track of the load average on the machine where it is running. Both the general system load average, and the load average being generated by Condor's activity there are determined. With this flag set, the *condor_startd* will log a message with the current state of both of these load averages whenever it computes them. This flag only affects the *condor_startd*.

**D_KEYBOARD** With this flag set, the *condor_startd* will print out a log message with the current values for remote and local keyboard idle time. This flag affects only the *condor_startd*.

**D_JOB** When this flag is set, the *condor_startd* will send to its log file the contents of any job ClassAd that the *condor_schedd* sends to claim the *condor_startd* for its use. This flag affects only the *condor_startd*.

**D_MACHINE** When this flag is set, the *condor_startd* will send to its log file the contents of its resource ClassAd when the *condor_schedd* tries to claim the *condor_startd* for its use. This flag affects only the *condor_startd*.

**D_SYSCALLS** This flag is used to make the *condor_shadow* log remote syscall requests and return values. This can help track down problems a user is having with a particular job by providing the system calls the job is performing. If any are failing, the reason for the failure is given. The *condor_schedd* also uses this flag for the server portion of the queue management code. With D_SYSCALLS defined in SCHEDD_DEBUG there will be verbose logging of all queue management operations the *condor_schedd* performs.

**D_MATCH** When this flag is set, the *condor_negotiator* logs a message for every match.

**D_NETWORK** When this flag is set, all Condor daemons will log a message on every TCP accept, connect, and close, and on every UDP send and receive. This flag is not yet fully supported in the *condor_shadow*.

**D_HOSTNAME** When this flag is set, the Condor daemons and/or tools will print verbose messages explaining how they resolve host names, domain names, and IP addresses. This is useful for sites that are having trouble getting Condor to work because of problems with DNS, NIS or other host name resolving systems in use.

**D_CKPT** When this flag is set, the Condor process checkpoint support code, which is linked into a STANDARD universe user job, will output some low-level details about the checkpoint procedure into the `$(SHADOW_LOG)`.

**D_SECURITY** This flag will enable debug messages pertaining to the setup of secure network communication, including messages for the negotiation of a socket authentication mechanism, the management of a session key cache. and messages about the authentication process itself. See section 3.6.1 for more information about secure communication configuration.

**D_PROCFAMILY** Condor often times needs to manage an entire family of processes, (that is, a process and all descendants of that process). This debug flag will turn on debugging output for the management of families of processes.

**D_ACCOUNTANT** When this flag is set, the *condor_negotiator* will output debug messages relating to the computation of user priorities (see section 3.4).

**D_PROTOCOL** Enable debug messages relating to the protocol for Condor's matchmaking and resource claiming framework.

**D_PID** This flag is different from the other flags, because it is used to change the formatting of all log messages that are printed, as opposed to specifying what kinds of messages should be printed. If `D_PID` is set, Condor will always print out the process identifier (PID) of the process writing each line to the log file. This is especially helpful for Condor daemons that can fork multiple helper-processes (such as the *condor_schedd* or *condor_collector*) so the log file will clearly show which thread of execution is generating each log message.

**D_FDS** This flag is different from the other flags, because it is used to change the formatting of all log messages that are printed, as opposed to specifying what kinds of messages should be printed. If `D_FDS` is set, Condor will always print out the file descriptor that the open of the log file was allocated by the operating system. This can be helpful in debugging Condor's use of system file descriptors as it will generally track the number of file descriptors that Condor has open.

**ALL_DEBUG** Used to make all subsystems share a debug flag. Set the parameter `ALL_DEBUG` instead of changing all of the individual parameters. For example, to turn on all debugging in all subsystems, set `ALL_DEBUG = D_ALL`.

**TOOL_DEBUG** Uses the same values (debugging levels) as `<SUBSYS>_DEBUG` to describe the amount of debugging information sent to `stderr` for Condor tools.

**SUBMIT_DEBUG** Uses the same values (debugging levels) as `<SUBSYS>_DEBUG` to describe the amount of debugging information sent to `stderr` for *condor_submit*.

Log files may optionally be specified per debug level as follows:

**<SUBSYS>_<LEVEL>_LOG** This is the name of a log file for messages at a specific debug level for a specific subsystem. If the debug level is included in `$(<SUBSYS>_DEBUG)`, then all messages of this debug level will be written both to the `$(<SUBSYS>_LOG)` file and the `$(<SUBSYS>_<LEVEL>_LOG)` file. For example, `$(SHADOW_SYSCALLS_LOG)` specifies a log file for all remote system call debug messages.

**MAX_<SUBSYS>_<LEVEL>_LOG** Similar to `MAX_<SUBSYS>_LOG` .

**TRUNC_<SUBSYS>_<LEVEL>_LOG_ON_OPEN** Similar to `TRUNC_<SUBSYS>_LOG_ON_OPEN` .

### 3.3.5 DaemonCore Configuration File Entries

Please read section 3.9 for details on DaemonCore. There are certain configuration file settings that DaemonCore uses which affect all Condor daemons (except the checkpoint server, shadow, and starter, none of which use DaemonCore yet).

**HOSTALLOW**... All macros that begin with either `HOSTALLOW` or `HOSTDENY` are settings for Condor's host-based security. See section 3.6.10 on Setting up IP/host-based security in Condor for details on these macros and how to configure them.

**ENABLE_RUNTIME_CONFIG** The *condor_config_val* tool has an option **-rset** for dynamically setting runtime configuration values (which only effect the in-memory configuration variables). Because of the potential security implications of this feature, by default, Condor daemons will not honor these requests. To use this functionality, Condor administrators must specifically enable it by setting `ENABLE_RUNTIME_CONFIG` to `True`, and specify what configuration variables can be changed using the `SETTABLE_ATTRS`... family of configuration options (described below). Defaults to `False`.

**ENABLE_PERSISTENT_CONFIG** The *condor_config_val* tool has a **-set** option for dynamically setting persistent configuration values. These values override options in the normal Condor configuration files. Because of the potential security implications of this feature, by default, Condor daemons will not honor these requests. To use this functionality, Condor administrators must specifically enable it by setting `ENABLE_PERSISTENT_CONFIG` to `True`, creating a directory where the Condor daemons will hold these dynamically-generated persistent configuration files (declared using `PERSISTENT_CONFIG_DIR`, described below) and specify what configuration variables can be changed using the `SETTABLE_ATTRS`... family of configuration options (described below). Defaults to `False`.

**PERSISTENT_CONFIG_DIR** Directory where daemons should store dynamically-generated persistent configuration files (used to support *condor_config_val* **-set**) This directory should **only** be writable by root, or the user the Condor daemons are running as (if non-root). There is no default, administrators that wish to use this functionality must create this directory and define this setting. This directory must not be shared by multiple Condor installations, though it can be shared by all Condor daemons on the same host. Keep in mind that this directory should not be placed on an NFS mount where "root-squashing" is in effect, or else Condor daemons running as root will not be able to write to them. A directory (only writable by root) on the local file system is usually the best location for this directory.

**SETTABLE_ATTRS**... All macros that begin with SETTABLE_ATTRS or <SUBSYS>_SETTABLE_ATTRS are settings used to restrict the configuration values that can be changed using the *condor_config_val* command. Section 3.6.10 on Setting up IP/Host-Based Security in Condor for details on these macros and how to configure them. In particular, section 3.6.10 on page 282 contains details specific to these macros.

**SHUTDOWN_GRACEFUL_TIMEOUT** Determines how long Condor will allow daemons try their graceful shutdown methods before they do a hard shutdown. It is defined in terms of seconds. The default is 1800 (30 minutes).

**<SUBSYS>_ADDRESS_FILE** A complete path to a file that is to contain an IP address and port number for a daemon. Every Condor daemon that uses DaemonCore has a command port where commands are sent. The IP/port of the daemon is put in that daemon's ClassAd, so that other machines in the pool can query the *condor_collector* (which listens on a well-known port) to find the address of a given daemon on a given machine. When tools and daemons are all executing on the same single machine, communications do not require a query of the *condor_collector* daemon. Instead, they look in a file on the local disk to find the IP/port. This macro causes daemons to write the IP/port of their command socket to a specified file. In this way, local tools will continue to operate, even if the machine running the *condor_collector* crashes. Using this file will also generate slightly less network traffic in the pool, since tools including *condor_q* and *condor_rm* do not need to send any messages over the network to locate the *condor_schedd* daemon. This macro is not necessary for the *condor_collector* daemon, since its command socket is at a well-known port.

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined in section 3.3.1.

**<SUBSYS>_ATTRS or <SUBSYS>_EXPRS** Allows any DaemonCore daemon to advertise arbitrary expressions from the configuration file in its ClassAd. Give the comma-separated list of entries from the configuration file you want in the given daemon's ClassAd. Frequently used to add attributes to machines so that the machines can discriminate between other machines in a job's **rank** and **requirements**.

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined in section 3.3.1.

<SUBSYS>_EXPRS is a historic setting that functions identically to <SUBSYS>_ATTRS. Use <SUBSYS>_ATTRS.

NOTE: The *condor_kbdd* does not send ClassAds now, so this entry does not affect it. The *condor_startd*, *condor_schedd*, *condor_master*, and *condor_collector* do send ClassAds, so those would be valid subsystems to set this entry for.

SUBMIT_EXPRS not part of the <SUBSYS>_EXPRS, it is documented in section 3.3.15

Because of the different syntax of the configuration file and ClassAds, a little extra work is required to get a given entry into a ClassAd. In particular, ClassAds require quote marks (") around strings. Numeric values and boolean expressions can go in directly. For example, if the *condor_startd* is to advertise a string macro, a numeric macro, and a boolean expression, do something similar to:

```
STRING = This is a string
NUMBER = 666
BOOL1 = True
BOOL2 = CurrentTime >= $(NUMBER) || $(BOOL1)
MY_STRING = "$(STRING)"
STARTD_ATTRS = MY_STRING, NUMBER, BOOL1, BOOL2
```

### 3.3.6 Network-Related Configuration File Entries

More information about networking in Condor can be found in section 3.7 on page 294.

**BIND_ALL_INTERFACES** For systems with multiple network interfaces, if this configuration setting is not defined, Condor binds all network sockets to first interface found, or the IP address specified with NETWORK_INTERFACE (described below). Starting with version 6.7.13, BIND_ALL_INTERFACES can be set to True to cause Condor to bind to all interfaces on the machine. However, currently Condor is still only able to advertise a single IP address, even if it is listening on multiple interfaces. By default, it will advertise the IP address of the network interface used to contact the collector, since this is the most likely to be accessible to other processes which query information from the same collector. More information about using this setting can be found in section 3.7.2 on page 298.

**NETWORK_INTERFACE** For systems with multiple network interfaces, if this configuration setting is not defined, Condor binds all network sockets to first interface found. To bind to a specific network interface other than the first one, this NETWORK_INTERFACE should be set to the IP address to use. When BIND_ALL_INTERFACES is set to True, this setting simply controls what IP address a given Condor host will advertise. More information about configuring Condor on machines with multiple network interfaces can be found in section 3.7.2 on page 298.

**HIGHPORT** Specifies an upper limit of given port numbers for Condor to use, such that Condor is restricted to a range of port numbers. If this macro is not explicitly specified, then Condor will not restrict the port numbers that it uses. Condor will use system-assigned port numbers. For this macro to work, both HIGHPORT and LOWPORT (given below) must be defined.

**LOWPORT** Specifies a lower limit of given port numbers for Condor to use, such that Condor is restricted to a range of port numbers. If this macro is not explicitly specified, then Condor will not restrict the port numbers that it uses. Condor will use system-assigned port numbers. For this macro to work, both HIGHPORT (given above) and LOWPORT must be defined.

**IN_LOWPORT** An integer value that specifies a lower limit of given port numbers for Condor to use on incoming connections (ports for listening), such that Condor is restricted to a range of port numbers. This range implies the use of both IN_LOWPORT and IN_HIGHPORT. A range of port numbers less than 1024 may be used for daemons running as root. Do not specify IN_LOWPORT in combination with IN_HIGHPORT such that the range crosses the

port 1024 boundary. Applies only to Unix machine configuration. Use of IN_LOWPORT and IN_HIGHPORT overrides any definition of LOWPORT and HIGHPORT.

**IN_HIGHPORT** An integer value that specifies an upper limit of given port numbers for Condor to use on incoming connections (ports for listening), such that Condor is restricted to a range of port numbers. This range implies the use of both IN_LOWPORT and IN_HIGHPORT. A range of port numbers less than 1024 may be used for daemons running as root. Do not specify IN_LOWPORT in combination with IN_HIGHPORT such that the range crosses the port 1024 boundary. Applies only to Unix machine configuration. Use of IN_LOWPORT and IN_HIGHPORT overrides any definition of LOWPORT and HIGHPORT.

**OUT_LOWPORT** An integer value that specifies a lower limit of given port numbers for Condor to use on outgoing connections, such that Condor is restricted to a range of port numbers. This range implies the use of both OUT_LOWPORT and OUT_HIGHPORT. A range of port numbers less than 1024 is inappropriate, as not all daemons and tools will be run as root. Applies only to Unix machine configuration. Use of OUT_LOWPORT and OUT_HIGHPORT overrides any definition of LOWPORT and HIGHPORT.

**OUT_HIGHPORT** An integer value that specifies an upper limit of given port numbers for Condor to use on outgoing connections, such that Condor is restricted to a range of port numbers. This range implies the use of both OUT_LOWPORT and OUT_HIGHPORT. A range of port numbers less than 1024 is inappropriate, as not all daemons and tools will be run as root. Applies only to Unix machine configuration. Use of OUT_LOWPORT and OUT_HIGHPORT overrides any definition of LOWPORT and HIGHPORT.

**UPDATE_COLLECTOR_WITH_TCP** If your site needs to use TCP connections to send ClassAd updates to your collector (which it almost certainly does NOT), set to True to enable this feature. Please read section 3.7.4 on "Using TCP to Send Collector Updates" on page 314 for more details and a discussion of when this functionality is needed. At this time, this setting only affects the main *condor_collector* for the site, not any sites that a *condor_schedd* might flock to. If enabled, also define COLLECTOR_SOCKET_CACHE_SIZE at the central manager, so that the collector will accept TCP connections for updates, and will keep them open for reuse. Defaults to False.

**TCP_UPDATE_COLLECTORS** The list of collectors which will be updated with TCP instead of UDP. Please read section 3.7.4 on "Using TCP to Send Collector Updates" on page 314 for more details and a discussion of when a site needs this functionality. If not defined, no collectors use TCP instead of UDP.

**<SUBSYS>_TIMEOUT_MULTIPLIER** An integer value that defaults to 1. This value multiplies configured timeout values for all targeted subsystem communications, thereby increasing the time until a timeout occurs. This configuration variable is intended for use by developers for debugging purposes, where communication timeouts interfere.

**NONBLOCKING_COLLECTOR_UPDATE** A boolean value that defaults to True. When True, the establishment of TCP connections to the *condor_collector* daemon for a security-enabled pool are done in a nonblocking manner.

**NEGOTIATOR_USE_NONBLOCKING_STARTD_CONTACT** A boolean value that defaults to `True`. When `True`, the establishment of TCP connections from the *condor_negotiator* daemon to the *condor_startd* daemon for a security-enabled pool are done in a nonblocking manner.

The following settings are specific to enabling Generic Connection Brokering or GCB in your Condor pool. More information about GCB and how to configure it can be found in section 3.7.3 on page 301.

**NET_REMAP_ENABLE** If defined to `True`, this setting will enable a network remapping service for Condor. The service to use is controlled by `NET_REMAP_SERVICE`, described below. This boolean value defaults to `False`.

**NET_REMAP_SERVICE** If `NET_REMAP_ENABLE` is defined to `True`, this setting controls what network remapping service should be used. Currently, the only value supported is `GCB`. The default is undefined.

**NET_REMAP_INAGENT** Hosts with the GCB network remapping service enabled that require a GCB broker (see the section on GCB referenced above for more details) specify the IP address of their broker with this setting. The default is undefined.

**NET_REMAP_ROUTE** Hosts with the GCB network remapping service enabled that would like to use a GCB routing table GCB broker (again, see the section on GCB for more details) specify The full path to their routing table with this setting. The default is undefined.

### 3.3.7 Shared File System Configuration File Macros

These macros control how Condor interacts with various shared and network file systems. If you are using AFS as your shared file system, be sure to read section 3.12.1 on Using Condor with AFS. For information on submitting jobs under shared file systems, see section 2.5.3.

**UID_DOMAIN** The `UID_DOMAIN` macro is used to decide under which user to run jobs. If the `$(UID_DOMAIN)` on the submitting machine is different than the `$(UID_DOMAIN)` on the machine that runs a job, then Condor runs the job as the user `nobody`. For example, if the submit machine has a `$(UID_DOMAIN)` of flippy.cs.wisc.edu, and the machine where the job will execute has a `$(UID_DOMAIN)` of cs.wisc.edu, the job will run as user `nobody`, because the two `$(UID_DOMAIN)`s are not the same. If the `$(UID_DOMAIN)` is the same on both the submit and execute machines, then Condor will run the job as the user that submitted the job.

A further check attempts to assure that the submitting machine can not lie about its `UID_DOMAIN`. Condor compares the submit machine's claimed value for `UID_DOMAIN` to its fully qualified name. If the two do not end the same, then the submit machine is presumed to be lying about its `UID_DOMAIN`. In this case, Condor will run the job as user `nobody`. For example, a job submission to the Condor pool at the UW Madison from flippy.example.com, claiming a `UID_DOMAIN` of of cs.wisc.edu, will run the job as the user `nobody`.

Because of this verification, $(UID_DOMAIN) must be a real domain name. At the Computer Sciences department at the UW Madison, we set the $(UID_DOMAIN) to be cs.wisc.edu to indicate that whenever someone submits from a department machine, we will run the job as the user who submits it.

Also see SOFT_UID_DOMAIN below for information about one more check that Condor performs before running a job as a given user.

A few details:

An administrator could set UID_DOMAIN to *. This will match all domains, but it is a gaping security hole. It is not recommended.

An administrator can also leave UID_DOMAIN undefined. This will force Condor to always run jobs as user nobody. Running standard universe jobs as user nobody enhances security and should cause no problems, because the jobs use remote I/O to access all of their files. However, if vanilla jobs are run as user nobody, then files that need to be accessed by the job will need to be marked as world readable/writable so the user nobody can access them.

When Condor sends e-mail about a job, Condor sends the e-mail to user@$(UID_DOMAIN). If UID_DOMAIN is undefined, the e-mail is sent to user@submitmachinename.

**TRUST_UID_DOMAIN** As an added security precaution when Condor is about to spawn a job, it ensures that the UID_DOMAIN of a given submit machine is a substring of that machine's fully-qualified host name. However, at some sites, there may be multiple UID spaces that do not clearly correspond to Internet domain names. In these cases, administrators may wish to use names to describe the UID domains which are not substrings of the host names of the machines. For this to work, Condor must not do this regular security check. If the TRUST_UID_DOMAIN setting is defined to True, Condor will not perform this test, and will trust whatever UID_DOMAIN is presented by the submit machine when trying to spawn a job, instead of making sure the submit machine's host name matches the UID_DOMAIN. When not defined, the default is False, since it is more secure to perform this test.

**SOFT_UID_DOMAIN** A boolean variable that defaults to False when not defined. When Condor is about to run a job as a particular user (instead of as user nobody), it verifies that the UID given for the user is in the password file and actually matches the given user name. However, under installations that do not have every user in every machine's password file, this check will fail and the execution attempt will be aborted. To cause Condor not to do this check, set this configuration variable to True. Condor will then run the job under the user's UID.

**VMx_USER** The name of a user for Condor to use instead of user nobody, as part of a solution that plugs a security hole whereby a lurker process can prey on a subsequent job run as user name nobody. x is an integer associated with virtual machines. On Windows, VMx_USER will only work if the credential of the specified user is stored on the execute machine using *condor_store_cred*. See Section 3.6.12 for more information.

**EXECUTE_LOGIN_IS_DEDICATED** A boolean value that defaults to False. When True, Condor knows that users given by the VMx_USER configuration variable exist exclusively to be the owner of Condor jobs. When a Condor job belonging to one of these users completes, Condor may use kill all jobs belonging to this user. This avoids lurker jobs that the Condor job may have started.

**FILESYSTEM_DOMAIN** The FILESYSTEM_DOMAIN macro is an arbitrary string that is used to decide if two machines (a submitting machine and an execute machine) share a file system. Although the macro name contains the word "DOMAIN", the macro is not required to be a domain name. It often is a domain name.

Note that this implementation is not ideal: machines may share some file systems but not others. Condor currently has no way to express this automatically. You can express the need to use a particular file system by adding additional attributes to your machines and submit files, similar to the example given in Frequently Asked Questions, section 7 on how to run jobs only on machines that have certain software packages.

Note that if you do not set $(FILESYSTEM_DOMAIN), Condor defaults to setting the macro's value to be the fully qualified hostname of the local machine. Since each machine will have a different $(FILESYSTEM_DOMAIN), they will not be considered to have shared file systems.

**RESERVE_AFS_CACHE** If your machine is running AFS and the AFS cache lives on the same partition as the other Condor directories, and you want Condor to reserve the space that your AFS cache is configured to use, set this macro to True. It defaults to False.

**USE_NFS** This macro influences how Condor jobs running in the standard universe access their files. Condor will redirect the file I/O requests of standard universe jobs to be executed on the machine which submitted the job. Because of this, as a Condor job migrates around the network, the file system always appears to be identical to the file system where the job was submitted. However, consider the case where a user's data files are sitting on an NFS server. The machine running the user's program will send all I/O over the network to the machine which submitted the job, which in turn sends all the I/O over the network a second time back to the NFS file server. Thus, all of the program's I/O is being sent over the network twice.

If this macro to True, then Condor will attempt to read/write files without redirecting I/O back to the submitting machine if both the submitting machine and the machine running the job are both accessing the same NFS servers (*if* they are both in the same $(FILESYSTEM_DOMAIN) and in the same $(UID_DOMAIN), as described above). The result is I/O performed by Condor standard universe jobs is only sent over the network once. While sending all file operations over the network twice might sound really bad, unless you are operating over networks where bandwidth as at a very high premium, practical experience reveals that this scheme offers very little real performance gain. There are also some (fairly rare) situations where this scheme can break down.

Setting $(USE_NFS) to False is always safe. It may result in slightly more network traffic, but Condor jobs are most often heavy on CPU and light on I/O. It also ensures that a remote standard universe Condor job will always use Condor's remote system calls mechanism to reroute I/O and therefore see the exact same file system that the user sees on the machine where she/he submitted the job.

Some gritty details for folks who want to know: If the you set $(USE_NFS) to True, and the $(FILESYSTEM_DOMAIN) of both the submitting machine and the remote machine about to execute the job match, and the $(FILESYSTEM_DOMAIN) claimed by the submit machine is indeed found to be a subset of what an inverse lookup to a DNS (domain name server) reports as the fully qualified domain name for the submit machine's IP address (this security measure

safeguards against the submit machine from lying), *then* the job will access files using a local system call, without redirecting them to the submitting machine (with NFS). Otherwise, the system call will get routed back to the submitting machine using Condor's remote system call mechanism. NOTE: When submitting a vanilla job, *condor_submit* will, by default, append requirements to the Job ClassAd that specify the machine to run the job must be in the same `$(FILESYSTEM_DOMAIN)` and the same `$(UID_DOMAIN)`.

**IGNORE_NFS_LOCK_ERRORS** When set to `True`, all errors related to file locking errors from NFS are ignored. Defaults to `False`, not ignoring errors.

**USE_AFS** If your machines have AFS, this macro determines whether Condor will use remote system calls for standard universe jobs to send I/O requests to the submit machine, or if it should use local file access on the execute machine (which will then use AFS to get to the submitter's files). Read the setting above on `$(USE_NFS)` for a discussion of why you might want to use AFS access instead of remote system calls.

One important difference between `$(USE_NFS)` and `$(USE_AFS)` is the AFS cache. With `$(USE_AFS)` set to `True`, the remote Condor job executing on some machine will start modifying the AFS cache, possibly evicting the machine owner's files from the cache to make room for its own. Generally speaking, since we try to minimize the impact of having a Condor job run on a given machine, we do not recommend using this setting.

While sending all file operations over the network twice might sound really bad, unless you are operating over networks where bandwidth as at a very high premium, practical experience reveals that this scheme offers very little real performance gain. There are also some (fairly rare) situations where this scheme can break down.

Setting `$(USE_AFS)` to `False` is always safe. It may result in slightly more network traffic, but Condor jobs are usually heavy on CPU and light on I/O. `False` ensures that a remote standard universe Condor job will always see the exact same file system that the user on sees on the machine where he/she submitted the job. Plus, it will ensure that the machine where the job executes does not have its AFS cache modified as a result of the Condor job being there.

However, things may be different at your site, which is why the setting is there.

### 3.3.8 Checkpoint Server Configuration File Macros

These macros control whether or not Condor uses a checkpoint server. If you are using a checkpoint server, this section describes the settings that the checkpoint server itself needs defined. A checkpoint server is installed separately. It is not included in the main Condor binary distribution or installation procedure. See section 3.8 on Installing a Checkpoint Server for details on installing and running a checkpoint server for your pool.

NOTE: If you are setting up a machine to join the UW-Madison CS Department Condor pool, you *should* configure the machine to use a checkpoint server, and use "condor-ckpt.cs.wisc.edu" as the checkpoint server host (see below).

**CKPT_SERVER_HOST** The hostname of a checkpoint server.

**STARTER_CHOOSES_CKPT_SERVER** If this parameter is `True` or undefined on the submit machine, the checkpoint server specified by `$(CKPT_SERVER_HOST)` on the execute machine is used. If it is `False` on the submit machine, the checkpoint server specified by `$(CKPT_SERVER_HOST)` on the submit machine is used.

**CKPT_SERVER_DIR** The checkpoint server needs this macro defined to the full path of the directory the server should use to store checkpoint files. Depending on the size of your pool and the size of the jobs your users are submitting, this directory (and its subdirectories) might need to store many Mbytes of data.

**USE_CKPT_SERVER** A boolean which determines if you want a given submit machine to use a checkpoint server if one is available. If a checkpoint server isn't available or `USE_CKPT_SERVER` is set to False, checkpoints will be written to the local `$(SPOOL)` directory on the submission machine.

**MAX_DISCARDED_RUN_TIME** If the shadow is unable to read a checkpoint file from the checkpoint server, it keeps trying only if the job has accumulated more than this many seconds of CPU usage. Otherwise, the job is started from scratch. Defaults to 3600 (1 hour). This setting is only used if `$(USE_CKPT_SERVER)` is `True`.

### 3.3.9 condor_master Configuration File Macros

These macros control the *condor_master*.

**DAEMON_LIST** This macro determines what daemons the *condor_master* will start and keep its watchful eyes on. The list is a comma or space separated list of subsystem names (listed in section 3.3.1). For example,

```
DAEMON_LIST = MASTER, STARTD, SCHEDD
```

NOTE: This configuration variable cannot be changed by using *condor_reconfig* or by sending a SIGHUP. To change this configuration variable, restart the *condor_master* daemon by using *condor_restart*. Only then will the change take effect.

NOTE: On your central manager, your `$(DAEMON_LIST)` will be different from your regular pool, since it will include entries for the *condor_collector* and *condor_negotiator*.

NOTE: On machines running Digital Unix, your `$(DAEMON_LIST)` will also include KBDD, for the *condor_kbdd*, which is a special daemon that runs to monitor keyboard and mouse activity on the console. It is only with this special daemon that we can acquire this information on those platforms.

**DC_DAEMON_LIST** This macro lists the daemons in `DAEMON_LIST` which use the Condor DaemonCore library. The *condor_master* must differentiate between daemons that use DaemonCore and those that don't so it uses the appropriate inter-process communication mechanisms. This list currently includes all Condor daemons except the checkpoint server by default.

**<SUBSYS>** Once you have defined which subsystems you want the *condor_master* to start, you must provide it with the full path to each of these binaries. For example:

```
MASTER          = $(SBIN)/condor_master
STARTD          = $(SBIN)/condor_startd
SCHEDD          = $(SBIN)/condor_schedd
```

These are most often defined relative to the `$(SBIN)` macro.

The macro is named by substituting `<SUBSYS>` with the appropriate subsystem string as defined in section 3.3.1.

**DAEMONNAME_ENVIRONMENT** For each subsystem defined in `DAEMON_LIST`, you may specify changes to the environment that daemon is started with by setting `DAEMONNAME_ENVIRONMENT`, where `DAEMONNAME` is the name of a daemon listed in `DAEMON_LIST`. It should use the same syntax for specifying the environment as the environment specification in a *condor_submit* file (see page 718). For example, if you wish to redefine the `TMP` and `CONDOR_CONFIG` environment variables seen by the *condor_schedd*, you could place the following in the config file:

```
SCHEDD_ENVIRONMENT = "TMP=/new/value CONDOR_CONFIG=/special/config"
```

When the *condor_schedd* was started by the *condor_master*, it would see the specified values of `TMP` and `CONDOR_CONFIG`.

**<SUBSYS>_ARGS** This macro allows the specification of additional command line arguments for any process spawned by the *condor_master*. List the desired arguments using the same syntax as the arguments specification in a *condor_submit* submit file (see page 717), with one exception: do not escape double-quotes when using the old-style syntax (this is for backward compatibility). Set the arguments for a specific daemon with this macro, and the macro will affect only that daemon. Define one of these for each daemon the *condor_master* is controlling. For example, set `$(STARTD_ARGS)` to specify any extra command line arguments to the *condor_startd*.

The macro is named by substituting `<SUBSYS>` with the appropriate subsystem string as defined in section 3.3.1.

**PREEN** In addition to the daemons defined in `$(DAEMON_LIST)`, the *condor_master* also starts up a special process, *condor_preen* to clean out junk files that have been left laying around by Condor. This macro determines where the *condor_master* finds the *condor_preen* binary. Comment out this macro, and *condor_preen* will not run.

**PREEN_ARGS** Controls how *condor_preen* behaves by allowing the specification of command-line arguments. This macro works as `$(<SUBSYS>_ARGS)` does. The difference is that you must specify this macro for *condor_preen* if you want it to do anything. *condor_preen* takes action only because of command line arguments. **-m** means you want e-mail about files *condor_preen* finds that it thinks it should remove. **-r** means you want *condor_preen* to actually remove these files.

**PREEN_INTERVAL** This macro determines how often *condor_preen* should be started. It is defined in terms of seconds and defaults to 86400 (once a day).

**PUBLISH_OBITUARIES** When a daemon crashes, the *condor_master* can send e-mail to the address specified by `$(CONDOR_ADMIN)` with an obituary letting the administrator know that the daemon died, the cause of death (which signal or exit status it exited with), and (optionally) the last few entries from that daemon's log file. If you want obituaries, set this macro to `True`.

**OBITUARY_LOG_LENGTH** This macro controls how many lines of the log file are part of obituaries. This macro has a default value of 20 lines.

**START_MASTER** If this setting is defined and set to `False` when the *condor_master* starts up, the first thing it will do is exit. This appears strange, but perhaps you do not want Condor to run on certain machines in your pool, yet the boot scripts for your entire pool are handled by a centralized This is an entry you would most likely find in a local configuration file, not a global configuration file.

**START_DAEMONS** This macro is similar to the `$(START_MASTER)` macro described above. However, the *condor_master* does not exit; it does not start any of the daemons listed in the `$(DAEMON_LIST)`. The daemons may be started at a later time with a *condor_on* command.

**MASTER_UPDATE_INTERVAL** This macro determines how often the *condor_master* sends a ClassAd update to the *condor_collector*. It is defined in seconds and defaults to 300 (every 5 minutes).

**MASTER_CHECK_NEW_EXEC_INTERVAL** This macro controls how often the *condor_master* checks the timestamps of the running daemons. If any daemons have been modified, the master restarts them. It is defined in seconds and defaults to 300 (every 5 minutes).

**MASTER_NEW_BINARY_DELAY** Once the *condor_master* has discovered a new binary, this macro controls how long it waits before attempting to execute the new binary. This delay exists because the *condor_master* might notice a new binary while it is in the process of being copied, in which case trying to execute it yields unpredictable results. The entry is defined in seconds and defaults to 120 (2 minutes).

**SHUTDOWN_FAST_TIMEOUT** This macro determines the maximum amount of time daemons are given to perform their fast shutdown procedure before the *condor_master* kills them outright. It is defined in seconds and defaults to 300 (5 minutes).

**MASTER_BACKOFF_CONSTANT and MASTER_<name>_BACKOFF_CONSTANT** When a daemon crashes, *condor_master* uses an exponential back off delay before restarting it; see the discussion at the end of this section for a detailed discussion on how these parameters work together. These settings define the constant value of the expression used to determine how long to wait before starting the daemon again (and, effectively becomes the initial backoff time). It is an integer in units of seconds, and defaults to 9 seconds.

`$(MASTER_<name>_BACKOFF_CONSTANT)` is the daemon-specific form of `MASTER_BACKOFF_CONSTANT`; if this daemon-specific macro is not defined for a specific daemon, the non-daemon-specific value will used.

**MASTER_BACKOFF_FACTOR and MASTER_<name>_BACKOFF_FACTOR** When a daemon crashes, *condor_master* uses an exponential back off delay before restarting it; see the discussion at the end of this section for a detailed discussion on how these parameters work together. This setting is the base of the exponent used to determine how long to wait before starting the daemon again. It defaults to 2 seconds.

$(MASTER_<name>_BACKOFF_FACTOR) is the daemon-specific form of MASTER_BACKOFF_FACTOR; if this daemon-specific macro is not defined for a specific daemon, the non-daemon-specific value will used.

**MASTER_BACKOFF_CEILING and MASTER_<name>_BACKOFF_CEILING** When a daemon crashes, *condor_master* uses an exponential back off delay before restarting it; see the discussion at the end of this section for a detailed discussion on how these parameters work together. This entry determines the maximum amount of time you want the master to wait between attempts to start a given daemon. (With 2.0 as the $(MASTER_BACKOFF_FACTOR), 1 hour is obtained in 12 restarts). It is defined in terms of seconds and defaults to 3600 (1 hour).

$(MASTER_<name>_BACKOFF_CEILING) is the daemon-specific form of MASTER_BACKOFF_CEILING; if this daemon-specific macro is not defined for a specific daemon, the non-daemon-specific value will used.

**MASTER_RECOVER_FACTOR and MASTER_<name>_RECOVER_FACTOR** A macro to set how long a daemon needs to run without crashing before it is considered *recovered*. Once a daemon has recovered, the number of restarts is reset, so the exponential back off returns to its initial state. The macro is defined in terms of seconds and defaults to 300 (5 minutes).

$(MASTER_<name>_RECOVER_FACTOR) is the daemon-specific form of MASTER_RECOVER_FACTOR; if this daemon-specific macro is not defined for a specific daemon, the non-daemon-specific value will used.

When a daemon crashes, *condor_master* will restart the daemon after a delay (a back off). The length of this delay is based on how many times it has been restarted, and gets larger after each crashes. The equation for calculating this backoff time is given by:

$$t = c + k^n$$

where $t$ is the calculated time, $c$ is the constant defined by $(MASTER_BACKOFF_CONSTANT), $k$ is the "factor" defined by $(MASTER_BACKOFF_FACTOR), and $n$ is the number of restarts already attempted (0 for the first restart, 1 for the next, etc.).

With default values, after the first crash, the delay would be $t = 9 + 2.0^0$, giving 10 seconds (remember, $n = 0$). If the daemon keeps crashing, the delay increases.

For example, take the $(MASTER_BACKOFF_FACTOR) (which defaults to 2.0) to the power the number of times the daemon has restarted, and add $(MASTER_BACKOFF_CONSTANT) (which defaults to 9). Thus:

$1^{st}$ crash: $n = 0$, so: $t = 9 + 2^0 = 9 + 1 = 10\ seconds$

$2^{nd}$ crash: $n = 1$, so: $t = 9 + 2^1 = 9 + 2 = 11\ seconds$

$3^{rd}$ crash: $n = 2$, so: $t = 9 + 2^2 = 9 + 4 = 13\ seconds$

...

$6^{th}$ crash: $n = 5$, so: $t = 9 + 2^5 = 9 + 32 = 41\ seconds$

...

$9^{th}$ crash: $n = 8$, so: $t = 9 + 2^8 = 9 + 256 = 265\ seconds$

And, after the 13 crashes, it would be:

$13^{th}$ crash: $n = 12$, so: $t = 9 + 2^{12} = 9 + 4096 = 4105\ seconds$

This is bigger than the $(MASTER_BACKOFF_CEILING), which defaults to 3600, so the daemon would really be restarted after only 3600 seconds, not 4105. The *condor_master* tries again every hour (since the numbers would get larger and would always be capped by the ceiling). Eventually, imagine that daemon finally started and did not crash. This might happen if, for example, an administrator reinstalled an accidentally deleted binary after receiving e-mail about the daemon crashing. If it stayed alive for $(MASTER_RECOVER_FACTOR) seconds (defaults to 5 minutes), the count of how many restarts this daemon has performed is reset to 0.

The moral of the example is that the defaults work quite well, and you probably will not want to change them for any reason.

**MASTER_NAME** Defines a unique name given for a *condor_master* daemon on a machine. Defaults to the fully qualified host name. If more than one *condor_master* is running on the same host (for example, because of multiple Personal Condor installations running as different users) the MASTER_NAME for each *condor_master* should be defined to uniquely identify the separate daemons.

If the MASTER_NAME contains more than a host name, it must have the form `identifying-string@full.host.name`. If the string specified with MASTER_NAME already includes an @ sign, Condor will replace whatever follows the @ sign with the fully qualified host name of the local machine. If the string does not include an @ sign, Condor will append one, followed by the host name. The `identifying-string` portion can contain any alphanumeric ASCII characters or punctuation marks except @ (which is used to delimit the name from the host name). We recommend that the string does not contain the : character, since that might cause problems with certain tools. In the example of many Personal Condor installations on the same host, the user name that each *condor_master* is executing as is, by convention, the `identifying-string`. This is easily accomplished by setting `MASTER_NAME = $(USERNAME)` in the configuration file.

If the MASTER_NAME setting is used, and the *condor_master* is configured to spawn a *condor_schedd*, the name defined with MASTER_NAME takes precedence over the SCHEDD_NAME setting (see section 3.3.11 on page 185). Since Condor makes the assumption that there is only one instance of the *condor_startd* running on a machine, the MASTER_NAME is not automatically propagated to the *condor_startd*. However, in situations where multiple *condor_startd* daemons are running on the same host (for example, when using *condor_glidein*), the STARTD_NAME should be set to uniquely identify the *condor_startd* daemons (this is done

automatically in the case of *condor_glidein*).

If a Condor daemon (master, schedd or startd) has been given a unique name, all Condor tools that need to contact that daemon can be told what name to use via the **-name** command-line option.

**MASTER_ATTRS** This macro is described in section 3.3.5 as <SUBSYS>_ATTRS.

**MASTER_DEBUG** This macro is described in section 3.3.4 as <SUBSYS>_DEBUG.

**MASTER_ADDRESS_FILE** This macro is described in section 3.3.5 as <SUBSYS>_ADDRESS_FILE.

**SECONDARY_COLLECTOR_LIST** This macro lists the host names of secondary collectors. A secondary collector is a machine running a *condor_collector* daemon that is not the central manager. A secondary collector makes it possible to execute administrative commands in the pool when the central manager is down by using the **-pool** argument to specify the name of a secondary collector to use to locate the *condor_master* daemon.

**ALLOW_ADMIN_COMMANDS** If set to NO for a given host, this macro disables administrative commands, such as *condor_restart*, *condor_on*, and *condor_off*, to that host.

**MASTER_INSTANCE_LOCK** Defines the name of a file for the *condor_master* daemon to lock in order to prevent multiple *condor_master*s from starting. This is useful when using shared file systems like NFS which do not technically support locking in the case where the lock files reside on a local disk. If this macro is not defined, the default file name will be `$(LOCK)/InstanceLock`. `$(LOCK)` can instead be defined to specify the location of all lock files, not just the *condor_master*'s `InstanceLock`. If `$(LOCK)` is undefined, then the master log itself is locked.

**ADD_WINDOWS_FIREWALL_EXCEPTION** When set to `False`, the *condor_master* will not automatically add Condor to the Windows Firewall list of trusted applications. Such trusted applications can accept incoming connections without interference from the firewall. This only affects machines running Windows XP SP2 or higher. The default is `True`.

**WINDOWS_FIREWALL_FAILURE_RETRY** An integer value (default value is 60) that represents the number of times the *condor_master* will retry to add firewall exceptions. When a Windows machine boots up, Condor starts up by default as well. Under certain conditions, the *condor_master* may have difficulty adding exceptions to the Windows Firewall because of a delay in other services starting up. Examples of services that may possibly be slow are the SharedAccess service, the Netman service, or the Workstation service. This configuration variable allows administrators to set the number of times (once every 10 seconds) that the *condor_master* will retry to add firewall exceptions. A value of 0 means that Condor will retry indefinitely.

### 3.3.10 condor_startd Configuration File Macros

NOTE: If you are running Condor on a multi-CPU machine, be sure to also read section 3.12.7 on page 347 which describes how to set up and configure Condor on SMP machines.

These settings control general operation of the *condor_startd*. Examples using these configuration macros, as well as further explanation is found in section 3.5 on Configuring The Startd Policy.

**START** A boolean expression that, when `True`, indicates that the machine is willing to start running a Condor job. `START` is considered when the *condor_negotiator* daemon is considering evicting the job to replace it with one that will generate a better rank for the *condor_startd* daemon, or a user with a higher priority.

**SUSPEND** A boolean expression that, when `True`, causes Condor to suspend running a Condor job. The machine may still be claimed, but the job makes no further progress, and Condor does not generate a load on the machine.

**PREEMPT** A boolean expression that, when `True`, causes Condor to stop a currently running job.

**CONTINUE** A boolean expression that, when `True`, causes Condor to continue the execution of a suspended job.

**KILL** A boolean expression that, when `True`, causes Condor to immediately stop the execution of a currently running job, without delay, and without taking the time to produce a checkpoint (for a standard universe job).

**RANK** A floating point value that Condor uses to compare potential jobs. A larger value for a specific job ranks that job above others with lower values for `RANK`.

**IS_VALID_CHECKPOINT_PLATFORM** A boolean expression that is logically anded with the with the `START` expression to limit which machines a standard universe job may continue execution on once they have produced a checkpoint. The default expression is

```
IS_VALID_CHECKPOINT_PLATFORM =
(
  ( (TARGET.JobUniverse == 1) == FALSE) ||

  (
    (MY.CheckpointPlatform =!= UNDEFINED) &&
    (
      (TARGET.LastCheckpointPlatform =?= MY.CheckpointPlatform) ||
      (TARGET.NumCkpts == 0)
    )
  )
)
```

**WANT_SUSPEND** A boolean expression that, when `True`, tells Condor to evaluate the `SUSPEND` expression.

**WANT_VACATE** A boolean expression that, when `True`, defines that a preempted Condor job is to be vacated, instead of killed.

**IS_OWNER** A boolean expression that defaults to being defined as

```
IS_OWNER = (START =?= FALSE)
```

Used to describe the state of the machine with respect to its use by its owner. Job ClassAd attributes are not used in defining IS_OWNER, as they would be Undefined.

**STARTER** This macro holds the full path to the *condor_starter* binary that the *condor_startd* should spawn. It is normally defined relative to $(SBIN).

**ALTERNATE_STARTER_1** This macro holds the full path to the *condor_starter.pvm* binary that the *condor_startd* spawns to service PVM jobs. It is normally defined relative to $(SBIN), since by default, *condor_starter.pvm* is installed in the regular Condor release directory.

**POLLING_INTERVAL** When a *condor_startd* enters the claimed state, this macro determines how often the state of the machine is polled to check the need to suspend, resume, vacate or kill the job. It is defined in terms of seconds and defaults to 5.

**UPDATE_INTERVAL** Determines how often the *condor_startd* should send a ClassAd update to the *condor_collector*. The *condor_startd* also sends update on any state or activity change, or if the value of its START expression changes. See section 3.5.6 on *condor_startd* states, section 3.5.7 on *condor_startd* Activities, and section 3.5.3 on *condor_startd* START expression for details on states, activities, and the START expression. This macro is defined in terms of seconds and defaults to 300 (5 minutes).

**MAXJOBRETIREMENTTIME** An integer value representing the number of seconds a preempted job will be allowed to run before being evicted. The default value of 0 (when the configuration variable is not present) implements the expected policy that there is no retirement time. See MAXJOBRETIREMENTTIME in section 3.5.9 for further explanation.

**CLAIM_WORKLIFE** If provided, this expression specifies the number of seconds during which a claim will continue accepting new jobs. Once this time expires, any existing job may continue to run as usual, but once it finishes or is preempted, the claim is closed. This may be useful if you want to force periodic renegotiation of resources without preemption having to occur. For example, if you have some low-priority jobs which should never be interrupted with kill signals, you could prevent them from being killed with MaxJobRetirementTime, but now high-priority jobs may have to wait in line when they match to a machine that is busy running one of these uninterruptible jobs. You can prevent the high-priority jobs from ever matching to such a machine by using a rank expression in the job or in the negotiator's rank expressions, but then the low-priority claim will never be interrupted; it can keep running more jobs. The solution is to use CLAIM_WORKLIFE to force the claim to stop running additional jobs after a certain amount of time. The default value for CLAIM_WORKLIFE is -1, which is treated as an infinite claim worklife, so claims may be held indefinitely (as long as they are not preempted and the schedd does not relinquish them, of course).

**MAX_CLAIM_ALIVES_MISSED** The *condor_schedd* sends periodic updates to each *condor_startd* as a keep alive (see the description of ALIVE_INTERVAL on page 183). If the *condor_startd* does not receive any keep alive messages, it assumes that something has gone wrong with the *condor_schedd* and that the resource is not being effectively used. Once this happens, the *condor_startd* considers the claim to have timed out, it releases the claim, and starts advertising itself as available for other jobs. Because these keep alive messages are sent via UDP, they are sometimes dropped by the network. Therefore, the *condor_startd* has some tolerance for

missed keep alive messages, so that in case a few keep alives are lost, the *condor_startd* will not immediately release the claim. This setting controls how many keep alive messages can be missed before the *condor_startd* considers the claim no longer valid. The default is 6.

**STARTD_HAS_BAD_UTMP** When the *condor_startd* is computing the idle time of all the users of the machine (both local and remote), it checks the `utmp` file to find all the currently active ttys, and only checks access time of the devices associated with active logins. Unfortunately, on some systems, `utmp` is unreliable, and the *condor_startd* might miss keyboard activity by doing this. So, if your `utmp` is unreliable, set this macro to `True` and the *condor_startd* will check the access time on all tty and pty devices.

**CONSOLE_DEVICES** This macro allows the *condor_startd* to monitor console (keyboard and mouse) activity by checking the access times on special files in `/dev`. Activity on these files shows up as `ConsoleIdle` time in the *condor_startd*'s ClassAd. Give a comma-separated list of the names of devices considered the console, without the `/dev/` portion of the pathname. The defaults vary from platform to platform, and are usually correct.

One possible exception to this is on Linux, where we use "mouse" as one of the entries. Most Linux installations put in a soft link from `/dev/mouse` that points to the appropriate device (for example, `/dev/psaux` for a PS/2 bus mouse, or `/dev/tty00` for a serial mouse connected to com1). However, if your installation does not have this soft link, you will either need to put it in (you will be glad you did), or change this macro to point to the right device.

Unfortunately, there are no such devices on Digital Unix (don't be fooled by `/dev/keyboard0`; the kernel does not update the access times on these devices), so this macro is not useful in these cases, and we must use the *condor_kbdd* to get this information by connecting to the X server.

**STARTD_JOB_EXPRS** When the machine is claimed by a remote user, the *condor_startd* can also advertise arbitrary attributes from the job ClassAd in the machine ClassAd. List the attribute names to be advertised. NOTE: Since these are already ClassAd expressions, do not do anything unusual with strings. This setting defaults to "JobUniverse".

**STARTD_ATTRS** This macro is described in section 3.3.5 as `<SUBSYS>_ATTRS`.

**STARTD_DEBUG** This macro (and other settings related to debug logging in the *condor_startd*) is described in section 3.3.4 as `<SUBSYS>_DEBUG`.

**STARTD_ADDRESS_FILE** This macro is described in section 3.3.5 as `<SUBSYS>_ADDRESS_FILE`

**STARTD_SHOULD_WRITE_CLAIM_ID_FILE** Starting with version 6.7.10, the *condor_startd* can be configured to write out the `ClaimId` for the next available claim on all virtual machines to separate files. This boolean attribute controls whether the *condor_startd* should write these files. The default value is true.

**STARTD_CLAIM_ID_FILE** This macro controls what file names are used if the above `STARTD_SHOULD_WRITE_CLAIM_ID_FILE` is true. By default, Condor will write the ClaimId into a file in the `$(LOG)` directory called `.startd_claim_id.vmX`, where `X` is the value of `VirtualMachineID`, the integer that identifies a given virtual machine on

the system, or `1` on a single-VM machine. If you define your own value for this setting, you should provide a full path, and Condor will automatically append the `.vmX` portion of the file name.

**NUM_CPUS** This macro can be used to "lie" to the *condor_startd* about how many CPUs your machine has. If you set this, it will override Condor's automatic computation of the number of CPUs in your machine, and Condor will use whatever integer you specify here. In this way, you can allow multiple Condor jobs to run on a single-CPU machine by having that machine treated like an SMP machine with multiple CPUs, which could have different Condor jobs running on each one. Or, you can have an SMP machine advertise more virtual machines than it has CPUs. However, using this parameter will hurt the performance of the jobs, since you would now have multiple jobs running on the same CPU, competing with each other. The option is only meant for people who specifically want this behavior and know what they are doing. It is disabled by default.

NOTE: This setting cannot be changed with a simple reconfig (either by sending a SIGHUP or using *condor_reconfig*. If you change this, you must restart the *condor_startd* for the change to take effect (by using "condor_restart -startd").

NOTE: If you use this setting on a given machine, you should probably advertise that fact in the machine's ClassAd by using the `STARTD_ATTRS` setting (described above). This way, jobs submitted in your pool could specify that they did or did not want to be matched with machines that were only really offering "fractional CPUs".

**MAX_NUM_CPUS** This macro will cap the number of CPUs detected by Condor on a machine. If you set `NUM_CPUS` this cap is ignored. If it is set to zero, there is no cap. If it is not defined in the config file, it defaults to zero and there is no cap.

NOTE: This setting cannot be changed with a simple reconfig (either by sending a SIGHUP or using *condor_reconfig*. If you change this, you must restart the *condor_startd* for the change to take effect (by using "condor_restart -startd").

**COUNT_HYPERTHREAD_CPUS** This macro controls how Condor sees hyper threaded processors. When set to `True` (the default), it includes virtual CPUs in the default value of `NUM_CPUS`. On dedicated cluster nodes, counting virtual CPUs can sometimes improve total throughput at the expense of individual job speed. However, counting them on desktop workstations can interfere with interactive job performance.

**MEMORY** Normally, Condor will automatically detect the amount of physical memory available on your machine. Define `MEMORY` to tell Condor how much physical memory (in MB) your machine has, overriding the value Condor computes automatically.

**RESERVED_MEMORY** How much memory would you like reserved from Condor? By default, Condor considers all the physical memory of your machine as available to be used by Condor jobs. If `RESERVED_MEMORY` is defined, Condor subtracts it from the amount of memory it advertises as available.

**STARTD_NAME** Used to give an alternative name in the *condor_startd*'s class ad. This esoteric configuration macro might be used in the situation where there are two *condor_startd* daemons running on one machine, and each reports to the same *condor_collector*. Different names

will distinguish the two daemons. See the description of MASTER_NAME in section 3.3.9 on page 167 for a description of valid Condor daemon names.

**RUNBENCHMARKS** Specifies when to run benchmarks. When the machine is in the Unclaimed state and this expression evaluates to True, benchmarks will be run. If RunBenchmarks is specified and set to anything other than False, additional benchmarks will be run when the *condor_startd* initially starts. To disable startup benchmarks, set RunBenchmarks to False, or comment it out of the configuration file.

**DedicatedScheduler** A string that identifies the dedicated scheduler. See section 3.12.8 on page 355 for details.

**STARTD_NOCLAIM_SHUTDOWN** The number of seconds to run without receiving a claim before shutting Condor down on this machine. Defaults to unset, which means to never shut down. This is primarily intended for condor_glidein. Use in other situations is not recommended.

These macros control if the *condor_startd* daemon should perform backfill computations whenever resources would otherwise be idle. See section 3.12.9 on page 359 on Configuring Condor for Running Backfill Jobs for details.

**ENABLE_BACKFILL** A boolean value that, when True, indicates that the machine is willing to perform backfill computations when it would otherwise be idle. This is not a policy expression that is evaluated, it is a simple True or False. This setting controls if any of the other backfill-related expressions should be evaluated. The default is False.

**BACKFILL_SYSTEM** A string that defines what backfill system to use for spawning and managing backfill computations. Currently, the only supported value for this is "BOINC", which stands for the *Berkeley Open Infrastructure for Network Computing*. See http://boinc.berkeley.edu for more information about BOINC. There is no default value, administrators must define this.

**START_BACKFILL** A boolean expression that is evaluated whenever a Condor resource is in the Unclaimed/Idle state and the ENABLE_BACKFILL expression is True. If START_BACKFILL evaluates to True, the machine will enter the Backfill state and attempt to spawn a backfill computation. This expression is analogous to the START expression that controls when a Condor resource is available to run normal Condor jobs. The default value is False (which means do not spawn a backfill job even if the machine is idle and ENABLE_BACKFILL expression is True). For more information about policy expressions and the Backfill state, see section 3.5 beginning on page 221, especially sections 3.5.6, 3.5.7, and 3.5.8.

**EVICT_BACKFILL** A boolean expression that is evaluated whenever a Condor resource is in the Backfill state which, when True, indicates the machine should immediately kill the currently running backfill computation and return to the Owner state. This expression is a way for administrators to define a policy where interactive users on a machine will cause backfill jobs to be removed. The default value is False. For more information about policy expressions and the Backfill state, see section 3.5 beginning on page 221, especially sections 3.5.6, 3.5.7, and 3.5.8.

These macros only apply to the *condor_startd* daemon when it is running on an SMP machine. See section 3.12.7 on page 347 on Configuring The Startd for SMP Machines for details.

**STARTD_RESOURCE_PREFIX** A string which specifies what prefix to give the unique Condor resources that are advertised on SMP machines. Currently, Condor uses the term *virtual machine* to describe these resources, so the default value for this setting is "vm". However, to avoid confusion with other kinds of virtual machines (the ones created using tools like VMware or Xen), in a future version of Condor, the current *virtual machine* terminology will change. To aid in the transition to the new terms (and to provide a means for sites that want to keep using "vm" even after the default has changed), this setting enables sites to define what string the *condor_startd* will use to name the individual resources on an SMP machine.

**VIRTUAL_MACHINES_CONNECTED_TO_CONSOLE** An integer which indicates how many of the virtual machines the *condor_startd* is representing should be "connected" to the console (in other words, notice when there's console activity). This defaults to all virtual machines (N in a machine with N CPUs).

**VIRTUAL_MACHINES_CONNECTED_TO_KEYBOARD** An integer which indicates how many of the virtual machines the *condor_startd* is representing should be "connected" to the keyboard (for remote tty activity, as well as console activity). Defaults to 1.

**DISCONNECTED_KEYBOARD_IDLE_BOOST** If there are virtual machines not connected to either the keyboard or the console, the corresponding idle time reported will be the time since the *condor_startd* was spawned, plus the value of this macro. It defaults to 1200 seconds (20 minutes). We do this because if the virtual machine is configured not to care about keyboard activity, we want it to be available to Condor jobs as soon as the *condor_startd* starts up, instead of having to wait for 15 minutes or more (which is the default time a machine must be idle before Condor will start a job). If you do not want this boost, set the value to 0. If you change your START expression to require more than 15 minutes before a job starts, but you still want jobs to start right away on some of your SMP nodes, increase this macro's value.

**STARTD_VM_EXPRS** The list of ClassAd attribute names that should be shared across all virtual machines on the same machine. For each attribute in the list, the attribute's value is taken from each virtual machine's machine ClassAd and placed into the machine ClassAd of all the other virtual machines within the machine. For example, if the configuration file for a 2-VM machine contains

```
STARTD_VM_EXPRS = State, Activity, EnteredCurrentActivity
```

then the machine ClassAd for both virtual machines will contain attributes that will be of the form:

```
vm1_State = "Claimed"
vm1_Activity = "Busy"
vm1_EnteredCurrentActivity = 1075249233
vm2_State = "Unclaimed"
vm2_Activity = "Idle"
vm2_EnteredCurrentActivity = 1075240035
```

The following settings control the number of virtual machines reported for a given SMP host, and what attributes each one has. They are only needed if you do not want to have an SMP machine report to Condor with a separate virtual machine for each CPU, with all shared system resources evenly divided among them. Please read section 3.12.7 on page 348 for details on how to properly configure these settings to suit your needs.

NOTE: You can only change the number of each type of virtual machine the *condor_startd* is reporting with a simple reconfig (such as sending a SIGHUP signal, or using the *condor_reconfig* command). You cannot change the definition of the different virtual machine types with a reconfig. If you change them, you must restart the *condor_startd* for the change to take effect (for example, using `condor_restart -startd`).

**MAX_VIRTUAL_MACHINE_TYPES** The maximum number of different virtual machine types. Note: this is the maximum number of different *types*, not of actual virtual machines. Defaults to 10. (You should only need to change this setting if you define more than 10 separate virtual machine types, which would be pretty rare.)

**VIRTUAL_MACHINE_TYPE_<N>** This setting defines a given virtual machine type, by specifying what part of each shared system resource (like RAM, swap space, etc) this kind of virtual machine gets. N can be any integer from 1 to the value of `$(MAX_VIRTUAL_MACHINE_TYPES)`, such as `VIRTUAL_MACHINE_TYPE_1`. The format of this entry can be somewhat complex, so please refer to section 3.12.7 on page 348 for details on the different possibilities.

**NUM_VIRTUAL_MACHINES_TYPE_<N>** This macro controls how many of a given virtual machine type are actually reported to Condor. There is no default.

**NUM_VIRTUAL_MACHINES** If your SMP machine is being evenly divided, and the virtual machine type settings described above are not being used, this macro controls how many virtual machines will be reported. The default is one virtual machine for each CPU. This setting can be used to reserve some CPUs on an SMP which would not be reported to the Condor pool.

The following macros describe the *cron* capabilities of Condor. The *cron* mechanism is used to run executables (called modules) directly from the *condor_startd* daemon. The output from modules is incorporated into the machine ClassAd generated by the *condor_startd*. These capabilities are used in Hawkeye, but can be used in other situations as well.

These configuration macros are divided into three sets. The three sets occurred as the functionality and usage of Condor's *cron* capabilities evolved. The first set applies to both new and older macros and syntax. The second set applies to the new macros and syntax. The third set applies only to the older (and outdated) macros and syntax.

This first set of configuration macros applies to both new and older macros and syntax.

**STARTD_CRON_NAME** Defines a logical name to be used in the formation of related configuration macro names. While not required, this macro makes other macros more readable and maintainable. A common example is

```
STARTD_CRON_NAME = HAWKEYE
```

This example allows the naming of other related macros to contain the string "HAWKEYE" in their name.

**STARTD_CRON_CONFIG_VAL** This configuration variable can be used to specify the *condor_config_val* program which the modules (jobs) should use to get configuration information from the daemon. If this is provided, a environment variable by the same name with the same value will be passed to all modules.

If STARTD_CRON_NAME is defined, then this configuration macro name is changed from STARTD_CRON_CONFIG_VAL to $(STARTD_CRON_NAME)_CONFIG_VAL. Example:

```
HAWKEYE_CONFIG_VAL = /usr/local/condor/bin/condor_config_val
```

**STARTD_CRON_AUTOPUBLISH** Optional setting that determines if the *condor_startd* should automatically publish a new update to the *condor_collector* after any of the cron modules produce output. Beware that enabling this setting can greatly increase the network traffic in a Condor pool, especially when many modules are executed, or if the period in which they run is short. There are three possible (case insensitive) values for this setting:

**Never** This default value causes the *condor_startd* to not automatically publish updates based on any cron modules. Instead, updates rely on the usual behavior for sending updates, which is periodic, based on the UPDATE_INTERVAL configuration setting, or whenever a given virtual machine changes state.

**Always** Causes the *condor_startd* to always send a new update to the *condor_collector* whenever any module exits.

**If_Changed** Causes the *condor_startd* to only send a new update to the *condor_collector* if the output produced by a given module is different than the previous output of the same module. The only exception is the LastUpdate attribute (automatically set for all cron modules to be the timestamp when the module last ran), which is ignored when STARTD_CRON_AUTOPUBLISH is set to If_Changed.

Beware that STARTD_CRON_AUTOPUBLISH does not honor the STARTD_CRON_NAME setting described above. Even if STARTD_CRON_NAME is defined, STARTD_CRON_AUTOPUBLISH will have the same name.

The following second set of configuration macros applies only to the new macros and syntax. This set is to be used for all new applications.

**STARTD_CRON_JOBLIST** This configuration variable is defined by a whitespace separated list of job names (called modules) to run. Each of these is the logical name of the module. This name must be unique (no two modules may have the same name).

If STARTD_CRON_NAME is defined, then this configuration macro name is changed from STARTD_CRON_JOBLIST to $(STARTD_CRON_NAME)_JOBLIST.

**STARTD_CRON_<ModuleName>_PREFIX** Specifies a string which is prepended by Condor to all attribute names that the module generates. For example, if a prefix is "xyz_", and an individual attribute is named "abc", the resulting attribute would be "xyz_abc". Although it can be quoted, the prefix can contain only alpha-numeric characters.

If STARTD_CRON_NAME is defined, then this configuration macro name is changed from STARTD_CRON_<ModuleName>_PREFIX to $(STARTD_CRON_NAME)_<ModuleName>_PREFIX.

**STARTD_CRON_<ModuleName>_EXECUTABLE** Used to specify the full path to the executable to run for this module. Note that multiple modules may specify the same executable (although they need to have different names).

If STARTD_CRON_NAME is defined, then this configuration macro name is changed from STARTD_CRON_<ModuleName>_EXECUTABLE to $(STARTD_CRON_NAME)_<ModuleName>_EXECUTABLE.

**STARTD_CRON_<ModuleName>_PERIOD** The period specifies time intervals at which the module should be run. For periodic modules, this is the time interval that passes between starting the execution of the module. The value may be specified in seconds (append value with the character 's'), in minutes (append value with the character 'm'), or in hours (append value with the character 'h'). As an example, 5m starts the execution of the module every five minutes. If no character is appended to the value, seconds are used as a default. For "Wait For Exit" mode, the value has a different meaning; in this case the period specifies the length of time after the module ceases execution before it is restarted. The minimum valid value of the period is 1 second.

If STARTD_CRON_NAME is defined, then this configuration macro name is changed from STARTD_CRON_<ModuleName>_PERIOD to $(STARTD_CRON_NAME)_<ModuleName>_PERIOD.

**STARTD_CRON_<ModuleName>_MODE** Used to specify the "Mode" in which the module operates. Legal values are "WaitForExit" and "Periodic" (the default).

If STARTD_CRON_NAME is defined, then this configuration macro name is changed from STARTD_CRON_<ModuleName>_MODE to $(STARTD_CRON_NAME)_<ModuleName>_MODE.

The default "Periodic" mode is used for most modules. In this mode, the module is expected to be started by the *condor_startd* daemon, gather and publish its data, and then exit.

The "WaitForExit" mode is used to specify a module which runs in the "Wait For Exit" mode. In this mode, the *condor_startd* daemon interprets the "period" differently. In this case, it refers to the amount of time to wait after the module exits before restarting it. With a value of 1, the module is kept running nearly continuously.

In general, "Wait For Exit" mode is for modules that produce a periodic stream of updated data, but it can be used for other purposes, as well.

**STARTD_CRON_<ModuleName>_RECONFIG** The "ReConfig" macro is used to specify whether a module can handle HUP signals, and should be sent a HUP signal when the *condor_startd*

daemon is reconfigured. The module is expected to reread its configuration at that time. A value of "True" enables this setting, and "False" disables it.

If STARTD_CRON_NAME is defined, then this configuration macro name is changed from STARTD_CRON_<ModuleName>_RECONFIG to $(STARTD_CRON_NAME)_<ModuleName>_RECONFIG.

**STARTD_CRON_<ModuleName>_KILL** The "Kill" macro is applicable on for modules running in the "Periodic" mode. Possible values are "True" and "False" (the default).

If STARTD_CRON_NAME is defined, then this configuration macro name is changed from STARTD_CRON_<ModuleName>_KILL to $(STARTD_CRON_NAME)_<ModuleName>_KILL.

This macro controls the behavior of the *condor_startd* when it detects that the module's executable is still running when it is time to start the module for a run. If enabled, the *condor_startd* will kill and restart the process in this condition. If not enabled, the existing process is allowed to continue running.

**STARTD_CRON_<ModuleName>_ARGS** The command line arguments to pass to the module to be executed.

If STARTD_CRON_NAME is defined, then this configuration macro name is changed from STARTD_CRON_<ModuleName>_ARGS to $(STARTD_CRON_NAME)_<ModuleName>_ARGS.

**STARTD_CRON_<ModuleName>_ENV** The environment string to pass to the module. The syntax is the same as that of DAEMONNAME_ENVIRONMENT in 3.3.9.

If STARTD_CRON_NAME is defined, then this configuration macro name is changed from STARTD_CRON_<ModuleName>_ENV to $(STARTD_CRON_NAME)_<ModuleName>_ENV.

**STARTD_CRON_<ModuleName>_CWD** The working directory in which to start the module.

If STARTD_CRON_NAME is defined, then this configuration macro name is changed from STARTD_CRON_<ModuleName>_CWD to $(STARTD_CRON_NAME)_<ModuleName>_CWD.

**STARTD_CRON_<ModuleName>_OPTIONS** A colon separated list of options. Not all combinations of options make sense; when a nonsense combination is listed, the last one in the list is followed.

If STARTD_CRON_NAME is defined, then this configuration macro name is changed from STARTD_CRON_<ModuleName>_OPTIONS to $(STARTD_CRON_NAME)_<ModuleName>_OPTIONS.

- The "WaitForExit" option enables the "Wait For Exit" mode (see above).
- The "ReConfig" option enables the "Reconfig" setting (see above).
- The "NoReConfig" option disables the "Reconfig" setting (see above).
- The "Kill" option enables the "Kill" setting (see above).
- The "NoKill" option disables the "Kill" setting (see above).

Here is a complete configuration example that uses Hawkeye.

```
# Hawkeye Job Definitions
STARTD_CRON_NAME = HAWKEYE

# Job 1
HAWKEYE_JOBLIST = job1
HAWKEYE_job1_PREFIX = prefix_
HAWKEYE_job1_EXECUTABLE = $(MODULES)/job1
HAWKEYE_job1_PERIOD = 5m
HAWKEYE_job1_MODE = WaitForExit
HAWKEYE_job1_KILL = false
HAWKEYE_job1_ARGS =-foo -bar
HAWKEYE_job1_ENV = xyzzy=somevalue

# Job 2
HAWKEYE_JOBLIST = $(HAWKEYE_JOBLIST) job2
HAWKEYE_job2_PREFIX = prefix_
HAWKEYE_job2_EXECUTABLE = $(MODULES)/job2
HAWKEYE_job2_PERIOD = 1h
HAWKEYE_job2_ENV = lwpi=somevalue
```

The following third set of configuration macros applies only to older macros and syntax. This set is documented for completeness and backwards compatibility. Do not use these configuration macros for any new application. Future releases of Condor may disable the use of this set.

**STARTD_CRON_JOBS** The list of the modules to execute. In Hawkeye, this is usually named HAWKEYE_JOBS. This configuration variable is defined by a whitespace or newline separated list of jobs (called modules) to run, where each module is specified using the format

```
modulename:prefix:executable:period[:options]
```

Each of these fields can be surrounded by matching quote characters (single quote or double quote, but they must match). This allows colon and whitespace characters to be specified. For example, the following specifies an executable name with a colon and a space in it:

```
foo:foo_:"c:/some dir/foo.exe":10m
```

These individual fields are described below:

- `modulename` The logical name of the module. This must be unique (no two modules may have the same name). See `STARTD_CRON_JOBLIST`
- `prefix` See `STARTD_CRON_<ModuleName>_PREFIX`
- `executable` See `STARTD_CRON_<ModuleName>_EXECUTABLE`
- `period` See `STARTD_CRON_<ModuleName>_PERIOD`

- Several options are available. Using more than one of these options for one module does not make sense. If this happens, the last one in the list is followed. See STARTD_CRON_<ModuleName>_OPTIONS
  - The “Continuous” option is used to specify a module which runs in continuous mode (as described above). See the “WaitForExit” and “ReConfig” options which replace “Continuous”.
    This option is now deprecated, and its functionality has been replaced by the new “WaitForExit” and “ReConfig” options, which together implement the capabilities of “Continuous”. This option will be removed from a future version of Condor.
  - The “WaitForExit” option
    See the discussion of “WaitForExit” in STARTD_CRON_<ModuleName>_OPTIONS above.
  - The “ReConfig” option
    See the discussion of “ReConfig in STARTD_CRON_<ModuleName>_OPTIONS above.
  - The ‘NoReConfig” option
    See the discussion of “NoReConfig in STARTD_CRON_<ModuleName>_OPTIONS above.
  - The “Kill” option
    See the discussion of “Kill” in STARTD_CRON_<ModuleName>_OPTIONS above.
  - The “NoKill” option
    See the discussion of “NoKill” in STARTD_CRON_<ModuleName>_OPTIONS above.

NOTE: The configuration file parsing logic will strip whitespace from the beginning and end of continuation lines. Thus, a job list like below will be misinterpreted and will not work as expected:

```
# Hawkeye Job Definitions
HAWKEYE_JOBS   =\
    JOB1:prefix_:$(MODULES)/job1:5m:nokill\
    JOB2:prefix_:$(MODULES)/job1_co:1h
HAWKEYE_JOB1_ARGS =-foo -bar
HAWKEYE_JOB1_ENV = xyzzy=somevalue
HAWKEYE_JOB2_ENV = lwpi=somevalue
```

Instead, write this as below:

```
# Hawkeye Job Definitions
HAWKEYE_JOBS   =

# Job 1
HAWKEYE_JOBS = $(HAWKEYE_JOBS) JOB1:prefix_:$(MODULES)/job1:5m:nokill
HAWKEYE_JOB1_ARGS =-foo -bar
```

```
HAWKEYE_JOB1_ENV = xyzzy=somevalue

# Job 2
HAWKEYE_JOBS = $(HAWKEYE_JOBS) JOB2:prefix_:$(MODULES)/job2:1h
HAWKEYE_JOB2_ENV = lwpi=somevalue
```

The following macros control the optional computation of resource availability statistics in the *condor_startd*.

**STARTD_COMPUTE_AVAIL_STATS** A boolean that determines if the *condor_startd* computes resource availability statistics. The default is False.

If STARTD_COMPUTE_AVAIL_STATS = True, the *condor_startd* will define the following ClassAd attributes for resources:

**AvailTime** The proportion of the time (between 0.0 and 1.0) that this resource has been in a state other than Owner.

**LastAvailInterval** The duration (in seconds) of the last period between Owner states.

The following attributes will also be included if the resource is not in the Owner state:

**AvailSince** The time at which the resource last left the Owner state. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

**AvailTimeEstimate** Based on past history, an estimate of how long the current period between Owner states will last.

**STARTD_AVAIL_CONFIDENCE** A floating point number representing the confidence level of the *condor_startd* daemon's AvailTime estimate. By default, the estimate is based on the 80th percentile of past values (that is, the value is initially set to 0.8).

**STARTD_MAX_AVAIL_PERIOD_SAMPLES** An integer that limits the number of samples of past available intervals stored by the *condor_startd* to limit memory and disk consumption. Each sample requires 4 bytes of memory and approximately 10 bytes of disk space.

The following configuration variables support java universe jobs.

**JAVA** The full path to the Java interpreter (the Java Virtual Machine).

**JAVA_MAXHEAP_ARGUMENT** An incomplete command line argument to the Java interpreter (the Java Virtual Machine) to specify the switch name for the Maxheap Argument. Condor uses it to construct the maximum heap size for the Java Virtual Machine. For example, the value for the Sun JVM is *-Xmx*.

**JAVA_CLASSPATH_ARGUMENT** The command line argument to the Java interpreter (the Java Virtual Machine) that specifies the Java Classpath. Classpath is a Java-specific term that denotes the list of locations (`.jar` files and/or directories) where the Java interpreter can look for the Java class files that a Java program requires.

**JAVA_CLASSPATH_SEPARATOR** The single character used to delimit constructed entries in the Classpath for the given operating system and Java Virtual Machine. If not defined, the operating system is queried for its default Classpath separator.

**JAVA_CLASSPATH_DEFAULT** A list of path names to `.jar` files to be added to the Java Classpath by default. The comma and/or space character delimits list entries.

**JAVA_EXTRA_ARGUMENTS** A list of additional arguments to be passed to the Java executable.

### 3.3.11 condor_schedd Configuration File Entries

These macros control the *condor_schedd*.

**SHADOW** This macro determines the full path of the *condor_shadow* binary that the *condor_schedd* spawns. It is normally defined in terms of `$(SBIN)`.

**SHADOW_PVM** This macro determines the full path of the special *condor_shadow.pvm* binary used for supporting PVM jobs that the *condor_schedd* spawns. It is normally defined in terms of `$(SBIN)`.

**START_LOCAL_UNIVERSE** A boolean value that defaults to `True`. The *condor_schedd* uses this macro to determine whether to start a **local** universe job. At intervals determined by `SCHEDD_INTERVAL`, the *condor_schedd* daemon evaluates this macro for each idle **local** universe job that it has. For each job, if the `START_LOCAL_UNIVERSE` macro is `True`, then the job's `Requirements` expression is evaluated. If both conditions are met, then the job is allowed to begin execution.

The following example only allows 10 **local** universe jobs to execute concurrently. The attribute `TotalLocalJobsRunning` is supplied by *condor_schedd*'s ClassAd:

```
START_LOCAL_UNIVERSE = TotalLocalJobsRunning < 10
```

**START_SCHEDULER_UNIVERSE** A boolean value that defaults to `True`. The *condor_schedd* uses this macro to determine whether to start a **scheduler** universe job. At intervals determined by `SCHEDD_INTERVAL`, the *condor_schedd* daemon evaluates this macro for each idle **scheduler** universe job that it has. For each job, if the `START_SCHEDULER_UNIVERSE` macro is `True`, then the job's `Requirements` expression is evaluated. If both conditions are met, then the job is allowed to begin execution.

The following example only allows 10 **scheduler** universe jobs to execute concurrently. The attribute `TotalSchedulerJobsRunning` is supplied by *condor_schedd*'s ClassAd:

```
START_SCHEDULER_UNIVERSE = TotalSchedulerJobsRunning < 10
```

**MAX_JOBS_RUNNING** This macro limits the number of processes spawned by a given *condor_schedd*, for all job universes except the grid universe. See section 2.4.1. This includes, but is not limited to *condor_shadow* processes, and scheduler universe processes, including *condor_dagman*. The actual number of *condor_shadow*s may be less if you have reached your $(RESERVED_SWAP) limit. This macro has a default value of 200.

**MAX_SHADOW_EXCEPTIONS** This macro controls the maximum number of times that *condor_shadow* processes can have a fatal error (exception) before the *condor_schedd* will relinquish the match associated with the dying shadow. Defaults to 5.

**SCHEDD_INTERVAL** This macro determines both how often the *condor_schedd* sends a ClassAd update to the *condor_collector* and how often the *condor_schedd* daemon evaluates jobs. It is defined in terms of seconds and defaults to 300 (every 5 minutes).

**JOB_START_COUNT** This macro works together with the JOB_START_DELAY macro to throttle job starts. The default and minimum values for this integer configuration variable are both 1. Small values for this macro are preferred and will minimize the load upon the *condor_schedd* daemon.

**JOB_START_DELAY** This integer-valued macro works together with the JOB_START_COUNT macro to throttle job starts. The *condor_schedd* daemon starts $(JOB_START_COUNT) jobs at a time, then delays for $(JOB_START_DELAY) seconds before starting the next set of jobs. This delay prevents a sudden, large load on the submit machine as it spawns many *condor_shadow* daemons simultaneously, and it prevents having to deal with their start up activity all at once. The resulting job start rate averages as fast as ($(JOB_START_COUNT)/$(JOB_START_DELAY)) jobs/second. This configuration variable is also used during the graceful shutdown of the *condor_schedd* daemon. During graceful shutdown, this macro determines the wait time in between requesting each *condor_shadow* daemon to gracefully shut down. It is defined in terms of seconds and defaults to 2. Setting this macro to a lower value is not advised, as it can overwhelm the *condor_schedd* daemon.

**JOB_IS_FINISHED_INTERVAL** The *condor_schedd* maintains a list of jobs that are ready to permanently leave the job queue, e.g. they have completed or been removed. This integer-valued macro specifies a delay in seconds to place between the taking jobs permanently out of the queue. The default value is 0, which tells the *condor_schedd* to not impose any delay.

**ALIVE_INTERVAL** This macro determines how often the *condor_schedd* should send a keep alive message to any *condor_startd* it has claimed. When the *condor_schedd* claims a *condor_startd*, it tells the *condor_startd* how often it is going to send these messages. If the *condor_startd* does not receive any of these keep alive messages during a certain period of time (defined via MAX_CLAIM_ALIVES_MISSED , described on page 170) the *condor_startd* releases the claim, and the *condor_schedd* no longer pays for the resource (in terms of user priority in the system). The macro is defined in terms of seconds and defaults to 300 (every 5 minutes).

**REQUEST_CLAIM_TIMEOUT** This macro sets the time (in seconds) that the *condor_schedd* will wait for a claim to be granted by the *condor_startd*. The default is 30 minutes. This is only likely to matter if the *condor_startd* has an existing claim and it takes a long time for the

existing claim to be preempted due to `MaxJobRetirementTime`. Once a request times out, the *condor_schedd* will simply begin the process of finding a machine for the job all over again.

**SHADOW_SIZE_ESTIMATE** This macro sets the estimated virtual memory size of each *condor_shadow* process. Specified in kilobytes. The default varies from platform to platform.

**SHADOW_RENICE_INCREMENT** When the *condor_schedd* spawns a new *condor_shadow*, it can do so with a *nice-level*. A nice-level is a Unix mechanism that allows users to assign their own processes a lower priority so that the processes do not interfere with interactive use of the machine. This is very handy for keeping a submit machine with lots of shadows running still useful to the owner of the machine. The value can be any integer between 0 and 19, with a value of 19 being the lowest priority. It defaults to 10.

**SCHED_UNIV_RENICE_INCREMENT** Analogous to `JOB_RENICE_INCREMENT` and `SHADOW_RENICE_INCREMENT`, scheduler universe jobs can be given a nice-level. See `JOB_RENICE_INCREMENT` in section 3.3.14 for further explanation; default value and range of legal values are the same.

**QUEUE_CLEAN_INTERVAL** The *condor_schedd* maintains the job queue on a given machine. It does so in a persistent way such that if the *condor_schedd* crashes, it can recover a valid state of the job queue. The mechanism it uses is a transaction-based log file (the `job_queue.log` file, not the `SchedLog` file). This file contains an initial state of the job queue, and a series of transactions that were performed on the queue (such as new jobs submitted, jobs completing, and checkpointing). Periodically, the *condor_schedd* will go through this log, truncate all the transactions and create a new file with containing only the new initial state of the log. This is a somewhat expensive operation, but it speeds up when the *condor_schedd* restarts since there are fewer transactions it has to play to figure out what state the job queue is really in. This macro determines how often the *condor_schedd* should rework this queue to cleaning it up. It is defined in terms of seconds and defaults to 86400 (once a day).

**WALL_CLOCK_CKPT_INTERVAL** The job queue contains a counter for each job's "wall clock" run time, i.e., how long each job has executed so far. This counter is displayed by *condor_q*. The counter is updated when the job is evicted or when the job completes. When the *condor_schedd* crashes, the run time for jobs that are currently running will not be added to the counter (and so, the run time counter may become smaller than the CPU time counter). The *condor_schedd* saves run time "checkpoints" periodically for running jobs so if the *condor_schedd* crashes, only run time since the last checkpoint is lost. This macro controls how often the *condor_schedd* saves run time checkpoints. It is defined in terms of seconds and defaults to 3600 (one hour). A value of 0 will disable wall clock checkpoints.

**ALLOW_REMOTE_SUBMIT** Starting with Condor Version 6.0, users can run *condor_submit* on one machine and actually submit jobs to another machine in the pool. This is called a *remote submit*. Jobs submitted in this way are entered into the job queue owned by the Unix user nobody. This macro determines whether this is allowed. It defaults to `False`.

**QUEUE_ALL_USERS_TRUSTED** . Defaults to False. If set to True, then unauthenticated users are allowed to write to the queue, and also we always trust whatever the `Owner` value is set to be

by the client in the job ad. This was added so users can continue to use the SOAP web-services interface over HTTP (w/o authenticating) to submit jobs in a secure, controlled environment – for instance, in a portal setting.

**QUEUE_SUPER_USERS** This macro determines what user names on a given machine have *super-user access* to the job queue, meaning that they can modify or delete the job ClassAds of other users. (Normally, you can only modify or delete ClassAds from the job queue that you own). Whatever user name corresponds with the UID that Condor is running as (usually the Unix user condor) will automatically be included in this list because that is needed for Condor's proper functioning. See section 3.6.12 on UIDs in Condor for more details on this. By default, we give root the ability to remove other user's jobs, in addition to user condor.

**SCHEDD_LOCK** This macro specifies what lock file should be used for access to the SchedLog file. It must be a separate file from the SchedLog, since the SchedLog may be rotated and synchronization across log file rotations is desired. This macro is defined relative to the $(LOCK) macro. If you decide to change this setting (not recommended), be sure to change the $(VALID_LOG_FILES) entry that *condor_preen* uses as well.

**SCHEDD_NAME** A unique name given for a *condor_schedd* daemon on a machine. Defaults to the fully qualified hostname of the machine where the *condor_schedd* is running. However, this configuration macro is used to uniquely identify *condor_schedd* ClassAds if more than one *condor_schedd* is running on the same host, for example, with many Personal Condor installations running as different users on the same machine. In that case, the recommended form for SCHEDD_NAME is username@full.host.name, where username is the user that a given *condor_schedd* is running as.

See the description of MASTER_NAME in section 3.3.9 on page 167 for a description of valid Condor daemon names. Also, note that if the MASTER_NAME setting is defined for the *condor_master* that spawned a given *condor_schedd*, that name will take precedence over whatever is defined in SCHEDD_NAME.

**SCHEDD_ATTRS** This macro is described in section 3.3.5 as <SUBSYS>_ATTRS.

**SCHEDD_DEBUG** This macro (and other settings related to debug logging in the *condor_schedd*) is described in section 3.3.4 as <SUBSYS>_DEBUG.

**SCHEDD_ADDRESS_FILE** This macro is described in section 3.3.5 as <SUBSYS>_ADDRESS_FILE.

**SCHEDD_EXECUTE** A directory to use as a temporary sandbox for local universe jobs. Defaults to $(SPOOL)/execute.

**FLOCK_NEGOTIATOR_HOSTS** This macro defines a list of negotiator host names (not including the local $(NEGOTIATOR_HOST) machine) for pools in which the *condor_schedd* should attempt to run jobs. Hosts in the list should be in order of preference. The *condor_schedd* will only send a request to a central manager in the list if the local pool and pools earlier in the list are not satisfying all the job requests. $(HOSTALLOW_NEGOTIATOR_SCHEDD) (see section 3.3.5) must also be configured to allow negotiators from all of the $(FLOCK_NEGOTIATOR_HOSTS) to contact the *condor_schedd*. Please make sure the

$(NEGOTIATOR_HOST) is first in the $(HOSTALLOW_NEGOTIATOR_SCHEDD) list. Similarly, the central managers of the remote pools must be configured to listen to requests from this *condor_schedd*.

**FLOCK_COLLECTOR_HOSTS** This macro defines a list of collector host names for pools in which the *condor_schedd* should attempt to run jobs. The collectors must be specified in order, corresponding to the $(FLOCK_NEGOTIATOR_HOSTS) list. In the typical case, where each pool has the collector and negotiator running on the same machine, $(FLOCK_COLLECTOR_HOSTS) should have the same definition as $(FLOCK_NEGOTIATOR_HOSTS).

**NEGOTIATE_ALL_JOBS_IN_CLUSTER** If this macro is set to False (the default), when the *condor_schedd* fails to start an idle job, it will not try to start any other idle jobs in the same cluster during that negotiation cycle. This makes negotiation much more efficient for large job clusters. However, in some cases other jobs in the cluster can be started even though an earlier job can't. For example, the jobs' requirements may differ, because of different disk space, memory, or operating system requirements. Or, machines may be willing to run only some jobs in the cluster, because their requirements reference the jobs' virtual memory size or other attribute. Setting this macro to True will force the *condor_schedd* to try to start all idle jobs in each negotiation cycle. This will make negotiation cycles last longer, but it will ensure that all jobs that can be started will be started.

**PERIODIC_EXPR_INTERVAL** This macro determines the period, in seconds, between evaluation of periodic job control expressions, such as periodic_hold, periodic_release, and periodic_remove, given by the user in a Condor submit file. By default, this value is 300 seconds (5 minutes). A value of 0 prevents the *condor_schedd* from performing the periodic evaluations.

**SCHEDD_ASSUME_NEGOTIATOR_GONE** This macro determines the period, in seconds, that the *condor_schedd* will wait for the *condor_negotiator* to initiate a negotiation cycle before the schedd will simply try to claim any local *condor_startd*. This allows for a machine that is acting as both a submit and execute node to run jobs locally if it cannot communicate with the central manager. The default value, if not specified, is 4 x $(NEGOTIATOR_INTERVAL). If $(NEGOTIATOR_INTERVAL) is not defined, then SCHEDD_ASSUME_NEGOTIATOR_GONE will default to 1200 (20 minutes).

**SCHEDD_ROUND_ATTR_<xxxx>** An integer-valued macro that instructs the *condor_schedd* to round up scalar job ClassAd attributes to a specific number of decimal places. This is desirable to improve job scheduling performance, since it allows Condor's auto-clustering algorithm to bucket attributes such as ImageSize. Replace <xxxx> with the name of the attribute to round, and set this macro equal to the number of decimal places to round up. For example, to round the value of job ClassAd attribute foo up to the nearest 100, set

```
SCHEDD_ROUND_ATTR_foo = 2
```

When the schedd rounds up an attribute value, it will save the raw (un-rounded) actual value in an attribute with the same name appended with "_RAW". So in the above example, the raw value will be stored in attribute foo_RAW in the job ClassAd. The following are set by default:

```
        SCHEDD_ROUND_ATTR_ImageSize = 4
        SCHEDD_ROUND_ATTR_ExecutableSize = 4
        SCHEDD_ROUND_ATTR_DiskUsage = 4
SCHEDD_ROUND_ATTR_NumCkpts = 4
```

Thus, the above attributes are all rounded up from Kbytes to the nearest 10 Megabytes.

**SCHEDD_BACKUP_SPOOL** This macro is used to enable the *condor_schedd* to make a backup of the job queue as it starts. If set to "True", the *condor_schedd* will create host specific a backup of the current spool file to the spool directory. This backup file will be overwritten each time the *condor_schedd* starts. SCHEDD_BACKUP_SPOOL defaults to "False".

**MPI_CONDOR_RSH_PATH** The complete path to the special version of *rsh* that is required to spawn MPI jobs under Condor. $(LIBEXEC) is the proper value for this configuration variable, required when running MPI dedicated jobs.

**SCHEDD_PREEMPTION_REQUIREMENTS** This boolean expression is utilized only for machines allocated by a dedicated scheduler. When True, a machine becomes a candidate for job preemption. This configuration variable has no default; when not defined, preemption will never be considered.

**SCHEDD_PREEMPTION_RANK** This floating point value is utilized only for machines allocated by a dedicated scheduler. It is evaluated in context of a job ClassAd, and it represents a machine's preference for running a job. This configuration variable has no default; when not defined, preemption will never be considered.

**ParallelSchedulingGroup** For parallel jobs which must be assigned within a group of machines (and not cross group boundaries), this configuration variable identifies members of a group. Each machine within a group sets this configuration variable with a string that identifies the group.

### 3.3.12 condor_shadow Configuration File Entries

These settings affect the *condor_shadow*.

**SHADOW_LOCK** This macro specifies the lock file to be used for access to the ShadowLog file. It must be a separate file from the ShadowLog, since the ShadowLog may be rotated and you want to synchronize access across log file rotations. This macro is defined relative to the $(LOCK) macro. If you decide to change this setting (not recommended), be sure to change the $(VALID_LOG_FILES) entry that *condor_preen* uses as well.

**SHADOW_DEBUG** This macro (and other settings related to debug logging in the shadow) is described in section 3.3.4 as <SUBSYS>_DEBUG.

**SHADOW_QUEUE_UPDATE_INTERVAL** The amount of time (in seconds) between ClassAd updates that the *condor_shadow* daemon sends to the *condor_schedd* daemon. Defaults to 900 (15 minutes).

**COMPRESS_PERIODIC_CKPT** This boolean macro specifies whether the shadow should instruct applications to compress periodic checkpoints (when possible). The default is `False`.

**COMPRESS_VACATE_CKPT** This boolean macro specifies whether the shadow should instruct applications to compress vacate checkpoints (when possible). The default is `False`.

**PERIODIC_MEMORY_SYNC** This boolean macro specifies whether the shadow should instruct applications to commit dirty memory pages to swap space during a periodic checkpoint. The default is `False`. This potentially reduces the number of dirty memory pages at vacate time, thereby reducing swapping activity on the remote machine.

**SLOW_CKPT_SPEED** This macro specifies the speed at which vacate checkpoints should be written, in kilobytes per second. If zero (the default), vacate checkpoints are written as fast as possible. Writing vacate checkpoints slowly can avoid overwhelming the remote machine with swapping activity.

### 3.3.13 condor_shadow.pvm Configuration File Entries

These macros control the *condor_shadow.pvm*, the special shadow that supports PVM jobs inside Condor. See section **??** on Installing PVM Support in Condor for details. *condor_shadow* macros also apply to this special shadow. See section 3.3.12.

**PVMD** This macro holds the full path to the special *condor_pvmd*, the Condor PVM daemon. This daemon is installed in the regular Condor release directory by default, so the macro is usually defined in terms of `$(SBIN)`.

**PVMGS** This macro holds the full path to the special *condor_pvmgs*, the Condor PVM Group Server daemon, which is needed to support PVM groups. This daemon is installed in the regular Condor release directory by default, so the macro is usually defined in terms of `$(SBIN)`.

### 3.3.14 condor_starter Configuration File Entries

These settings affect the *condor_starter*.

**EXEC_TRANSFER_ATTEMPTS** Sometimes due to a router misconfiguration, kernel bug, or other Act of God network problem, the transfer of the initial checkpoint from the submit machine to the execute machine will fail midway through. This parameter allows a retry of the transfer a certain number of times that must be equal to or greater than 1. If this parameter is not specified, or specified incorrectly, then it will default to three. If the transfer of the initial executable fails every attempt, then the job goes back into the idle state until the next renegotiation cycle.

NOTE: : This parameter does not exist in the NT starter.

**JOB_RENICE_INCREMENT** When the *condor_starter* spawns a Condor job, it can do so with a *nice-level*. A nice-level is a Unix mechanism that allows users to assign their own processes a

lower priority, such that these processes do not interfere with interactive use of the machine. For machines with lots of real memory and swap space, such that the only scarce resource is CPU time, use this macro in conjunction with a policy that allows Condor to always start jobs on the machines. Condor jobs would always run, but interactive response on the machines would never suffer. A user most likely will not notice Condor is running jobs. See section 3.5 on Startd Policy Configuration for more details on setting up a policy for starting and stopping jobs on a given machine.

The integer value is set by the *condor_starter* daemon for each job just before the job runs. The range of allowable values are integers in the range of 0 to 19 (inclusive), with a value of 19 being the lowest priority. If the integer value is outside this range, then on a Unix machine, a value greater than 19 is auto-decreased to 19; a value less than 0 is treated as 0. For values outside this range, a Windows machine ignores the value and uses the default instead. The default value is 10, which maps to the idle priority class on a Windows machine.

**STARTER_LOCAL_LOGGING** This macro determines whether the starter should do local logging to its own log file, or send debug information back to the *condor_shadow* where it will end up in the ShadowLog. It defaults to `True`.

**STARTER_DEBUG** This setting (and other settings related to debug logging in the starter) is described above in section 3.3.4 as `$(<SUBSYS>_DEBUG)`.

**STARTER_UPDATE_INTERVAL** The amount of time (in seconds) between ClassAd updates that the *condor_starter* daemon sends to the *condor_shadow* daemon. Defaults to 1200 (20 minutes).

**USER_JOB_WRAPPER** The full path to an executable or script. This macro allows an administrator to specify a wrapper script to handle the execution of all user jobs. If specified, Condor never directly executes a job, but instead invokes the program specified by this macro. The command-line arguments passed to this program will include the full-path to the actual user job which should be executed, followed by all the command-line parameters to pass to the user job. This wrapper program must ultimately replace its image with the user job; in other words, it must `exec()` the user job, not `fork()` it. For instance, if the wrapper program is a C/Korn shell script, the last line of execution should be:

```
exec $*
```

This can potentially lose information about the arguments. Any argument with embedded whitespace will be split into multiple arguments. For example the argument ”argument one” will become the two arguments ”argument” and ”one”. For Bourne type shells (sh, bash, ksh), the following preserves the arguments:

```
exec "$@"
```

For the C type shells (csh, tcsh), the following preserves the arguments:

```
exec $*:q
```

For Windows machines, the wrapper will either be a batch script (with a file extension of `.bat` or `.cmd`) or an executable (with a file extension of `.exe` or `.com`).

**USE_VISIBLE_DESKTOP** This setting is only meaningful on Windows machines. If True, Condor will allow the job to create windows on the desktop of the execute machine and interact with the job. This is particularly useful for debugging why an application will not run under Condor. If False, Condor uses the default behavior of creating a new, non-visible desktop to run the job on. See section 6.2 for details on how Condor interacts with the desktop.

**STARTER_JOB_ENVIRONMENT** This macro sets the default environment inherited by jobs. The syntax is the same as the syntax for environment settings in the job submit file (see page 718). If the same environment variable is assigned by this macro and by the user in the submit file, the user's setting takes precedence.

**JOB_INHERITS_STARTER_ENVIRONMENT** A boolean value that defaults to `False`. When `True`, it causes jobs to inherit all environment variables from the *condor_starter*. This is useful for glidein jobs that need to access environment variables from the batch system running the glidein daemons. When both the user job and `STARTER_JOB_ENVIRONMENT` define an environment variable that is in the *condor_starter*'s environment, the user job's definition takes precedence. This variable does not apply to standard or pvm universe jobs.

### 3.3.15 condor_submit Configuration File Entries

**DEFAULT_UNIVERSE** The universe under which a job is executed may be specified in the submit description file. If it is not specified in the submit description file, then this variable specifies the universe (when defined). If the universe is not specified in the submit description file, and if this variable is not defined, then the default universe for a job will be the standard universe.

If you want *condor_submit* to automatically append an expression to the `Requirements` expression or `Rank` expression of jobs at your site use the following macros:

**APPEND_REQ_VANILLA** Expression to be appended to vanilla job requirements.

**APPEND_REQ_STANDARD** Expression to be appended to standard job requirements.

**APPEND_REQUIREMENTS** Expression to be appended to any type of universe jobs. However, if `APPEND_REQ_VANILLA` or `APPEND_REQ_STANDARD` is defined, then ignore the `APPEND_REQUIREMENTS` for those universes.

**APPEND_RANK** Expression to be appended to job rank. `APPEND_RANK_STANDARD` or `APPEND_RANK_VANILLA` will override this setting if defined.

**APPEND_RANK_STANDARD** Expression to be appended to standard job rank.

**APPEND_RANK_VANILLA** Expression to append to vanilla job rank.

NOTE: The APPEND_RANK_STANDARD and APPEND_RANK_VANILLA macros were called APPEND_PREF_STANDARD and APPEND_PREF_VANILLA in previous versions of Condor.

In addition, you may provide default Rank expressions if your users do not specify their own with:

**DEFAULT_RANK_VANILLA** Default Rank for vanilla jobs.

**DEFAULT_RANK_STANDARD** Default Rank for standard jobs.

Both of these macros default to the jobs preferring machines where there is more main memory than the image size of the job, expressed as:

```
((Memory*1024) > Imagesize)
```

**DEFAULT_IO_BUFFER_SIZE** Condor keeps a buffer of recently-used data for each file an application opens. This macro specifies the default maximum number of bytes to be buffered for each open file at the executing machine. The *condor_status* buffer_size command will override this default. If this macro is undefined, a default size of 512 KB will be used.

**DEFAULT_IO_BUFFER_BLOCK_SIZE** When buffering is enabled, Condor will attempt to consolidate small read and write operations into large blocks. This macro specifies the default block size Condor will use. The *condor_status* buffer_block_size command will override this default. If this macro is undefined, a default size of 32 KB will be used.

**SUBMIT_SKIP_FILECHECK** If True, *condor_submit* behaves as if the **-d** command-line option is used. This tells *condor_submit* to disable file permission checks when submitting a job. This can significantly decrease the amount of time required to submit a large group of jobs. The default value is False.

**SUBMIT_SEND_RESCHEDULE** A boolean expression that when False, prevents *condor_submit* from automatically sending a *condor_reschedule* command as it completes. The *condor_reschedule* command causes the *condor_schedd* daemon to start searching for machines with which to match the submitted jobs. When True, this step always occurs. In the case that the machine where the job(s) are submitted is managing a huge number of jobs (thousands or tens of thousands), this step would hurt performance in such a way that it became an obstacle to scalability. The default value is True.

**SUBMIT_EXPRS** The given comma-separated, named expressions are inserted into all the job ClassAds that *condor_submit* creates. This is equivalent to the "+" syntax in submit files. See the the *condor_submit* manual page on page 716 for details on using the "+" syntax to add attributes to the job ClassAd. Attributes defined in the submit description file with "+" will override attributes defined in the config file with SUBMIT_EXPRS.

**LOG_ON_NFS_IS_ERROR** A boolean value that controls whether *condor_submit* prohibits job submit files with user log files on NFS. If LOG_ON_NFS_IS_ERROR is set to True, such submit files will be rejected. If LOG_ON_NFS_IS_ERROR is set to False, submitting such a file

results in a warning, but the job will be submitted. If not defined, LOG_ON_NFS_IS_ERROR defaults to False.

**SUBMIT_MAX_PROCS_IN_CLUSTER** An integer value that limits the maximum number of jobs that would be assigned within a single cluster. Job submissions that would exceed the defined value fail, issuing an error message, and with no jobs submitted. The default value is 0, which does not limit the number of jobs assigned a single cluster number.

### 3.3.16 condor_preen Configuration File Entries

These macros affect *condor_preen*.

**PREEN_ADMIN** This macro sets the e-mail address where *condor_preen* will send e-mail (if it is configured to send email at all... see the entry for PREEN). Defaults to $(CONDOR_ADMIN).

**VALID_SPOOL_FILES** This macro contains a (comma or space separated) list of files that *condor_preen* considers valid files to find in the $(SPOOL) directory. Defaults to all the files that are valid. A change to the $(HISTORY) macro requires a change to this macro as well.

**VALID_LOG_FILES** This macro contains a (comma or space separated) list of files that *condor_preen* considers valid files to find in the $(LOG) directory. Defaults to all the files that are valid. A change to the names of any of the log files above requires a change to this macro as well. In addition, the defaults for the $(<SUBSYS>_ADDRESS_FILE) are listed here, so a change to those requires a change this entry as well.

### 3.3.17 condor_collector Configuration File Entries

These macros affect the *condor_collector*.

**CLASSAD_LIFETIME** This macro determines the default maximum age for ClassAds collected by the *condor_collector*. ClassAd older than the maximum age are discarded by the *condor_collector* as stale.

If present, the ClassAd attribute "ClassAdLifetime" specifies the ad's lifetime in seconds. If "ClassAdLifetime" is not present in the ad, the *condor_collector* will use the value of $(CLASSAD_LIFETIME). The macro is defined in terms of seconds, and defaults to 900 (15 minutes).

**MASTER_CHECK_INTERVAL** This macro defines how often the collector should check for machines that have ClassAds from some daemons, but not from the *condor_master* (*orphaned daemons*) and send e-mail about it. It is defined in seconds and defaults to 10800 (3 hours).

**COLLECTOR_REQUIREMENTS** A boolean expression that filters out unwanted ClassAd updates. The expression is evaluated for ClassAd updates that have passed through enabled security

authorization checks. The default behavior when this expression is not defined is to allow all ClassAd updates to take place. If `False`, a ClassAd update will be rejected.

Stronger security mechanisms are the better way to authorize or deny updates to the *condor_collector*. This configuration variable exists to help those that use host-based security, and do not trust all processes that run on the hosts in the pool. This configuration variable may be used to throw out ClassAds that should not be allowed. For example, for *condor_startd* daemons that run on a fixed port, configure this expression to ensure that only machine ClassAds advertising the expected fixed port are accepted. As a convenience, before evaluating the expression, some basic sanity checks are performed on the ClassAd to ensure that all of the ClassAd attributes used by Condor to contain IP:port information are consistent. To validate this information, the attribute to check is `TARGET.MyAddress`.

**CLIENT_TIMEOUT** Network timeout that the *condor_collector* uses when talking to any daemons or tools that are sending it a ClassAd update. It is defined in seconds and defaults to 30.

**QUERY_TIMEOUT** Network timeout when talking to anyone doing a query. It is defined in seconds and defaults to 60.

**CONDOR_DEVELOPERS** By default, Condor will send e-mail once per week to this address with the output of the *condor_status* command, which lists how many machines are in the pool and how many are running jobs. The default value of condor-admin@cs.wisc.edu will send this report to the Condor Team developers at the University of Wisconsin-Madison. The Condor Team uses these weekly status messages in order to have some idea as to how many Condor pools exist in the world. We appreciate getting the reports, as this is one way we can convince funding agencies that Condor is being used in the real world. If you do not wish this information to be sent to the Condor Team, explicitly set the value to `NONE` to disable this feature, or replace the address with a desired location. If undefined (commented out) in the configuration file, Condor follows its default behavior.

**COLLECTOR_NAME** This macro is used to specify a short description of your pool. It should be about 20 characters long. For example, the name of the UW-Madison Computer Science Condor Pool is "UW-Madison CS". While this macro might seem similar to `MASTER_NAME` or `SCHEDD_NAME`, it is totally unrelated. Those settings are used to unique identify (and locate) a specific set of Condor daemons if there are more than one running on the same machine. The `COLLECTOR_NAME` setting is just used as a human-readable string to describe the pool, which is included in the updates set to the `CONDOR_DEVELOPERS_COLLECTOR` (see below).

**CONDOR_DEVELOPERS_COLLECTOR** By default, every pool sends periodic updates to a central *condor_collector* at UW-Madison with basic information about the status of your pool. This includes only the number of total machines, the number of jobs submitted, the number of machines running jobs, the host name of your central manager, and the `$(COLLECTOR_NAME)` specified above. These updates help the Condor Team see how Condor is being used around the world. By default, they will be sent to `condor.cs.wisc.edu`. If you do not want these updates to be sent from your pool, explicitly set this macro to `NONE`. If undefined (commented out) in the configuration file, Condor follows its default behavior.

**COLLECTOR_SOCKET_BUFSIZE** This specifies the buffer size, in bytes, reserved for *condor_collector* network UDP sockets. The default is 10240000, or a ten megabyte buffer. This is a healthy size, even for a large pool. The larger this value, the less likely the *condor_collector* will have stale information about the pool due to dropping update packets. If your pool is small or your central manager has very little RAM, considering setting this parameter to a lower value (perhaps 256000 or 128000).

NOTE: For some Linux distributions, it may be necessary to configure a larger value than the default; this parameter is /proc/sys/net/core/rmem_max . You can see the values that the *condor_collector* actually used by enabling D_FULLDEBUG for the collector and looking at the log line that looks like this:

Reset OS socket buffer size to 2048k (UDP), 255k (TCP).

**COLLECTOR_TCP_SOCKET_BUFSIZE** This specifies the TCP buffer size, in bytes, reserved for *condor_collector* network sockets. The default is 131072, or a 128 kilobyte buffer. This is a healthy size, even for a large pool. The larger this value, the less likely the *condor_collector* will have stale information about the pool due to dropping update packets. If your pool is small or your central manager has very little RAM, considering setting this parameter to a lower value (perhaps 65536 or 32768).

NOTE: See the note for `COLLECTOR_SOCKET_BUFSIZE` .

**COLLECTOR_SOCKET_CACHE_SIZE** If your site wants to use TCP connections to send ClassAd updates to the collector, you must use this setting to enable a cache of TCP sockets (in addition to enabling `UPDATE_COLLECTOR_WITH_TCP` ). Please read section 3.7.4 on "Using TCP to Send Collector Updates" on page 314 for more details and a discussion of when you would need this functionality. If you do not enable a socket cache, TCP updates will be refused by the collector. The default value for this setting is 0, with no cache enabled. If you lower this number, you must run *condor_restart* and not just *condor_reconfig* for the change to take effect.

**KEEP_POOL_HISTORY** This boolean macro is used to decide if the collector will write out statistical information about the pool to history files. The default is `False`. The location, size and frequency of history logging is controlled by the other macros.

**POOL_HISTORY_DIR** This macro sets the name of the directory where the history files reside (if history logging is enabled). The default is the `SPOOL` directory.

**POOL_HISTORY_MAX_STORAGE** This macro sets the maximum combined size of the history files. When the size of the history files is close to this limit, the oldest information will be discarded. Thus, the larger this parameter's value is, the larger the time range for which history will be available. The default value is 10000000 (10 Mbytes).

**POOL_HISTORY_SAMPLING_INTERVAL** This macro sets the interval, in seconds, between samples for history logging purposes. When a sample is taken, the collector goes through the information it holds, and summarizes it. The information is written to the history file once for each 4 samples. The default (and recommended) value is 60 seconds. Setting this macro's value too low will increase the load on the collector, while setting it to high will produce less precise statistical information.

**COLLECTOR_DAEMON_STATS** This macro controls whether or not the Collector keeps update statistics on incoming updates. The default value is True. If this option is enabled, the collector will insert several attributes into ClassAds that it stores and sends. ClassAds without the "UpdateSequenceNumber" and "DaemonStartTime" attributes will not be counted, and will not have attributes inserted (all modern Condor daemons which publish ClassAds publish these attributes).

The attributes inserted are "UpdatesTotal", "UpdatesSequenced", and "UpdatesLost". "UpdatesTotal" is the total number of updates (of this ad type) the Collector has received from this host. "UpdatesSequenced" is the number of updates that the Collector could have as lost. In particular, for the first update from a daemon it is impossible to tell if any previous ones have been lost or not. "UpdatesLost" is the number of updates that the Collector has detected as being lost.

**COLLECTOR_DAEMON_HISTORY_SIZE** This macro controls the size of the published update history that the Collector inserts into the ClassAds it stores and sends. The default value is 128, which means that history is stored and published for the latest 128 updates. This macro is ignored if `$(COLLECTOR_DAEMON_STATS)` is not enabled.

If this has a non-zero value, the Collector will insert "UpdatesHistory" into the ClassAd (similar to "UpdatesTotal" above). "UpdatesHistory" is a hexadecimal string which represents a bitmap of the last `COLLECTOR_DAEMON_HISTORY_SIZE` updates. The most significant bit (MSB) of the bitmap represents the most recent update, and the least significant bit (LSB) represents the least recent. A value of zero means that the update was not lost, and a value of 1 indicates that the update was detected as lost.

For example, if the last update was not lost, the previous lost, and the previous two not, the bitmap would be 0100, and the matching hex digit would be "4". Note that the MSB can never be marked as lost because its loss can only be detected by a non-lost update (a "gap" is found in the sequence numbers). Thus, UpdatesHistory = "0x40" would be the history for the last 8 updates. If the next updates are all successful, the values published, after each update, would be: 0x20, 0x10, 0x08, 0x04, 0x02, 0x01, 0x00.

**COLLECTOR_CLASS_HISTORY_SIZE** This macro controls the size of the published update history that the Collector inserts into the Collector ClassAds it produces. The default value is zero.

If this has a non-zero value, the Collector will insert "UpdatesClassHistory" into the Collector ClassAd (similar to "UpdatesHistory" above). These are added "per class" of ClassAd, however. The classes refer to the "type" of ClassAds (i.e. "Start"). Additionally, there is a "Total" class created which represents the history of all ClassAds that this Collector receives.

Note that the collector always publishes Lost, Total and Sequenced counts for all ClassAd "classes". This is similar to the statistics gathered if `$(COLLECTOR_DAEMON_STATS)` is enabled.

**COLLECTOR_QUERY_WORKERS** This macro sets the maximum number of "worker" processes that the Collector can have. When receiving a query request, the UNIX Collector will "fork" a new process to handle the query, freeing the main process to handle other requests. When the number of outstanding "worker" processes reaches this maximum, the request is handled

by the main process. This macro is ignored on Windows, and its default value is zero. The default configuration, however, has this set to 16.

**COLLECTOR_DEBUG** This macro (and other macros related to debug logging in the collector) is described in section 3.3.4 as <SUBSYS>_DEBUG.

### 3.3.18 condor_negotiator Configuration File Entries

These macros affect the *condor_negotiator*.

**NEGOTIATOR_INTERVAL** Sets how often the negotiator starts a negotiation cycle. It is defined in seconds and defaults to 300 (5 minutes).

**NEGOTIATOR_CYCLE_DELAY** An integer value that represents the minimum number of seconds that must pass before a new negotiation cycle may start. The default value is 20. NEGOTIATOR_CYCLE_DELAY is intended only for use by Condor experts.

**NEGOTIATOR_TIMEOUT** Sets the timeout that the negotiator uses on its network connections to the *condor_schedd* and *condor_startd*s. It is defined in seconds and defaults to 30.

**PRIORITY_HALFLIFE** This macro defines the half-life of the user priorities. See section 2.7.2 on User Priorities for details. It is defined in seconds and defaults to 86400 (1 day).

**DEFAULT_PRIO_FACTOR** This macro sets the priority factor for local users. See section 2.7.2 on User Priorities for details. Defaults to 1.

**NICE_USER_PRIO_FACTOR** This macro sets the priority factor for nice users. See section 2.7.2 on User Priorities for details. Defaults to 10000000.

**REMOTE_PRIO_FACTOR** This macro defines the priority factor for remote users (users who who do not belong to the accountant's local domain - see below). See section 2.7.2 on User Priorities for details. Defaults to 10000.

**ACCOUNTANT_LOCAL_DOMAIN** This macro is used to decide if a user is local or remote. A user is considered to be in the local domain if the UID_DOMAIN matches the value of this macro. Usually, this macro is set to the local UID_DOMAIN. If it is not defined, all users are considered local.

**MAX_ACCOUNTANT_DATABASE_SIZE** This macro defines the maximum size (in bytes) that the accountant database log file can reach before it is truncated (which re-writes the file in a more compact format). If, after truncating, the file is larger than one half the maximum size specified with this macro, the maximum size will be automatically expanded. The default is 1 megabyte (1000000).

**NEGOTIATOR_DISCOUNT_SUSPENDED_RESOURCES** This macro tells the negotiator to not count resources that are suspended when calculating the number of resources a user is using. Defaults to false, that is, a user is still charged for a resource even when that resource has suspended the job.

**NEGOTIATOR_SOCKET_CACHE_SIZE** This macro defines the maximum number of sockets that the negotiator keeps in its open socket cache. Caching open sockets makes the negotiation protocol more efficient by eliminating the need for socket connection establishment for each negotiation cycle. The default is currently 16. To be effective, this parameter should be set to a value greater than the number of *condor_schedd*s submitting jobs to the negotiator at any time. If you lower this number, you must run *condor_restart* and not just *condor_reconfig* for the change to take effect.

**NEGOTIATOR_PRE_JOB_RANK** Resources that match a request are first sorted by this expression. If there are any ties in the rank of the top choice, the top resources are sorted by the user-supplied rank in the job ClassAd, then by NEGOTIATOR_POST_JOB_RANK, then by PREEMPTION_RANK (if the match would cause preemption and there are still any ties in the top choice). MY refers to attributes of the machine ClassAd and TARGET refers to the job ClassAd. The purpose of the pre job rank is to allow the pool administrator to override any other rankings, in order to optimize overall throughput. For example, it is commonly used to minimize preemption, even if the job rank prefers a machine that is busy. If undefined, this expression has no effect on the ranking of matches. The standard configuration file shipped with Condor specifies an expression to steer jobs away from busy resources:

```
NEGOTIATOR_PRE_JOB_RANK = RemoteOwner =?= UNDEFINED
```

**NEGOTIATOR_POST_JOB_RANK** Resources that match a request are first sorted by NEGOTIATOR_PRE_JOB_RANK. If there are any ties in the rank of the top choice, the top resources are sorted by the user-supplied rank in the job ClassAd, then by NEGOTIATOR_POST_JOB_RANK, then by PREEMPTION_RANK (if the match would cause preemption and there are still any ties in the top choice). MY refers to attributes of the machine ClassAd and TARGET refers to the job ClassAd. The purpose of the post job rank is to allow the pool administrator to choose between machines that the job ranks equally. The default value is undefined, which causes this rank to have no effect on the ranking of matches. The following example expression steers jobs toward faster machines and tends to fill a cluster of multi-processors by spreading across all machines before filling up individual machines. In this example, the expression is chosen to have no effect when preemption would take place, allowing control to pass on to PREEMPTION_RANK.

```
UWCS_NEGOTIATOR_POST_JOB_RANK = \
 (RemoteOwner =?= UNDEFINED) * (KFlops - VirtualMachineID)
```

**PREEMPTION_REQUIREMENTS** When considering user priorities, the negotiator will not preempt a job running on a given machine unless the PREEMPTION_REQUIREMENTS expression evaluates to True and the owner of the idle job has a better priority than the owner of the running job. The PREEMPTION_REQUIREMENTS expression is evaluated within the context of the candidate machine ClassAd and the candidate idle job ClassAd; thus the MY scope prefix refers to the machine ClassAd, and the TARGET scope prefix refers to the ClassAd of the idle (candidate) job. If not explicitly set in the Condor configuration file, the default value for this expression is True. Note that this setting does not influence other potential causes of preemption, such as startd RANK, or PREEMPT expressions. See section 3.5.10 for a general discussion of limiting preemption.

**PREEMPTION_RANK** Resources that match a request are first sorted by NEGOTIATOR_PRE_JOB_RANK. If there are any ties in the rank of the top choice, the top resources are sorted by the user-supplied rank in the job ClassAd, then by NEGOTIATOR_POST_JOB_RANK, then by PREEMPTION_RANK (if the match would cause preemption and there are still any ties in the top choice). MY refers to attributes of the machine ClassAd and TARGET refers to the job ClassAd. This expression is used to rank machines that the job and the other negotiation expressions rank the same. For example, if the job has no preference, it is usually preferable to preempt a job with a small ImageSize instead of a job with a large ImageSize. The default is to rank all preemptable matches the same. However, the negotiator will always prefer to match the job with an idle machine over a preemptable machine, if none of the other ranks express a preference between them.

**NEGOTIATOR_DEBUG** This macro (and other settings related to debug logging in the negotiator) is described in section 3.3.4 as <SUBSYS>_DEBUG.

**NEGOTIATOR_MAX_TIME_PER_SUBMITTER** This macro limits the amount of time the negotiator will spend with a submitter. It defaults to one year.

The following configuration macros affect negotiation for group users.

**GROUP_NAMES** A comma-separated list of the recognized group names, case insensitive. If undefined (the default), group support is disabled. Group names must not conflict with any user names. That is, if there is a physics group, there may not be a physics user. Any group that is defined here must also have a quota, or the group will be ignored. Example:

```
GROUP_NAMES = group_physics, group_chemistry
```

**GROUP_QUOTA_<groupname>** A positive integer to represent a static quota specifying the exact number of machines owned by this group. Note that Condor does not verify or check consistency of quota values. Example:

```
GROUP_QUOTA_group_physics = 20
GROUP_QUOTA_group_chemistry = 10
```

**GROUP_PRIO_FACTOR_<groupname>** A floating point value greater than or equal to 1.0 to specify the default user priority factor for <groupname>. The group name must also be specified in the GROUP_NAMES list. GROUP_PRIO_FACTOR_<groupname> is evaluated when the negotiator first negotiates for the user as a member of the group. All members of the group inherit the default priority factor when no other value is present. For example, the following setting specifies that all members of the group named group_physics inherit a default user priority factor of 2.0:

```
GROUP_PRIO_FACTOR_group_physics = 2.0
```

**GROUP_AUTOREGROUP** A boolean value (defaults to `False`) that when `True`, causes users who submitted to a specific group to also negotiate a second time with the `none` group, to be considered with the independent job submitters. This allows group submitted jobs to be matched with idle machines even if the group is over its quota.

**GROUP_AUTOREGROUP_<groupname>** This is the same as `GROUP_AUTOREGROUP`, but it is settable on a per-group basis. If no value is specified for a given group, the default behavior is determined by `GROUP_AUTOREGROUP`, which in turn defaults to `False`.

**NEGOTIATOR_CONSIDER_PREEMPTION** For expert users only. A boolean value (defaults to `True`), that when `False`, can cause the negotiator to run faster and also have better spinning pie accuracy. *Only set this to `False` if `PREEMPTION_REQUIREMENTS` is `False`, and if all condor_startd rank expressions are `False`.*

### 3.3.19 condor_credd Configuration File Macros

**CREDD_HOST** The host name of the machine running the *condor_credd* daemon.

**CREDD_CACHE_LOCALLY** A boolean value that defaults to `False`. When `True`, the first successful password fetch operation to the *condor_credd* daemon causes the password to be stashed in a local, secure password store. Subsequent uses of that password do not require communication with the *condor_credd* daemon.

### 3.3.20 condor_gridmanager Configuration File Entries

These macros affect the *condor_gridmanager*.

**GRIDMANAGER_LOG** Logs for the *condor_gridmanager* will be written as the user who submitted the individual jobs. As a result `GRIDMANAGER_LOG` should defined to incorporate `$(USERNAME)` to specify a different file for each user in a directory that users will be able to write into. An example definition:

```
GRIDMANAGER_LOG = /tmp/GridmanagerLog.$(USERNAME)
```

**GRIDMANAGER_CHECKPROXY_INTERVAL** The number of seconds between checks for an updated X509 proxy credential. The default is 10 minutes (600 seconds).

**GRIDMANAGER_MINIMUM_PROXY_TIME** The minimum number of seconds before expiration of the X509 proxy credential for the gridmanager to continue operation. If seconds until expiration is less than this number, the gridmanager will shutdown and wait for a refreshed proxy credential. The default is 3 minutes (180 seconds).

**HOLD_JOB_IF_CREDENTIAL_EXPIRES** True or False. Defaults to True. If True, and for grid universe jobs only, Condor-G will place a job on hold `GRIDMANAGER_MINIMUM_PROXY_TIME` seconds before the proxy expires. If False, the job will stay in the last known state, and Condor-G will periodically check to see if the job's proxy has been refreshed, at which point management of the job will resume.

**GRIDMANAGER_CONTACT_SCHEDD_DELAY** The minimum number of seconds between connections to the *condor_schedd*. The default is 5 seconds.

**GRIDMANAGER_JOB_PROBE_INTERVAL** The number of seconds between active probes of the status of a submitted job. The default is 5 minutes (300 seconds).

**CONDOR_JOB_POLL_INTERVAL** After a condor grid type job is submitted, how often (in seconds) the *condor_gridmanager* should probe the remote *condor_schedd* to check the jobs status. This defaults to 300 seconds (5 minutes). Setting this to a lower number will decrease latency (Condor will discover that a job has finished more quickly), but will increase network traffic.

**GRIDMANAGER_RESOURCE_PROBE_INTERVAL** When a resource appears to be down, how often (in seconds) the *condor_gridmanager* should ping it to test if it is up again.

**GRIDMANAGER_RESOURCE_PROBE_DELAY** The number of seconds between pings of a remote resource that is currently down. The default is 5 minutes (300 seconds).

**GRIDMANAGER_EMPTY_RESOURCE_DELAY** The number of seconds that the *condor_gridmanager* retains information about a grid resource, once the *condor_gridmanager* has no active jobs on that resource. An active job is a grid universe job that is in the queue, but is not in the HELD state. Defaults to 300 seconds.

**GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE** Limits the number of jobs that a *condor_gridmanager* daemon will submit to a resource. It is useful for controlling the number of *jobmanager* processes running on the front-end node of a cluster. This number may be exceeded if it is reduced through the use of *condor_reconfig* while the *condor_gridmanager* is running or if the *condor_gridmanager* receives new jobs from the *condor_schedd* that were already submitted (that is, their `GridJobId` is not undefined). In these cases, submitted jobs will not be killed, but no new jobs can be submitted until the number of submitted jobs falls below the current limit. Defaults to 100.

**GRIDMANAGER_MAX_PENDING_SUBMITS_PER_RESOURCE** The maximum number of jobs that can be in the process of being submitted at any time (that is, how many `globus_gram_client_job_request()` calls are pending). It is useful for controlling the number of new connections/processes created at a given time. The default value is 5. This variable allows you to set different limits for each resource. After the first integer in the value comes a list of resourcename/number pairs, where each number is the limit for that resource. If a resource is not in the list, Condor uses the first integer. An example usage:

```
GRIDMANAGER_MAX_PENDING_SUBMITS_PER_RESOURCE=20,nostos,5,beak,50
```

**GRIDMANAGER_MAX_PENDING_SUBMITS** Configuration variable still recognized, but the name has changed to be `GRIDMANAGER_MAX_PENDING_SUBMITS_PER_RESOURCE`.

**GRIDMANAGER_MAX_JOBMANAGERS_PER_RESOURCE** For grid jobs of type **gt2**, limits the number of globus-job-manager processes that the *condor_gridmanager* lets run at a time on the remote head node. Allowing too many globus-job-managers to run causes severe load on the headnote, possibly making it non-functional. This number may be exceeded if it is reduced through the use of *condor_reconfig* while the *condor_gridmanager* is running or if some

globus-job-managers take a few extra seconds to exit. The value 0 means there is no limit. The default value is 10.

**GAHP** The full path to the binary of the GAHP server. This configuration variable is no longer used. Use GT2_GAHP at section 3.3.20 instead.

**GAHP_ARGS** Arguments to be passed to the GAHP server. This configuration variable is no longer used.

**GRIDMANAGER_GAHP_CALL_TIMEOUT** The number of seconds after which a pending GAHP command should time out. The default is 5 minutes (300 seconds).

**GRIDMANAGER_MAX_PENDING_REQUESTS** The maximum number of GAHP commands that can be pending at any time. The default is 50.

**GRIDMANAGER_CONNECT_FAILURE_RETRY_COUNT** The number of times to retry a command that failed due to a timeout or a failed connection. The default is 3.

**GRIDMANAGER_GLOBUS_COMMIT_TIMEOUT** The duration, in seconds, of the two phase commit timeout to Globus for gt2 jobs only. This maps directly to the two_phase setting in the Globus RSL.

**GLOBUS_GATEKEEPER_TIMEOUT** The number of seconds after which if a gt2 grid universe job fails to ping the gatekeeper, the job will be put on hold. Defaults to 5 days (in seconds).

**C_GAHP_LOG** The complete path and file name of the Condor GAHP server's log. There is no default value. The expected location as defined in the example configuration is /temp/CGAHPLog.$(USERNAME).

**MAX_C_GAHP_LOG** The maximum size of the C_GAHP_LOG.

**C_GAHP_WORKER_THREAD_LOG** The complete path and file name of the Condor GAHP worker process' log. There is no default value. The expected location as defined in the example configuration is /temp/CGAHPWorkerLog.$(USERNAME).

**GLITE_LOCATION** The complete path to the directory containing the Glite software. There is no default value. The expected location as given in the example configuration is $(LIB)/glite. The necessary Glite software is included with Condor, and is required for pbs and lsf jobs.

**CONDOR_GAHP** The complete path and file name of the Condor GAHP executable. There is no default value. The expected location as given in the example configuration is $(SBIN)/condor_c-gahp.

**GT2_GAHP** The complete path and file name of the GT2 GAHP executable. There is no default value. The expected location as given in the example configuration is $(SBIN)/gahp_server.

**GT3_GAHP** The complete path and file name of the wrapper script that invokes the GT3 GAHP executable. There is no default value. The expected location as given in the example configuration is $(SBIN)/gt3_gahp.

**GT4_GAHP** The complete path and file name of the wrapper script that invokes the GT4 GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/gt4_gahp`.

**PBS_GAHP** The complete path and file name of the PBS GAHP executable. There is no default value. The expected location as given in the example configuration is `$(GLITE_LOCATION)/bin/batch_gahp`.

**LSF_GAHP** The complete path and file name of the LSF GAHP executable. There is no default value. The expected location as given in the example configuration is `$(GLITE_LOCATION)/bin/batch_gahp`.

**UNICORE_GAHP** The complete path and file name of the wrapper script that invokes the Unicore GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/unicore_gahp`.

**NORDUGRID_GAHP** The complete path and file name of the wrapper script that invokes the NorduGrid GAHP executable. There is no default value. The expected location as given in the example configuration is `$(SBIN)/nordugrid_gahp`.

### 3.3.21 grid_monitor Configuration File Entries

These macros affect the *grid_monitor*.

**ENABLE_GRID_MONITOR** When set to `True` enables the *grid_monitor* tool. The *grid_monitor* tool is used to reduce load on Globus gatekeepers. This parameter only affects grid jobs of type **gt2**. `GRID_MONITOR` must also be correctly configured. Defaults to `False`. See section 5.3.2 on page 431 for more information.

**GRID_MONITOR** The complete pathname of the *grid_monitor* tool used to reduce load on Globus gatekeepers. This parameter only affects grid jobs of type **gt2**. This parameter is not referenced unless `ENABLE_GRID_MONITOR` is set to `True`. See section 5.3.2 on page 431 for more information.

**GRID_MONITOR_HEARTBEAT_TIMEOUT** If this many seconds pass without hearing from a *grid_monitor*, it is assumed to be dead. Defaults to 300 (5 minutes). Increasing this number will improve the ability of the *grid_monitor* to survive in the face of transient problems but will also increase the time before Condor notices a problem.

**GRID_MONITOR_RETRY_DURATION** If something goes wrong with the *grid_monitor* at a particular site (like `GRID_MONITOR_HEARTBEAT_TIMEOUT` expiring), Condor-G will attempt to restart the *grid_monitor* for this many seconds. Defaults to 900 (15 minutes). If this duration passes without success the *grid_monitor* will be disabled for the site in question until 60 minutes have passed.

### 3.3.22 Configuration File Entries Relating to Grid Usage and Glidein

These macros affect the Condor's usage of grid resources and glidein.

**GLIDEIN_SERVER_URLS** A comma or space-separated list of URLs that contain the binaries that must be copied by *condor_glidein*. There are no default values, but working URLs that copy from the UW site are provided in the distributed sample configuration files.

### 3.3.23 Configuration File Entries for DAGMan

These macros affect the operation of DAGMan and DAGMan jobs within Condor.

**DAGMAN_MAX_SUBMITS_PER_INTERVAL** An integer that controls how many individual jobs *condor_dagman* will submit in a row before servicing other requests (such as a *condor_rm*). The legal range of values is 1 to 1000. If defined with a value less than 1, the value 1 will be used. If defined with a value greater than 1000, the value 1000 will be used. If not defined, it defaults to 5.

**DAGMAN_MAX_SUBMIT_ATTEMPTS** An integer that controls how many times in a row *condor_dagman* will attempt to execute *condor_submit* for a given job before giving up. Note that consecutive attempts use an exponential backoff, starting with 1 second. The legal range of values is 1 to 16. If defined with a value less than 1, the value 1 will be used. If defined with a value greater than 16, the value 16 will be used. Note that a value of 16 would result in *condor_dagman* trying for approximately 36 hours before giving up. If not defined, it defaults to 6 (approximately two minutes before giving up).

**DAGMAN_SUBMIT_DELAY** An integer that controls the number of seconds that *condor_dagman* will sleep before submitting consecutive jobs. It can be increased to help reduce the load on the *condor_schedd* daemon. The legal range of values is 0 to 60. If defined with a value less than 0, the value 0 will be used. If defined with a value greater than 60, the value 60 will be used. The default value is 0.

**DAGMAN_STARTUP_CYCLE_DETECT** A boolean value that when `True` causes *condor_dagman* to check for cycles in the DAG before submitting DAG node jobs, in addition to its run time cycle detection. If not defined, it defaults to `False`.

**DAGMAN_RETRY_SUBMIT_FIRST** A boolean value that controls whether a failed submit is retried first (before any other submits) or last (after all other ready jobs are submitted). If this value is set to `True`, when a job submit fails, the job is placed at the head of the queue of ready jobs, so that it will be submitted again before any other jobs are submitted (this has been the behavior of *condor_dagman* up to this point). If this value is set to `False`, when a job submit fails, the job is placed at the tail of the queue of ready jobs. If not defined, it defaults to `True`.

**DAGMAN_RETRY_NODE_FIRST** A boolean value that controls whether a failed node (with retries) is retried first (before any other ready nodes) or last (after all other ready nodes). If this value

is set to `True`, when a node with retries fails (after the submit succeeded), the node is placed at the head of the queue of ready nodes, so that it will be tried again before any other jobs are submitted. If this value is set to `False`, when a node with retries fails, the node is placed at the tail of the queue of ready nodes (this has been the behavior of *condor_dagman* up to this point). If not defined, it defaults to `False`.

**DAGMAN_MAX_JOBS_IDLE** An integer value that controls the maximum number of idle node jobs allowed within the DAG before *condor_dagman* temporarily stops submitting jobs. Once idle jobs start to run, *condor_dagman* will resume submitting jobs. If both the command-line flag and the configuration parameter are specified, the command-line flag overrides the configuration parameter. The default is that there is no limit on the maximum number of idle jobs.

**DAGMAN_MAX_JOBS_SUBMITTED** An integer value that controls the maximum number of node jobs within the DAG that will be submitted to Condor at one time. Note that this parameter is the same as the **-maxjobs** *c*ommand-line flag to condor_submit_dag. If both the command-line flag and the configuration parameter are specified, the command-line flag overrides the configuration parameter. The default is that there is no limit on the maximum number of jobs run at one time.

**DAGMAN_MUNGE_NODE_NAMES** A boolean value that controls whether *condor_dagman* automatically renames nodes when running multiple DAGs (the renaming is done to avoid possible name conflicts). If this value is set to `True`, all node names have the "DAG number" prepended to them. For example, the first DAG specified on the *condor_submit_dag* command line is considered DAG number 0, the second is DAG number 1, etc. So if DAG number 2 has a node B, that node will internally be renamed to "2.B". If not defined, `DAGMAN_MUNGE_NODE_NAMES` defaults to `True`.

**DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION** This macro is no longer used. The improved functionality of the `DAGMAN_ALLOW_EVENTS` macro eliminates the need for this variable.

A boolean value that controls whether *condor_dagman* aborts or continues with a DAG in the rare case that Condor erroneously executes the job within a DAG node more than once. A bug in Condor very occasionally causes a job to run twice. Running a job twice is contrary to the semantics of a DAG. The configuration macro `DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION` determines whether *condor_dagman* considers this a fatal error or not. The default value is `False`; *condor_dagman* considers running the job more than once a fatal error, logs this fact, and aborts the DAG. When set to `True`, *condor_dagman* still logs this fact, but continues with the DAG.

This configuration macro is to remain at its default value except in the case where a site encounters the Condor bug in which DAG job nodes are executed twice, and where it is certain that having a DAG job node run twice will not corrupt the DAG. The logged messages within `*.dagman.out` files in the case of that a node job runs twice contain the string "EVENT ERROR."

**DAGMAN_ALLOW_EVENTS** An integer that controls which "bad" events are considered fatal errors by *condor_dagman*. This macro replaces and expands upon the functionality of the

DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION macro. If DAGMAN_ALLOW_EVENTS is set, it overrides the setting of DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION.

The DAGMAN_ALLOW_EVENTS value is a bitwise-OR of the following values:

0 = allow no ”bad” events

1 = allow almost all ”bad” events (all except ”job re-run after terminated event”)

2 = allow terminated/aborted event combination

4 = allow ”job re-run after terminated event” bug

8 = allow garbage/orphan events

16 = allow execute or terminate event before job’s submit event

32 = allow two terminated events per job (sometimes seen with grid jobs)

64 = allow duplicated events in general

The default value is 114 (allow terminated/aborted event combination, allow execute and/or terminated event before job’s submit event, allow double terminated events, and allow general duplicate events).

For example, a value of 6 instructs *condor_dagman* to allow both the terminated/aborted event combination and the ”job re-run after terminated event” bug. A value of 0 means that any ”bad” event will be considered a fatal error.

A value of 5 (1 + 4) will never abort the DAG because of a ”bad” event – but you should almost never use this setting, because the ”job re-run after terminated event” bug breaks the semantics of the DAG.

This macro should almost always remain set to the default value!

**DAGMAN_DEBUG** This macro is described in section 3.3.4 as <SUBSYS>_DEBUG.

**MAX_DAGMAN_LOG** This macro is described in section 3.3.4 as MAX_<SUBSYS>_LOG.

**DAGMAN_CONDOR_SUBMIT_EXE** The executable that *condor_dagman* will use to submit Condor jobs. If not defined, *condor_dagman* looks for *condor_submit* in the PATH.

**DAGMAN_STORK_SUBMIT_EXE** The executable that *condor_dagman* will use to submit Stork jobs. If not defined, *condor_dagman* looks for *stork_submit* in the PATH.

**DAGMAN_CONDOR_RM_EXE** The executable that *condor_dagman* will use to remove Condor jobs. If not defined, *condor_dagman* looks for *condor_rm* in the PATH.

**DAGMAN_STORK_RM_EXE** The executable that *condor_dagman* will use to remove Stork jobs. If not defined, *condor_dagman* looks for *stork_rm* in the PATH.

**DAGMAN_PROHIBIT_MULTI_JOBS** A boolean value that controls whether *condor_dagman* prohibits node job submit files that queue multiple job procs (other than parallel universe). If a DAG references such a submit file, the DAG will abort during the initialization process. If not defined, DAGMAN_PROHIBIT_MULTI_JOBS defaults to False.

**DAGMAN_LOG_ON_NFS_IS_ERROR** A boolean value that controls whether *condor_dagman* prohibits node job submit files with user log files on NFS. If a DAG references such a submit file and DAGMAN_LOG_ON_NFS_IS_ERROR is True, the DAG will abort during the initialization process. If DAGMAN_LOG_ON_NFS_IS_ERROR is False, a warning will be issued but the DAG will still be submitted. It is *strongly* recommended that DAGMAN_LOG_ON_NFS_IS_ERROR remain set to the default value, because running a DAG with node job log files on NFS will often cause errors. If not defined, DAGMAN_LOG_ON_NFS_IS_ERROR defaults to True.

**DAGMAN_ABORT_DUPLICATES** A boolean value that controls whether to attempt to abort duplicate instances of *condor_dagman* running the same DAG on the same machine. When *condor_dagman* starts up, if no DAG lock file exists, *condor_dagman* creates the lock file and writes its PID into it. If the lock file does exist, and DAGMAN_ABORT_DUPLICATES is set to True, *condor_dagman* checks whether a process with the given PID exists, and if so, it assumes that there is already another instance of *condor_dagman* running on the same DAG. Note that this test is not foolproof: it is possible that, if *condor_dagman* crashes, the same PID gets reused by another process before *condor_dagman* gets rerun on that DAG. This should be quite rare, however. If not defined, DAGMAN_ABORT_DUPLICATES defaults to False.

**DAGMAN_PENDING_REPORT_INTERVAL** An integer value (in seconds) that controls how often *condor_dagman* will print a report of pending nodes to the dagman.out file. Note that the report will only be printed if *condor_dagman* has been waiting at least DAGMAN_PENDING_REPORT_INTERVAL seconds without seeing any node job user log events, in order to avoid cluttering the dagman.out file. (This feature is mainly intended to help diagnose ”stuck” *condor_dagman* processes that are waiting indefinitely for a job to finish.) If not defined, DAGMAN_PENDING_REPORT_INTERVAL defaults to 600 seconds (10 minutes).

### 3.3.24 Configuration File Entries Relating to Security

These macros affect the secure operation of Condor. Many of these macros are described in section 3.6 on Security.

**SEC_*_AUTHENTICATION** This section has not yet been written

**SEC_*_ENCRYPTION** This section has not yet been written

**SEC_*_INTEGRITY** This section has not yet been written

**SEC_*_NEGOTIATION** This section has not yet been written

**SEC_*_AUTHENTICATION_METHODS** This section has not yet been written

**SEC_*_CRYPTO_METHODS** This section has not yet been written

**GSI_DAEMON_NAME** A comma separated list of the subject name(s) of the certificate(s) that the daemons use.

**GSI_DAEMON_DIRECTORY** A directory name used in the construction of complete paths for the configuration variables GSI_DAEMON_CERT, GSI_DAEMON_KEY, and GSI_DAEMON_TRUSTED_CA_DIR, for any of these configuration variables are not explicitly set.

**GSI_DAEMON_CERT** A complete path and file name to the X.509 certificate to be used in GSI authentication. If this configuration variable is not defined, and GSI_DAEMON_DIRECTORY is defined, then Condor uses GSI_DAEMON_DIRECTORY to construct the path and file name as

```
GSI_DAEMON_CERT  = $(GSI_DAEMON_DIRECTORY)/hostcert.pem
```

**GSI_DAEMON_KEY** A complete path and file name to the X.509 private key to be used in GSI authentication. If this configuration variable is not defined, and GSI_DAEMON_DIRECTORY is defined, then Condor uses GSI_DAEMON_DIRECTORY to construct the path and file name as

```
GSI_DAEMON_KEY  = $(GSI_DAEMON_DIRECTORY)/hostkey.pem
```

**GSI_DAEMON_TRUSTED_CA_DIR** The directory that contains the list of trusted certification authorities to be used in GSI authentication. The files in this directory are the public keys and signing policies of the trusted certification authorities. If this configuration variable is not defined, and GSI_DAEMON_DIRECTORY is defined, then Condor uses GSI_DAEMON_DIRECTORY to construct the directory path as

```
GSI_DAEMON_TRUSTED_CA_DIR  = $(GSI_DAEMON_DIRECTORY)/certificates
```

**GSI_DAEMON_PROXY** A complete path and file name to the X.509 proxy to be used in GSI authentication. When this configuration variable is defined, use of this proxy takes precedence over use of a certificate and key.

**DELEGATE_JOB_GSI_CREDENTIALS** A boolean value that defaults to True for Condor version 6.7.19 and more recent versions. When True, a job's GSI X.509 credentials are delegated, instead of being copied. This results in a more secure communication when not encrypted.

**GRIDMAP** The complete path and file name of the Globus Gridmap file. The Gridmap file is used to map X.509 distinguished names to Condor user ids.

**SEC_DEFAULT_SESSION_DURATION** The amount of time in seconds before a communication session expires. Defaults to 8640000 seconds (100 days) to avoid a bug in session renegotiation for Condor Version 6.6.0. A session is a record of necessary information to do communication between a client and daemon, and is protected by a shared secret key. The session expires to reduce the window of opportunity where the key may be compromised by attack.

**FS_REMOTE_DIR** The location of a file visible to both server and client in Remote File System authentication. The default when not defined is the directory /shared/scratch/tmp.

**ENCRYPT_EXECUTE_DIRECTORY** The execute directory for jobs on Windows platforms may be encrypted by setting this configuration variable to `True`. Defaults to `False`. The method of encryption uses the EFS (Encrypted File System) feature of Windows NTFS v5.

**SEC_TCP_SESSION_TIMEOUT** The length of time in seconds until the timeout when establishing a UDP security session via TCP. The default value is 20 seconds. Scalability issues with a large pool would be the only basis for a change from the default value.

**SEC_PASSWORD_FILE** For Unix machines, the path and file name of the file containing the pool password for password authentication.

**AUTH_SSL_SERVER_CAFILE** The path and file name of a file containing one or more trusted CA's certificates for the server side of a communication authenticating with SSL.

**AUTH_SSL_CLIENT_CAFILE** The path and file name of a file containing one or more trusted CA's certificates for the client side of a communication authenticating with SSL.

**AUTH_SSL_SERVER_CADIR** The path to a directory that may contain the certificates (each in its own file) for multiple trusted CAs for the server side of a communication authenticating with SSL. When defined, the authenticating entity's certificate is utilized to identify the trusted CA's certificate within the directory.

**AUTH_SSL_CLIENT_CADIR** The path to a directory that may contain the certificates (each in its own file) for multiple trusted CAs for the client side of a communication authenticating with SSL. When defined, the authenticating entity's certificate is utilized to identify the trusted CA's certificate within the directory.

**AUTH_SSL_SERVER_CERTFILE** The path and file name of the file containing the public certificate for the server side of a communication authenticating with SSL.

**AUTH_SSL_CLIENT_CERTFILE** The path and file name of the file containing the public certificate for the client side of a communication authenticating with SSL.

**AUTH_SSL_SERVER_KEYFILE** The path and file name of the file containing the private key for the server side of a communication authenticating with SSL.

**AUTH_SSL_CLIENT_KEYFILE** The path and file name of the file containing the private key for the client side of a communication authenticating with SSL.

**CERTIFICATE_MAPFILE** A path and file name of the unified map file.

### 3.3.25 Configuration File Entries Relating to High Availability

These macros affect the high availability operation of Condor.

**MASTER_HA_LIST** Similar to `DAEMON_LIST`, this macro defines a list of daemons that the *condor_master* starts and keeps its watchful eyes on. However, the `MASTER_HA_LIST` daemons are run in a *High Availability* mode. The list is a comma or space separated list of subsystem names (as listed in section 3.3.1). For example,

```
MASTER_HA_LIST = SCHEDD
```

The *High Availability* feature allows for several *condor_master* daemons (most likely on separate machines) to work together to insure that a particular service stays available. These *condor_master* daemons ensure that one and only one of them will have the listed daemons running.

To use this feature, the lock URL must be set with HA_LOCK_URL.

Currently, only file URLs are supported (those with file:...). The default value for MASTER_HA_LIST is the empty string, which disables the feature.

**HA_LOCK_URL** This macro specifies the URL that the *condor_master* processes use to synchronize for the *High Availability* service. Currently, only file URLs are supported; for example, file:/share/spool. Note that this URL must be identical for all *condor_master* processes sharing this resource. For *condor_schedd* sharing, we recommend setting up SPOOL on an NFS share and having all *High Availability condor_schedd* processes sharing it, and setting the HA_LOCK_URL to point at this directory as well. For example:

```
MASTER_HA_LIST = SCHEDD
SPOOL = /share/spool
HA_LOCK_URL = file:/share/spool
```

A separate lock is created for each *High Availability* daemon.

There is no default value for HA_LOCK_URL.

**HA_<SUBSYS>_LOCK_URL** This macro controls the *High Availability* lock URL for a specific subsystem as specified in the configuration variable name, and it overrides the system-wide lock URL specified by HA_LOCK_URL. If not defined for each subsystem, HA_<SUBSYS>_LOCK_URL is ignored, and the value of HA_LOCK_URL is used.

**HA_LOCK_HOLD_TIME** This macro specifies the number of seconds that the *condor_master* will hold the lock for each *High Availability* daemon. Upon gaining the shared lock, the *condor_master* will hold the lock for this number of seconds. Additionally, the *condor_master* will periodically renew each lock as long as the *condor_master* and the daemon are running. When the daemon dies, or the *condor_master* exists, the *condor_master* will immediately release the lock(s) it holds.

HA_LOCK_HOLD_TIME defaults to 3600 seconds (one hour).

**HA_<SUBSYS>_LOCK_HOLD_TIME** This macro controls the *High Availability* lock hold time for a specific subsystem as specified in the configuration variable name, and it overrides the system wide poll period specified by HA_LOCK_HOLD_TIME. If not defined for each subsystem, HA_<SUBSYS>_LOCK_HOLD_TIME is ignored, and the value of HA_LOCK_HOLD_TIME is used.

**HA_POLL_PERIOD** This macro specifies how often the *condor_master* polls the *High Availability* locks to see if any locks are either stale (meaning not updated for HA_LOCK_HOLD_TIME seconds), or have been released by the owning *condor_master*. Additionally, the *condor_master* renews any locks that it holds during these polls.

HA_POLL_PERIOD defaults to 300 seconds (five minutes).

**HA_<SUBSYS>_POLL_PERIOD** This macro controls the *High Availability* poll period for a specific subsystem as specified in the configuration variable name, and it overrides the system wide poll period specified by HA_POLL_PERIOD. If not defined for each subsystem, HA_<SUBSYS>_POLL_PERIOD is ignored, and the value of HA_POLL_PERIOD is used.

**MASTER_<SUBSYS>_CONTROLLER** Used only in HA configurations involving the *condor_had*.

The *condor_master* has the concept of a controlling and controlled daemon, typically with the *condor_had* daemon serving as the controlling process. In this case, all *condor_on* and *condor_off* commands directed at controlled daemons are given to the controlling daemon, which then handles the command, and, when required, sends appropriate commands to the *condor_master* to do the actual work. This allows the controlling daemon to know the state of the controlled daemon.

As of 6.7.14, this configuration variable must be specified for all configurations using *condor_had*. To configure the *condor_negotiator* controlled by *condor_had*:

```
MASTER_NEGOTIATOR_CONTROLLER = HAD
```

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined in section 3.3.1.

**HAD_LIST** A comma-separated list of all *condor_had* daemons in the form IP:port or hostname:port. Each central manager machine that runs the *condor_had* daemon should appear in this list. If HAD_USE_PRIMARY is set to True, then the first machine in this list is the primary central manager, and all others in the list are backups.

All central manager machines must be configured with an identical HAD_LIST. The machine addresses are identical to the addresses defined in COLLECTOR_HOST.

**HAD_USE_PRIMARY** Boolean value to determine if the first machine in the HAD_LIST configuration variable is a primary central manager. Defaults to False.

**HAD_CONNECTION_TIMEOUT** The time (in seconds) that the *condor_had* daemon waits before giving up on the establishment of a TCP connection. The failure of the communication connection is the detection mechanism for the failure of a central manager machine. For a LAN, a recommended value is 2 seconds. The use of authentication (by Condor) increases the connection time. The default value is 5 seconds. If this value is set too low, *condor_had* daemons will incorrectly assume the failure of other machines.

**HAD_ARGS** Command line arguments passed by the *condor_master* daemon as it invokes the *condor_had* daemon. To make high availability work, the *condor_had* daemon requires the port number it is to use. This argument is of the form

```
-p $(HAD_PORT_NUMBER)
```

where HAD_PORT_NUMBER is a helper configuration variable defined with the desired port number. Note that this port number must be the same value here as used in HAD_LIST. There is no default value.

**HAD** The path to the *condor_had* executable. Normally it is defined relative to $(SBIN). This configuration variable has no default value.

**MAX_HAD_LOG** Controls the maximum length in bytes to which the *condor_had* daemon log will be allowed to grow. It will grow to the specified length, then be saved to a file with the suffix .old. The .old file is overwritten each time the log is saved, thus the maximum space devoted to logging is twice the maximum length of this log file. A value of 0 specifies that this file may grow without bounds. The default is 1 Mbyte.

**HAD_DEBUG** Logging level for the *condor_had* daemon. See <SUBSYS>_DEBUG for values.

**HAD_LOG** Full path and file name of the log file. There is no default value.

**REPLICATION_LIST** A comma-separated list of all *condor_replication* daemons in the form IP:port or hostname:port. Each central manager machine that runs the *condor_had* daemon should appear in this list. All potential central manager machines must be configured with an identical REPLICATION_LIST.

**STATE_FILE** A full path and file name of the file protected by the replication mechanism. When not defined, the default path and file used is

```
$(SPOOL)/Accountantnew.log
```

**REPLICATION_INTERVAL** Sets how often the *condor_replication* daemon initiates its tasks of replicating the $(STATE_FILE). It is defined in seconds and defaults to 300 (5 minutes). This is the same as the default NEGOTIATOR_INTERVAL.

**MAX_TRANSFER_LIFETIME** A timeout period within which the process that transfers the state file must complete its transfer. The recommended value is 2 * average size of state file / network rate. It is defined in seconds and defaults to 300 (5 minutes).

**HAD_UPDATE_INTERVAL** Like UPDATE_INTERVAL, determines how often the *condor_had* is to send a ClassAd update to the *condor_collector*. Updates are also sent at each and every change in state. It is defined in seconds and defaults to 300 (5 minutes).

**HAD_USE_REPLICATION** A boolean value that defaults to False. When True, the use of *condor_replication* daemons is enabled.

**REPLICATION_ARGS** Command line arguments passed by the *condor_master* daemon as it invokes the *condor_replication* daemon. To make high availability work, the *condor_replication* daemon requires the port number it is to use. This argument is of the form

```
-p $(REPLICATION_PORT_NUMBER)
```

where REPLICATION_PORT_NUMBER is a helper configuration variable defined with the desired port number. Note that this port number must be the same value as used in REPLICATION_LIST. There is no default value.

**REPLICATION** The full path and file name of the *condor_replication* executable. It is normally defined relative to $(SBIN). There is no default value.

**MAX_REPLICATION_LOG** Controls the maximum length in bytes to which the *condor_replication* daemon log will be allowed to grow. It will grow to the specified length, then be saved to a file with the suffix .old. The .old file is overwritten each time the log is saved, thus the maximum space devoted to logging is twice the maximum length of this log file. A value of 0 specifies that this file may grow without bounds. The default is 1 Mbyte.

**REPLICATION_DEBUG** Logging level for the *condor_replication* daemon. See <SUBSYS>_DEBUG for values.

**REPLICATION_LOG** Full path and file name to the log file. There is no default value.

### 3.3.26 Configuration File Entries Relating to Quill

These macros affect the Quill database management and interface to its representation of the job queue.

**QUILL** The full path name to the *condor_quill* daemon.

**QUILL_ARGS** Arguments to be passed to the *condor_quill* daemon upon its invocation.

**QUILL_LOG** Path to the Quill daemon's log file.

**QUILL_ENABLED** A boolean variable that defaults to False. When True, Quill functionality is enabled. When False, the Quill daemon writes a message to its log and exits. The *condor_q* and *condor_history* tools then do not use Quill.

**QUILL_NAME** A string that uniquely identifies an instance of the *condor_quill* daemon, as there be more than *condor_quill* daemon per pool. The string must not be the same as for any *condor_schedd* daemon. A convenient definition to choose is of the form

```
quill-for-schedd_name@machinename.fully.qualified.address
```

**QUILL_DB_NAME** A string that identifies a database within a database server.

**QUILL_DB_IP_ADDR** An IP address of the database server.

**QUILL_POLLING_PERIOD** The frequency, in number of seconds, at which the Quill daemon polls the file job_queue.log for updates. New information in the log file is sent to the database. The default value is 10.

**QUILL_HISTORY_CLEANING_INTERVAL** The interval, in hours, between scans of the database to identify and delete jobs that are beyond their QUILL_HISTORY_DURATION time. The default value is 24.

**QUILL_HISTORY_DURATION** The number of days after entry into the database that a job will remain in the database. After QUILL_HISTORY_DURATION days, the job is deleted.

**QUILL_MANAGE_VACUUM** A boolean value that defaults to False. When True, Quill takes on the maintenance task of vacuuming the database. As of *PostgreSQL* version 8.1, the database can perform this task automatically; therefore having Quill vacuum is not necessary. A value of True causes warnings to be written to the log file.

**QUILL_IS_REMOTELY_QUERYABLE** A boolean value that defaults to True. When False, the remote database tables may not be remotely queryable.

**QUILL_DB_QUERY_PASSWORD** Defines the password string needed by *condor_q* to gain read access for remotely querying the Quill database.

**QUILL_ADDRESS_FILE** When defined, it specifies the path and file name of a local file containing the IP address and port number of the Quill daemon. By using the file, tools executed on the local machine do not need to query the central manager in order to find the Quill daemon.

### 3.3.27 MyProxy Configuration File Macros

In some cases, Condor can autonomously refresh GSI certificate proxies via *MyProxy*, available from http://myproxy.ncsa.uiuc.edu/.

**MYPROXY_GET_DELEGATION** The full path name to the *myproxy-get-delegation* executable, installed as part of the *MyProxy* software. Often, it is necessary to wrap the actual executable with a script that sets the environment, such as the LD_LIBRARY_PATH, correctly. If this macro is defined, Condor-G and *condor_credd* will have the capability to autonomously refresh proxy certificates. By default, this macro is undefined.

### 3.3.28 Configuration File Macros Affecting APIs

**ENABLE_SOAP** A boolean value that defaults to False. When True, Condor daemons will respond to HTTP PUT commands as if they were SOAP calls. When False, all HTTP PUT commands are denied.

**ENABLE_WEB_SERVER** A boolean value that defaults to False. When True, Condor daemons will respond to HTTP GET commands, and send the static files sitting in the subdirectory defined by the configuration variable WEB_ROOT_DIR. In addition, web commands are considered a READ command, so the client will be checked by host-based security.

**SOAP_LEAVE_IN_QUEUE** A boolean value that when True, causes a job in the completed state to remain in the queue, instead of being removed based on the completion of file transfer. There is no default value.

**WEB_ROOT_DIR** A complete path to the directory containing all the files served by the web server.

**<SUBSYS>_ENABLE_SOAP_SSL** A boolean value that defaults to `False`. When `True`, enables SOAP over SSL for the specified <SUBSYS>. Any specific <SUBSYS>_ENABLE_SOAP_SSL setting overrides the value of ENABLE_SOAP_SSL.

**ENABLE_SOAP_SSL** A boolean value that defaults to `False`. When `True`, enables SOAP over SSL for all daemons.

**<SUBSYS>_SOAP_SSL_PORT** A required port number on which SOAP over SSL messages are accepted, when SOAP over SSL is enabled. The <SUBSYS> must be specified, because multiple daemons running on a single machine may not share a port. There is no default value.

The macro is named by substituting <SUBSYS> with the appropriate subsystem string as defined in section 3.3.1.

**SOAP_SSL_SERVER_KEYFILE** A required complete path and file name to specify the daemon's identity, as used in authentication when SOAP over SSL is enabled. The file is to be an OpenSSL PEM file containing a certificate and private key. There is no default value.

**SOAP_SSL_SERVER_KEYFILE_PASSWORD** An optional complete path and file name to specify a password for unlocking the daemon's private key. There is no default value.

**SOAP_SSL_CA_FILE** A required complete path and file name to specify a file containing certificates of trusted Certificate Authorities (CAs). Only clients who present a certificate signed by a trusted CA will be authenticated. There is no default value.

**SOAP_SSL_CA_DIR** A required complete path to a directory containing certificates of trusted Certificate Authorities (CAs). Only clients who present a certificate signed by a trusted CA will be authenticated. There is no default value.

**SOAP_SSL_DH_FILE** An optional complete path and file name to a DH file containing keys for a DH key exchange. There is no default value.

### 3.3.29 Stork Configuration File Macros

**STORK_MAX_NUM_JOBS** An integer limit on the number of concurrent data placement jobs handled by Stork. The default value when not defined is 10.

**STORK_MAX_RETRY** An integer limit on the number of attempts for a single data placement job. For data transfers, this includes transfer attempts on the primary protocol, all alternate protocols, and all retries. The default value when not defined is 10.

**STORK_MAXDELAY_INMINUTES** An integer limit (in minutes) on the run time for a data placement job, after which the job is considered failed. The default value when not defined is 10, and the minimum legal value is 1.

**STORK_TMP_CRED_DIR** The full path to the temporary credential storage directory used by Stork. The default value is `/tmp` when not defined.

**STORK_MODULE_DIR** The full path to the directory containing Stork modules. The default value when not defined is as defined by `$(LIBEXEC)`. It is a fatal error for both `STORK_MODULE_DIR` and `LIBEXEC` to be undefined.

**CRED_SUPER_USERS** Access to a stored credential is restricted to the user who submitted the credential, and any user names specified in this macro. The format is a space or comma separated list of user names which are valid on the *stork_credd* host. The default value of this macro is `root` on Unix systems, and `Administrator` on Windows systems.

**CRED_STORE_DIR** Directory for storing credentials. This directory must exist prior to starting *stork_credd*. It is highly recommended to restrict access permissions to *only* the directory owner. The default value is `$(SPOOL_DIR)/cred`.

**CRED_INDEX_FILE** Index file path of saved credentials. This file will be automatically created if it does not exist. The default value is `$(CRED_STORE_DIR)/cred-index`.

**DEFAULT_CRED_EXPIRE_THRESHOLD** *stork_credd* will attempt to refresh credentials when their remaining lifespan is less than this value. Units = seconds. Default value = 3600 seconds (1 hour).

**CRED_CHECK_INTERVAL** *stork_credd* periodically checks remaining lifespan of stored credentials, at this interval. Units = seconds. Default value = 60 seconds (1 minute).

## 3.4 User Priorities and Negotiation

Condor uses priorities to determine machine allocation for jobs. This section details the priorities and the allocation of machines (negotiation).

For accounting purposes, each user is identified by username@uid_domain. Each user is assigned a priority value even if submitting jobs from different machines in the same domain, or even if submitting from multiple machines in the different domains.

The numerical priority value assigned to a user is inversely related to the *goodness* of the priority. A user with a numerical priority of 5 gets more resources than a user with a numerical priority of 50. There are two priority values assigned to Condor users:

- Real User Priority (RUP), which measures resource usage of the user.
- Effective User Priority (EUP), which determines the number of resources the user can get.

This section describes these two priorities and how they affect resource allocations in Condor. Documentation on configuring and controlling priorities may be found in section 3.3.18.

### 3.4.1 Real User Priority (RUP)

A user's RUP measures the resource usage of the user through time. Every user begins with a RUP of one half (0.5), and at steady state, the RUP of a user equilibrates to the number of resources used by that user. Therefore, if a specific user continuously uses exactly ten resources for a long period of time, the RUP of that user stabilizes at ten.

However, if the user decreases the number of resources used, the RUP gets better. The rate at which the priority value decays can be set by the macro `PRIORITY_HALFLIFE` , a time period defined in seconds. Intuitively, if the `PRIORITY_HALFLIFE` in a pool is set to 86400 (one day), and if a user whose RUP was 10 removes all his jobs, the user's RUP would be 5 one day later, 2.5 two days later, and so on.

### 3.4.2 Effective User Priority (EUP)

The effective user priority (EUP) of a user is used to determine how many resources that user may receive. The EUP is linearly related to the RUP by a *priority factor* which may be defined on a per-user basis. Unless otherwise configured, the priority factor for all users is 1.0, and so the EUP is the same as the the RUP. However, if desired, the priority factors of specific users (such as remote submitters) can be increased so that others are served preferentially.

The number of resources that a user may receive is inversely related to the ratio between the EUPs of submitting users. Therefore user $A$ with EUP=5 will receive twice as many resources as user $B$ with EUP=10 and four times as many resources as user $C$ with EUP=20. However, if $A$ does not use the full number of allocated resources, the available resources are repartitioned and distributed among remaining users according to the inverse ratio rule.

Condor supplies mechanisms to directly support two policies in which EUP may be useful:

**Nice users** A job may be submitted with the parameter `nice_user` set to TRUE in the submit command file. A nice user job gets its RUP boosted by the `NICE_USER_PRIO_FACTOR` priority factor specified in the configuration file, leading to a (usually very large) EUP. This corresponds to a low priority for resources. These jobs are therefore equivalent to Unix background jobs, which use resources not used by other Condor users.

**Remote Users** The flocking feature of Condor (see section 5.2) allows the *condor_schedd* to submit to more than one pool. In addition, the submit-only feature allows a user to run a *condor_schedd* that is submitting jobs into another pool. In such situations, submitters from other domains can submit to the local pool. It is often desirable to have Condor treat local users preferentially over these remote users. If configured, Condor will boost the RUPs of remote users by `REMOTE_PRIO_FACTOR` specified in the configuration file, thereby lowering their priority for resources.

The priority boost factors for individual users can be set with the **setfactor** option of *condor_userprio*. Details may be found in the *condor_userprio* manual page on page 755.

### 3.4.3 Priorities and Preemption

Priorities are used to ensure that users get their fair share of resources. The priority values are used at allocation time. In addition, Condor preempts machine claims and reallocates them when conditions change.

To ensure that preemptions do not lead to *thrashing*, a `PREEMPTION_REQUIREMENTS` expression is defined to specify the conditions that must be met for a preemption to occur. It is usually defined to deny preemption if a current running job has been running for a relatively short period of time. This effectively limits the number of preemptions per resource per time interval.

Note that `PREEMPTION_REQUIREMENTS` only applies to preemptions due to user priority. It does not have any effect if the machine rank expression prefers a different job, or if the startd policy expression causes the job to vacate due to other activity on the machine. See section 3.5.10 for a general discussion of limiting preemption.

### 3.4.4 Priority Calculation

This section may be skipped if the reader so feels, but for the curious, here is Condor's priority calculation algorithm.

The RUP of a user $u$ at time $t$, $\pi_r(u,t)$, is calculated every time interval $\delta t$ using the formula

$$\pi_r(u,t) = \beta \times \pi(u,t-\delta t) + (1-\beta) \times \rho(u,t)$$

where $\rho(u,t)$ is the number of resources used by user $u$ at time $t$, and $\beta = 0.5^{\delta t/h}$. $h$ is the half life period set by `PRIORITY_HALFLIFE` .

The EUP of user $u$ at time $t$, $\pi_e(u,t)$ is calculated by

$$\pi_e(u,t) = \pi_r(u,t) \times f(u,t)$$

where $f(u,t)$ is the priority boost factor for user $u$ at time $t$.

As mentioned previously, the RUP calculation is designed so that at steady state, each user's RUP stabilizes at the number of resources used by that user. The definition of $\beta$ ensures that the calculation of $\pi_r(u,t)$ can be calculated over non-uniform time intervals $\delta t$ without affecting the calculation. The time interval $\delta t$ varies due to events internal to the system, but Condor guarantees that unless the central manager machine is down, no matches will be unaccounted for due to this variance.

### 3.4.5 Negotiation

Negotiation is the method Condor undergoes periodically to match queued jobs with resources capable of running jobs. The *condor_negotiator* daemon is responsible for negotiation.

During a negotiation cycle, the *condor_negotiator* daemon accomplishes the following ordered list of items.

1. Build a list of all possible resources, regardless of the state of those resources.
2. Obtain a list of all job submitters (for the entire pool).
3. Sort the list of all job submitters based on EUP (see section 3.4.2 for an explanation of EUP). The submitter with the best priority is first within the sorted list.
4. Iterate until there are either no more resources to match, or no more jobs to match.

   For each submitter (in EUP order):

   For each submitter, get each job. Since jobs may be submitted from more than one machine (hence to more than one *condor_schedd* daemon), here is a further definition of the ordering of these jobs. With jobs from a single *condor_schedd* daemon, jobs are typically returned in job priority order. When more than one *condor_schedd* daemon is involved, they are contacted in an undefined order. All jobs from a single *condor_schedd* daemon are considered before moving on to the next. For each job:

   - For each machine in the pool that can execute jobs:
     (a) If `machine.requirements` evaluates to `False` or `job.requirements` evaluates to `False`, skip this machine
     (b) If the machine is in the Claimed state, but not running a job, skip this machine.
     (c) If this machine is not running a job, add it to the potential match list by reason of No Preemption.
     (d) If the machine is running a job
       – If the `machine.RANK` on this job is better than the running job, add this machine to the potential match list by reason of Rank.
       – If the EUP of this job is better than the EUP of the currently running job, and `PREEMPTION_REQUIREMENTS` is `True`, and the `machine.RANK` on this job is not worse than the currently running job, add this machine to the potential match list by reason of Priority.
   - Of machines in the potential match list, sort by `NEGOTIATOR_PRE_JOB_RANK`, `job.RANK`, `NEGOTIATOR_POST_JOB_RANK`, Reason for claim (No Preemption, then Rank, then Priority), `PREEMPTION_RANK`
   - The job is assigned to the top machine on the potential match list. The machine is removed from the list of resources to match (on this negotiation cycle).

The *condor_negotiatior* asks the *condor_schedd* for the ”next job” from a given submitter/user. Typically, the *condor_schedd* returns jobs in the order of job priority. If priorities are the same, job submission time is used; older jobs go first. If a cluster has multiple procs in it and one of the jobs cannot be matched, the *condor_schedd* will not return any more jobs in that cluster on that negotiation pass. This is an optimization based on the theory that the cluster jobs are similar.

The configuration variable NEGOTIATE_ALL_JOBS_IN_CLUSTER disables the cluster-skipping optimization. Use of the configuration variable SIGNIFICANT_ATTRIBUTES will change the definition of what the *condor_schedd* considers a cluster from the default definition of all jobs that share the same ClusterId.

### 3.4.6 Group Accounting

By default, Condor does all accounting on a per-user basis, and this accounting is primarily used to compute priorities for Condor's fair-share scheduling algorithms. However, accounting can also be done on a per-group basis. Multiple users can all submit jobs into the same accounting group, and all of the jobs will be treated with the same priority.

To use an accounting group, each job inserts an attribute into the job ClassAd which defines the accounting group name for the job. A common name is decided upon and used for the group. The following line is an example that defines the attribute within the job's submit description file:

```
+AccountingGroup = "group_physics"
```

The AccountingGroup attribute is a string, and it therefore must be enclosed in double quote marks. The string may have a maximum length of 40 characters. The name should *not* be qualified with a domain. Certain parts of the Condor system do append the value $(UID_DOMAIN) (as specified in the configuration file on the submit machine) to this string for internal use. For example, if the value of UID_DOMAIN is example.com, and the accounting group name is as specified, *condor_userprio* will show statistics for this accounting group using the appended domain, for example

```
                                  Effective
User Name                         Priority
------------------------------    ---------
group_physics@example.com              0.50
user@example.com                      23.11
heavyuser@example.com                111.13
...
```

Additionally, the *condor_userprio* command allows administrators to remove an entity from the accounting system in Condor. The **-delete** option to *condor_userprio* accomplishes this if all the jobs from a given accounting group are completed, and the administrator wishes to remove that group from the system. The **-delete** option identifies the accounting group with the fully-qualified name of the accounting group. For example

```
condor_userprio -delete group_physics@example.com
```

Condor removes entities itself as they are no longer relevant. Intervention by an administrator to delete entities can be beneficial when the use of thousands of short term accounting groups leads to scalability issues.

Note that the name of an accounting group may include a period (.), and the period character has no special interpretation for group accounting. For group users, as described in the next section, a period does have special meaning.

### 3.4.7 Group Quotas

The use of group quotas modifies the negotiation for available resources (machines) within a Condor pool. This solves the difficulties inherent when priorities assigned based on each single user are insufficient. This may be the case when different groups (of varying size) own computers, and the groups choose to combine their computers to form a Condor pool. Consider an imaginary Condor pool example with thirty computers. Twenty computers are owned by the physics group and ten computers are owned by the chemistry group. One notion of fair allocation could be implemented by configuring the twenty machines owned by the physics group to prefer (using the RANK configuration macro) jobs submitted by the users identified as associated with the physics group. Likewise, the ten machines owned by the chemistry group are configured to prefer jobs from users associated with the the chemistry group. This routes jobs to execute on specific machines, perhaps causing more preemption than necessary. The (fair allocation) policy desired is likely somewhat different, if these thirty machines have been pooled. It does not tie users to specific sets of machines, but to numbers of machines (a quota). Given thirty similar machines, the desired policy allows users within the physics group to have preference on up to twenty of the machines within the pool, and the machines can be any of the machines that are available.

A quota for a set of users requires an identification for the set; members are called group users. Jobs to be negotiated for under the group quota specify the group user with the `AccountingGroup` job ClassAd, as described above.

The syntax for specifying a group user is

```
group.user
```

The `group` is a name chosen for the group. Group names are not required to begin with the string `"group_"`, as in the examples `"group_physics.newton"` and `"group_chemistry.curie"`, but it is a useful convention because group names must not conflict with user names. The period character between the group and the user name is a required part of the syntax.

Configuration controls the order of negotiation for groups and individual users, as well as sets quotas (preferentially allocated number of machines) for the groups. A declared number of virtual machines specifies the quota for each group (see `GROUP_QUOTA_<groupname>` in section 3.3.18). The sum of the quotas for all groups must be less than or equal to the number of virtual machines in the entire pool. If the sum is less than the number of virtual machines in the entire pool, the remaining machines are allocated to the `none` group, comprised of the general users not submitting jobs in a group.

Where group users are specified for jobs, accounting is done per group user. It is no longer done by group, or by individual user.

Negotiation is changed when group quotas are used. Condor negotiates first for defined groups, and then for independent job submitters. Given jobs belonging to different groups, Condor negotiates first for the group currently utilizing the smallest percentage of machines in its quota. After this, Condor negotiates for the group currently utilizing the second smallest percentage of machines

in its quota. The last group will be the one with the highest percentage of machines in its quota. As an example, again use the imaginary pool and groups given above. If various users within `group_physics` have jobs running on 15 computers, then the physics group has 75% of the machines within its quota. If various users within `group_chemistry` have jobs running on 5 computers, then the chemistry group has 50% of the machines within its quota. Negotiation will take place for the chemistry group first. For independent job submissions (those not part of any group), the classic Condor user fair share algorithm still applies.

Note that there is no verification that a user is a member of the group that he claims. We rely on societal pressure for enforcement.

Configuration variables affect group quotas. See section 3.3.18 for detailed descriptions of the variables mentioned. Group names that may be given quotas to be used in negotiation are listed in the `GROUP_NAMES` macro. The names chosen must not conflict with Condor user names. Quotas (by group) are defined in numbers of virtual machines. Each group may assign an initial value for each group user's user priority factor with the `GROUP_PRIO_FACTOR_<groupname>` macro. If a group is currently allocated its entire quota of machines, and a group user has a submitted job that is not running, the `GROUP_AUTOREGROUP` macro allows the job to be considered a second time within the negotiation cycle along with all other individual users' jobs.

## 3.5 Startd Policy Configuration

This section describes the configuration of the *condor_startd* to implement the desired policy for when remote jobs should start, be suspended, (possibly) resumed, vacate (with a checkpoint) or be killed (no checkpoint). This policy is the heart of Condor's balancing act between the needs and wishes of resource owners (machine owners) and resource users (people submitting their jobs to Condor). Please read this section carefully if you plan to change any of the settings described here, as a wrong setting can have a severe impact on either the owners of machines in your pool (they may ask to be removed from the pool entirely) or the users of your pool (they may stop using Condor).

Before we get into the details, there are a few things to note:

- Much of this section refers to ClassAd expressions. You probably want to read through section 4.1 on ClassAd expressions before continuing with this.
- If you are primarily familiar with the version 6.0 policy expressions and what they do, read section 3.5.11 on page 259. This section explains the differences between the version 6.0 policy expressions and later versions.
- If you are defining the policy for an SMP machine (a multi-CPU machine), also read section 3.12.7 for specific information on configuring the *condor_startd* for SMP machines. Each *virtual machine* represented by the *condor_startd* on an SMP machine has its own *state* and *activity* (as described below). In the future, each virtual machine will be able to have its own individual policy expressions defined. Within this manual section, the word "machine" refers to an individual virtual machine within an SMP machine.

To define your policy, set expressions in the configuration file (see section 3.3 on Configuring Condor for an introduction to Condor's configuration files). The expressions are evaluated in the context of the machine's ClassAd and a job ClassAd. The expressions can therefore reference attributes from either ClassAd. Listed in this section are both the attributes that are included in the machine's ClassAd and the attributes that are included in a job ClassAd. The START expression is explained. It describes the conditions that must be met for a machine to start a job. The RANK expression is described. It allows the specification of the kinds of jobs a machine prefers to run. A final discussion details how the *condor_startd* daemon works. Included are the machine *states* and *activities*, to give an idea of what is possible in policy decisions. Two example policy settings are presented.

### 3.5.1 Startd ClassAd Attributes

The *condor_startd* daemon represents the machine on which it is running to the Condor pool. The daemon publishes characteristics about the machine in the machine's ClassAd to aid matchmaking with resource requests. The values of these attributes may be listed by using the command: *condor_status -l hostname*. On an SMP machine, the *condor_startd* will break the machine up and advertise it as separate virtual machines, each with its own name and ClassAd. The attributes themselves and what they represent are described below:

**Activity:** String which describes Condor job activity on the machine. Can have one of the following values:

- **"Idle":** There is no job activity
- **"Busy":** A job is busy running
- **"Suspended":** A job is currently suspended
- **"Vacating":** A job is currently checkpointing
- **"Killing":** A job is currently being killed
- **"Benchmarking":** The startd is running benchmarks

**Arch:** String with the architecture of the machine. Typically one of the following:

- **"INTEL":** Intel x86 CPU (Pentium, Xeon, etc).
- **"IA64":** Intel 64-bit CPU
- **"ALPHA":** Digital Alpha CPU
- **"SGI":** Silicon Graphics MIPS CPU
- **"SUN4u":** Sun UltraSparc CPU
- **"SUN4x":** A Sun Sparc CPU other than an UltraSparc, i.e. sun4m or sun4c CPU found in older Sparc workstations such as the Sparc 10, Sparc 20, IPC, IPX, etc.
- **"PPC":** Power Macintosh
- **"HPPA1":** Hewlett Packard PA-RISC 1.x CPU (i.e. PA-RISC 7000 series CPU) based workstation

**"HPPA2":** Hewlett Packard PA-RISC 2.x CPU (i.e. PA-RISC 8000 series CPU) based workstation

**CheckpointPlatform:** A string which opaquely encodes various aspects about a machine's operating system, hardware, and kernel attributes. It is used to identify systems where previously taken checkpoints for the standard universe may resume.

**ClockDay:** The day of the week, where 0 = Sunday, 1 = Monday, . . ., 6 = Saturday.

**ClockMin:** The number of minutes passed since midnight.

**CondorLoadAvg:** The portion of the load average generated by Condor (either from remote jobs or running benchmarks).

**ConsoleIdle:** The number of seconds since activity on the system console keyboard or console mouse has last been detected.

**Cpus:** Number of CPUs in this machine, i.e. 1 = single CPU machine, 2 = dual CPUs, etc.

**CurrentRank:** A float which represents this machine owner's affinity for running the Condor job which it is currently hosting. If not currently hosting a Condor job, `CurrentRank` is 0.0. When a machine is claimed, the attribute's value is computed by evaluating the machine's `Rank` expression with respect to the current job's ClassAd.

**Disk:** The amount of disk space on this machine available for the job in Kbytes ( e.g. 23000 = 23 megabytes ). Specifically, this is the amount of disk space available in the directory specified in the Condor configuration files by the `EXECUTE` macro, minus any space reserved with the `RESERVED_DISK` macro.

**EnteredCurrentActivity:** Time at which the machine entered the current Activity (see `Activity` entry above). On all platforms (including NT), this is measured in the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

**FileSystemDomain:** A "domain" name configured by the Condor administrator which describes a cluster of machines which all access the same, uniformly-mounted, networked file systems usually via NFS or AFS. This is useful for Vanilla universe jobs which require remote file access.

**KeyboardIdle:** The number of seconds since activity on any keyboard or mouse associated with this machine has last been detected. Unlike `ConsoleIdle`, `KeyboardIdle` also takes activity on pseudo-terminals into account (i.e. virtual "keyboard" activity from telnet and rlogin sessions as well). Note that `KeyboardIdle` will always be equal to or less than `ConsoleIdle`.

**KFlops:** Relative floating point performance as determined via a Linpack benchmark.

**LastHeardFrom:** Time when the Condor central manager last received a status update from this machine. Expressed as the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970). Note: This attribute is only inserted by the central manager once it receives the ClassAd. It is not present in the *condor_startd* copy of the ClassAd. Therefore, you could not use this attribute in defining *condor_startd* expressions (and you would not want to).

**LoadAvg:** A floating point number with the machine's current load average.

**Machine:** A string with the machine's fully qualified hostname.

**Memory:** The amount of RAM in megabytes.

**Mips:** Relative integer performance as determined via a Dhrystone benchmark.

**MyType:** The ClassAd type; always set to the literal string `"Machine"`.

**Name:** The name of this resource; typically the same value as the `Machine` attribute, but could be customized by the site administrator. On SMP machines, the *condor_startd* will divide the CPUs up into separate virtual machines, each with with a unique name. These names will be of the form "vm#@full.hostname", for example, "vm1@vulture.cs.wisc.edu", which signifies virtual machine 1 from vulture.cs.wisc.edu.

**OpSys:** String describing the operating system running on this machine. For Condor Version 6.8.6 typically one of the following:

**"HPUX10":** for HPUX 10.20

**"HPUX11":** for HPUX B.11.00

**"LINUX":** for LINUX 2.0.x, LINUX 2.2.x, LINUX 2.4.x, or LINUX 2.6.x kernel systems

**"OSF1":** for Digital Unix 4.x

**"SOLARIS25":** for Solaris 2.4 or 5.5

**"SOLARIS251":** for Solaris 2.5.1 or 5.5.1

**"SOLARIS26":** for Solaris 2.6 or 5.6

**"SOLARIS27":** for Solaris 2.7 or 5.7

**"SOLARIS28":** for Solaris 2.8 or 5.8

**"SOLARIS29":** for Solaris 2.9 or 5.9

**"WINNT50":** for Windows 2000

**"WINNT51":** for Windows XP

**"WINNT52":** for Windows Server 2003

**"OSX":** for Darwin

**"OSX10_2":** for Darwin 6.4

**Requirements:** A boolean, which when evaluated within the context of the machine ClassAd and a job ClassAd, must evaluate to TRUE before Condor will allow the job to use this machine.

**MaxJobRetirementTime:** An expression giving the maximum time in seconds that the startd will wait for the job to finish before kicking it off if it needs to do so. This is evaluated in the context of the job ClassAd, so it may refer to job attributes as well as machine attributes.

**StartdIpAddr:** String with the IP and port address of the *condor_startd* daemon which is publishing this machine ClassAd.

**State:** String which publishes the machine's Condor state. Can be:

**"Owner":** The machine owner is using the machine, and it is unavailable to Condor.

**"Unclaimed":** The machine is available to run Condor jobs, but a good match is either not available or not yet found.

**"Matched":** The Condor central manager has found a good match for this resource, but a Condor scheduler has not yet claimed it.

**"Claimed":** The machine is claimed by a remote *condor_schedd* and is probably running a job.

**"Preempting":** A Condor job is being preempted (possibly via checkpointing) in order to clear the machine for either a higher priority job or because the machine owner wants the machine back.

**TargetType:** Describes what type of ClassAd to match with. Always set to the string literal "Job", because machine ClassAds always want to be matched with jobs, and vice-versa.

**UidDomain:** a domain name configured by the Condor administrator which describes a cluster of machines which all have the same passwd file entries, and therefore all have the same logins.

**VirtualMachineID:** For SMP machines, the integer that identifies the VM. The value will be X for the VM with

```
name="vmX@full.hostname"
```

For non-SMP machines with one virtual machine, the value will be 1.

**VirtualMemory:** The amount of currently available virtual memory (swap space) expressed in Kbytes.

In addition, there are a few attributes that are automatically inserted into the machine ClassAd whenever a resource is in the Claimed state:

**ClientMachine:** The hostname of the machine that has claimed this resource

**RemoteOwner:** The name of the user who originally claimed this resource.

**RemoteUser:** The name of the user who is currently using this resource. In general, this will always be the same as the RemoteOwner, but in some cases, a resource can be claimed by one entity that hands off the resource to another entity which uses it. In that case, RemoteUser would hold the name of the entity currently using the resource, while RemoteOwner would hold the name of the entity that claimed the resource.

**PreemptingOwner:** The name of the user who is preempting the job that is currently running on this resource.

**PreemptingUser:** The name of the user who is preempting the job that is currently running on this resource. The relationship between `PreemptingUser` and `PreemptingOwner` is the same as the relationship between `RemoteUser` and `RemoteOwner`.

**PreemptingRank:** A float which represents this machine owner's affinity for running the Condor job which is waiting for the current job to finish or be preempted. If not currently hosting a Condor job, `PreemptingRank` is undefined. When a machine is claimed and there is already a job running, the attribute's value is computed by evaluating the machine's `Rank` expression with respect to the preempting job's ClassAd.

**TotalClaimRunTime:** A running total of the amount of time (in seconds) that all jobs (under the same claim) ran (have spent in the Claimed/Busy state).

**TotalClaimSuspendTime:** A running total of the amount of time (in seconds) that all jobs (under the same claim) have been suspended (in the Claimed/Suspended state).

**TotalJobRunTime:** A running total of the amount of time (in seconds) that a single job ran (has spent in the Claimed/Busy state).

**TotalJobSuspendTime:** A running total of the amount of time (in seconds) that a single job has been suspended (in the Claimed/Suspended state).

There are a few attributes that are only inserted into the machine ClassAd if a job is currently executing. If the resource is claimed but no job are running, none of these attributes will be defined.

**JobId:** The job's identifier (for example, `152.3`), as seen from *condor_q* on the submitting machine.

**JobStart:** The time stamp in integer seconds of when the job began executing, since the Unix epoch (00:00:00 UTC, Jan 1, 1970). For idle machines, the value is UNDEFINED.

**LastPeriodicCheckpoint:** If the job has performed a periodic checkpoint, this attribute will be defined and will hold the time stamp of when the last periodic checkpoint was begun. If the job has yet to perform a periodic checkpoint, or cannot checkpoint at all, the `LastPeriodicCheckpoint` attribute will not be defined.

Finally, the single attribute, `CurrentTime`, is defined by the ClassAd environment.

**CurrentTime:** Evaluates to the the number of integer seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970).

### 3.5.2 Job ClassAd Attributes

**Args:** String representing the arguments passed to the job.

**CkptArch:** String describing the architecture of the machine this job executed on at the time it last produced a checkpoint. If the job has never produced a checkpoint, this attribute is `undefined`.

**CkptOpSys:** String describing the operating system of the machine this job executed on at the time it last produced a checkpoint. If the job has never produced a checkpoint, this attribute is `undefined`.

**ClusterId:** Integer cluster identifier for this job. A cluster is a group of jobs that were submitted together. Each job has its own unique job identifier within the cluster, but shares a common cluster identifier. The value changes each time a job or set of jobs are queued for execution under Condor.

**Cmd:** The path to and the file name of the job to be executed.

**CompletionDate:** The time when the job completed, or the value 0 if the job has not yet completed. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

**CumulativeSuspensionTime:** A running total of the number of seconds the job has spent in suspension for the life of the job.

**CurrentHosts:** The number of hosts in the claimed state, due to this job.

**EnteredCurrentStatus:** An integer containing the epoch time of when the job entered into its current status So for example, if the job is on hold, the ClassAd expression

```
CurrentTime - EnteredCurrentStatus
```

will equal the number of seconds that the job has been on hold.

**ExecutableSize:** Size of the executable in Kbytes.

**ExitBySignal:** An attribute that is `True` when a user job exits via a signal and `False` otherwise. For some grid universe jobs, how the job exited is unavailable. In this case, `ExitBySignal` is set to `False`.

**ExitCode:** When a user job exits by means other than a signal, this is the exit return code of the user job. For some grid universe jobs, how the job exited is unavailable. In this case, `ExitCode` is set to 0.

**ExitSignal:** When a user job exits by means of an unhandled signal, this attribute takes on the numeric value of the signal. For some grid universe jobs, how the job exited is unavailable. In this case, `ExitSignal` will be undefined.

**ExitStatus:** The way that Condor previously dealt with a job's exit status. This attribute should no longer be used. It is not always accurate in heterogeneous pools, or if the job exited with a signal. Instead, see the attributes: `ExitBySignal`, `ExitCode`, and `ExitSignal`.

**HoldReasonCode:** An integer value that represents the reason that a job was put on hold.

| *Integer Code* | *Reason for Hold* |
|---|---|
| 1 | The user put the job on hold with *condor_hold*. |
| 2 | Globus middleware reported an error. `HoldReasonSubCode` is the GRAM error number. |
| 3 | The `PERIODIC_HOLD` expression evaluated to `True`. |
| 4 | The credentials for the job are invalid. |
| 5 | A job policy expression evaluated to `Undefined`. |
| 6 | The *condor_starter* failed to start the executable. `HoldReasonSubCode` is the Unix error number. |
| 7 | The standard output file for the job could not be opened. `HoldReasonSubCode` is the Unix error number |
| 8 | The standard input file for the job could not be opened. `HoldReasonSubCode` is the Unix error number. |
| 9 | The standard output stream for the job could not be opened. `HoldReasonSubCode` is the Unix error num |
| 10 | The standard input stream for the job could not be opened. `HoldReasonSubCode` is the Unix error numb |
| 11 | An internal Condor protocol error was encountered when transferring files. |
| 12 | The *condor_starter* failed to download input files. `HoldReasonSubCode` is the Unix error number. |
| 13 | The *condor_starter* failed to upload output files. `HoldReasonSubCode` is the Unix error number. |
| 14 | The initial working directory of the job cannot be accessed. `HoldReasonSubCode` is the Unix error num |

**`HoldReasonSubCode`:** An integer value that represents further information to go along with the `HoldReasonCode`, for some values of `HoldReasonCode`. See `HoldReasonCode` for the values.

**`HoldKillSig`:** Currently only for scheduler and local universe jobs, a string containing a name of a signal to be sent to the job if the job is put on hold.

**`HoldReason`:** A string containing a human-readable message about why a job is on hold. This is the message that will be displayed in response to the command `condor_q -hold`. It can be used to determine if a job should be released or not.

**`ImageSize`:** Estimate of the memory image size of the job in Kbytes. The initial estimate may be specified in the job submit file. Otherwise, the initial value is equal to the size of the executable. When the job checkpoints, the `ImageSize` attribute is set to the size of the checkpoint file (since the checkpoint file contains the job's memory image). A vanilla universe job's `ImageSize` is recomputed internally every 15 seconds.

**`JobLeaseDuration`:** The number of seconds set for a job lease, the amount of time that a job may continue running on a remote resource, despite its submitting machine's lack of response. See section 2.14.4 for details on job leases.

**`JobPrio`:** Integer priority for this job, set by *condor_submit* or *condor_prio*. The default value is 0. The higher the number, the greater (better) the priority.

**`JobStartDate`:** Time at which the job first began running. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

**`JobStatus`:** Integer which indicates the current status of the job.

| *Value* | *Status* |
|---|---|
| 0 | Unexpanded (the job has never run) |
| 1 | Idle |
| 2 | Running |
| 3 | Removed |
| 4 | Completed |
| 5 | Held |

| *Value* | *Universe* |
|---|---|
| 1 | standard |
| 4 | PVM |
| 5 | vanilla |
| 7 | scheduler |
| 8 | MPI |
| 9 | grid |
| 10 | java |
| 11 | parallel |
| 12 | local |

**JobUniverse:** Integer which indicates the job universe.

**LastCheckpointPlatform:** An opaque string which is the `CheckpointPlatform` identifier from the last machine where this standard universe job had successfully produced a checkpoint.

**LastCkptServer:** Host name of the last checkpoint server used by this job. When a pool is using multiple checkpoint servers, this tells the job where to find its checkpoint file.

**LastCkptTime:** Time at which the job last performed a successful checkpoint. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

**LastMatchTime:** An integer containing the epoch time when the job was last successfully matched with a resource (gatekeeper) Ad.

**LastRejMatchReason:** If, at any point in the past, this job failed to match with a resource ad, this attribute will contain a string with a human-readable message about why the match failed.

**LastRejMatchTime:** An integer containing the epoch time when Condor-G last tried to find a match for the job, but failed to do so.

**LastSuspensionTime:** Time at which the job last performed a successful suspension. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

**LastVacateTime:** Time at which the job was last evicted from a remote workstation. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

**LocalSysCpu:** An accumulated number of seconds of system CPU time that the job caused to the machine upon which the job was submitted.

**LocalUserCpu:** An accumulated number of seconds of user CPU time that the job caused to the machine upon which the job was submitted.

**MaxHosts:** The maximum number of hosts that this job would like to claim. As long as `CurrentHosts` is the same as `MaxHosts`, no more hosts are negotiated for.

**MaxJobRetirementTime:** Maximum time in seconds to let this job run uninterrupted before kicking it off when it is being preempted. This can only decrease the amount of time from what the corresponding startd expression allows.

**MinHosts:** The minimum number of hosts that must be in the claimed state for this job, before the job may enter the running state.

**NiceUser:** Boolean value which indicates whether this is a nice-user job.

**NumCkpts:** A count of the number of checkpoints written by this job during its lifetime.

**NumGlobusSubmits:** An integer that is incremented each time the *condor_gridmanager* receives confirmation of a successful job submission into Globus.

**NumJobMatches:** An integer that is incremented by the *condor_schedd* each time the job is matched with a resource ad by the negotiator.

**NumRestarts:** A count of the number of restarts from a checkpoint attempted by this job during its lifetime.

**NumSystemHolds:** An integer that is incremented each time Condor-G places a job on hold due to some sort of error condition. This counter is useful, since Condor-G will always place a job on hold when it gives up on some error condition. Note that if the user places the job on hold using the *condor_hold* command, this attribute is not incremented.

**Owner:** String describing the user who submitted this job.

**ProcId:** Integer process identifier for this job. Within a cluster of many jobs, each job has the same `ClusterId`, but will have a unique `ProcId`. Within a cluster, assignment of a `ProcId` value will start with the value 0. The job (process) identifier described here is unrelated to operating system PIDs.

**QDate:** Time at which the job was submitted to the job queue. Measured in the number of seconds since the epoch (00:00:00 UTC, Jan 1, 1970).

**ReleaseReason:** A string containing a human-readable message about why the job was released from hold.

**RemoteIwd:** The path to the directory in which a job is to be executed on a remote machine.

**RemoteSysCpu:** The total number of seconds of system CPU time (the time spent at system calls) the job used on remote machines.

**RemoteUserCpu:** The total number of seconds of user CPU time the job used on remote machines.

**RemoteWallClockTime:** Cumulative number of seconds the job has been allocated a machine. This also includes time spent in suspension (if any), so the total real time spent running is

```
RemoteWallClockTime - CumulativeSuspensionTime
```

Note that this number does not get reset to zero when a job is forced to migrate from one machine to another.

**RemoveKillSig:** Currently only for scheduler universe jobs, a string containing a name of a signal to be sent to the job if the job is removed.

**StreamErr:** An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, and `TransferErr` is `True`, then standard error is streamed back to the submit machine, instead of doing the transfer (as a whole) after the job completes. If `False`, then standard error is transfered back to the submit machine (as a whole) after the job completes. If `TransferErr` is `False`, then this job attribute is ignored.

**StreamOut:** An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, and `TransferOut` is `True`, then job output is streamed back to the submit machine, instead of doing the transfer (as a whole) after the job completes. If `False`, then job output is transferred back to the submit machine (as a whole) after the job completes. If `TransferOut` is `False`, then this job attribute is ignored.

**TotalSuspensions:** A count of the number of times this job has been suspended during its lifetime.

**TransferErr:** An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the error output from the job is transferred from the remote machine back to the submit machine. The name of the file after transfer is the file referred to by job attribute `Err`. If `False`, no transfer takes place (remote to submit machine), and the name of the file is the file referred to by job attribute `Err`.

**TransferExecutable:** An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the job executable is transferred from the submit machine to the remote machine. The name of the file (on the submit machine) that is transferred is given by the job attribute `Cmd`. If `False`, no transfer takes place, and the name of the file used (on the remote machine) will be as given in the job attribute `Cmd`.

**TransferIn:** An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the job input is transferred from the submit machine to the remote machine. The name of the file that is transferred is given by the job attribute `In`. If `False`, then the job's input is taken from a file on the remote machine (pre-staged), and the name of the file is given by the job attribute `In`.

**TransferOut:** An attribute utilized only for grid universe jobs. The default value is `True`. If `True`, then the output from the job is transferred from the remote machine back to the submit

machine. The name of the file after transfer is the file referred to by job attribute `Out`. If `False`, no transfer takes place (remote to submit machine), and the name of the file is the file referred to by job attribute `Out`.

### 3.5.3 The START expression

The most important expression to the *condor_startd* is the `START` expression. This expression describes the conditions that must be met for a machine to run a job. This expression can reference attributes in the machine's ClassAd (such as `KeyboardIdle` and `LoadAvg`) and attributes in a job ClassAd (such as `Owner`, `Imagesize`, and `Cmd`, the name of the executable the job will run). The value of the `START` expression plays a crucial role in determining the state and activity of a machine.

The `Requirements` expression is used for matching machines with jobs.

The *condor_startd* defines the `Requirements` expression by logically **and**ing the `START` expression and the `IS_VALID_CHECKPOINT_PLATFORM` expression.

In situations where a machine wants to make itself unavailable for further matches, the `Requirements` expression is set to FALSE. When the `START` expression locally evaluates to TRUE, the machine advertises the `Requirements` expression as TRUE and does not publish the `START` expression.

Normally, the expressions in the machine ClassAd are evaluated against certain request ClassAds in the *condor_negotiator* to see if there is a match, or against whatever request ClassAd currently has claimed the machine. However, by locally evaluating an expression, the machine only evaluates the expression against its own ClassAd. If an expression cannot be locally evaluated (because it references other expressions that are only found in a request ad, such as `Owner` or `Imagesize`), the expression is (usually) undefined. See section 4.1 for specifics on how undefined terms are handled in ClassAd expression evaluation.

A note of caution is in order when modifying the `START` to reference job ClassAd attributes. The default `IsOwner` expression is a function of the `START` expression

```
START =?= FALSE
```

See a detailed discussion of the `IsOwner` expression in section 3.5.8. However, the machine locally evaluates the `IsOwner` expression to determine if it is capable of running jobs for Condor. Any job ClassAd attributes appearing in the `START` expression, and hence in the `IsOwner` expression are undefined in this context, and may lead to unexpected behavior. Whenever the `START` expression is modified to reference job ClassAd attributes, the `IsOwner` expression should also be modified to reference only machine ClassAd attributes.

NOTE: If you have machines with lots of real memory and swap space such that the only scarce resource is CPU time, consider defining `JOB_RENICE_INCREMENT` so that Condor starts jobs on the machine with low priority. Then, further configure to set up the machines with:

```
START = True
SUSPEND = False
PREEMPT = False
KILL = False
```

In this way, Condor jobs always run and can never be kicked off from activity on the machine. However, because they would run with "nice priority", interactive response on the machines will not suffer. You probably would not notice Condor was running the jobs, assuming you had enough free memory for the Condor jobs that there was little swapping.

### 3.5.4 The IS_VALID_CHECKPOINT_PLATFORM expression

A checkpoint is the platform-dependent information necessary to continue the execution of a standard universe job. Therefore, the machine (platform) upon which a job executed and produced a checkpoint limits the machines (platforms) which may use the checkpoint to continue job execution. This platform-dependent information is no longer the obvious combination of architecture and operating system, but may include subtle items such as the difference between the normal, bigmem, and hugemem kernels within the Linux operating system. This results in the incorporation of a separate expression to indicate the ability of a machine to resume and continue the execution of a job that has produced a checkpoint. The REQUIREMENTS expression is dependent on this information.

At a high level, IS_VALID_CHECKPOINT_PLATFORM is an expression which becomes true when a job's checkpoint platform matches the current checkpointing platform of the machine. Since this expression is **and**ed with the START expression to produce the REQUIREMENTS expression, it must also behave correctly when evaluating in the context of jobs that are not standard universe.

In words, the current default policy for this expression:

**Any non standard universe job may run on this machine. A standard universe job may run on machines with the new checkpointing identification system. A standard universe job may run if it has not yet produced a first checkpoint. If a standard universe job has produced a checkpoint, then make sure the checkpoint platforms between the job and the machine match.**

The following is the default boolean expression for this policy. A JobUniverse value of 1 denotes the standard universe. This expression may be overridden in the Condor configuration files.

```
IS_VALID_CHECKPOINT_PLATFORM =
(
  ( (TARGET.JobUniverse == 1) == FALSE) ||

  (
    (MY.CheckpointPlatform =!= UNDEFINED) &&
    (
      (TARGET.LastCheckpointPlatform =?= MY.CheckpointPlatform) ||
      (TARGET.NumCkpts == 0)
    )
  )
)
```

IS_VALID_CHECKPOINT_PLATFORM is a separate policy expression because the complexity of IS_VALID_CHECKPOINT_PLATFORM can be very high. While this functionality is conceptually separate from the normal START policies usually constructed, it is also a part of the Requirements to allow the job to run.

### 3.5.5 The RANK expression

A machine may be configured to prefer certain jobs over others using the RANK expression. It is an expression, like any other in a machine ClassAd. It can reference any attribute found in either the machine ClassAd or a request ad (normally, in fact, it references things in the request ad). The most common use of this expression is likely to configure a machine to prefer to run jobs from the owner of that machine, or by extension, a group of machines to prefer jobs from the owners of those machines.

For example, imagine there is a small research group with 4 machines called tenorsax, piano, bass, and drums. These machines are owned by the 4 users coltrane, tyner, garrison, and jones, respectively.

Assume that there is a large Condor pool in your department, but you spent a lot of money on really fast machines for your group. You want to implement a policy that gives priority on your machines to anyone in your group. To achieve this, set the RANK expression on your machines to reference the Owner attribute and prefer requests where that attribute matches one of the people in your group as in

```
RANK = Owner == "coltrane" || Owner == "tyner" \
       || Owner == "garrison" || Owner == "jones"
```

The RANK expression is evaluated as a floating point number. However, like in C, boolean expressions evaluate to either 1 or 0 depending on if they are TRUE or FALSE. So, if this expression evaluated to 1 (because the remote job was owned by one of the preferred users), it would be a larger value than any other user (for whom the expression would evaluate to 0).

A more complex RANK expression has the same basic set up, where anyone from your group has priority on your machines. Its difference is that the machine owner has better priority on their own machine. To set this up for Jimmy Garrison, place the following entry in Jimmy Garrison's local configuration file bass.local:

```
RANK = (Owner == "coltrane") + (Owner == "tyner") \
       + ((Owner == "garrison") * 10) + (Owner == "jones")
```

NOTE: The parentheses in this expression are important, because "+" operator has higher default precedence than "==".

The use of "+" instead of "| | " allows us to distinguish which terms matched and which ones didn't. If anyone not in the John Coltrane quartet was running a job on the machine called bass, the

RANK would evaluate numerically to 0, since none of the boolean terms evaluates to 1, and 0+0+0+0 still equals 0.

Suppose Elvin Jones submits a job. His job would match this machine (assuming the START was True for him at that time) and the RANK would numerically evaluate to 1. Therefore, Elvin would preempt the Condor job currently running. Assume that later Jimmy submits a job. The RANK evaluates to 10, since the boolean that matches Jimmy gets multiplied by 10. Jimmy would preempt Elvin, and Jimmy's job would run on Jimmy's machine.

The RANK expression is not required to reference the Owner of the jobs. Perhaps there is one machine with an enormous amount of memory, and others with not much at all. You can configure your large-memory machine to prefer to run jobs with larger memory requirements:

```
RANK = ImageSize
```

That's all there is to it. The bigger the job, the more this machine wants to run it. It is an altruistic preference, always servicing the largest of jobs, no matter who submitted them. A little less altruistic is John's RANK that prefers his jobs over those with the largest Imagesize:

```
RANK = (Owner == "coltrane" * 1000000000000) + Imagesize
```

This RANK breaks if a job is submitted with an image size of more $10^{12}$ Kbytes. However, with that size, this RANK expression preferring that job would not be Condor's only problem!

### 3.5.6 Machine States

A machine is assigned a *state* by Condor. The state depends on whether or not the machine is available to run Condor jobs, and if so, what point in the negotiations has been reached. The possible states are

**Owner** The machine is being used by the machine owner, and/or is not available to run Condor jobs. When the machine first starts up, it begins in this state.

**Unclaimed** The machine is available to run Condor jobs, but it is not currently doing so.

**Matched** The machine is available to run jobs, and it has been matched by the negotiator with a specific schedd. That schedd just has not yet claimed this machine. In this state, the machine is unavailable for further matches.

**Claimed** The machine has been claimed by a schedd.

**Preempting** The machine was claimed by a schedd, but is now preempting that claim for one of the following reasons.

1. the owner of the machine came back

2. another user with higher priority has jobs waiting to run
3. another request that this resource would rather serve was found

**Backfill** The machine is running a backfill computation while waiting for either the machine owner to come back or to be matched with a Condor job. This state is only entered if the machine is specifically configured to enable backfill jobs.

Figure 3.3 shows the states and the possible transitions between the states.

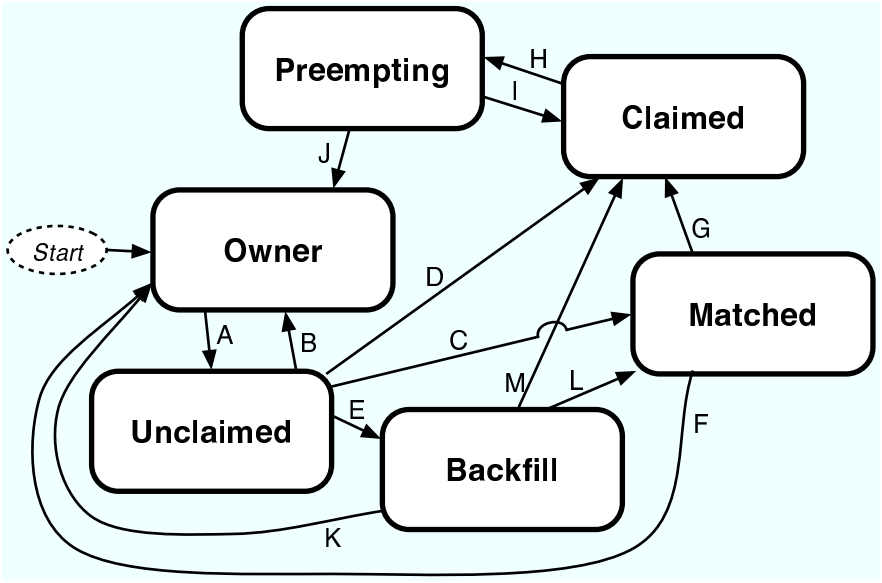



Figure 3.3: Machine States

Each transition is labeled with a letter. The cause of each transition is described below.

- Transitions out of the Owner state

  **A** The machine switches from Owner to Unclaimed whenever the START expression no longer locally evaluates to FALSE. This indicates that the machine is potentially available to run a Condor job.

- Transitions out of the Unclaimed state

  **B** The machine switches from Unclaimed back to Owner whenever the START expression locally evaluates to FALSE. This indicates that the machine is unavailable to run a Condor job and is in use by the resource owner.

  **C** The transition from Unclaimed to Matched happens whenever the *condor_negotiator* matches this resource with a Condor job.

**D** The transition from Unclaimed directly to Claimed also happens if the *condor_negotiator* matches this resource with a Condor job. In this case the *condor_schedd* recieves the match and initiates the claiming protocol with the machine before the *condor_startd* recieves the match notification from the *condor_negotiator*.

**E** The transition from Unclaimed to Backfill happens if the machine is configured to run backfill computations (see section 3.12.9) and the `START_BACKFILL` expression evaluates to TRUE.

- Transitions out of the Matched state

**F** The machine moves from Matched to Owner if either the `START` expression locally evalutes to FALSE, or if the `MATCH_TIMEOUT` timer expires. This timeout is used to ensure that if a machine is matched with a given *condor_schedd*, but that *condor_schedd* does not contact the *condor_startd* to claim it, that the machine will give up on the match and become available to be matched again. In this case, since the `START` expression does not locally evaluate to FALSE, as soon as transition **F** is complete, the machine will immediately enter the Unclaimed state again (via transition **A**). The machine might also go from Matched to Owner if the *condor_schedd* attempts to perform the claiming protocol but encounters some sort of error. Finally, the machine will move into the Owner state if the *condor_startd* recieves a *condor_vacate* command while it is in the Matched state.

**G** The transition from Matched to Claimed occurs when the *condor_schedd* successfully completes the claiming protocol with the *condor_startd*.

- Transitions out of the Claimed state

**H** From the Claimed state, the only possible destination is the Preempting state. This transition can be caused by many reasons:

  – The *condor_schedd* that has claimed the machine has no more work to perform and releases the claim
  – The `PREEMPT` expression evaluates to TRUE (which usually means the resource owner has started using the machine again and is now using the keyboard, mouse, CPU, etc)
  – The *condor_startd* receives a *condor_vacate* command
  – The *condor_startd* is told to shutdown (either via a signal or a *condor_off* command)
  – The resource is matched to a job with a better priority (either a better user priority, or one where the machine rank is higher)

- Transitions out of the Preempting state

**I** The resource will move from Preempting back to Claimed if the resource was matched to a job with a better priority.

**J** The resource will move from Preempting to Owner if the `PREEMPT` expression had evaluated to TRUE, if *condor_vacate* was used, or if the `START` expression locally evalutes to FALSE when the *condor_startd* has finished evicting whatever job it was running when it entered the Preempting state.

- Transitions out of the Backfill state

  **K** The resource will move from Backfill to Owner for the following reasons:
  - The `EVICT_BACKFILL` expression evalutes to TRUE
  - The *condor_startd* receives a *condor_vacate* command
  - The *condor_startd* is being shutdown

  **L** The transition from Backfill to Matched occurs whenever a resource running a backfill computation is matched with a *condor_schedd* that wants to run a Condor job.

  **M** The transition from Backfill directly to Claimed is similar to the transition from Unclaimed directly to Claimed. It only occurs if the *condor_schedd* completes the claiming protocol before the *condor_startd* receives the match notification from the *condor_negotiator*.

### 3.5.7 Machine Activities

Within some machine states, *activities* of the machine are defined. The state has meaning regardless of activity. Differences between activities are significant. Therefore, a "state/activity" pair describes a machine. The following list describes all the possible state/activity pairs.

- Owner

  **Idle** This is the only activity for Owner state. As far as Condor is concerned the machine is Idle, since it is not doing anything for Condor.

- Unclaimed

  **Idle** This is the normal activity of Unclaimed machines. The machine is still Idle in that the machine owner is willing to let Condor jobs run, but Condor is not using the machine for anything.

  **Benchmarking** The machine is running benchmarks to determine the speed on this machine. This activity only occurs in the Unclaimed state. How often the activity occurs is determined by the `RunBenchmarks` expression.

- Matched

  **Idle** When Matched, the machine is still Idle to Condor.

- Claimed

  **Idle** In this activity, the machine has been claimed, but the schedd that claimed it has yet to *activate* the claim by requesting a *condor_starter* to be spawned to service a job. The machine returns to this state (usually briefly) when jobs (and therefore *condor_starter*) finish.

  **Busy** Once a *condor_starter* has been started and the claim is active, the machine moves to the Busy activity to signify that it is doing something as far as Condor is concerned.

**Suspended** If the job is suspended by Condor, the machine goes into the Suspended activity. The match between the schedd and machine has not been broken (the claim is still valid), but the job is not making any progress and Condor is no longer generating a load on the machine.

**Retiring** When an active claim is about to be preempted for any reason, it enters retirement, while it waits for the current job to finish. The `MaxJobRetirementTime` expression determines how long to wait (counting since the time the job started). Once the job finishes or the retirement time expires, the Preempting state is entered.

- Preempting The preempting state is used for evicting a Condor job from a given machine. When the machine enters the Preempting state, it checks the `WANT_VACATE` expression to determine its activity.

  **Vacating** In the Vacating activity, the job that was running is in the process of checkpointing. As soon as the checkpoint process completes, the machine moves into either the Owner state or the Claimed state, depending on the reason for its preemption.

  **Killing** Killing means that the machine has requested the running job to exit the machine immediately, without checkpointing.

- Backfill

  **Idle** The machine is configured to run backfill jobs and is ready to do so, but it has not yet had a chance to spawn a backfill manager (for example, the BOINC client).

  **Busy** The machine is performing a backfill computation.

  **Killing** The machine was running a backfill computation, but it is now killing the job to either return resources to the machine owner, or to make room for a regular Condor job.

Figure 3.4 on page 240 gives the overall view of all machine states and activities and shows the possible transitions from one to another within the Condor system. Each transition is labeled with a number on the diagram, and transition numbers referred to in this manual will be **bold**.

Various expressions are used to determine when and if many of these state and activity transitions occur. Other transitions are initiated by parts of the Condor protocol (such as when the *condor_negotiator* matches a machine with a schedd). The following section describes the conditions that lead to the various state and activity transitions.

### 3.5.8 State and Activity Transitions

This section traces through all possible state and activity transitions within a machine and describes the conditions under which each one occurs. Whenever a transition occurs, Condor records when the machine entered its new activity and/or new state. These times are often used to write expressions that determine when further transitions occurred. For example, enter the Killing activity if a machine has been in the Vacating activity longer than a specified amount of time.

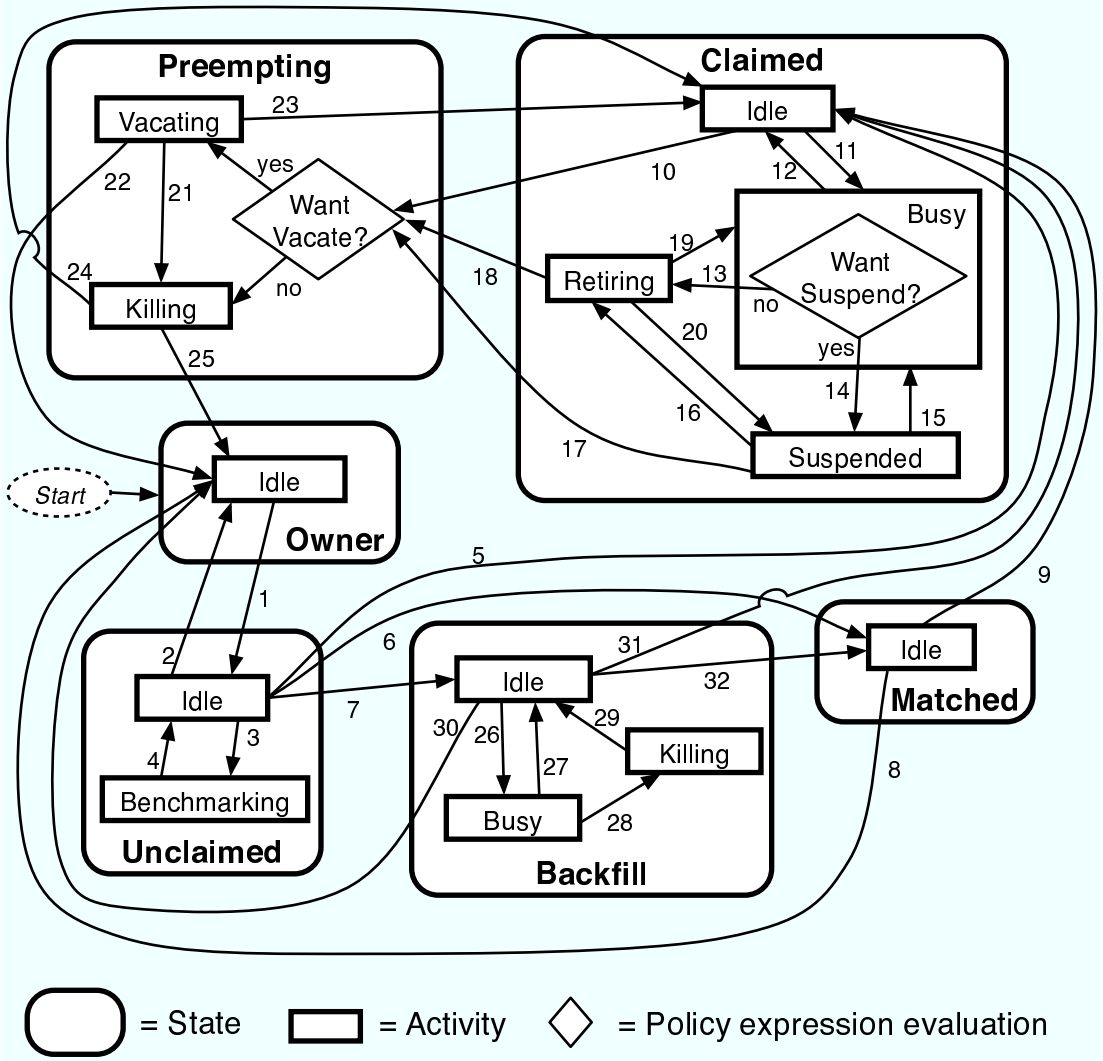



Figure 3.4: Machine States and Activities

**Owner State**

When the startd is first spawned, the machine it represents enters the Owner state. The machine remains in the Owner state while the expression `IsOwner` is TRUE. If the `IsOwner` expression is FALSE, then the machine transitions to the Unclaimed state. The default value for the `IsOwner` expression is optimized for a shared resource

```
START =?= FALSE
```

So, the machine will remain in the Owner state as long as the START expression locally evaluates to FALSE. Section 3.5.3 provides more detail on the START expression. If the START locally evaluates to TRUE or cannot be locally evaluated (it evaluates to UNDEFINED), transition **1** occurs and the machine enters the Unclaimed state. The IsOwner expression is locally evaluated by the machine, and should not reference job ClassAd attributes, which would be UNDEFINED.

For dedicated resources, the recommended value for the IsOwner expression is FALSE.

The Owner state represents a resource that is in use by its interactive owner (for example, if the keyboard is being used). The Unclaimed state represents a resource that is neither in use by its interactive user, nor the Condor system. From Condor's point of view, there is little difference between the Owner and Unclaimed states. In both cases, the resource is not currently in use by the Condor system. However, if a job matches the resource's START expression, the resource is available to run a job, regardless of if it is in the Owner or Unclaimed state. The only differences between the two states are how the resource shows up in *condor_status* and other reporting tools, and the fact that Condor will not run benchmarking on a resource in the Owner state. As long as the IsOwner expression is TRUE, the machine is in the Owner State. When the IsOwner expression is FALSE, the machine goes into the Unclaimed State.

Here is an example that assumes that an IsOwner expression is not present in the configuration. If the START expression is

```
START = KeyboardIdle > 15 * $(MINUTE) && Owner == "coltrane"
```

and if KeyboardIdle is 34 seconds, then the machine would remain in the Owner state. Owner is undefined, and anything && FALSE is FALSE.

If, however, the START expression is

```
START = KeyboardIdle > 15 * $(MINUTE) || Owner == "coltrane"
```

and KeyboardIdle is 34 seconds, then the machine leaves the Owner state and becomes Unclaimed. This is because FALSE || UNDEFINED is UNDEFINED. So, while this machine is not available to just anybody, if user coltrane has jobs submitted, the machine is willing to run them. Any other user's jobs have to wait until KeyboardIdle exceeds 15 minutes. However, since coltrane might claim this resource, but has not yet, the machine goes to the Unclaimed state.

While in the Owner state, the startd polls the status of the machine every UPDATE_INTERVAL to see if anything has changed that would lead it to a different state. This minimizes the impact on the Owner while the Owner is using the machine. Frequently waking up, computing load averages, checking the access times on files, computing free swap space take time, and there is nothing time critical that the startd needs to be sure to notice as soon as it happens. If the START expression evaluates to TRUE and five minutes pass before the startd notices, that's a drop in the bucket of high-throughput computing.

The machine can only transition to the Unclaimed state from the Owner state. It does so when the IsOwner expression no longer evaluates to FALSE. By default, that happens when START no longer locally evaluates to FALSE.

Whenever the machine is not actively running a job, it will transition back to the Owner state if `IsOwner` evaluates to TRUE. Once a job is started, the value of `IsOwner` does not matter; the job either runs to completion or is preempted. Therefore, you must configure the preemption policy if you want to transition back to the Owner state from Claimed Busy.

**Unclaimed State**

If the `IsOwner` expression becomes TRUE, then the machine returns to the Owner state. If the `IsOwner` expression becomes FALSE, then the machine remains in the Unclaimed state. If the `IsOwner` expression is not present in the configuration files, then the default value for the `IsOwner` expression is

```
START =?= FALSE
```

so that while in the Unclaimed state, if the `START` expression locally evaluates to FALSE, the machine returns to the Owner state by transition **2**.

When in the Unclaimed state, the `RunBenchmarks` expression is relevant. If `RunBenchmarks` evaluates to TRUE while the machine is in the Unclaimed state, then the machine will transition from the Idle activity to the Benchmarking activity (transition **3**) and perform benchmarks to determine `MIPS` and `KFLOPS`. When the benchmarks complete, the machine returns to the Idle activity (transition **4**).

The startd automatically inserts an attribute, `LastBenchmark`, whenever it runs benchmarks, so commonly `RunBenchmarks` is defined in terms of this attribute, for example:

```
BenchmarkTimer = (CurrentTime - LastBenchmark)
RunBenchmarks = $(BenchmarkTimer) >= (4 * $(HOUR))
```

Here, a macro, `BenchmarkTimer` is defined to help write the expression. This macro holds the time since the last benchmark, so when this time exceeds 4 hours, we run the benchmarks again. The startd keeps a weighted average of these benchmarking results to try to get the most accurate numbers possible. This is why it is desirable for the startd to run them more than once in its lifetime.

NOTE: `LastBenchmark` is initialized to 0 before benchmarks have ever been run. To have the *condor_startd* run benchmarks as soon as the machine is Unclaimed (if it has not done so already), include a term using `LastBenchmark` as in the example above.

NOTE: If `RunBenchmarks` is defined and set to something other than FALSE, the startd will automatically run one set of benchmarks when it first starts up. To disable benchmarks, both at startup and at any time thereafter, set `RunBenchmarks` to FALSE or comment it out of the configuration file.

From the Unclaimed state, the machine can go to four other possible states: Owner (transition **2**), Backfill/Idle, Matched, or Claimed/Idle.

Once the *condor_negotiator* matches an Unclaimed machine with a requester at a given schedd, the negotiator sends a command to both parties, notifying them of the match. If the schedd receives that notification and initiates the claiming procedure with the machine before the negotiator's message gets to the machine, the Match state is skipped, and the machine goes directly to the Claimed/Idle state (transition **5**). However, normally the machine will enter the Matched state (transition **6**), even if it is only for a brief period of time.

If the machine has been configured to perform backfill jobs (see section 3.12.9), while it is in Unclaimed/Idle it will evaluate the START_BACKFILL expression. Once START_BACKFILL evaluates to TRUE, the machine will enter the Backfill/Idle state (transition **7**) to begin the process of running backfill jobs.

#### Matched State

The Matched state is not very interesting to Condor. Noteworthy in this state is that the machine lies about its START expression while in this state and says that Requirements are false to prevent being matched again before it has been claimed. Also interesting is that the startd starts a timer to make sure it does not stay in the Matched state too long. The timer is set with the MATCH_TIMEOUT configuration file macro. It is specified in seconds and defaults to 120 (2 minutes). If the schedd that was matched with this machine does not claim it within this period of time, the machine gives up, and goes back into the Owner state via transition **8**. It will probably leave the Owner state right away for the Unclaimed state again and wait for another match.

At any time while the machine is in the Matched state, if the START expression locally evaluates to FALSE, the machine enters the Owner state directly (transition **8**).

If the schedd that was matched with the machine claims it before the MATCH_TIMEOUT expires, the machine goes into the Claimed/Idle state (transition **9**).

#### Claimed State

The Claimed state is certainly the most complex state. It has the most possible activities and the most expressions that determine its next activities. In addition, the *condor_checkpoint* and *condor_vacate* commands affect the machine when it is in the Claimed state. In general, there are two sets of expressions that might take effect. They depend on the universe of the request: standard or vanilla. The standard universe expressions are the normal expressions. For example:

```
WANT_SUSPEND            = True
WANT_VACATE             = $(ActivationTimer) > 10 * $(MINUTE)
SUSPEND                 = $(KeyboardBusy) || $(CPUBusy)
...
```

The vanilla expressions have the string"_VANILLA" appended to their names. For example:

```
WANT_SUSPEND_VANILLA    = True
```

```
WANT_VACATE_VANILLA     = True
SUSPEND_VANILLA         = $(KeyboardBusy) || $(CPUBusy)
...
```

Without specific vanilla versions, the normal versions will be used for all jobs, including vanilla jobs. In this manual, the normal expressions are referenced. The difference exists for the the resource owner that might want the machine to behave differently for vanilla jobs, since they cannot checkpoint. For example, owners may want vanilla jobs to remain suspended for longer than standard jobs.

While Claimed, the POLLING_INTERVAL takes effect, and the startd polls the machine much more frequently to evaluate its state.

If the machine owner starts typing on the console again, it is best to notice this as soon as possible to be able to start doing whatever the machine owner wants at that point. For SMP machines, if any virtual machine is in the Claimed state, the startd polls the machine frequently. If already polling one virtual machine, it does not cost much to evaluate the state of all the virtual machines at the same time.

There are a variety of events that may cause the startd to try to get rid of or temporarily suspend a running job. Activity on the machine's console, load from other jobs, or shutdown of the startd via an administrative command are all possible sources of interference. Another one is the appearance of a higher priority claim to the machine by a different Condor user.

Depending on the configuration, the startd may respond quite differently to activity on the machine, such as keyboard activity or demand for the cpu from processes that are not managed by Condor. The startd can be configured to completely ignore such activity or to suspend the job or even to kill it. A standard configuration for a desktop machine might be to go through successive levels of getting the job out of the way. The first and least costly to the job is suspending it. This works for both standard and vanilla jobs. If suspending the job for a short while does not satisfy the machine owner (the owner is still using the machine after a specific period of time), the startd moves on to vacating the job. Vacating a standard universe job involves performing a checkpoint so that the work already completed is not lost. Vanilla jobs are sent a *soft kill signal* so that they can gracefully shut down if necessary; the default is SIGTERM. If vacating does not satisfy the machine owner (usually because it is taking too long and the owner wants their machine back *now*), the final, most drastic stage is reached: killing. Killing is a quick death to the job, using a hard-kill signal that cannot be intercepted by the application. For vanilla jobs that do no special signal handling, vacating and killing are equivalent.

The WANT_SUSPEND expression determines if the machine will evaluate the SUSPEND expression to consider entering the Suspended activity. The WANT_VACATE expression determines what happens when the machine enters the Preempting state. It will go to the Vacating activity or directly to Killing. If one or both of these expressions evaluates to FALSE, the machine will skip that stage of getting rid of the job and proceed directly to the more drastic stages.

When the machine first enters the Claimed state, it goes to the Idle activity. From there, it has two options. It can enter the Preempting state via transition **10** (if a *condor_vacate* arrives, or if the START expression locally evaluates to FALSE), or it can enter the Busy activity (transition **11**) if

the schedd that has claimed the machine decides to activate the claim and start a job.

From Claimed/Busy, the machine can transition to three other state/activity pairs. The startd evaluates the WANT_SUSPEND expression to decide which other expressions to evaluate. If WANT_SUSPEND is TRUE, then the startd evaluates the SUSPEND expression. If WANT_SUSPEND is FALSE, then the startd will evaluate the PREEMPT expression and skip the Suspended activity entirely. By transition, the possible state/activity destinations from Claimed/Busy:

**Claimed/Idle** If the starter that is serving a given job exits (for example because the jobs completes), the machine will go to Claimed/Idle (transition **12**).

**Claimed/Retiring** If WANT_SUSPEND is FALSE and the PREEMPT expression is TRUE, the machine enters the Retiring activity (transition **13**). From there, it waits for a configurable amount of time for the job to finish before moving on to preemption.

Another reason the machine would go from Claimed/Busy to Claimed/Retiring is if the *condor_negotiator* matched the machine with a "better" match. This better match could either be from the machine's perspective using the startd RANK expression, or it could be from the negotiator's perspective due to a job with a higher user priority.

Another case resulting in a transition to Claimed/Retiring is when the startd is being shut down. The only exception is a "fast" shutdown, which bypasses retirement completely.

**Claimed/Suspended** If both the WANT_SUSPEND and SUSPEND expressions evaluate to TRUE, the machine suspends the job (transition **14**).

If a *condor_checkpoint* command arrives, or the PeriodicCheckpoint expression evaluates to TRUE, there is no state change. The startd has no way of knowing when this process completes, so periodic checkpointing can not be another state. Periodic checkpointing remains in the Claimed/Busy state and appears as a running job.

From the Claimed/Suspended state, the following transitions may occur:

**Claimed/Busy** If the CONTINUE expression evaluates to TRUE, the machine resumes the job and enters the Claimed/Busy state (transition **15**) or the Claimed/Retiring state (transition **16**), depending on whether the claim has been preempted.

**Claimed/Retiring** If the PREEMPT expression is TRUE, the machine will enter the Claimed/Retiring activity (transition **16**).

**Preempting** If the claim is in suspended retirement and the retirement time expires, the job enters the Preempting state (transition **17**). This is only possible if MaxJobRetirementTime *decreases* during the suspension.

For the Claimed/Retiring state, the following transitions may occur:

**Preempting** If the job finishes or the job's runtime exceeds MaxJobRetirementTime, the Preempting state is entered (transition **18**). The runtime is computed from the time when the job

was started by the startd minus any suspension time. (When retiring due to startd shutdown or restart, it is possible for the admin to issue a "peaceful" shutdown command, which causes `MaxJobRetirementTime` to effectively be infinite, avoiding any killing of jobs.)

**Claimed/Busy** If the startd was retiring because of a preempting claim only and the preempting claim goes away, the normal Claimed/Busy state is resumed (transition **19**). If instead the retirement is due to owner activity (`PREEMPT`) or the startd is being shut down, no unretirement is possible.

**Claimed/Suspended** In exactly the same way that suspension may happen from the Claimed/Busy state, it may also happen during the Claimed/Retiring state (transition **20**). In this case, when the job continues from suspension, it moves back into Claimed/Retiring (transition **16**) instead of Claimed/Busy (transition **15**).

#### Preempting State

The Preempting state is less complex than the Claimed state. There are two activities. Depending on the value of `WANT_VACATE`, a machine will be in the Vacating activity (if TRUE) or the Killing activity (if FALSE).

While in the Preempting state (regardless of activity) the machine advertises its `Requirements` expression as FALSE to signify that it is not available for further matches, either because it is about to transition to the Owner state, or because it has already been matched with one preempting match, and further preempting matches are disallowed until the machine has been claimed by the new match.

The main function of the Preempting state is to get rid of the starter associated with the resource. If the *condor_starter* associated with a given claim exits while the machine is still in the Vacating activity, then the job successfully completed a graceful shutdown. For standard universe jobs, this means that a checkpoint was saved. For other jobs, this means the application was given an opportunity to do a graceful shutdown, by intercepting the soft kill signal.

If the machine is in the Vacating activity, it keeps evaluating the `KILL` expression. As soon as this expression evaluates to TRUE, the machine enters the Killing activity (transition **21**).

When the starter exits, or if there was no starter running when the machine enters the Preempting state (transition **10**), the other purpose of the Preempting state is completed: notifying the schedd that had claimed this machine that the claim is broken.

At this point, the machine enters either the Owner state by transition **22** (if the job was preempted because the machine owner came back) or the Claimed/Idle state by transition **23** (if the job was preempted because a better match was found).

If the machine enters the Killing activity, (because either `WANT_VACATE` was FALSE or the `KILL` expression evaluated to TRUE), it attempts to force the *condor_starter* to immediately kill the underlying Condor job. Once the machine has begun to hard kill the Condor job, the *condor_startd* starts a timer, the length of which is defined by the `KILLING_TIMEOUT` macro. This macro is defined in seconds and defaults to 30. If this timer expires and the machine is still in the Killing

activity, something has gone seriously wrong with the *condor_starter* and the startd tries to vacate the job immediately by sending SIGKILL to all of the *condor_starter*'s children, and then to the *condor_starter* itself.

Once the *condor_starter* has killed off all the processes associated with the job and exited, and once the schedd that had claimed the machine is notified that the claim is broken, the machine will leave the Preempting/Killing state. If the job was preempted because a better match was found, the machine will enter Claimed/Idle (transition **24**). If the preemption was caused by the machine owner (the PREEMPT expression evaluated to TRUE, *condor_vacate* was used, etc), the machine will enter the Owner state (transition **25**).

#### Backfill State

The Backfill state is used whenever the machine is performing low priority background tasks to keep itself busy. For more information about backfill support in Condor, see section 3.12.9 on page 359. This state is only used if the machine has been configured to enable backfill computation, if a specific backfill manager has been installed and configured, and if the machine is otherwise idle (not being used interactively or for regular Condor computations). If the machine meets all these requirements, and the START_BACKFILL expression evalutes to TRUE, the machine will move from the Unclaimed/Idle state to Backfill/Idle (transition **7**).

Once a machine is in Backfill/Idle, it will immediately attempt to spawn whatever backfill manager it has been configured to use (currently, only the BOINC client is supported as a backfill manager in Condor). Once the BOINC client is running, the machine will enter Backfill/Busy (transition **26**) to indicate that it is now performing a backfill computation.

NOTE: On SMP machines, the *condor_startd* will only spawn a single instance of the BOINC client, even if multiple virtual machines are available to run backfill jobs. Therefore, only the first machine to enter Backfill/Idle will cause a copy of the BOINC client to start running. If a given virtual machine on an SMP enters the Backfill state and a BOINC client is already running under this *condor_startd*, the virtual machine will immediately enter Backfill/Busy without waiting to spawn another copy of the BOINC client.

If the BOINC client ever exits on its own (which normally wouldn't happen), the machine will go back to Backfill/Idle (transition **27**) where it will immediately attempt to respawn the BOINC client (and return to Backfill/Busy via transition **26**).

As the BOINC client is running a backfill computation, a number of events can occur that will drive the machine out of the Backfill state. The machine can get matched or claimed for a Condor job, interactive users can start using the machine again, the machine might be evicted with *condor_vacate*, or the *condor_startd* might be shutdown. All of these events cause the *condor_startd* to kill the BOINC client and all its descendants, and enter the Backfill/Killing state (transition **28**).

Once the BOINC client and all its children have exited the system, the machine will enter the Backfill/Idle state to indicate that the BOINC client is now gone (transition **29**). As soon as it enters Backfill/Idle after the BOINC client exits, the machine will go into another state, depending on what caused the BOINC client to be killed in the first place.

If the EVICT_BACKFILL expression evaluates to TRUE while a machine is in Backfill/Busy, after the BOINC client is gone, the machine will go back into the Owner/Idle state (transition **30**). The machine will also return to the Owner/Idle state after the BOINC client exits if *condor_vacate* was used, or if the *condor_startd* is being shutdown.

When a machine running backfill jobs is matched with a requester that wants to run a Condor job, the machine will either enter the Matched state, or go directly into Claimed/Idle. As with the case of a machine in Unclaimed/Idle (described above), the *condor_negotiator* informs both the *condor_startd* and the *condor_schedd* of the match, and the exact state transitions at the machine depend on what order the various entities initiate communication with each other. If the *condor_schedd* is notified of the match and sends a request to claim the *condor_startd* before the *condor_negotiator* has a chance to notify the *condor_startd*, once the BOINC client exits, the machine will immediately enter Claimed/Idle (transition **31**). Normally, the notification from the *condor_negotiator* will reach the *condor_startd* before the *condor_schedd* attempts to claim it. In this case, once the BOINC client exits, the machine will enter Matched/Idle (transition **32**).

### 3.5.9 State/Activity Transition Expression Summary

This section is a summary of the information from the previous sections. It serves as a quick reference.

**START** When TRUE, the machine is willing to spawn a remote Condor job.

**RunBenchmarks** While in the Unclaimed state, the machine will run benchmarks whenever TRUE.

**MATCH_TIMEOUT** If the machine has been in the Matched state longer than this value, it will transition to the Owner state.

**WANT_SUSPEND** If TRUE, the machine evaluates the SUSPEND expression to see if it should transition to the Suspended activity. If FALSE, the machine look at the PREEMPT expression.

**SUSPEND** If WANT_SUSPEND is TRUE, and the machine is in the Claimed/Busy state, it enters the Suspended activity if SUSPEND is TRUE.

**CONTINUE** If the machine is in the Claimed/Suspended state, it enter the Busy activity if CONTINUE is TRUE.

**PREEMPT** If the machine is either in the Claimed/Suspended activity, or is in the Claimed/Busy activity and WANT_SUSPEND is FALSE, the machine enters the Claimed/Retiring state whenever PREEMPT is TRUE.

**CLAIM_WORKLIFE** If provided, this expression specifies the number of seconds during which a claim will continue accepting new jobs. Once this time expires, any existing job may continue to run as usual, but once it finishes or is preempted, the claim is closed. This may be useful if you want to force periodic renegotiation of resources without preemption having to occur. For example, if you have some low-priority jobs which should never be interrupted with kill

signals, you could prevent them from being killed with `MaxJobRetirementTime`, but now high-priority jobs may have to wait in line when they match to a machine that is busy running one of these uninterruptible jobs. You can prevent the high-priority jobs from ever matching to such a machine by using a rank expression in the job or in the negotiator's rank expressions, but then the low-priority claim will never be interrupted; it can keep running more jobs. The solution is to use `CLAIM_WORKLIFE` to force the claim to stop running additional jobs after a certain amount of time. The default value for `CLAIM_WORKLIFE` is -1, which is treated as an infinite claim worklife, so claims may be held indefinitely (as long as they are not preempted and the schedd does not relinquish them, of course).

**MAXJOBRETIREMENTTIME** If the machine is in the Claimed/Retiring state, this expression specifies the maximum time (in seconds) that the startd will wait for the job to finish naturally (without any kill signals from the startd). The clock starts when the job is started and is paused during any suspension. The job may provide its own expression for `MaxJobRetirementTime`, but this can only be used to take *less* than the time granted by the startd, never more. (For convenience, standard universe and nice_user jobs are submitted with a default retirement time of 0, so they will never wait in retirement unless the user overrides the default.)

Once the job finishes or if the retirement time expires, the machine enters the Preempting state.

This expression is evaluated in the context of the job ClassAd, so it may refer to attributes of the current job as well as machine attributes. The expression is continually re-evaluated while the job is running, so it is possible, though unusual, to have an expression that changes over time. For example, if you want the retirement time to drop to 0 if an especially high priority job is waiting for the current job to retire, you could use `PreemptingRank` in the expression. Example:

```
MaxJobRetirementTime = 3600 * ( \
  MY.PreemptingRank =?= UNDEFINED || \
  PreemptingRank < 600)
```

In this example, the retirement time is 3600 seconds, but if a job gets matched to this machine and it has a `PreemptingRank` of 600 or more, the retirement time drops to 0 and the current job is immediately preempted.

**WANT_VACATE** This is checked only when the `PREEMPT` expression is TRUE and the machine enters the Preempting state. If `WANT_VACATE` is TRUE, the machine enters the Vacating activity. If it is FALSE, the machine will proceed directly to the Killing activity.

**KILL** If the machine is in the Preempting/Vacating state, it enters Preempting/Killing whenever `KILL` is TRUE.

**KILLING_TIMEOUT** If the machine is in the Preempting/Killing state for longer than `KILLING_TIMEOUT` seconds, the startd sends a SIGKILL to the *condor_starter* and all its children to try to kill the job as quickly as possible.

**PERIODIC_CHECKPOINT** If the machine is in the Claimed/Busy state and `PERIODIC_CHECKPOINT` is TRUE, the user's job begins a periodic checkpoint.

**RANK** If this expression evaluates to a higher number for a pending resource request than it does for the current request, the machine preempts the current request (enters the Preempting/Vacating state). When the preemption is complete, the machine enters the Claimed/Idle state with the new resource request claiming it.

**START_BACKFILL** When TRUE, if the machine is otherwise idle, it will enter the Backfill state and spawn a backfill computation (using BOINC).

**EVICT_BACKFILL** When TRUE, if the machine is currently running a backfill computation, it will kill the BOINC client and return to the Owner/Idle state.

### 3.5.10 Policy Settings

This section describes the default configuration policy and then provides examples of extensions to these policies.

#### Default Policy Settings

These settings are the default as shipped with Condor. They have been used for many years with no problems. The vanilla expressions are identical to the regular ones. (They are not listed here. If not defined, the standard expressions are used for vanilla jobs as well).

The following are macros to help write the expressions clearly.

**StateTimer** Amount of time in the current state.

**ActivityTimer** Amount of time in the current activity.

**ActivationTimer** Amount of time the job has been running on this machine.

**LastCkpt** Amount of time since the last periodic checkpoint.

**NonCondorLoadAvg** The difference between the system load and the Condor load (the load generated by everything but Condor).

**BackgroundLoad** Amount of background load permitted on the machine and still start a Condor job.

**HighLoad** If the `$(NonCondorLoadAvg)` goes over this, the CPU is considered too busy, and eviction of the Condor job should start.

**StartIdleTime** Amount of time the keyboard must to be idle before Condor will start a job.

**ContinueIdleTime** Amount of time the keyboard must to be idle before resumption of a suspended job.

**MaxSuspendTime** Amount of time a job may be suspended before more drastic measures are taken.

**MaxVacateTime** Amount of time a job may be checkpointing before we give up and kill it outright.

**KeyboardBusy** A boolean expression that evaluates to TRUE when the keyboard is being used.

**CPUIdle** A boolean expression that evaluates to TRUE when the CPU is idle.

**CPUBusy** A boolean expression that evaluates to TRUE when the CPU is busy.

**MachineBusy** The CPU or the Keyboard is busy.

**CPUIsBusy** A boolean value set to the same value as `CPUBusy`.

**CPUBusyTime** The value 0 if `CPUBusy` is False; the time in seconds since `CPUBusy` became True.

```
##  These macros are here to help write legible expressions:
MINUTE          = 60
HOUR            = (60 * $(MINUTE))
StateTimer      = (CurrentTime - EnteredCurrentState)
ActivityTimer   = (CurrentTime - EnteredCurrentActivity)
ActivationTimer = (CurrentTime - JobStart)
LastCkpt        = (CurrentTime - LastPeriodicCheckpoint)

NonCondorLoadAvg        = (LoadAvg - CondorLoadAvg)
BackgroundLoad          = 0.3
HighLoad                = 0.5
StartIdleTime           = 15 * $(MINUTE)
ContinueIdleTime        = 5 * $(MINUTE)
MaxSuspendTime          = 10 * $(MINUTE)
MaxVacateTime           = 10 * $(MINUTE)

KeyboardBusy            = KeyboardIdle < $(MINUTE)
ConsoleBusy             = (ConsoleIdle  < $(MINUTE))
CPUIdle                 = $(NonCondorLoadAvg) <= $(BackgroundLoad)
CPUBusy                 = $(NonCondorLoadAvg) >= $(HighLoad)
KeyboardNotBusy         = ($(KeyboardBusy) == False)
MachineBusy             = ($(CPUBusy) || $(KeyboardBusy)
```

Macros are defined to want to suspend jobs (instead of killing them) in the case of jobs that use little memory, when the keyboard is not being used, and for vanilla universe and PVM universe jobs. We want to gracefully vacate jobs which have been running for more than 10 minutes or are vanilla universe or PVM universe jobs.

```
WANT_SUSPEND       = ( $(SmallJob) || $(KeyboardNotBusy) \
                       || $(IsPVM) || $(IsVanilla) )
WANT_VACATE        = ( $(ActivationTimer) > 10 * $(MINUTE) \
                       || $(IsPVM) || $(IsVanilla) )
```

Finally, definitions of the actual expressions. Start a job if the keyboard has been idle long enough and the load average is low enough OR the machine is currently running a Condor job. Note that Condor would only run one job at a time. It just may prefer to run a different job, as defined by the machine rank or user priorities.

```
START        = ( (KeyboardIdle > $(StartIdleTime)) \
                 && ( $(CPUIdle) || \
                      (State != "Unclaimed" && State != "Owner")) )
```

Suspend a job if the keyboard has been touched. Alternatively, suspend if the CPU has been busy for more than two minutes and the job has been running for more than 90 seconds.

```
SUSPEND         = ( $(KeyboardBusy) || \
                  ( (CpuBusyTime > 2 * $(MINUTE)) \
                     && $(ActivationTimer) > 90 ) )
```

Continue a suspended job if the CPU is idle, the Keyboard has been idle for long enough, and the job has been suspended more than 10 seconds.

```
CONTINUE        = ( $(CPUIdle) && ($(ActivityTimer) > 10) \
                  && (KeyboardIdle > $(ContinueIdleTime)) )
```

There are two conditions that signal preemption. The first condition is if the job is suspended, but it has been suspended too long. The second condition is if suspension is not desired and the machine is busy.

```
PREEMPT         = ( ((Activity == "Suspended") && \
                    ($(ActivityTimer) > $(MaxSuspendTime))) \
                    || (SUSPEND && (WANT_SUSPEND == False)) )
```

Do not give jobs any time to retire on their own when they are about to be preempted.

```
MaxJobRetirementTime = 0
```

Kill jobs that take too long leaving gracefully.

```
KILL            = $(ActivityTimer) > $(MaxVacateTime)
```

Finally, specify periodic checkpointing. For jobs smaller than 60 Mbytes, do a periodic checkpoint every 6 hours. For larger jobs, only checkpoint every 12 hours.

```
PERIODIC_CHECKPOINT     = ( (ImageSize < 60000) && \
                            ($(LastCkpt) > (6 * $(HOUR))) ) || \
                          ( $(LastCkpt) > (12 * $(HOUR)) )
```

At UW-Madison, we have a fast network. We simplify our expression considerably to

```
PERIODIC_CHECKPOINT     = $(LastCkpt) > (3 * $(HOUR))
```

For reference, the entire set of policy settings are included once more without comments:

```
##  These macros are here to help write legible expressions:
MINUTE          = 60
HOUR            = (60 * $(MINUTE))
StateTimer      = (CurrentTime - EnteredCurrentState)
ActivityTimer   = (CurrentTime - EnteredCurrentActivity)
ActivationTimer = (CurrentTime - JobStart)
LastCkpt        = (CurrentTime - LastPeriodicCheckpoint)

NonCondorLoadAvg        = (LoadAvg - CondorLoadAvg)
BackgroundLoad          = 0.3
HighLoad                = 0.5
StartIdleTime           = 15 * $(MINUTE)
ContinueIdleTime        = 5 * $(MINUTE)
MaxSuspendTime          = 10 * $(MINUTE)
MaxVacateTime           = 10 * $(MINUTE)

KeyboardBusy            = KeyboardIdle < $(MINUTE)
ConsoleBusy             = (ConsoleIdle  < $(MINUTE))
CPUIdle                 = $(NonCondorLoadAvg) <= $(BackgroundLoad)
CPUBusy                 = $(NonCondorLoadAvg) >= $(HighLoad)
KeyboardNotBusy         = ($(KeyboardBusy) == False)
MachineBusy             = ($(CPUBusy) || $(KeyboardBusy)

WANT_SUSPEND       = ( $(SmallJob) || $(KeyboardNotBusy) \
                       || $(IsPVM) || $(IsVanilla) )
WANT_VACATE        = ( $(ActivationTimer) > 10 * $(MINUTE) \
                       || $(IsPVM) || $(IsVanilla) )
START        = ( (KeyboardIdle > $(StartIdleTime)) \
                 && ( $(CPUIdle) || \
                      (State != "Unclaimed" && State != "Owner")) )
SUSPEND         = ( $(KeyboardBusy) || \
                  ( (CpuBusyTime > 2 * $(MINUTE)) \
```

```
                    && $(ActivationTimer) > 90 ) )
CONTINUE        = ( $(CPUIdle) && ($(ActivityTimer) > 10) \
                  && (KeyboardIdle > $(ContinueIdleTime)) )
PREEMPT         = ( ((Activity == "Suspended") && \
                    ($(ActivityTimer) > $(MaxSuspendTime))) \
                    || (SUSPEND && (WANT_SUSPEND == False)) )
MaxJobRetirementTime = 0
KILL            = $(ActivityTimer) > $(MaxVacateTime)
PERIODIC_CHECKPOINT     = ( (ImageSize < 60000) && \
                            ($(LastCkpt) > (6 * $(HOUR))) ) || \
                          ( $(LastCkpt) > (12 * $(HOUR)) )
```

**Test-job Policy Example**

This example shows how the default macros can be used to set up a machine for running test jobs from a specific user. Suppose we want the machine to behave normally, except if user coltrane submits a job. In that case, we want that job to start regardless of what is happening on the machine. We do not want the job suspended, vacated or killed. This is reasonable if we know coltrane is submitting very short running programs for testing purposes. The jobs should be executed right away. This works with any machine (or the whole pool, for that matter) by adding the following 5 expressions to the existing configuration:

```
START      = ($(START)) || Owner == "coltrane"
SUSPEND    = ($(SUSPEND)) && Owner != "coltrane"
CONTINUE   = $(CONTINUE)
PREEMPT    = ($(PREEMPT)) && Owner != "coltrane"
KILL       = $(KILL)
```

Notice that there is nothing special in either the `CONTINUE` or `KILL` expressions. If Coltrane's jobs never suspend, they never look at `CONTINUE`. Similarly, if they never preempt, they never look at `KILL`.

**Time of Day Policy**

Condor can be configured to only run jobs at certain times of the day. In general, we discourage configuring a system like this, since you can often get lots of good cycles out of machines, even when their owners say "I'm always using my machine during the day." However, if you submit mostly vanilla jobs or other jobs that cannot checkpoint, it might be a good idea to only allow the jobs to run when you know the machines will be idle and when they will not be interrupted.

To configure this kind of policy, you should use the `ClockMin` and `ClockDay` attributes, defined in section 3.5.1 on "Startd ClassAd Attributes". These are special attributes which are automatically inserted by the *condor_startd* into its ClassAd, so you can always reference them in your policy expressions. `ClockMin` defines the number of minutes that have passed since midnight.

For example, 8:00am is 8 hours after midnight, or 8 * 60 minutes, or 480. 5:00pm is 17 hours after midnight, or 17 * 60, or 1020. `ClockDay` defines the day of the week, Sunday = 0, Monday = 1, and so on.

To make the policy expressions easy to read, we recommend using macros to define the time periods when you want jobs to run or not run. For example, assume regular "work hours" at your site are from 8:00am until 5:00pm, Monday through Friday:

```
WorkHours = ( (ClockMin >= 480 && ClockMin < 1020) && \
              (ClockDay > 0 && ClockDay < 6) )
AfterHours = ( (ClockMin < 480 || ClockMin >= 1020) || \
               (ClockDay == 0 || ClockDay == 6) )
```

Of course, you can fine-tune these settings by changing the definition of `AfterHours` and `WorkHours` for your site.

Assuming you are using the default policy expressions discussed above, there are only a few minor changes required to force Condor jobs to stay off of your machines during work hours:

```
# Only start jobs after hours.
START = $(AfterHours) && $(CPUIdle) && KeyboardIdle > $(StartIdleTime)

# Consider the machine busy during work hours, or if the keyboard or
# CPU are busy.
MachineBusy = ( $(WorkHours) || $(CPUBusy) || $(KeyboardBusy) )
```

By default, the `MachineBusy` macro is used to define the `SUSPEND` and `PREEMPT` expressions. If you have changed these expressions at your site, you will need to add `$(WorkHours)` to your `SUSPEND` and `PREEMPT` expressions as appropriate.

Depending on your site, you might also want to avoid suspending jobs during work hours, so that in the morning, if a job is running, it will be immediately preempted, instead of being suspended for some length of time:

```
WANT_SUSPEND = $(AfterHours)
```

**Desktop/Non-Desktop Policy**

Suppose you have two classes of machines in your pool: desktop machines and dedicated cluster machines. In this case, you might not want keyboard activity to have any effect on the dedicated machines. For example, when you log into these machines to debug some problem, you probably do not want a running job to suddenly be killed. Desktop machines, on the other hand, should do whatever is necessary to remain responsive to the user.

There are many ways to achieve the desired behavior. One way is to make a standard desktop policy and a standard non-desktop policy and to copy the desired one into the local configuration file for each machine. Another way is to define one standard policy (in condor_config) with a simple toggle that can be set in the local configuration file. The following example illustrates the latter approach.

For ease of use, an entire policy is included in this example. Some of the expressions are just the usual default settings.

```
# If "IsDesktop" is configured, make it an attribute of the machine ClassAd.
STARTD_EXPRS = IsDesktop

# Only consider starting jobs if:
# 1) the load average is low enough OR the machine is currently
#    running a Condor job
# 2) AND the user is not active (if a desktop)
START = ( ($(CPUIdle) || (State != "Unclaimed" && State != "Owner")) \
          && (IsDesktop =!= True || (KeyboardIdle > $(StartIdleTime))) )

# Suspend (instead of vacating/killing) for the following cases:
WANT_SUSPEND = ( $(SmallJob) || $(JustCpu) || $(IsPVM) \
                 || $(IsVanilla) )

# When preempting, vacate (instead of killing) in the following cases:
WANT_VACATE  = ( $(ActivationTimer) > 10 * $(MINUTE) \
                 || $(IsPVM) || $(IsVanilla) )

# Suspend jobs if:
# 1) The CPU has been busy for more than 2 minutes, AND
# 2) the job has been running for more than 90 seconds
# 3) OR suspend if this is a desktop and the user is active
SUSPEND = ( ((CpuBusyTime > 2 * $(MINUTE)) && ($(ActivationTimer) > 90)) \
            || ( IsDesktop =?= True && $(KeyboardBusy) ) )

# Continue jobs if:
# 1) the CPU is idle, AND
# 2) we've been suspended more than 5 minutes AND
# 3) the keyboard has been idle for long enough (if this is a desktop)
CONTINUE = ( $(CPUIdle) && ($(ActivityTimer) > 300) \
             && (IsDesktop =!= True || (KeyboardIdle > $(ContinueIdleTime))) )

# Preempt jobs if:
# 1) The job is suspended and has been suspended longer than we want
# 2) OR, we don't want to suspend this job, but the conditions to
#    suspend jobs have been met (someone is using the machine)
PREEMPT = ( ((Activity == "Suspended") && \
```

```
            ($(ActivityTimer) > $(MaxSuspendTime))) \
           || (SUSPEND && (WANT_SUSPEND == False)) )

# Replace 0 in the following expression with whatever amount of
# retirement time you want dedicated machines to provide.  The other part
# of the expression forces the whole expression to 0 on desktop
# machines.
MaxJobRetirementTime = (IsDesktop =!= True) * 0

# Kill jobs if they have taken too long to vacate gracefully
KILL = $(ActivityTimer) > $(MaxVacateTime)
```

With this policy in condor_config, the local configuration files for desktops can be easily configured with the following line:

```
IsDesktop = True
```

In all other cases, the default policy described above will ignore keyboard activity.

**Disabling Preemption**

Preemption can result in jobs being killed by Condor. When this happens, the jobs remain in the queue and will be automatically rescheduled. We highly recommend designing jobs that work well in this environment, rather than simply disabling preemption.

Planning for preemption makes jobs more robust in the face of other sources of failure. One way to live happily with preemption is to use Condor's standard universe, which provides the ability to produce checkpoints. If a job is incompatible with the requirements of standard universe, the job can still gracefully shutdown and restart by intercepting the soft kill signal.

All that being said, there may be cases where it is appropriate to force Condor to never kill jobs within some upper time limit. This can be achieved with the following configuration policy:

```
# When we want to kick a job off, let it run uninterrupted for
# up to 2 days before forcing it to vacate.
MAXJOBRETIREMENTTIME = $(HOUR) * 24 * 2
```

Construction of this expression may be more complicated. For example, it could provide a different retirement time to different users or different types of jobs. Also be aware that the job may come with its own definition of `MaxJobRetirementTime`, but this may only cause *less* retirement time to be used, never more than what the machine offers.

The longer the retirement time that is given, the slower reallocation of resources in the pool can become if there are long-running jobs. However, by preventing jobs from being killed, you may decrease the number of cycles that are wasted on non-checkpointable jobs that are killed.

Note that the use of MAXJOBRETIREMENTTIME limits the killing of jobs, but it does not prevent the preemption of resource claims. Therefore, it is technically not a way of disabling preemption, but simply a way of forcing preempting claims to wait until an existing job finishes or runs out of time. In other words, it limits the preemption of jobs but not the preemption of claims.

Limiting the preemption of jobs is often more desirable than limiting the preemption of resource claims. The following configuration limits the preemptions of resource claims:

```
#Disable preemption by machine activity.
PREEMPT = False
#Disable preemption by user priority.
PREEMPTION_REQUIREMENTS = False
#Disable preemption by machine RANK by ranking all jobs equally.
RANK = 0
#Since we are disabling claim preemption, we
# may as well optimize negotiation for this case:
NEGOTIATOR_CONSIDER_PREEMPTION = False
```

Be aware of the consequences of this policy. Without any preemption of resource claims, once the *condor_negotiator* gives the *condor_schedd* a match to a machine, the *condor_schedd* may hold onto this claim indefinitely, as long as the user keeps supplying more jobs to run. If this is not desired, force claims to be retired after some amount of time using CLAIM_WORKLIFE . This enforces a time limit, beyond which no new jobs may be started on an existing claim; therefore the *condor_schedd* daemon is forced to go back to the *condor_negotiator* to request a new match, if there is still more work to do.

Also be aware that in all versions of Condor prior to 6.8.1, it is not advisable to set NEGOTIATOR_CONSIDER_PREEMPTION to False, because of a bug that can lead to some machines never being matched to jobs.

#### Job Suspension

As new jobs are submitted that receive a higher priority than currently executing jobs, the executing jobs may be preempted. These jobs lose whatever forward progress they have made, and are sent back to the job queue to await starting over again as another machine becomes available.

Condor may be configured with a policy that allows these potentially evicted jobs to be suspended instead. The configuration introduces the notion of a virtual machine, even in the case of a single, physical CPU. The policy utilizes two virtual machines, one (called VM1 in the example) that runs jobs following the normal policy implemented for all machines in the pool. The second virtual machine (called VM2 in the example) is set to run jobs whenever it can, and suspend these jobs when VM1 is utilized.

Section 3.3.10 contains details of the STARTD_VM_EXPRS configuration macro, utilized in this policy example. The configuration, with comments to illustrate the intended policy will appear much like this example.

```
# Lie to Condor, to achieve 2 virtual machines with only a single CPU
```

```
NUM_CPUS = 2

# VM1 is the high-prio VM, while VM2 is the background VM...
START = (VirtualMachineID == 1) && $(VM1_START) || \
        (VirtualMachineID == 2) && $(VM2_START)

# Only start jobs on VM1 if the job is marked as a high-priority job
VM1_START = (TARGET.IsHighPrioJob =?= TRUE)

# Only start jobs on VM2 if there is no job on VM1, and if the machine is
# otherwise idle...  NOTE: the "Busy" activity is only in the Claimed
# state, and only when there is an active job, so that is good enough
# for our needs...
VM2_START = ( (vm1_Activity != "Busy") && \
              (KeyboardIdle > $(StartIdleTime)) && \
      ($(CPUIdle) || (State != "Unclaimed" && State != "Owner")) )

# Only suspend jobs on VM2.  Suspend if there is keyboard activity or
# if a job starts on VM1...
SUSPEND = (VirtualMachineID == 2) && \
          ( (vm1_Activity == "Busy") || ($(KeyboardBusy)) )

# Since SUSPEND can only be TRUE on VM2, we can write the
# CONTINUE expression specifically for VM2:
CONTINUE = (KeyboardIdle > $(ContinueIdleTime)) && \
           (vm1_Activity != "Busy")
```

### 3.5.11 Differences from the Version 6.0 Policy Settings

This section describes how the current policy expressions differ from the policy expressions in previous versions of Condor. If you have never used Condor version 6.0 or earlier, or you never looked closely at the policy settings, skip this section.

In summary, there is no longer a VACATE expression, and the KILL expression is not evaluated while a machine is claimed. There is a PREEMPT expression which describes the conditions when a machine will move from the Claimed state to the Preempting state. Once a machine is transitioning into the Preempting state, the WANT_VACATE expression controls whether the job should be vacated with a checkpoint or directly killed. The KILL expression determines the transition from Preempting/Vacating to Preempting/Killing.

In previous versions of Condor, the KILL expression handled three distinct cases (the transitions from Claimed/Busy, Claimed/Suspended and Preempting/Vacating), and the VACATE expression handled two cases (the transitions from Claimed/Busy and Claimed/Suspended). In the current version of Condor, PREEMPT handles the same two cases as the previous VACATE expression, but the KILL expression handles one case. Very complex policies can now be specified using all of the default expressions, only tuning the WANT_VACATE and WANT_SUSPEND expressions. In previous versions, heavy use of the WANT_* expressions caused a complex KILL expression.

## 3.6 Security

Security in Condor is a broad issue, with many aspects to consider. Because Condor's main purpose is to allow users to run arbitrary code on large numbers of computers, it is important to try to limit who can access a Condor pool and what privileges they have when using the pool. This section covers these topics.

There is a distinction between the kinds of resource attacks Condor can defeat, and the kinds of attacks Condor cannot defeat. Condor cannot prevent security breaches of users that can elevate their privilege to the root or administrator account. Condor does not run user jobs in sandboxes (standard universe jobs are a partial exception to this), so Condor cannot defeat all malicious actions by user jobs. An example of a malicious job is one that launches a distributed denial of service attack. Condor assumes that users are trustworthy. Condor can prevent unauthorized access to the Condor pool, to help ensure that only trusted users have access to the pool. In addition, Condor provides encryption and integrity checking, to ensure that data (both Condor's data and user jobs' data) has not been examined or tampered with.

Broadly speaking, the aspects of security in Condor may be categorized and described:

**Users** Authorization or capability in an operating system is based on a process owner. Both those that submit jobs and Condor daemons become process owners. The Condor system prefers that Condor daemons are run as the user `root`, while other common operations are owned by a user of Condor. Operations that do not belong to either `root` or a Condor user are often owned by the `condor` user. See Section 3.6.12 for more detail.

**Authentication** Proper identification of a user is accomplished by the process of authentication. It attempts to distinguish between real users and impostors. By default, Condor's authentication uses the user id (UID) to determine identity, but Condor can choose among a variety of authentication mechanisms, including the stronger authentication methods Kerberos and GSI.

**Authorization** Authorization specifies who is allowed to do what. Some users are allowed to submit jobs, while other users are allowed administrative privileges over Condor itself. Condor provides authorization on either a per-user or on a per-machine basis.

**Privacy** Condor may encrypt data sent across the network, which prevents others from viewing the data. With persistence and sufficient computing power, decryption is possible. Condor can encrypt the data sent for internal communication, as well as user data, such as files and executables. Encryption operates on network transmissions: unencrypted data is stored on disk.

**Integrity** The *man-in-the-middle* attack tampers with data without the awareness of either side of the communication. Condor's integrity check sends additional cryptographic data to verify that network data transmissions have not been tampered with. Note that the integrity information is only for network transmissions: data stored on disk does not have this integrity information.

### 3.6.1 Condor's Security Model

At the heart of Condor's security model is the notion that communications are subject to various security checks. A request from one Condor daemon to another may require authentication to prevent subversion of the system. A request from a user of Condor may need to be denied due to the confidential nature of the request. The security model handles these example situations and many more.

Requests to Condor are categorized into groups of *access levels*, based on the type of operation requested. The user of a specific request must be authorized at the required access level. For example, the executing the *condor_status* command requires the `READ` access level. Actions that accomplish management tasks, such as shutting down or restarting of a daemon require an `ADMINISTRATOR` access level. See Section 3.6.8 for a full list of Condor's access levels and their meanings.

There are two sides to any communication or command invocation in Condor. One side is identified as the *client*, and the other side is identified as the *daemon*. The client is the party that initiates the command, and the daemon is the party that processes the command and responds. In some cases it is easy to distinguish the client from the daemon, while in other cases it is not as easy. Condor tools such as *condor_submit* and *condor_config_val* are clients. They send commands to daemons and act as clients in all their communications. For example, the *condor_submit* command communicates with the *condor_schedd*. Behind the scenes, Condor daemons also communicate with each other; in this case the daemon initiating the command plays the role of the client. For instance, the *condor_negotiator* daemon acts as a client when contacting the *condor_schedd* daemon to initiate matchmaking. Once a match has been found, the *condor_schedd* daemon acts as a client and contacts the *condor_startd* daemon.

Condor's security model is implemented using configuration. Commands in Condor are executed over TCP/IP network connections. While network communication enables Condor to manage resources that are distributed across an organization (or beyond), it also brings in security challenges. Condor must have ways of ensuring that commands are being sent by trustworthy users. Jobs that are operating on sensitive data must be allowed to use encryption such that the data is not seen by outsiders. Jobs may need assurance that data has not been tampered with. These issues can be addressed with Condor's authentication, encryption, and integrity features.

#### Access Level Descriptions

Authorization is granted based on specified access levels. This list describes each access level, and provides examples of their usage. The levels implement a partial hierarchy; a higher level often implies a `READ` or both a `WRITE` and a `READ` level of access as described.

**`READ`** This access level access can obtain or read information about Condor. Examples that require only `READ` access are viewing the status of the pool with *condor_status*, checking a job queue with *condor_q*, or viewing user priorities with *condor_userprio*. `READ` access does not allow any changes, and it does not allow job submission.

**WRITE** This access level is required to send (write) information to Condor. Examples that require WRITE access are job submission with *condor_submit* and advertising a machine so it appears in the pool (this is usually done automatically by the *condor_startd* daemon). The WRITE level of access implies READ access.

**ADMINISTRATOR** This access level has additional Condor administrator rights to the pool. It includes the ability to change user priorities (with the command *condor_userprio -set*), as well as the ability to turn Condor on and off (as with the commands *condor_on* and *condor_off*). The ADMINISTRATOR level of access implies both READ and WRITE access.

**CONFIG** This access level is required to modify a daemon's configuration using the *condor_config_val* command. By default, this level of access can change any configuration parameters of a Condor pool, except those specified in the condor_config.root configuration file. The CONFIG level of access implies READ access.

**OWNER** This level of access is required for commands that the owner of a machine (any local user) should be able to use, in addition to the Condor administrators. An example that requires the OWNER access level is the *condor_vacate* command. The command causes the *condor_startd* daemon to vacate any Condor job currently running on a machine. The owner of that machine should be able to cause the removal of a job running on the machine.

**DAEMON** This access level is used for commands that are internal to the operation of Condor. An example of this internal operation is when the *condor_startd* daemon sends its ClassAd updates to the *condor_collector* daemon. Authorization at this access level should only be given to the user account under which the Condor daemons run. The DAEMON level of access implies both READ and WRITE access.

**NEGOTIATOR** This access level is used specifically to verify that commands are sent by the *condor_negotiator* daemon. The *condor_negotiator* daemon runs on the central manager of the pool. Commands requiring this access level are the ones that tell the *condor_schedd* daemon to begin negotiating, and those that tell an available *condor_startd* daemon that it has been matched to a *condor_schedd* with jobs to run. The NEGOTIATOR level of access implies READ access.

### 3.6.2 Security Negotiation

Because of the wide range of environments and security demands necessary, Condor must be flexible. Configuration provides this flexibility. The process by which Condor determines the security settings that will be used when a connection is established is called *security negotiation*. Security negotiation's primary purpose is to determine which of the features of authentication, encryption, and integrity checking will be enabled for a connection. In addition, since Condor supports multiple technologies for authentication and encryption, security negotiation also determines which technology is chosen for the connection.

Security negotiation is a completely separate process from matchmaking, and should not be confused with any specific function of the *condor_negotiator* daemon.

**Configuration**

The configuration macro names that determine what features will be used during client-daemon communication follow the pattern:

```
SEC_<context>_<feature>
```

The `<feature>` portion of the macro name determines which security feature's policy is being set. `<feature>` may be any one of

```
AUTHENTICATION
ENCRYPTION
INTEGRITY
NEGOTIATION
```

The `<context>` component of the security policy macros can be used to craft a fine-grained security policy based on the type of communication taking place. `<context>` may be any one of

```
CLIENT
READ
WRITE
ADMINISTRATOR
CONFIG
OWNER
DAEMON
NEGOTIATOR
DEFAULT
```

Any of these constructed configuration macros may be set to any of the following values:

```
REQUIRED
PREFERRED
OPTIONAL
NEVER
```

Security negotiation resolves various client-daemon combinations of desired security features in order to set a policy.

As an example, consider Frida the scientist. Frida wants to avoid authentication when possible. She sets

```
SEC_DEFAULT_AUTHENTICATION = OPTIONAL
```

The machine running the *condor_schedd* to which Frida will remotely submit jobs, however, is operated by a security-conscious system administrator who dutifully sets:

| | | Daemon Setting | | |
|---|---|---|---|---|
| | | NEVER | OPTIONAL | REQUIRED |
| Client Setting | NEVER | No | No | Fail |
| | REQUIRED | Fail | Yes | Yes |

Table 3.1: Resolution of security negotiation.

| | | Daemon Setting | | | |
|---|---|---|---|---|---|
| | | NEVER | OPTIONAL | PREFERRED | REQUIRED |
| Client Setting | NEVER | No | No | No | Fail |
| | OPTIONAL | No | No | Yes | Yes |
| | PREFERRED | No | Yes | Yes | Yes |
| | REQUIRED | Fail | Yes | Yes | Yes |

Table 3.2: Resolution of security features.

```
SEC_DEFAULT_AUTHENTICATION = REQUIRED
```

When Frida submits her jobs, Condor's security negotiation determines that authentication will be used, and allows the command to continue.

Whether or not security negotiation occurs depends on the setting at both the client and daemon side of the configuration variable(s) defined by `SEC_*_NEGOTIATION`. `SEC_DEFAULT_NEGOTIATION` is a variable representing the entire set of configuration variables for `NEGOTIATION`. For the client side setting, the only definitions that make sense are `REQUIRED` and `NEVER`. For the daemon side setting, the `PREFERRED` value makes no sense. Table 3.1 shows how security negotiation resolves various client-daemon combinations of security negotiation policy settings. Within the table, Yes means the security negotiation will take place. No means it will not. Fail means that the policy settings are incompatible and the communication cannot continue.

Enabling authentication, encryption, and integrity checks is dependent on security negotiation taking place. The enabled security negotiation further sets the policy for these other features. Table 3.2 shows how security features are resolved for client-daemon combinations of security feature policy settings. Like Table 3.1, Yes means the feature will be utilized. No means it will not. Fail implies incompatibility and the feature cannot be resolved.

The enabling of encryption and/or integrity checks is dependent on authentication taking place. The authentication provides a key exchange. The key is needed for both encryption and integrity checks.

Setting `SEC_CLIENT_<feature>` determines the policy for all outgoing commands. The policy for incoming commands (the daemon side of the communication) takes a more fine-grained approach that implements a set of access levels for the received command. For example, it is desirable to have all incoming administrative requests require authentication. Inquiries on pool status may not be so restrictive. To implement this, the administrator configures the policy:

```
SEC_ADMINISTRATOR_AUTHENTICATION = REQUIRED
```

```
SEC_READ_AUTHENTICATION          = OPTIONAL
```

The DEFAULT value for <context> provides a way to set a policy for all access levels (READ, WRITE, etc.) that do not have a specific configuration variable defined.

**Configuration for Security Methods**

Authentication and encryption can each be accomplished by a variety of methods or technologies. Which method is utilized is determined during security negotiation.

The configuration macros that determine the methods to use for authentication and/or encryption are

```
SEC_<context>_AUTHENTICATION_METHODS
SEC_<context>_CRYPTO_METHODS
```

These macros are defined by a comma or space delimited list of possible methods to use. Section 3.6.3 lists all implemented authentication methods. Section 3.6.5 lists all implemented encryption methods.

### 3.6.3 Authentication

The client side of any communication uses one of two macros to specify whether authentication is to occur:

```
SEC_DEFAULT_AUTHENTICATION
SEC_CLIENT_AUTHENTICATION
```

For the daemon side, there are seven macros to specify whether authentication is to take place, based upon the necessary access level:

```
SEC_DEFAULT_AUTHENTICATION
SEC_READ_AUTHENTICATION
SEC_WRITE_AUTHENTICATION
SEC_ADMINISTRATOR_AUTHENTICATION
SEC_CONFIG_AUTHENTICATION
SEC_OWNER_AUTHENTICATION
SEC_DAEMON_AUTHENTICATION
SEC_NEGOTIATOR_AUTHENTICATION
```

As an example, the macro defined in the configuration file for a daemon as

```
SEC_WRITE_AUTHENTICATION = REQUIRED
```

signifies that the daemon must authenticate the client for any communication that requires the WRITE access level. If the daemon's configuration contains

```
SEC_DEFAULT_AUTHENTICATION = REQUIRED
```

and does not contain any other security configuration for AUTHENTICATION, then this default defines the daemon's needs for authentication over all access levels. Where a specific macro is defined, the more specific value takes precedence over the default definition.

If authentication is to be done, then the communicating parties must negotiate a mutually acceptable method of authentication to be used. A list of acceptable methods may be provided by the client, using the macros

```
SEC_DEFAULT_AUTHENTICATION_METHODS
SEC_CLIENT_AUTHENTICATION_METHODS
```

A list of acceptable methods may be provided by the daemon, using the macros

```
SEC_DEFAULT_AUTHENTICATION_METHODS
SEC_READ_AUTHENTICATION_METHODS
SEC_WRITE_AUTHENTICATION_METHODS
SEC_ADMINISTRATOR_AUTHENTICATION_METHODS
SEC_CONFIG_AUTHENTICATION_METHODS
SEC_OWNER_AUTHENTICATION_METHODS
SEC_DAEMON_AUTHENTICATION_METHODS
SEC_NEGOTIATOR_AUTHENTICATION_METHODS
```

The methods are given as a comma-separated list of acceptable values. These variables list the authentication methods that are available to be used. The ordering of the list defines preference; the first item in the list indicates the highest preference. Defined values are

```
GSI
SSL
KERBEROS
PASSWORD
FS
FS_REMOTE
NTSSPI
CLAIMTOBE
ANONYMOUS
```

For example, a client may be configured with:

```
SEC_CLIENT_AUTHENTICATION_METHODS = FS, GSI
```

and a daemon the client is trying to contact with:

```
SEC_DEFAULT_AUTHENTICATION_METHODS = GSI
```

Security negotiation will determine that GSI authentication is the only compatible choice. If there are multiple compatible authentication methods, security negotiation will make a list of acceptable methods and they will be tried in order until one succeeds.

As another example, the macro

```
SEC_DEFAULT_AUTHENTICATION_METHODS = KERBEROS, NTSSPI
```

indicates that either Kerberos or Windows authentication may be used, but Kerberos is preferred over Windows. Note that if the client and daemon agree that multiple authentication methods may be used, then they are tried in turn. For instance, if they both agree that Kerberos or NTSSPI may be used, then Kerberos will be tried first, and if there is a failure for any reason, then NTSSPI will be tried.

If the configuration for a machine does not define any variable for `SEC_<access-level>_AUTHENTICATION`, then Condor uses a default value of `OPTIONAL`. Authentication will be required for any operation which modifies the job queue, such as *condor_qedit* and *condor_rm*. If the configuration for a machine does not define any variable for `SEC_<access-level>_AUTHENTICATION_METHODS`, the default value for a Unix machine is `FS`, `KERBEROS`, `GSI`. This default value for a Windows machine is `NTSSPI`, `KERBEROS`, `GSI`.

**GSI Authentication**

The GSI (Grid Security Infrastructure) protocol provides an avenue for Condor to do PKI-based (Public Key Infrastructure) authentication using X.509 certificates. The basics of GSI are well-documented elsewhere, such as http://www.globus.org/.

A simple introduction to this type of authentication defines Condor's use of terminology, and it illuminates the needed items that Condor must access to do this authentication. Assume that A authenticates to B. In this example, A is the client, and B is the daemon within their communication. This example's one-way authentication implies that B is verifying the identity of A, using the certificate A provides, and utilizing B's own set of trusted CAs (Certification Authorities). Client A provides its certificate (or proxy) to daemon B. B does two things: B checks that the certificate is valid, and B checks to see that the CA that signed A's certificate is one that B trusts.

For the GSI authentication protocol, an X.509 certificate is required. Files with predetermined names hold a certificate, a key, and optionally, a proxy. A separate directory has one or more files that become the list of trusted CAs.

Allowing Condor to do this GSI authentication requires knowledge of the locations of the client A's certificate and the daemon B's list of trusted CAs. When one side of the communication (as either client A or daemon B) is a Condor daemon, these locations are determined by configuration or by default locations. When one side of the communication (as a client A) is a user of Condor (the process owner of a Condor tool, for example *condor_submit*), these locations are determined by the pre-set values of environment variables or by default locations.

**GSI certificate locations for Condor daemons** For a Condor daemon, the certificate may be a single host certificate, and all Condor daemons on the same machine may share the same certificate. In some cases, the certificate can also be copied to other machines, where local copies are necessary. This may occur only in cases where a single host certificate can match multiple host names, something that is beyond the scope of this manual. The certificates must be protected by access rights to files, since the password file is not encrypted.

The specification of the location of the necessary files through configuration uses the following precedence.

1. Configuration variable `GSI_DAEMON_DIRECTORY` gives the complete path name to the directory that contains the certificate, key, and directory with trusted CAs. Condor uses this directory as follows in its construction of the following configuration variables:

   ```
   GSI_DAEMON_CERT           = $(GSI_DAEMON_DIRECTORY)/hostcert.pem
   GSI_DAEMON_KEY            = $(GSI_DAEMON_DIRECTORY)/hostkey.pem
   GSI_DAEMON_TRUSTED_CA_DIR = $(GSI_DAEMON_DIRECTORY)/certificates
   ```

   Note that no proxy is assumed in this case.

2. If the `GSI_DAEMON_DIRECTORY` is not defined, or when defined, the location may be overridden with specific configuration variables that specify the complete path and file name of the certificate with

   `GSI_DAEMON_CERT`

   the key with

   `GSI_DAEMON_KEY`

   a proxy with

   `GSI_DAEMON_PROXY`

   the complete path to the directory containing the list of trusted CAs with

   `GSI_DAEMON_TRUSTED_CA_DIR`

3. The default location assumed is `/etc/grid-security`. Note that this implemented by setting the value of `GSI_DAEMON_DIRECTORY`.

Here is an example portion of the configuration file that would enable and require GSI authentication, along with a minimal set of other variables to make it work. Note that the last entry (`GSI_DAEMON_NAME`) in this example must be on a single line; this example is broken onto two lines for formatting reasons.

```
SEC_DEFAULT_AUTHENTICATION = REQUIRED
SEC_DEFAULT_AUTHENTICATION_METHODS = GSI
GSI_DAEMON_DIRECTORY = /path/to/daemon/credential.location
GSI_DAEMON_NAME = /C=US/O=Condor/O=University of Wisconsin
/OU=Computer Sciences Department/CN=condor@cs.wisc.edu
```

The SEC_DEFAULT_AUTHENTICATION macro specifies that authentication is required for all communications. This single macro covers all communications, but could be replaced with a set of macros that require authentication for only specific communications.

The macro GSI_DAEMON_DIRECTORY is specified to give Condor a single place to find the daemon's certificate. This path may be a directory on a shared file system such as AFS. Alternatively, this path name can point to local copies of the certificate stored in a local file system.

When a daemon acts as the client within authentication, the daemon needs a listing of those from which it will accept certificates.

The macro GSI_DAEMON_NAME configuration macro provides daemons with a distinguished name to use for X.509 authentication. This name is specified with the following format

```
GSI_DAEMON_NAME = /C=?/O=?/O=?/OU=?/CN=<daemon_name@domain>
```

A complete example that has the question marks filled in and the daemon's user name filled in is given in the example configuration above.

Condor will also need a way to map an X.509 distinguished name to a Condor user id. There are two ways to accomplish this mapping. For a first way to specify the mapping, see section 3.6.4 to use Condor's unified map file. The second way to do the mapping is within an administrator-maintained GSI-specific file called an X.509 map file, mapping from X509 Distinguished Name (DN) to Condor user id. It is similar to a Globus grid map file except that it is only used for mapping to a user id, not for authorization. Information about authorization can be found in Section 3.6.8. Entries (lines) in the file each contain two items. The first item in an entry is the X.509 certificate subject name, and it is enclosed in quotes (using the character "). The second item is the Condor user id. The two items in an entry are separated by tab or space character(s). Here is an example of an entry in an X.509 map file. Entries must be on a single line; this example is broken onto two lines for formatting reasons.

```
"/C=US/O=Globus/O=University of Wisconsin/
OU=Computer Sciences Department/CN=Alice Smith" asmith
```

Condor finds the map file in one of three ways. If the configuration variable GRIDMAP is defined, it gives the full path name to the map file. When not defined, Condor looks for the map file in

```
$(GSI_DAEMON_DIRECTORY)/grid-mapfile
```

If GSI_DAEMON_DIRECTORY is not defined, then the third place Condor looks for the map file is given by

```
/etc/grid-security/grid-mapfile
```

**GSI certificate locations for Users** The user specifies the location of a certificate, proxy, etc. in one of two ways:

1. Environment variables give the location of necessary items.
   X509_USER_PROXY gives the path and file name of the proxy. This proxy will have been created using the *grid-proxy-init* program, which will place the proxy in the /tmp directory with the file name being determined by the format:

   ```
   /tmp/x509up_uXXXX
   ```

   The specific file name is given by substituting the XXXX characters with the UID of the user. Note that when a valid proxy is used, the certificate and key locations are not needed.
   X509_USER_CERT gives the path and file name of the certificate. It is also used if a proxy location has been checked, but the proxy is no longer valid.
   X509_USER_KEY gives the path and file name of the key. Note that most keys are password encrypted, such that knowing the location could not lead to using the key.
   X509_CERT_DIR gives the path to the directory containing the list of trusted CAs.
2. Without environment variables to give locations of necessary certificate information, Condor uses a default directory for the user. This directory is given by

   ```
   $(HOME)/.globus
   ```

**SSL Authentication**

SSL authentication is similar to GSI authentication, but without GSI's delegation (proxy) capabilities. SSL utilizes X.509 certificates.

All SSL authentication is mutual authentication in Condor. This means that when SSL authentication is used and when one process communicates with another, each process must be able to verify the signature on the certificate presented by the other process. The process that initiates the connection is the client, and the process that receives the connection is the server. For example, when a *condor_startd* daemon authenticates with a *condor_collector* daemon to provide a machine ClassAd, the *condor_startd* daemon initiates the connection and acts as the client, and the *condor_collector* daemon acts as the server.

The names and locations of keys and certificates for clients, servers, and the files used to specify trusted certificate authorities (CAs) are defined by settings in the configuration files. The contents of the files are identical in format and interpretation to those used by other systems which use SSL, such as Apache httpd.

The configuration variables AUTH_SSL_CLIENT_CERTFILE and AUTH_SSL_SERVER_CERTFILE specify the file location for the certificate file for the initiator and recipient of connections, respectively. Similarly, the configuration variables AUTH_SSL_CLIENT_KEYFILE and AUTH_SSL_SERVER_KEYFILE specify the locations for keys.

The configuration variables AUTH_SSL_SERVER_CAFILE and AUTH_SSL_CLIENT_CAFILE each specify a path and file name, providing the location of a file containing one or more certificates issued by trusted certificate authorities. Similarly, AUTH_SSL_SERVER_CADIR and

AUTH_SSL_CLIENT_CADIR each specify a directory with one or more files, each which may contain a single CA certificate. The directories must be prepared using the OpenSSL c_rehash utility.

**Kerberos Authentication**

If Kerberos is used for authentication, then a mapping from a Kerberos domain (called a realm) to a Condor UID domain is necessary. There are two ways to accomplish this mapping. For a first way to specify the mapping, see section 3.6.4 to use Condor's unified map file. A second way to specify the mapping defines the configuration variable KERBEROS_MAP_FILE to define a path to an administrator-maintained Kerberos-specific map file. The configuration syntax is

```
KERBEROS_MAP_FILE = /path/to/etc/condor.kmap
```

Lines within this map file have the syntax

```
KERB.REALM = UID.domain.name
```

Here are two lines from a map file to use as an example:

```
CS.WISC.EDU   = cs.wisc.edu
ENGR.WISC.EDU = ee.wisc.edu
```

If a KERBEROS_MAP_FILE configuration variable is defined and set, then all permitted realms must be explicitly mapped. If no map file is specified, then Condor assumes that the Kerberos realm is the same as the Condor UID domain.

The configuration variable CONDOR_SERVER_PRINCIPAL defines the name of a Kerberos principal. If CONDOR_SERVER_PRINCIPAL is not defined, then the default value used is "host". A principal specifies a unique name to which a set of credentials may be assigned.

Condor takes the specified (or default) principal and appends a slash character, the host name, an '@' (at sign character), and the Kerberos realm. As an example, the configuration

```
CONDOR_SERVER_PRINCIPAL = condor-daemon
```

results in Condor's use of

```
condor-daemon/the.host.name@YOUR.KERB.REALM
```

as the server principal.

Here is an example of configuration settings that use Kerberos for authentication and require authentication of all communications of the write or administrator access level.

```
SEC_WRITE_AUTHENTICATION                 = REQUIRED
SEC_WRITE_AUTHENTICATION_METHODS         = KERBEROS
SEC_ADMINISTRATOR_AUTHENTICATION         = REQUIRED
SEC_ADMINISTRATOR_AUTHENTICATION_METHODS = KERBEROS
```

Kerberos authentication on Unix platforms requires access to various files that usually are only accessible by the root user. At this time, the only supported way to use KERBEROS authentication on Unix platforms is to start daemons Condor as user `root`.

**Password Authentication**

The password method provides mutual authentication through the use of a shared secret. This is often a good choice when strong security is desired, but an existing Kerberos or X.509 infrastructure is not in place. Password authentication is available on both Unix and Windows. It currently can only be used for daemon-to-daemon authentication. The shared secret in this context is referred to as the *pool password*.

Before a daemon can use password authentication, the pool password must be stored on the daemon's local machine. On Unix, the password will be placed in a file defined by the configuration variable `SEC_PASSWORD_FILE` . This file will be accessible only by the UID that Condor is started as. On Windows, the same secure password store that is used for user passwords will be used for the pool password (see section 6.2.3).

Storing the pool password is done via the **-c** option to *condor_store_cred*. Running

```
condor_store_cred -c add
```

will prompt for the pool password and store it on the local machine, making it available for daemons to use for authentication. The *condor_master* must be running for this command to work.

In addition, storing the pool password to a given machine requires `CONFIG`-level access. For example, if the pool password should only be set locally, and only by root, the following would be placed in the global configuration file.

```
ALLOW_CONFIG = root@mydomain/$(IP_ADDRESS)
```

It is also possible to set the pool password remotely, but this is recommended only if it can be done over an encrypted channel. This is possible on Windows, for example, in an environment where common accounts exist across all the machines in the pool. In this case, `ALLOW_CONFIG` can be set to allow the Condor administrator (who in this example has an account `condor` common to all machines in the pool) to set the password from the central manager as follows.

```
ALLOW_CONFIG = condor@mydomain/$(CONDOR_HOST)
```

The Condor administrator then executes

```
condor_store_cred -c -n host.mydomain add
```

from the central manager to store the password to a given machine. Since the `condor` account exists on both the central manager and `host.mydomain`, the NTSSPI authentication method can be used to authenticate and encrypt the connection. *condor_store_cred* will warn and prompt for cancellation, if the channel is not encrypted for whatever reason (typically because common accounts do not exist or Condor's security is misconfigured).

When a daemon is authenticated using a pool password, its security principle is `condor_pool@$(UID_DOMAIN)`, where `$(UID_DOMAIN)` is taken from the daemon's configuration. The `ALLOW_DAEMON` and `ALLOW_NEGOTIATOR` configuration variables for authorization should restrict access using this name. For example,

```
ALLOW_DAEMON = condor_pool@mydomain/*, condor@mydomain/$(IP_ADDRESS)
ALLOW_NEGOTIATOR = condor_pool@mydomain/$(CONDOR_HOST)
```

This configuration allows remote `DAEMON`-level and `NEGOTIATOR`-level access, if the pool password is known. Local daemons authenticated as `condor@mydomain` are also allowed access. This is done so local authentication can be done using another method such as `FS`.

#### File System Authentication

This form of authentication utilizes the ownership of a file in the identity verification of a client. A daemon authenticating a client requires the client to write a file in a specific location (`/tmp`). The daemon then checks the ownership of the file. The file's ownership verifies the identity of the client. In this way, the file system becomes the trusted authority. This authentication method is only appropriate for clients and daemons that are on the same computer.

#### File System Remote Authentication

Like file system authentication, this form of authentication utilizes the ownership of a file in the identity verification of a client. In this case, a daemon authenticating a client requires the client to write a file in a specific location, but the location is not restricted to `/tmp`. The location of the file is specified by the configuration variable `FS_REMOTE_DIR` .

#### Windows Authentication

This authentication is done only among Windows machines using a proprietary method. The Windows security interface SSPI is used to enforce NTLM (NT LAN Manager). The authentication is based on challenge and response, using the user's password as a key. This is similar to Kerberos. The main difference is that Kerberos provides an access token that typically grants access to an entire network, whereas NTLM authentication only verifies an identity to one machine at a time. NTSSPI

is best-used in a way similar to file system authentication in Unix, and probably should not be used for authentication between two computers.

#### Claim To Be Authentication

Claim To Be authentication accepts any identity claimed by the client. As such, it does not authenticate. It is included in Condor and in the list of authentication methods for testing purposes only.

#### Anonymous Authentication

Anonymous authentication causes authentication to be skipped entirely. As such, it does not authenticate. It is included in Condor and in the list of authentication methods for testing purposes only.

### 3.6.4 The Unified Map File for Authentication

Condor's unified map file allows the mappings from authenticated names to a Condor canonical user name to be specified as a single list within a single file. The location of the unified map file is defined by the configuration variable `CERTIFICATE_MAPFILE` ; it specifies the path and file name of the unified map file. Each mapping is on its own line of the unified map file. Each line contains 3 fields, separated by white space (space or tab characters):

1. The name of the authentication method to which the mapping applies.
2. A regular expression representing the authenticated name to be mapped.
3. The canonical Condor user name.

Allowable authentication method names are the same as used to define any of the configuration variables `SEC_*_AUTHENTICATION_METHODS`, as repeated here:

```
GSI
SSL
KERBEROS
PASSWORD
FS
FS_REMOTE
NTSSPI
CLAIMTOBE
ANONYMOUS
```

The fields that represent an authenticated name and the canonical Condor user name may utilize regular expressions as defined by PCRE (Perl-Compatible Regular Expressions). Due to this, more than one line (mapping) within the unified map file may match. Lookups are therefore defined to use the first mapping that matches.

The default map file contains a mapping that matches for each authentication method that Condor implements. The unusual string for the GSI authentication entry instructs Condor to use GSI-specific way of locating the needed GSI map file, as shown in section 3.6.3. The default map file:

```
FS (.*) \1
FS_REMOTE (.*) \1
GSI (.*) GSS_ASSIST_GRIDMAP
SSL (.*) \1
KERBEROS (.*) \1
NTSSPI (.*) \1
CLAIMTOBE (.*) \1
PASSWORD (.*) \1
```

### 3.6.5 Encryption

Encryption provides privacy support between two communicating parties. Through configuration macros, both the client and the daemon can specify whether encryption is required for further communication.

The client uses one of two macros to enable or disable encryption:

```
SEC_DEFAULT_ENCRYPTION
SEC_CLIENT_ENCRYPTION
```

For the daemon, there are seven macros to enable or disable encryption:

```
SEC_DEFAULT_ENCRYPTION
SEC_READ_ENCRYPTION
SEC_WRITE_ENCRYPTION
SEC_ADMINISTRATOR_ENCRYPTION
SEC_CONFIG_ENCRYPTION
SEC_OWNER_ENCRYPTION
SEC_DAEMON_ENCRYPTION
SEC_NEGOTIATOR_ENCRYPTION
```

As an example, the macro defined in the configuration file for a daemon as

```
SEC_CONFIG_ENCRYPTION = REQUIRED
```

signifies that any communication that changes a daemon's configuration must be encrypted. If a daemon's configuration contains

```
SEC_DEFAULT_ENCRYPTION = REQUIRED
```

and does not contain any other security configuration for ENCRYPTION, then this default defines the daemon's needs for encryption over all access levels. Where a specific macro is present, its value takes precedence over any default given.

If encryption is to be done, then the communicating parties must find (negotiate) a mutually acceptable method of encryption to be used. A list of acceptable methods may be provided by the client, using the macros

```
SEC_DEFAULT_CRYPTO_METHODS
SEC_CLIENT_CRYPTO_METHODS
```

A list of acceptable methods may be provided by the daemon, using the macros

```
SEC_DEFAULT_CRYPTO_METHODS
SEC_READ_CRYPTO_METHODS
SEC_WRITE_CRYPTO_METHODS
SEC_ADMINISTRATOR_CRYPTO_METHODS
SEC_CONFIG_CRYPTO_METHODS
SEC_OWNER_CRYPTO_METHODS
SEC_DAEMON_CRYPTO_METHODS
SEC_NEGOTIATOR_CRYPTO_METHODS
```

The methods are given as a comma-separated list of acceptable values. These variables list the encryption methods that are available to be used. The ordering of the list gives preference; the first item in the list indicates the highest preference. Possible values are

```
3DES
BLOWFISH
```

### 3.6.6 Integrity

An integrity check assures that the messages between communicating parties have not been tampered with. Any change, such as addition, modification, or deletion can be detected. Through configuration macros, both the client and the daemon can specify whether an integrity check is required of further communication.

The client uses one of two macros to enable or disable an integrity check:

```
SEC_DEFAULT_INTEGRITY
SEC_CLIENT_INTEGRITY
```

For the daemon, there are seven macros to enable or disable an integrity check:

```
SEC_DEFAULT_INTEGRITY
SEC_READ_INTEGRITY
SEC_WRITE_INTEGRITY
SEC_ADMINISTRATOR_INTEGRITY
SEC_CONFIG_INTEGRITY
SEC_OWNER_INTEGRITY
SEC_NEGOTIATOR_INTEGRITY
```

As an example, the macro defined in the configuration file for a daemon as

```
SEC_CONFIG_INTEGRITY = REQUIRED
```

signifies that any communication that changes a daemon's configuration must have its integrity assured. If a daemon's configuration contains

```
SEC_DEFAULT_INTEGRITY = REQUIRED
```

and does not contain any other security configuration for `INTEGRITY`, then this default defines the daemon's needs for integrity checks over all access levels. Where a specific macro is present, its value takes precedence over any default given.

A signed MD5 checksum is currently the only available method for integrity checking. Its use is implied whenever integrity checks occur. If more methods are implemented, then there will be further macros to allow both the client and the daemon to specify which methods are acceptable.

### 3.6.7 Example of Daemon-Side Security Configuration

A configuration file is provided when Condor is installed. No security features are enabled within the configuration as distributed. Included as comments within the configuration file is an example suggesting settings that enable security features. Here is that example of the daemon-side portion.

```
SEC_DEFAULT_NEGOTIATION     = REQUIRED
SEC_DEFAULT_AUTHENTICATION  = REQUIRED
SEC_DEFAULT_ENCRYPTION      = REQUIRED
SEC_DEFAULT_INTEGRITY       = REQUIRED

SEC_DEFAULT_AUTHENTICATION_METHODS = KERBEROS, FS
SEC_DEFAULT_CRYPTO_METHODS         = 3DES, BLOWFISH
```

This set of configuration macros forces security negotiation to occur, and sets the features to be used at all times. All communication is authenticated with Kerberos, unless the client does not use Kerberos, but supports File System (FS) authentication, in which case FS authentication is used. All

communication is both encrypted and integrity checked to make sure that messages are not modified or corrupted. The encryption is preferably with triple DES, but Blowfish will be used if the client does not use 3DES, but does use Blowfish.

### 3.6.8 Authorization

Authorization protects resource usage by granting or denying access requests made to the resources. It defines who is allowed to do what.

Authorization is defined in terms of users. An initial implementation provided authorization based on hosts (machines), while the current implementation relies on user-based authorization. Section 3.6.10 on Setting Up IP/Host-Based Security in Condor describes the previous implementation. This IP/Host-Based security still exists, and it can be used, but significantly stronger and more flexible security can be achieved with the newer authorization based on fully qualified user names. This section discusses user-based authorization.

Unlike authentication, encryption, and integrity checks, which can be configured by both client and server, authorization is used only by a server. The authorization portion of the security of a Condor pool is based on a set of configuration macros. The macros list which user/daemon will be authorized to issue what request given a specific access level.

These configuration macros define a set of users that will be allowed to (or denied from) carrying out various Condor commands. Each access level may have its own list of authorized users. A complete list of the authorization macros:

```
ALLOW_READ
ALLOW_WRITE
ALLOW_ADMINISTRATOR
ALLOW_CONFIG
ALLOW_OWNER
ALLOW_NEGOTIATOR
DENY_READ
DENY_WRITE
DENY_ADMINISTRATOR
DENY_CONFIG
DENY_OWNER
DENY_NEGOTIATOR
```

Each macro is defined by a comma-separated list of fully qualified users. Each fully qualified user is described using the following format:

```
username@domain/hostname
```

The information to the left of the slash character describes a user within a domain. The information to the right of the slash character describes one or more machines from which the user would be

issuing a command. This host name may take the form of either a fully qualified host name of the form

```
bird.cs.wisc.edu
```

or an IP address of the form

```
128.105.128.0
```

An example is

```
zmiller@cs.wisc.edu/bird.cs.wisc.edu
```

Within the format, wild card characters (the asterisk, *) are allowed. The use of wild cards is limited to one wild card on either side of the slash character. A wild card character used in the host name is further limited to come at the beginning of a fully qualified host name or at the end of an IP address. For example,

```
*@cs.wisc.edu/bird.cs.wisc.edu
```

refers to any user that comes from `cs.wisc.edu`, where the command is originating from the machine `bird.cs.wisc.edu`. Another valid example,

```
zmiller@cs.wisc.edu/*.cs.wisc.edu
```

refers to commands coming from any machine within the `cs.wisc.edu` domain, and issued by `zmiller`. A third valid example,

```
*@cs.wisc.edu/*
```

refers to commands coming from any user within the `cs.wisc.edu` domain where the command is issued from any machine. A fourth valid example,

```
*@cs.wisc.edu/128.105.*
```

refers to commands coming from any user within the `cs.wisc.edu` domain where the command is issued from machines within the network that match the first two octets of the IP address.

If the set of machines is specified by an IP address, then further specification using a net mask identifies a physical set (subnet) of machines. This physical set of machines is specified using the form

```
network/netmask
```

The `network` is an IP address. The net mask takes one of two forms. It may be a decimal number which refers to the number of leading bits of the IP address that are used in describing a subnet. Or, the net mask may take the form of

`a.b.c.d`

where `a`, `b`, `c`, and `d` are decimal numbers that each specify an 8-bit mask. An example net mask is

`255.255.192.0`

which specifies the bit mask

`11111111.11111111.11000000.00000000`

A single complete example of a configuration variable that uses a net mask is

```
ALLOW_WRITE = joesmith@cs.wisc.edu/128.105.128.0/17
```

User `joesmith` within the `cs.wisc.edu` domain is given write authorization when originating from machines that match their leftmost 17 bits of the IP address.

This flexible set of configuration macros could used to define conflicting authorization. Therefore, the following protocol defines the precedence of the configuration macros.

1. `DENY_*` macros take precedence over `ALLOW_*` `macros` where there is a conflict. This implies that if a specific user is both denied and granted authorization, the conflict is resolved by denying access.
2. If macros are omitted, the default behavior is to grant authorization for every user.

**Example of Authorization Security Configuration**

An example of the configuration variables for the user-side authorization is derived from the necessary access levels as described in Section 3.6.1.

```
ALLOW_READ            = *@cs.wisc.edu/*
ALLOW_WRITE           = *@cs.wisc.edu/*.cs.wisc.edu
ALLOW_ADMINISTRATOR   = condor-admin@cs.wisc.edu/*.cs.wisc.edu
ALLOW_CONFIG          = condor-admin@cs.wisc.edu/*.cs.wisc.edu
ALLOW_NEGOTIATOR      = condor@cs.wisc.edu/$(COLLECTOR_HOST)
```

This example configuration presumes that the *condor_collector* and *condor_negotiator* daemons are running on the same machine.

This example configuration authorizes any user in the cs.wisc.edu domain to carry out a request that requires the READ access level from any machine. Any user in the cs.wisc.edu domain may carry out a request that requires the WRITE access level from any machine in the cs.wisc.edu domain. Only the user called condor-admin may carry out a request that requires the ADMINISTRATOR access level from any machine in the cs.wisc.edu domain. The administrator, logged into any machine within the cs.wisc.edu domain is authorized at the CONFIG access level. Only the negotiator daemon, running as condor on the machine defined by the NEGOTIATOR_HOST macro is authorized with the NEGOTIATOR access level. And, the last line of the example presumes that there is a user called condor, and that the daemons have all been started up as this user.

In the local configuration file for each host, the host's owner should be authorized as the owner of the machine. An example of the entry in the local configuration file:

```
ALLOW_OWNER           = username@cs.wisc.edu/hostname.cs.wisc.edu
```

In this example the owner has a login of username, and the machine's name is represented by hostname.

### 3.6.9 Security Sessions

To set up and configure secure communications in Condor, authentication, encryption, and integrity checks can be used. However, these come at a cost: performing strong authentication can take a significant amount of time, and generating the cryptographic keys for encryption and integrity checks can take a significant amount of processing power.

The Condor system makes many network connections between different daemons. If each one of these was to be authenticated, and new keys were generated for each connection, Condor would not be able to scale well. Therefore, Condor uses the concept of *sessions* to cache relevant security information for future use and greatly speed up the establishment of secure communications between the various Condor daemons.

A new session is established the first time a connection is made from one daemon to another. Each session has a fixed lifetime after which it will expire and a new session will need to be created again. But while a valid session exists, it can be re-used as many times as needed, thereby preventing the need to continuously re-establish secure connections. Each entity of a connection will have access to a *session key* that proves the identity of the other entity on the opposing side of the connection. This session key is exchanged securely using a strong authentication method, such as Kerberos or GSI. Other authentication methods, such as NTSSPI, FS_REMOTE, CLAIMTOBE, and ANONYMOUS, do not support secure key exchange. An entity listening on the wire may be able to impersonate the client or server in a session that does not use a strong authentication method.

Establishing a secure session requires that either the encryption or the integrity options be enabled. If the encryption capability is enabled, then the session will be restarted using the session key as the encryption key. If integrity capability is enabled, then the checksum includes the session key even though it is not transmitted. Without either of these two methods enabled, it is possible

for an attacker to use an open session to make a connection to a daemon and use that connection for nefarious purposes. It is strongly recommended that if *you have authentication turned on, you should also turn on integrity and/or encryption.*

The configuration parameter SEC_DEFAULT_NEGOTIATION will allow a user to set the default level of secure sessions in Condor. Like other security settings, the possible values for this parameter can be REQUIRED, PREFERRED, OPTIONAL, or NEVER. If you disable sessions and you have authentication turned on, then most authentication (other than commands like *condor_submit*) will fail because Condor requires sessions when you have security turned on. On the other hand, if you are not using strong security in Condor, but you are relying on the default host-based security, turning off sessions may be useful in certain situations. These might include debugging problems with the security session management or slightly decreasing the memory consumption of the daemons, which keep track of the sessions in use.

Session lifetimes for specific daemons are already properly configured in the default installation of Condor. Condor tools such as *condor_q* and *condor_status* create a session that expires after one minute. Theoretically they should not create a session at all, because the session cannot be reused between program invocations, but this is difficult to do in the general case. This allows a very small window of time for any possible attack, and it helps keep the memory footprint of running daemons down, because they are not keeping track of all of the sessions. The session durations may be manually tuned by using macros in the configuration file, but this is not recommended.

### 3.6.10 Host-Based Security in Condor

This section describes the mechanisms for setting up Condor's host-based security. This is now an outdated form of implementing security levels for machine access. It remains available and documented for purposes of backward compatibility. If used at the same time as the user-based authorization, the two specifications are merged together.

The host-based security paradigm allows control over which machines can join a Condor pool, which machines can find out information about your pool, and which machines within a pool can perform administrative commands. By default, Condor is configured to allow anyone to view or join a pool. It is recommended that this parameter is changed to only allow access from machines that you trust.

This section discusses how the host-based security works inside Condor. It lists the different levels of access and what parts of Condor use which levels. There is a description of how to configure a pool to grant or deny certain levels of access to various machines. Configuration examples and the settings of configuration variables using the *condor_config_val* command complete this section.

Inside the Condor daemons or tools that use DaemonCore (see section 3.9 for details), most tasks are accomplished by sending commands to another Condor daemon. These commands are represented by an integer value to specify which command is being requested, followed by any optional information that the protocol requires at that point (such as a ClassAd, capability string, etc). When the daemons start up, they will register which commands they are willing to accept, what to do with arriving commands, and the access level required for each command. When a

command request is received by a daemon, Condor identifies the access level required and checks the IP address of the sender to verify that it satisfies the allow/deny settings from the configuration file. If permission is granted, the command request is honored; otherwise, the request will be aborted.

Settings for the access levels in the global configuration file will affect all the machines in the pool. Settings in a local configuration file will only affect the specific machine. The settings for a given machine determine what other hosts can send commands to that machine. If a machine foo is to be given administrator access on machine bar, place foo in bar's configuration file access list (not the other way around).

The following are the various access levels that commands within Condor can be registered with:

**READ** Machines with READ access can read information from the Condor daemons. For example, they can view the status of the pool, see the job queue(s), and view user permissions. READ access does not allow a machine to alter any information, and does not allow job submission. A machine listed with READ permission will be unable join a Condor pool; the machine can only view information about the pool.

**WRITE** Machines with WRITE access can write information to the Condor daemons. Most important for granting a machine with this access is that the machine will be able to join a pool since they are allowed to send ClassAd updates to the central manager. The machine can talk to the other machines in a pool in order to submit or run jobs. In addition, any machine with WRITE access can request the *condor_startd* daemon to perform periodic checkpoints on an executing job. After the checkpoint is completed, the job will continue to execute and the machine will still be claimed by the original *condor_schedd* daemon. This allows users on the machines where they submitted their jobs to use the *condor_checkpoint* command to get their jobs to periodically checkpoint, even if the users do not have an account on the machine where the jobs execute.

**IMPORTANT:** For a machine to join a Condor pool, the machine must have both WRITE permission **AND** READ permission. WRITE permission is not enough.

**ADMINISTRATOR** Machines with ADMINISTRATOR access are granted additional Condor administrator rights to the pool. This includes the ability to change user priorities (with the command `userprio -set`), and the ability to turn Condor on and off (with the command *condor_off* <machine>). It is recommended that few machines be granted administrator access in a pool; typically these are the machines that are used by Condor and system administrators as their primary workstations, or the machines running as the pool's central manager.

**IMPORTANT:** Giving ADMINISTRATOR privileges to a machine grants administrator access for the pool to **ANY USER** on that machine. This includes any users who can run Condor jobs on that machine. It is recommended that ADMINISTRATOR access is granted with due diligence.

**OWNER** This level of access is required for commands that the owner of a machine (any local user) should be able to use, in addition to the Condor administrators. For example, the *condor_vacate* command causes the *condor_startd* daemon to vacate any running Condor job. It

requires OWNER permission, so that any user logged into a local machine can issue a *condor_vacate* command.

**NEGOTIATOR** This access level is used specifically to verify that commands are sent by the *condor_negotiator* daemon. The *condor_negotiator* daemon runs on the central manager of the pool. Commands requiring this access level are the ones that tell the *condor_schedd* daemon to begin negotiating, and those that tell an available *condor_startd* daemon that it has been matched to a *condor_schedd* with jobs to run.

**CONFIG** This access level is required to modify a daemon's configuration using the *condor_config_val* command. By default, machines with this level of access are able to change any configuration parameter, except those specified in the condor_config.root configuration file. Therefore, one should exercise extreme caution before granting this level of host-wide access. Because of the implications caused by CONFIG privileges, it is disabled by default for all hosts.

Starting with version 6.3.2, Condor provides a mechanism for more fine-grained control over the configuration settings that can be modified remotely with *condor_config_val*.

Host-based security access permissions are specified in configuration files.

ADMINISTRATOR and NEGOTIATOR access default to the central manager machine. OWNER access defaults to the local machine, as well as any machines given with ADMINISTRATOR access. CONFIG access is not granted to any machine as its default. These defaults are sufficient for most pools, and should not be changed without a compelling reason. If machines other than the default are to have to have OWNER access, they probably should also have ADMINISTRATOR access. By granting machines ADMINISTRATOR access, they will automatically have OWNER access, given how OWNER access is set within the configuration.

The default access configuration is

```
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST)
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
HOSTALLOW_READ = *
HOSTALLOW_WRITE = *
HOSTALLOW_NEGOTIATOR = $(COLLECTOR_HOST)
HOSTALLOW_NEGOTIATOR_SCHEDD = $(COLLECTOR_HOST), $(FLOCK_NEGOTIATOR_HOSTS)
HOSTALLOW_WRITE_COLLECTOR = $(HOSTALLOW_WRITE), $(FLOCK_FROM)
HOSTALLOW_WRITE_STARTD    = $(HOSTALLOW_WRITE), $(FLOCK_FROM)
HOSTALLOW_READ_COLLECTOR  = $(HOSTALLOW_READ), $(FLOCK_FROM)
HOSTALLOW_READ_STARTD     = $(HOSTALLOW_READ), $(FLOCK_FROM)
```

This example configuration presumes that the *condor_collector* and *condor_negotiator* daemons are running on the same machine.

For each access level, an ALLOW or a DENY may be added.

- If you have an ALLOW, it means "only allow these machines". No ALLOW means allow anyone.

- If you have a DENY, it means ”deny these machines”. No DENY means to deny nobody.
- If you have both an ALLOW and a DENY, it means allow the machines listed in ALLOW except for the machines listed in DENY.
- Exclusively for the `CONFIG` access, no ALLOW means allow no one. Note that this is different than the other ALLOW configurations. It is different to enable more stringent security where older configurations are used, since older configuration files would not have a `CONFIG` configuration entry.

Multiple machine entries in the configuration files may be separated by either a space or a comma. The machines may be listed by

- Individual host names - for example: condor.cs.wisc.edu
- Individual IP address - for example: 128.105.67.29
- IP subnets (use a trailing “*”) - for example: 144.105.*, 128.105.67.*
- Host names with a wild card “*” character (only one “*” is allowed per name) - for example: *.cs.wisc.edu, sol*.cs.wisc.edu

To resolve an entry that falls into both allow and deny: individual machines have a higher order of precedence than wild card entries, and host names with a wild card have a higher order of precedence than IP subnets. Otherwise, DENY has a higher order of precedence than ALLOW. (this is how most people would intuitively expect it to work).

In addition, the above access levels may be specified on a per-daemon basis, instead of machine-wide for all daemons. Do this with the subsystem string (described in section 3.3.1 on Subsystem Names), which is one of: STARTD, SCHEDD, MASTER, NEGOTIATOR, or COLLECTOR. For example, to grant different read access for the *condor_schedd*:

```
HOSTALLOW_READ_SCHEDD = <list of machines>
```

The following is a list of registered commands that daemons will accept. The list is ordered by daemon. For each daemon, the commands are grouped by the access level required for a daemon to accept the command from a given machine.

ALL DAEMONS:

**`WRITE`** The command sent as a result of *condor_reconfig* to reconfigure a daemon.

**`ADMINISTRATOR`** The command sent as a result of `reconfig -full` to perform a full reconfiguration on a daemon.

STARTD:

**WRITE** All commands that relate to a *condor_schedd* daemon claiming a machine, starting jobs there, or stopping those jobs.

The command that *condor_checkpoint* sends to periodically checkpoint all running jobs.

**READ** The command that *condor_preen* sends to request the current state of the *condor_startd* daemon.

**OWNER** The command that *condor_vacate* sends to cause any running jobs to stop running.

**NEGOTIATOR** The command that the *condor_negotiator* daemon sends to match a machine's *condor_startd* daemon with a given *condor_schedd* daemon.

NEGOTIATOR:

**WRITE** The command that initiates a new negotiation cycle. It is sent by the *condor_schedd* when new jobs are submitted or a *condor_reschedule* command is issued.

**READ** The command that can retrieve the current state of user priorities in the pool (sent by the *condor_userprio* command).

**ADMINISTRATOR** The command that can set the current values of user priorities (sent as a result of the `userprio -set` command).

COLLECTOR:

**WRITE** All commands that update the *condor_collector* daemon with new ClassAds.

**READ** All commands that query the *condor_collector* daemon for ClassAds.

SCHEDD:

**NEGOTIATOR** The command that the *condor_negotiator* sends to begin negotiating with this *condor_schedd* to match its jobs with available *condor_startds*.

**WRITE** The command which *condor_reschedule* sends to the *condor_schedd* to get it to update the *condor_collector* with a current ClassAd and begin a negotiation cycle.

The commands that a *condor_startd* sends to the *condor_schedd* when it must vacate its jobs and release the *condor_schedd's* claim.

The commands which write information into the job queue (such as *condor_submit* and *condor_hold*). Note that for most commands which attempt to write to the job queue, Condor will perform an additional user-level authentication step. This additional user-level authentication prevents, for example, an ordinary user from removing a different user's jobs.

**READ** The command from any tool to view the status of the job queue.

MASTER: All commands are registered with ADMINISTRATOR access:

**restart** : Master restarts itself (and all its children)

**off** : Master shuts down all its children

**off -master** : Master shuts down all its children and exits

**on** : Master spawns all the daemons it is configured to spawn

This section provides examples of configuration settings. Notice that ADMINISTRATOR access is only granted through a HOSTALLOW setting to explicitly grant access to a small number of machines. We recommend this.

- Let any machine join your pool. Only the central manager has administrative access (this is the default that ships with Condor)

```
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST)
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

- Only allow machines at NCSA to join or view the pool. The central manager is the only machine with ADMINISTRATOR access.

```
HOSTALLOW_READ = *.ncsa.uiuc.edu
HOSTALLOW_WRITE = *.ncsa.uiuc.edu
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST)
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

- Only allow machines at NCSA and the U of I Math department join the pool, EXCEPT do **not** allow lab machines to do so. Also, do not allow the 177.55 subnet (perhaps this is the dial-in subnet). Allow anyone to view pool statistics. The machine named bigcheese administers the pool (not the central manager).

```
HOSTALLOW_WRITE = *.ncsa.uiuc.edu, *.math.uiuc.edu
HOSTDENY_WRITE = lab-*.edu, *.lab.uiuc.edu, 177.55.*
HOSTALLOW_ADMINISTRATOR = bigcheese.ncsa.uiuc.edu
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

- Only allow machines at NCSA and UW-Madison's CS department to view the pool. Only NCSA machines and the machine raven.cs.wisc.edu can join the pool. (Note: the machine raven has the read access it needs through the wild card setting in HOSTALLOW_READ ). This example also shows how to use "\" to continue a long list of machines onto multiple lines, making it more readable (this works for all configuration file entries, not just host access entries)

```
HOSTALLOW_READ = *.ncsa.uiuc.edu, *.cs.wisc.edu
HOSTALLOW_WRITE = *.ncsa.uiuc.edu, raven.cs.wisc.edu
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST), bigcheese.ncsa.uiuc.edu, \
                          biggercheese.uiuc.edu
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

- Allow anyone except the military to view the status of the pool, but only let machines at NCSA view the job queues. Only NCSA machines can join the pool. The central manager, bigcheese, and biggercheese can perform most administrative functions. However, only biggercheese can update user priorities.

```
HOSTDENY_READ = *.mil
HOSTALLOW_READ_SCHEDD = *.ncsa.uiuc.edu
HOSTALLOW_WRITE = *.ncsa.uiuc.edu
HOSTALLOW_ADMINISTRATOR = $(CONDOR_HOST), bigcheese.ncsa.uiuc.edu, \
                          biggercheese.uiuc.edu
HOSTALLOW_ADMINISTRATOR_NEGOTIATOR = biggercheese.uiuc.edu
HOSTALLOW_OWNER = $(FULL_HOSTNAME), $(HOSTALLOW_ADMINISTRATOR)
```

A new security feature introduced in Condor version 6.3.2 enables more fine-grained control over the configuration settings that can be modified remotely with the *condor_config_val* command. The manual page for *condor_config_val* on page 625 details how to use *condor_config_val* to modify configuration settings remotely. Since certain configuration attributes can have a large impact on the functioning of the Condor system and the security of the machines in a Condor pool, it is important to restrict the ability to change attributes remotely.

For each security access level described, the Condor administrator can define which configuration settings a host at that access level is allowed to change. Optionally, the administrator can define separate lists of settable attributes for each Condor daemon, or the administrator can define one list that is used by all daemons.

For each command that requests a change in configuration setting, Condor searches all the different possible security access levels to see which, if any, the request satisfies. (Some hosts can qualify for multiple access levels. For example, any host with ADMINISTRATOR permission probably has WRITE permission also). Within the qualified access level, Condor searches for the list of attributes that may be modified. If the request is covered by the list, the request will be granted. If not covered, the request will be refused.

The default configuration shipped with Condor is exceedingly restrictive. Condor users or administrators cannot set configuration values from remote hosts with *condor_config_val*. Enabling this feature requires a change to the settings in the configuration file. Use this security feature carefully. Grant access only for attributes which you need to be able to modify in this manner, and grant access only at the most restrictive security level possible.

The most secure use of this feature allows Condor users to set attributes in the configuration file which are not used by Condor directly. These are custom attributes published by various Condor daemons with the <SUBSYS>_ATTRS setting described in section 3.3.5 on page 156. It is secure to grant access only to modify attributes that are used by Condor to publish information. Granting access to modify settings used to control the behavior of Condor is not secure. The goal is to ensure no one can use the power to change configuration attributes to compromise the security of your Condor pool.

The control lists are defined by configuration settings that contain SETTABLE_ATTRS in their name. The name of the control lists have the following form:

```
<SUBSYS>_SETTABLE_ATTRS_PERMISSION-LEVEL
```

The two parts of this name that can vary are PERMISSION-LEVEL and the ¡SUBSYS¿. The PERMISSION-LEVEL can be any of the security access levels described earlier in this section. Examples include `WRITE`, `OWNER`, and `CONFIG`.

The ¡SUBSYS¿ is an optional portion of the name. It can be used to define separate rules for which configuration attributes can be set for each kind of Condor daemon (for example, STARTD, SCHEDD, MASTER). There are many configuration settings that can be defined differently for each daemon that use this ¡SUBSYS¿ naming convention. See section 3.3.1 on page 142 for a list. If there is no daemon-specific value for a given daemon, Condor will look for `SETTABLE_ATTRS_PERMISSION-LEVEL` .

Each control list is defined by a comma-separated list of attribute names which should be allowed to be modified. The lists can contain wild cards characters ('*').

Some examples of valid definitions of control lists with explanations:

- `SETTABLE_ATTRS_CONFIG = *`

  Grant unlimited access to modify configuration attributes to any request that came from a machine in the `CONFIG` access level. This was the default behavior before Condor version 6.3.2.

- `SETTABLE_ATTRS_ADMINISTRATOR = *_DEBUG, MAX_*_LOG`

  Grant access to change any configuration setting that ended with "_DEBUG" (for example, `STARTD_DEBUG` ) and any attribute that matched "MAX_*_LOG" (for example, `MAX_SCHEDD_LOG` ) to any host with `ADMINISTRATOR` access.

- `STARTD_SETTABLE_ATTRS_OWNER = HasDataSet`

  Allows any request to modify the `HasDataSet` attribute that came from a host with `OWNER` access. By default, `OWNER` covers any request originating from the local host, plus any machines listed in the `ADMINISTRATOR` level. Therefore, any Condor job would qualify for OWNER access to the machine where it is running. So, this setting would allow any process running on a given host, including a Condor job, to modify the `HasDataSet` variable for that host. `HasDataSet` is not used by Condor, it is an invented attribute included in the `STARTD_ATTRS` setting in order for this example to make sense.

### 3.6.11 Using Condor w/ Firewalls, Private Networks, and NATs

This topic is now addressed in more detail in section 3.7, which explains network communication in Condor.

### 3.6.12 User Accounts in Condor

On a Unix system, UIDs (User IDentification numbers) form part of an operating system's tools for maintaining access control. Each executing program has a UID, a unique identifier of a user executing the program. This is also called the real UID. A common situation has one user executing the

program owned by another user. Many system commands work this way, with a user (corresponding to a person) executing a program belonging to (owned by) `root`. Since the program may require privileges that `root` has which the user does not have, a special bit in the program's protection specification (a setuid bit) allows the program to run with the UID of the program's owner, instead of the user that executes the program. This UID of the program's owner is called an effective UID.

Condor works most smoothly when its daemons run as `root`. The daemons then have the ability to switch their effective UIDs at will. When the daemons run as `root`, they normally leave their effective UID and GID (Group IDentification) to be those of user and group `condor`. This allows access to the log files without changing the ownership of the log files. It also allows access to these files when the user `condor`'s home directory resides on an NFS server. `root` can not normally access NFS files.

If there is no `condor` user and group on the system, an administrator can specify which UID and GID the Condor daemons should use when they do not need root privileges in two ways: either with the `CONDOR_IDS` environment variable or the `CONDOR_IDS` configuration file setting. In either case, the value should be the UID integer, followed by a period, followed by the GID integer. For example, if a Condor administrator does not want to create a `condor` user, and instead wants their Condor daemons to run as the `daemon` user (a common non-root user for system daemons to execute as), the `daemon` user's UID was 2, and group `daemon` had a GID of 2, the corresponding setting in the Condor configuration file would be `CONDOR_IDS = 2.2`.

On a machine where a job is submitted, the *condor_schedd* daemon changes its effective UID to `root` such that it has the capability to start up a *condor_shadow* daemon for the job. Before a *condor_shadow* daemon is created, the *condor_schedd* daemon switches back to `root`, so that it can start up the *condor_shadow* daemon with the (real) UID of the user who submitted the job. Since the *condor_shadow* runs as the owner of the job, all remote system calls are performed under the owner's UID and GID. This ensures that as the job executes, it can access only files that its owner could access if the job were running locally, without Condor.

On the machine where the job executes, the job runs either as the submitting user or as user `nobody`, to help ensure that the job cannot access local resources or do harm. If the `UID_DOMAIN` matches, and the user exists as the same UID in password files on both the submitting machine and on the execute machine, the job will run as the submitting user. If the user does not exist in the execute machine's password file and `SOFT_UID_DOMAIN` is True, then the job will run under the submitting user's UID anyway (as defined in the submitting machine's password file). If `SOFT_UID_DOMAIN` is False, and `UID_DOMAIN` matches, and the user is not in the execute machine's password file, then the job execution attempt will be aborted.

#### Running Condor as Non-Root

While we strongly recommend starting up the Condor daemons as `root`, we understand that it is not always possible to do so. The main problems appear when one Condor installation is shared by many users on a single machine, or if machines are set up to only execute Condor jobs. With a submit-only installation for a single user, there is no need for (or benefit from) running as `root`.

What follows are the effects on the various parts of Condor of running both with and without root access.

***condor_startd*** If you're setting up a machine to run Condor jobs and don't start the *condor_startd* as `root`, you're basically relying on the goodwill of your Condor users to agree to the policy you configure the *condor_startd* to enforce as far as starting, suspending, vacating and killing Condor jobs under certain conditions. If you run as `root`, however, you can enforce these policies regardless of malicious users. By running as `root`, the Condor daemons run with a different UID than the Condor job that gets started (since the user's job is started as either the UID of the user who submitted it, or as user `nobody`, depending on the `UID_DOMAIN` settings). Therefore, the Condor job cannot do anything to the Condor daemons. If you don't start the daemons as `root`, all processes started by Condor, including the end user's job, run with the same UID (since you can't switch UIDs unless you're `root`). Therefore, a user's job could just kill the *condor_startd* and *condor_starter* as soon as it starts up and by doing so, avoid getting suspended or vacated when a user comes back to the machine. This is nice for the user, since they get unlimited access to the machine, but awful for the machine owner or administrator. If you trust the users submitting jobs to Condor, this might not be a concern. To ensure, however, that the policy you choose is effectively enforced by Condor, the *condor_startd* should be started as `root`.

In addition, some system information cannot be obtained without `root` access on some platforms (such as load average on IRIX). As a result, when running without `root` access, the *condor_startd* must call other programs (for example, *uptime*) to get this information. This is much less efficient than getting the information directly from the kernel (which is what we do if we're running as `root`). On Linux and Solaris, we can get this information directly without root access, so this is not a concern on those platforms.

If you cannot have all of Condor running as `root`, at least consider whether you can install the *condor_startd* as setuid root. That would solve both of these problems. If you cannot do that, you could also install it as a setgid sys or kmem program (depending on whatever group has read access to `/dev/kmem` on your system), and that would at least solve the system information problem.

***condor_schedd*** The biggest problem running the *condor_schedd* without root access is that the *condor_shadow* processes which it spawns are stuck with the same UID the *condor_schedd* has. This means that users submitting their jobs must go out of their way to grant write access to user or group `condor` (or whoever the *condor_schedd* is running as) for any files or directories their jobs write or create. Similarly, read access must be granted to their input files.

Consider installing *condor_submit* as a setgid condor program so that at least the `stdout`, `stderr` and `UserLog` files get created with the right permissions. If *condor_submit* is a setgid program, it will automatically set it's umask to 002, and create group-writable files. This way, the simple case of a job that only writes to `stdout` and `stderr` will work. If users have programs that open their own files, they will need to know and set the proper permissions on the directories they submit from.

***condor_master*** The *condor_master* is what spawns the *condor_startd* and *condor_schedd*. To have both running as `root`, have the *condor_master* run as `root`. This happens automatically if you start the master from your boot scripts.

***condor_negotiator* and *condor_collector*** There is no need to have either of these daemons running as `root`.

***condor_kbdd*** On platforms that need the *condor_kbdd* (Digital Unix and IRIX) the *condor_kbdd* must run as `root`. If it is started as any other user, it will not work. You might consider installing this program as a setuid root binary if you cannot run the *condor_master* as `root`. Without the *condor_kbdd*, the startd has no way to monitor mouse activity at all, and the only keyboard activity it will notice is activity on ttys (such as xterms, remote logins, etc).

If you do choose to run Condor as non-root, then you may choose almost any user you like. A common choice is to use the `condor` user; this simplifies the setup because Condor will look for its configuration files in the `condor` user's directory. If you do not select the `condor` user, then you will need to ensure that the configuration is set properly so that Condor can find its configuration files.

If users will be submitting jobs as a user different than the user Condor is running as (perhaps you are running as the `condor` user and users are submitting as themselves), then users have to be careful to only have file permissions properly set up to be accessible by the user Condor is using. In practice, this means creating world-writable directories for output from Condor jobs. This creates a potential security risk, in that any user on the machine where the job is submitted can alter the data, remove it, or do other undesirable things. It is only acceptable in an environment where users can trust other users.

Normally, users without root access who wish to use Condor on their machines create a `condor` home directory somewhere within their own accounts and start up the daemons (to run with the UID of the user). As in the case where the daemons run as user `condor`, there is no ability to switch UIDs or GIDs. The daemons run as the UID and GID of the user who started them. On a machine where jobs are submitted, the *condor_shadow* daemons all run as this same user. But if other users are using Condor on the machine in this environment, the *condor_shadow* daemons for these other users' jobs execute with the UID of the user who started the daemons. This is a security risk, since the Condor job of the other user has access to all the files and directories of the user who started the daemons. Some installations have this level of trust, but others do not. Where this level of trust does not exist, it is best to set up a `condor` account and group, or to have each user start up their own Personal Condor submit installation.

When a machine is an execution site for a Condor job, the Condor job executes with the UID of the user who started the *condor_startd* daemon. This is also potentially a security risk, which is why we do not recommend starting up the execution site daemons as a regular user. Use either `root` or a user (such as the user `condor`) that exists only to run Condor jobs.

#### Running Jobs as the Nobody User

Under Unix, Condor runs jobs either as the user that submitted the jobs, or as the user called `nobody`. Condor uses user `nobody` if the value of the `UID_DOMAIN` configuration variable of the submitting and executing machines are different.

When Condor cleans up after a executing a vanilla universe job, it does the best that it can by deleting all of the processes started by the job. Unfortunately, it is possible to fool Condor, and leave processes behind after Condor has cleaned up. If the job is running as user nobody, it is possible for it to leave a lurker process lying in wait for the next job run as nobody. The lurker process may prey maliciously on the next nobody user job, wreaking havoc.

Condor could prevent this problem by simply killing all processes run by the nobody user, but this would annoy many system administrators. The nobody user is often used for non-Condor system processes.

Condor provides a two-part solution to this difficulty. First, create user accounts specifically for Condor to use instead of user nobody. These can be low-privilege accounts, as the nobody user is. Create one of these accounts for each virtual machine per computer, so that distinct users can be used for concurrent processes. This prevents malicious behavior between processes running on distinct virtual machines. Section 3.12.7 details virtual machines. For a sample machine with two virtual machines, create two users that are intended only to be used by Condor. As an example, call them nobody1 and nobody2. Tell Condor about these users with the VMx_USER configuration variables, where x is replaced with the virtual machine number. In this example:

```
VM1_USER = nobody1
VM2_USER = nobody2
```

Reconfigure Condor, so that Condor will make use of these users instead of the nobody user. One more change is required to prevent lurker processes: tell Condor that these accounts are intended only to be used by Condor, so Condor can kill all the processes belonging to these users upon job completion. The configuration variable EXECUTE_LOGIN_IS_DEDICATED is introduced and set to True for this purpose.

```
EXECUTE_LOGIN_IS_DEDICATED = TRUE
```

Notes:

1. If UID_DOMAIN is not set in the configuration, do not set EXECUTE_LOGIN_IS_DEDICATED. In this case, lurker processes are not a concern, and other processes that a user may have running would be killed improperly.
2. This only applies to vanilla universe and Java universe jobs. Standard universe jobs are not a concern, because they are not allowed to create new processes.
3. On Windows, VMx_USER will only work if the credential of the specified user is stored on the execute machine using *condor_store_cred*. See the *condor_store_cred* manual page (in section 9) for details of this command.

**Working Directories for Jobs**

Every executing process has a notion of its current working directory. This is the directory that acts as the base for all file system access. There are two current working directories for any Condor job: one where the job is submitted and a second where the job executes. When a user submits a job, the submit-side current working directory is the same as for the user when the *condor_submit* command is issued. The **initialdir** submit command may change this, thereby allowing different jobs to have different working directories. This is useful when submitting large numbers of jobs. This submit-side current working directory remains unchanged for the entire life of a job. The submit-side current working directory is also the working directory of the *condor_shadow* daemon. This is particularly relevant for standard universe jobs, since file system access for the job goes through the *condor_shadow* daemon, and therefore all accesses behave as if they were executing without Condor.

There is also an execute-side current working directory. For standard universe jobs, it is set to the `execute` subdirectory of Condor's home directory. This directory is world-writable, since a Condor job usually runs as user `nobody`. Normally, standard universe jobs would never access this directory, since all I/O system calls are passed back to the *condor_shadow* daemon on the submit machine. In the event, however, that a job crashes and creates a core dump file, the execute-side current working directory needs to be accessible by the job so that it can write the core file. The core file is moved back to the submit machine, and the *condor_shadow* daemon is informed. The *condor_shadow* daemon sends e-mail to the job owner announcing the core file, and provides a pointer to where the core file resides in the submit-side current working directory.

## 3.7 Networking

This section on network communication in Condor discusses which network ports are used, how Condor behaves on machines with multiple network interfaces and IP addresses, and how to facilitate functionality in a pool that spans firewalls and private networks.

The security section of the manual contains some information that is relevant to the discussion of network communication which will not be duplicated here, so please see section 3.6 as well.

Firewalls, private networks, and network address translation (NAT) pose special problems for Condor. There are currently two main mechanisms for dealing with firewalls within Condor:

1. Restrict Condor to use a specific range of port numbers, and allow connections through the firewall that use any port within the range.
2. Use *Generic Connection Brokering* (GCB).

Each method has its own advantages and disadvantages, as described below.

### 3.7.1 Port Usage in Condor

#### Default Port Usage

Every Condor daemon listens on a network port for incoming commands. Most daemons listen on a dynamically assigned port. In order to send a message, Condor daemons and tools locate the correct port to use by querying the *condor_collector*, extracting the port number from the ClassAd. One of the attributes included in every daemon's ClassAd is the full IP address and port number upon which the daemon is listening.

To access the *condor_collector* itself, all Condor daemons and tools must know the port number where the *condor_collector* is listening. The *condor_collector* is the only daemon with a well-known, fixed port. By default, Condor uses port 9618 for the *condor_collector* daemon. However, this port number can be changed (see below).

As an optimization for daemons and tools communicating with another daemon that is running on the same host, each Condor daemon can be configured to write its IP address and port number into a well-known file. The file names are controlled using the `<SUBSYS>_ADDRESS_FILE` configuration variables, as described in section 3.3.5 on page 156.

NOTE: In the 6.6 stable series, and Condor versions earlier than 6.7.5, the *condor_negotiator* also listened on a fixed, well-known port (the default was 9614). However, beginning with version 6.7.5, the *condor_negotiator* behaves like all other Condor daemons, and publishes its own ClassAd to the *condor_collector* which includes the dynamically assigned port the *condor_negotiator* is listening on. All Condor tools and daemons that need to communicate with the *condor_negotiator* will either use the `NEGOTIATOR_ADDRESS_FILE` or will query the *condor_collector* for the *condor_negotiator*'s ClassAd.

Sites that configure any checkpoint servers will introduce other fixed ports into their network. Each *condor_cktp_server* will listen to 4 fixed ports: 5651, 5652, 5653, and 5654. There is currently no way to configure alternative values for any of these ports.

#### Using a Non Standard, Fixed Port for the *condor_collector*

By default, Condor uses port 9618 for the *condor_collector* daemon. To use a different port number for this daemon, the configuration variables that tell Condor these communication details are modified. Instead of

```
CONDOR_HOST = machX.cs.wisc.edu
COLLECTOR_HOST = $(CONDOR_HOST)
```

the configuration might be

```
CONDOR_HOST = machX.cs.wisc.edu
COLLECTOR_HOST = $(CONDOR_HOST):9650
```

If a non standard port is defined, the same value of COLLECTOR_HOST (including the port) must be used for all machines in the Condor pool. Therefore, this setting should be modified in the global configuration file (condor_config file), or the value must be duplicated across all configuration files in the pool if a single configuration file is not being shared.

When querying the *condor_collector* for a remote pool that is running on a non standard port, any Condor tool that accepts the **-pool** argument can optionally be given a port number. For example:

```
% condor_status -pool foo.bar.org:1234
```

#### Using a Dynamically Assigned Port for the *condor_collector*

On single machine pools, it is permitted to configure the *condor_collector* daemon to use a dynamically assigned port, as given out by the operating system. This prevents port conflicts with other services on the same machine. However, a dynamically assigned port is only to be used on single machine Condor pools, and only if the COLLECTOR_ADDRESS_FILE configuration variable has also been defined. This mechanism allows all of the Condor daemons and tools running on the same machine to find the port upon which the *condor_collector* daemon is listening, even when this port is not defined in the configuration file and is not known in advance.

To enable the *condor_collector* daemon to use a dynamically assigned port, the port number is set to 0 in the COLLECTOR_HOST variable. The COLLECTOR_ADDRESS_FILE configuration variable must also be defined, as it provides a known file where the IP address and port information will be stored. All Condor clients know to look at the information stored in this file. For example:

```
COLLECTOR_HOST = $(CONDOR_HOST):0
COLLECTOR_ADDRESS_FILE = $(LOG)/.collector_address
```

NOTE: Using a port of 0 for the *condor_collector* and specifying a COLLECTOR_ADDRESS_FILE only works in Condor version 6.6.8 or later in the 6.6 stable series, and in version 6.7.4 or later in the 6.7 development series. Do not attempt to do this with older versions of Condor.

Configuration definition of COLLECTOR_ADDRESS_FILE is in section 3.3.5 on page 156, and COLLECTOR_HOST is in section 3.3.3 on page 145.

#### Restricting Port Usage to Operate with Firewalls

If a Condor pool is completely behind a firewall, then no special consideration or port usage is needed. However, if there is a firewall between the machines within a Condor pool, then configuration variables may be set to force the usage of specific ports, and to utilize a specific range of ports.

By default, Condor uses port 9618 for the *condor_collector* daemon, and dynamic (apparently random) ports for everything else. See section 3.7.1, if a dynamically assigned port is desired for the *condor_collector* daemon.

The configuration variables HIGHPORT and LOWPORT facilitate setting a restricted range of ports that Condor will use. This may be useful when some machines are behind a firewall. The configuration macros HIGHPORT and LOWPORT will restrict dynamic ports to the range specified. The configuration variables are fully defined in section 3.3.3. All of these ports must be greater than 0 and less than 65,536. Note that both HIGHPORT and LOWPORT must be at least 1024 for Condor version 6.6.8. In general, use ports greater than 1024, in order to avoid port conflicts with standard services on the machine. Another reason for using ports greater than 1024 is that daemons and tools are often not run as root, and only root may listen to a port lower than 1024. Also, the range must include enough ports that are not in use, or Condor cannot work.

The range of ports assigned may be restricted based on incoming (listening) and outgoing (connect) ports with the configuration variables IN_HIGHPORT , IN_LOWPORT , OUT_HIGHPORT , and OUT_LOWPORT . See section 3.3.6 for complete definitions of these configuration variables. A range of ports lower than 1024 for daemons running as root is appropriate for incoming ports, but not for outgoing ports. The use of ports below 1024 (versus above 1024) has security implications; therefore, it is inappropriate to assign a range that crosses the 1024 boundary.

NOTE: Setting HIGHPORT and LOWPORT will not automatically force the *condor_collector* to bind to a port within the range. The only way to control what port the *condor_collector* uses is by setting the COLLECTOR_HOST (as described above).

The total number of ports needed depends on the size of the pool, the usage of the machines within the pool (which machines run which daemons), and the number of jobs that may execute at one time. Here we discuss how many ports are used by each participant in the system.

The central manager of the pool needs 5 + NEGOTIATOR_SOCKET_CACHE_SIZE ports for daemon communication, where NEGOTIATOR_SOCKET_CACHE_SIZE is specified in the configuration or defaults to the value 16.

Each execute machine (those machines running a *condor_startd* daemon) requires 5 + (5 * number of virtual machines advertised by that machine) ports. By default, the number of virtual machines advertised will equal the number of physical CPUs in that machine.

Submit machines (those machines running a *condor_schedd* daemon) require 5 + (5 * MAX_JOBS_RUNNING) ports. The configuration variable MAX_JOBS_RUNNING limits (on a per-machine basis, if desired) the maximum number of jobs. Without this configuration macro, the maximum number of jobs that could be simultaneously executing at one time is a function of the number of reachable execute machines.

Also be aware that HIGHPORT and LOWPORT only impact dynamic port selection used by the Condor system, and they do not impact port selection used by jobs submitted to Condor. Thus, jobs submitted to Condor that may create network connections may not work in a port restricted environment. For this reason, specifying HIGHPORT and LOWPORT is not going to produce the expected results if a user submits jobs to be executed under the PVM or MPI job universes.

Where desired, a local configuration for machines *not* behind a firewall can override the usage of HIGHPORT and LOWPORT, such that the ports used for these machines are not restricted. This can be accomplished by adding the following to the local configuration file of those machines *not*

behind a firewall:

```
HIGHPORT = UNDEFINED
LOWPORT  = UNDEFINED
```

If the maximum number of ports allocated using HIGHPORT and LOWPORT is too few, socket binding errors of the form

```
failed to bind any port within <$LOWPORT> - <$HIGHPORT>
```

are likely to appear repeatedly in log files.

#### Multiple Collectors

This section has not yet been written

#### Port Conflicts

This section has not yet been written

### 3.7.2 Configuring Condor for Machines With Multiple Network Interfaces

Beginning with Condor version 6.1.5, Condor can run on machines with multiple network interfaces. However, starting with Condor version 6.7.13, new functionality is available that allows even better support for multi-homed machines, using the configuration variable BIND_ALL_INTERFACES. A multi-homed machine is one that has more than one NIC (Network Interface Card). Further improvements to this new functionality will remove the need for any special configuration in the common case. For now, care must still be given to machines with multiple NICs, even when using this new configuration variable.

#### Using BIND_ALL_INTERFACES

Starting with Condor version 6.7.13, machines can be configured such that whenever Condor daemons or tools call bind(), the daemons or tools use all network interfaces on the machine. This means that outbound connections will always use the appropriate network interface to connect to a remote host, instead of being forced to use an interface that might not have a route to the given destination. Furthermore, sockets upon which a daemon listens for incoming connections will be bound to all network interfaces on the machine. This means that so long as remote clients know the right port, they can use any IP address on the machine and still contact a given Condor daemon.

To enable this functionality, the boolean configuration variable BIND_ALL_INTERFACES is defined and set to True:

```
BIND_ALL_INTERFACES = TRUE
```

This functionality has limitations, and therefore has a default value of `False`. Here are descriptions of the limitations.

**Using all network interfaces does not work with Kerberos.** Every Kerberos ticket contains a specific IP address within it. Authentication over a socket (using Kerberos) requires the socket to also specify that same specific IP address. Use of `BIND_ALL_INTERFACES` causes outbound connections from a multi-homed machine to originate over any of the interfaces. Therefore, the IP address of the outbound connection and the IP address in the Kerberos ticket will not necessarily match, causing the authentication to fail. Sites using Kerberos authentication on multi-homed machines are strongly encouraged not to enable `BIND_ALL_INTERFACES`, at least until Condor's Kerberos functionality supports using multiple Kerberos tickets together with finding the right one to match the IP address a given socket is bound to.

**There is a potential security risk.** Consider the following example of a security risk. A multi-homed machine is at a network boundary. One interface is on the public Internet, while the other connects to a private network. Both the multi-homed machine and the private network machines comprise a Condor pool. If the multi-homed machine enables `BIND_ALL_INTERFACES`, then it is at risk from hackers trying to compromise the security of the pool. Should this multi-homed machine be compromised, the entire pool is vulnerable. Most sites in this situation would run an *sshd* on the multi-homed machine so that remote users who wanted to access the pool could log in securely and use the Condor tools directly. In this case, remote clients do not need to use Condor tools running on machines in the public network to access the Condor daemons on the multi-homed machine. Therefore, there is no reason to have Condor daemons listening on ports on the public Internet, causing a potential security threat.

**Only one IP address will be advertised.** At present, even though a given Condor daemon will be listening to ports on multiple interfaces, each with their own IP address, there is currently no mechanism for that daemon to advertise all of the possible IP addresses where it can be contacted. Therefore, Condor clients (other Condor daemons or tools) will not necessarily able to locate and communicate with a given daemon running on a multi-homed machine where `BIND_ALL_INTERFACES` has been enabled.

Currently, Condor daemons can only advertise a single IP address in the ClassAd they send to their *condor_collector*. Condor tools and other daemons only know how to look up a single IP address, and they attempt to use that single IP address when connecting to the daemon. So, even if the daemon is listening on 2 or more different interfaces, each with a separate IP, the daemon must choose what IP address to publicly advertise so that other daemons and tools can locate it.

By default, Condor advertises the IP address of the network interface used to contact the collector, since this is the most likely to be accessible to other processes that query the same collector. The `NETWORK_INTERFACE` setting can still be used to specify the IP address Condor should advertise, even if `BIND_ALL_INTERFACES` is set to `True`. Therefore, some of the considerations described below regarding what interface should be used in various situations still apply when deciding what interface is to be advertised.

Sites that make heavy use of private networks and multi-homed machines should consider if using Generic Connection Brokering, GCB, is right for them. More information about GCB and Condor can be found in section 3.7.3 on page 301.

### Central Manager with Two or More NICs

Often users of Condor wish to set up "compute farms" where there is one machine with two network interface cards (one for the public Internet, and one for the private net). It is convenient to set up the "head" node as a central manager in most cases and so here are the instructions required to do so.

Setting up the central manager on a machine with more than one NIC can be a little confusing because there are a few external variables that could make the process difficult. One of the biggest mistakes in getting this to work is that either one of the separate interfaces is not active, or the host/domain names associated with the interfaces are incorrectly configured.

Given that the interfaces are up and functioning, and they have good host/domain names associated with them here is how to configure Condor:

In this example, **farm-server.farm.org** maps to the private interface.

On the central manager's global (to the cluster) configuration file:
`CONDOR_HOST` = **farm-server.farm.org**

On your central manager's local configuration file:
`NETWORK_INTERFACE` = ip address of **farm-server.farm.org**
`NEGOTIATOR` = `$(SBIN)`/condor_negotiator
`COLLECTOR` = `$(SBIN)`/condor_collector
`DAEMON_LIST` = `MASTER`, `COLLECTOR`, `NEGOTIATOR`, `SCHEDD`, `STARTD`

If your central manager and farm machines are all NT, then you only have vanilla universe and it will work now. However, if you have this setup for UNIX, then at this point, standard universe jobs should be able to function in the pool, but if you did not configure the `UID_DOMAIN` macro to be homogeneous across the farm machines, the standard universe jobs will run as **nobody** on the farm machines.

In order to get vanilla jobs and file server load balancing for standard universe jobs working (under Unix), do some more work both in the cluster you have put together and in Condor to make everything work. First, you need a file server (which could also be the central manager) to serve files to all of the farm machines. This could be NFS or AFS, it does not really matter to Condor. The mount point of the directories you wish your users to use must be the same across all of the farm machines. Now, configure `UID_DOMAIN` and `FILESYSTEM_DOMAIN` to be homogeneous across the farm machines and the central manager. Now, you will have to inform Condor that an NFS or AFS filesystem exists and that is done in this manner. In the global (to the farm) configuration file:

```
# If you have NFS
USE_NFS = True
# If you have AFS
```

```
HAS_AFS = True
USE_AFS = True
# if you want both NFS and AFS, then enable both sets above
```

Now, if you've set up your cluster so that it is possible for a machine name to never have a domain name (for example: there is machine name but no fully qualified domain name in `/etc/hosts`), you must configure `DEFAULT_DOMAIN_NAME` to be the domain that you wish to be added on to the end of your host name.

#### A Client Machine with Multiple Interfaces

If you have a client machine with two or more NICs, then there might be a specific network interface with which you desire a client machine to communicate with the rest of the Condor pool. In this case, in the local configuration file for that machine, place:
`NETWORK_INTERFACE` = ip address of interface desired

#### A Checkpoint Server on a Machine with Multiple NICs

If your Checkpoint Server is on a machine with multiple interfaces, the only way to get things to work is if your different interfaces have different host names associated with them, and you set `CKPT_SERVER_HOST` to the host name that corresponds with the IP address you want to use in the global configuration file for your pool. You will still need to specify `NETWORK_INTERFACE` in the local config file for your Checkpoint Server.

### 3.7.3 Generic Connection Brokering (GCB)

Generic Connection Brokering, or GCB, is a system for managing network connections across private network and firewall boundaries. Starting with Condor version 6.7.13, Condor's Linux releases are linked with GCB, and can use GCB functionality to run jobs (either directly or via flocking) on pools that span public and private networks.

While GCB provides numerous advantages over restricting Condor to use a range of ports which are then opened on the firewall (see section 3.7.1 on page 296), GCB is also a very complicated system, with major implications for Condor's networking and security functionality. Therefore, sites must carefully weigh the advantages and disadvantages of attempting to configure and use GCB before making a decision.

Advantages:

- Better connectivity. GCB works with pools that have multiple private networks (even multiple private networks that use the same IP addresses (for example, 192.168.2.*). GCB also works with sites that use network address translation (NAT).

- More secure. Administrators never need to allow inbound connections through the firewall. With GCB, only outbound connections from behind the firewall must be allowed (which is a standard firewall configuration). It is possible to trade decreased performance for better security, and configure the firewall to only allow outbound connections to a single public IP address.
- Does not require `root` access to any machines. All parts of a GCB system can be run as an unprivileged user, and in the common case, no changes to the firewall configuration are required.

Disadvantages:

- The GCB broker (section 3.7.3 describes the broker) node(s) is a potential failure point to the pool. Any private nodes that want to communicate outside their own network must be represented by a GCB broker. This machine must be highly reliable, since if the broker is ever down, all inbound communication with the private nodes is impossible. Furthermore, no other Condor services should be run on a GCB broker (for example, the Condor pool's central manager). While it is possible to do so, it is not recommended. In general, no other services should be run on the machine at all, and the host should be dedicated to the task of serving as a GCB broker.
- All Condor nodes behind a given firewall share a single IP address (the public IP address of their GCB broker). All Condor daemons using a GCB broker will advertise themselves with this single IP address, and in some cases, connections to/from those daemons will actually originate at the broker. This has implications for Condor's host/IP based security, and the general level of confusion for users and administrators of the pool. Debugging problems will be more difficult, as any log messages which only print the IP address (not the name and/or port) will become ambiguous. Even log or error messages that include the port will not necessarily be helpful, as it is difficult to correlate ports on the broker with the corresponding private nodes.
- Can not function with Kerberos authentication. Kerberos tickets include the IP address of the machine where they were created. However, when Condor daemons are using GCB, they use a different IP address, and therefore, any attempt to authenticate using Kerberos will fail, as Kerberos will consider this a (poor) attempt to fool it into using an invalid host principle.
- Scalability and performance degradation:
  - Connections are more expensive to establish.
  - In some cases, connections must be forwarded through a proxy server on the GCB broker.
  - Each network port on each private node must correspond to a unique port on the broker host, so there is a fixed limit to how many private nodes a given broker can service (which is a function of the number of ports each private node requires and the total number of available ports on the broker).

  - Each private node must maintain an open TCP connection to its GCB broker. GCB will attempt to recover in the case of the socket being closed, but this means the broker must have at least as many sockets open as there are private nodes.

- It is more complex to configure and debug.

Given the increased complexity, use of GCB requires a careful read of this entire manual section, followed by a thorough installation.

Details of GCB and how it works can be found at the GCB homepage:

http://www.cs.wisc.edu/condor/gcb

This information is useful for understanding the technical details of how GCB works, and the various parts of the system. While some of the information is partly out of date (especially the discussion of how to configure GCB) most of the sections are perfectly accurate and worth reading. Ignore the section on "GCBnize", which describes how to get a given application to use GCB, as the Linux port of all Condor daemons and tools have already been converted to use GCB.

The rest of this section gives the details for configuring a Condor pool to use GCB. It is divided into the following topics:

- Introduction to the GCB broker
- Configuring the GCB broker
- Spawning a GCB broker (with a *condor_master* or using *initd*)
- How to configure Condor machines to use GCB
- Configuring the GCB routing table
- Implications for Condor's host/IP security settings
- Implications for other Condor configuration settings

**Introduction to the GCB Broker**

At the heart of GCB is a logical entity known as a *broker* or *inagent*. In reality, the entity is made up of daemon processes running on the same machine comprised of the *gcb_broker* and a set of *gcb_relay_server* processes, each one spawned by the *gcb_broker*.

Every private network using GCB must have at least one broker to arrange connections. The broker must be installed on a machine that nodes in both the public and the private (firewalled) network can directly talk to. The broker need not be able to initiate connections to the private nodes. It can take advantage of the case where it can initiate connections to the private nodes, and that will improve performance. The broker is generally installed on a machine with multiple network interfaces (on the network boundary) or just outside of a network that allows outbound connections.

If the private network contains many hosts, sites can configure multiple GCB brokers, and partition the private nodes so that different subsets of the nodes use different brokers.

For a more thorough explanation of what a GCB broker is, check out: http://www.cs.wisc.edu/~sschang/firewall/gcb/mechanism.htm

A GCB broker should generally be installed on a dedicated machine. These are machines that are not running other Condor daemons or services. If running any other Condor service (for example, the central manager of the pool) on the same machine as the GCB broker, all other machines attempting to use this Condor service (for example, to connect to the *condor_collector* or *condor_negotiator*) will incur additional connection costs and latency. It is possible that future versions of GCB and Condor will be able to overcome these limitations, but for now, we recommend that a broker is run on a dedicated machine with no other Condor daemons (except perhaps a single *condor_master* used to spawn the *gcb_broker* daemon, as described below).

In principle, a GCB broker is a network element that functions almost like a router. It allows certain connections through the firewall by redirecting connections or forwarding connections. In general, it is not a good idea to run a lot of other services on the network elements, especially not services like Condor which can spawn arbitrary jobs. Furthermore, the GCB broker relies on listening to many network ports. If other applications are running on the same host as the broker, problems exist where the broker does not have enough network ports available to forward all the connections that might be required of it. Also, all nodes inside a private network rely on the GCB broker for all incoming communication. For performance reasons, avoid forcing the GCB broker to contend with other processes for system resources, such that it is always available to handle communication requests. There is nothing in GCB or Condor requiring the broker to run on a separate machine, but it is the recommended configuration.

The *gcb_broker* daemon listens on two hard-coded, fixed ports (65432 and 65430). A future version of Condor and GCB will remove this limitation. However, for now, to run a *gcb_broker* on a given host, ensure that ports 65432 and 65430 are not already in use.

If `root` access on a machine where a GCB broker is planned, one good option is to have *initd* configured to spawn (and re-spawn) the *gcb_broker* binary (which is located in the `<release_dir>/libexec` directory). This way, the *gcb_broker* will be automatically restarted on reboots, or in the event that the broker itself crashes or is killed. Without `root` access, use a *condor_master* to manage the *gcb_broker* binary.

**Configuring the GCB broker**

Since the *gcb_broker* and *gcb_relay_server* are not Condor daemons, they do not read the Condor configuration files. Therefore, they must be configured by other means, namely the environment and through the use of command-line arguments.

There is one required command-line argument for the *gcb_broker*. This argument defines the public IP address this broker will use to represent itself and any private network nodes that are configured to use this broker. This information is defined with **-i xxx.xxx.xxx.xxx** on the command-line when the *gcb_broker* is executed. If the broker is being setup outside the private network, it is

likely that the machine will only have one IP address, which is clearly the one to use. However, if the broker is being run on a machine on the network boundary (a multi-homed machine with interfaces into both the private and public networks), be sure to use the IP address of the interface on the public network.

Additionally, specify environment variables to control how the *gcb_broker* (and the *gcb_relay_server* processes it spawns) will behave. Some of these settings can also be specified as command-line arguments to the *gcb_broker*. All of them have reasonable defaults if not defined.

- General daemon behavior

  The environment variable `GCB_RELAY_SERVER` defines the full path to the `gcb_relay_server` binary the broker should use. The command-line override for this is **-r /full/path/to/relayserver**. If not set either on the command-line or in the environment, the *gcb_broker* process will search for a program named `gcb_relay_server` in the same directory where the `gcb_broker` binary is located, and attempt to use that one.

  The environment variable `GCB_ACTIVE_TO_CLIENT` is a boolean that defines whether the GCB broker can directly talk to servers running inside the network that it manages The value must be `yes` or `no`, case sensitive. `GCB_ACTIVE_TO_CLIENT` should be set to `yes` only if this GCB broker is running on a network boundary and can connect to both the private and public nodes. If the broker is running in the public network, it should be left undefined or set to `no`.

- Log file locations

  The environment variable `GCB_LOG_DIR` defines a directory to use for all GCB-related log files. If defined, and the per-daemon log file settings (described below) are not defined, the broker will write to `$GCB_LOG_DIR/BrokerLog` and the relay server will write to `$GCB_LOG_DIR/RelayServerLog.<pid>`

  The environment variable `GCB_BROKER_LOG` defines the full path for the GCB broker's log file. The command-line override is **-l /full/path/to/log/file**. This definition overrides `GCB_LOG_DIR`.

  The environment variable `GCB_RELAY_SERVER_LOG` defines the full path to the GCB relay server's log file. Each relay server writes its own log file, so the actual filename will be: `$GCB_RELAY_SERVER_LOG.<pid>` where `<pid>` is replaced with the process id of the corresponding *gcb_relay_server*. When defined, this setting overrides `GCB_LOG_DIR`.

- Verbose logging

  The environment variable `GCB_DEBUG_LEVEL` controls how verbose all the GCB daemon's log files should be. Can be either `fulldebug` (more verbose) or `basic`. This defines logging behavior for all GCB daemons, unless the following daemon-specific settings are defined.

The environment variable GCB_BROKER_DEBUG controls verbose logging specifically for the GCB broker. The command-line override for this is **-d level**. Overrides GCB_DEBUG_LEVEL.

The environment variable GCB_RELAY_SERVER_DEBUG controls verbose logging specifically for the GCB relay server. Overrides GCB_DEBUG_LEVEL.

- Maximum log file size

  The environment variable GCB_MAX_LOG defines the maximum size in bytes of all GCB log files. When the log file reaches this size, the content of the file will be moved to `filename.old`, and a new log is started. This defines logging behavior for all GCB daemons, unless the following daemon-specific settings are used.

  The environment variable GCB_BROKER_MAX_LOG defines the maximum size in bytes of the GCB broker log file.

  The environment variable GCB_RELAY_SERVER_MAX_LOG defines the maximum size in bytes of the GCB relay server log file.

**Spawning the GCB Broker**

There are two ways to spawn the GCB broker:

- Use a *condor_master*.

  To spawn the GCB broker with a *condor_master*, here are the recommended `condor_config` settings that will work:

```
# Specify that you only want the master and the broker running
DAEMON_LIST = MASTER, GCB_BROKER

# Define the path to the broker binary for the master to spawn
GCB_BROKER = $(RELEASE_DIR)/libexec/gcb_broker

# Define the path to the release_server binary for the broker to use
GCB_RELAY = $(RELEASE_DIR)/libexec/gcb_relay_server

# Setup the gcb_broker's environment.  We use a macro to build up the
# environment we want in pieces, and then finally define
# GCB_BROKER_ENVIRONMENT, the setting that condor_master uses.

# Initialize an empty macro
GCB_BROKER_ENV =

# (recommended) Provide the full path to the gcb_relay_server
GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_RELAY_SERVER=$(GCB_RELAY)

# (recommended) Tell GCB to write all log files into the Condor log
# directory (the directory used by the condor_master itself)
GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_LOG_DIR=$(LOG)
# Or, you can specify a log file separately for each GCB daemon:
```

```
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_BROKER_LOG=$(LOG)/GCB_Broker_Log
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_RELAY_SERVER_LOG=$(LOG)/GCB_RS_Log

# (optional -- only set if true) Tell the GCB broker that it can
# directly connect to machines in the private network which it is
# handling communication for.  This should only be enabled if the GCB
# broker is running directly on a network boundary and can open direct
# connections to the private nodes.
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_ACTIVE_TO_CLIENT=yes

# (optional) turn on verbose logging for all of GCB
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_DEBUG_LEVEL=fulldebug
# Or, you can turn this on separately for each GCB daemon:
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_BROKER_DEBUG=fulldebug
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_RELAY_SERVER_DEBUG=fulldebug

# (optional) specify the maximum log file size (in bytes)
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_MAX_LOG=640000
# Or, you can define this separately for each GCB daemon:
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_BROKER_MAX_LOG=640000
#GCB_BROKER_ENV = $(GCB_BROKER_ENV);GCB_RELAY_SERVER_MAX_LOG=640000

# Finally, set the value the condor_master really uses
GCB_BROKER_ENVIRONMENT = $(GCB_BROKER_ENV)

# If your Condor installation on this host already has a public
# interface as the default (either because it is the first interface
# listed in this machine's host entry, or because you've already
# defined NETWORK_INTERFACE), you can just use Condor's special macro
# that holds the IP address for this.
GCB_BROKER_IP = $(ip_address)
# Otherwise, you could define it yourself with your real public IP:
# GCB_BROKER_IP = 123.123.123.123

# (required) define the command-line arguments for the broker
GCB_BROKER_ARGS = -i $(GCB_BROKER_IP)
```

Once those settings are in place, either spawn or restart the *condor_master* and the *gcb_broker* should be started. Ensure the broker is running by reading the log file specified with `GCB_BROKER_LOG`, or in `$(LOG)/BrokerLog` if using the default.

- Use *initd*.

  The system's *initd* may be used to manage the *gcb_broker* without running the *condor_master* on the broker node, but this requires `root` access. Generally, this involves adding a line to the `/etc/inittab` file. Some sites use other means to manage and generate the `/etc/inittab`, such as *cfengine* or other system configuration management tools, so check with the local system administrator to be sure. An example line might be something like:

  ```
  GB:23:respawn:/path/to/gcb_broker -i 123.123.123.123 -r /path/to/relay_server
  ```

  It may be easier to wrap the *gcb_broker* binary in a shell script, in order to change the command-line arguments (and set environment variables) without having to edit `/etc/inittab` all the time. This will be similar to:

```
GB:23:respawn:/opt/condor-6.7.13/libexec/gcb_broker.sh
```

Then, create the wrapper, as similar to:

```
#!/bin/sh

libexec=/opt/condor-6.7.13/libexec
ip=123.123.123.123
relay=$libexec/gcb_relay_server

exec $libexec/gcb_broker -i $ip -r $relay
```

You will probably also want to set some environment variables to tell the GCB daemons where to write their log files (GCB_LOG_DIR), and possibly some of the other variables described above.

Either way, after updating the /etc/inittab, send the *initd* process (always PID 1) a SIGHUP signal, and it will re-read the inittab and spawn the *gcb_broker*.

**Configuring Condor nodes to be GCB clients**

In general, before configuring a node in a Condor pool to use GCB, the GCB broker node(s) for the pool must be set up and running. Set up, configure, and spawn the broker first.

To enable the use of GCB on a given Condor host, set the following Condor configuration variables:

```
# Tell Condor to use a network remapping service (currently only GCB
# is supported, but in the future, there might be other options)
NET_REMAP_ENABLE = true
NET_REMAP_SERVICE = GCB
```

Only GCB clients within a private network need to define the following variable, which specifies the IP address of the broker serving this network. Note that this IP address must be the same as the IP address that was specified on the broker's command-line with the **-i** option.

```
# Public IP address (in standard dot notation) of the GCB broker
# serving this private node.
NET_REMAP_INAGENT = xxx.xxx.xxx.xxx
```

Obviously, because the NET_REMAP_INAGENT setting is only valid on private nodes, it should not be defined in a global Condor configuration file (condor_config). Furthermore, with a large number of hosts in a given private network, if multiple brokers are to be run to alleviate scalability issues, each subset of the private nodes that uses a specific broker will need a different value for this variable.

Finally, if setting up the recommended (but optional) GCB routing table, tell Condor daemons where to find their table. Define the following variable:

```
# The full path to the routing table used by GCB
NET_REMAP_ROUTE = /full/path/to/GCB-routing-table
```

Setting NET_REMAP_ENABLE causes the BIND_ALL_INTERFACES variable to be automatically set. More information about this setting can be found in section 3.7.2 on page 298. It would not hurt to place the following in the configuration file near the other GCB-related settings, just to remember it:

```
# Tell Condor to bind to all network interfaces, instead of a single
# interface.
BIND_ALL_INTERFACES = true
```

Once a GCB broker is set up and running to manage connections for a each private network, and the Condor installation for all the nodes in either private and public networks are configured to enable GCB, restart the Condor daemons, and all of the different machines should be able to communicate with each other.

**Configuring the GCB routing table**

By default, a GCB-enabled application will always attempt to directly connect to a given IP/port pair. In the case of a private nodes being represented by a GCB broker, the IP/port will be a proxy socket on the broker node, not the real address at each private node. When the GCB broker receives a direct connection to one of its proxy sockets, it notifies the corresponding private node, which establishes a new connection to the broker. The broker then forwards packets between these two sockets, establishing a communication pathway into the private node. This allows clients which are not linked with the GCB libraries to communicate with private nodes using a GCB broker.

This mechanism is expensive in terms of latency (time between messages) and total bandwidth (how much data can be moved in a given time period), as well as expensive in terms of the broker's system resources such as network I/O, processor time, and memory. This expensive mechanism is unnecessary in the case of GCB-aware clients trying to connect to private nodes that can directly communicate with the public host. The alternative is to contact the GCB broker's command interface (the fixed port where the broker is listening for GCB management commands), and use a GCB-specific protocol to request a connection to the given IP/port. In this case, the GCB broker will notify the private node to directly connect to the public client (technically, to a new socket created by the GCB client library linked in with the client's application), and a direct socket between the two is established, removing the need for packet forwarding between the proxy sockets at the GCB broker.

On the other hand, in cases where a direct connection from the client to a given server is possible (for example, two GCB-aware clients in the same public network attempting to communicate with each other), it is expensive and unnecessary to attempt to contact a GCB broker, and the client should connect directly.

To allow a GCB-enabled client to know if it should make a direct connection (which might involve packet forwarding through proxy sockets), or if it should use the GCB protocol to commu-

nicate with the broker's command port and arrange a direct socket, GCB provides a *routing table*. Using this table, an administrator can define what IP addresses should be considered private nodes where the GCB connection protocol will be used, and what nodes are public, where a direct connection (without incurring the latency of contacting the GCB broker, only to find out there is no information about the given IP/port) should be made immediately.

If the attempt to contact the GCB broker for a given IP/port fails, or if the desired port is not being managed by the broker, the GCB client library making the connection will fall back and attempt a direct connection. Therefore, configuring a GCB routing table is not required for communication to work within a GCB-enabled environment. However, the GCB routing table can significantly improve performance for communication with private nodes being represented by a GCB broker.

One confusing aspect of GCB is that all of the nodes on a private network believe that their own IP address is the address of their GCB broker. Due to this, all the Condor daemons on a private network advertise themselves with the same IP address (though the broker will map the different ports to different nodes within the private network). Therefore, a given node in the public network needs to be told that if it is contacting this IP address, it should know that the IP address is really a GCB broker representing a node in the private network, so that the public network node can contact the broker to arrange a single socket from the private node to the public one, instead of relying on forwarding packets between proxy sockets at the broker. Any other addresses, such as other public IP addresses, can be contacted directly, without going through a GCB broker. Similarly, other nodes within the same private network will still be advertising their address with their GCB broker's public IP address. So, nodes within the same private network also have to know that the public IP address of the broker is really a GCB broker, yet all other public IP addresses are valid for direct communication.

In general, all connections can be made directly, except to a host represented by a GCB broker. Furthermore, the default behavior of the GCB client library is to make a direct connection. The routing table is a (somewhat complicated) way to tell a given GCB installation what GCB brokers it might have to communicate with, and that it should directly communicate with anything else. In practice, the routing table should have a single entry for each GCB broker in the system. Future versions of GCB will be able to make use of more complicated routing behavior, which is why the full routing table infrastructure described below is implemented, even if the current version of GCB is not taking advantage of all of it.

**Format of the GCB routing table**

The routing table is a plain ASCII text file. Each line of the file contains one rule. Each rule consists of a *target* and a *method*. The target specifies destination IP address(es) to match, and the method defines what mechanism must be used to connect to the given target. The target must be a valid IP address string in the standard dotted notation, followed by a slash character (`/`), as well as an integer *mask*. The mask specifies how many bits of the destination IP address and target IP address must match. The method must be one of the strings

```
GCB
direct
```

GCB stops searching the table as soon as it finds a matching rule, therefore place more specific rules (rules with a larger value for the mask and without wildcards) before generic rules (rules with wildcards or smaller mask values). The default when no rule is matched is to use direct communication. Some examples and the corresponding routing tables may help clarify this syntax.

**Simple GCB routing table example (1 private, 1 public)**

Consider an example with a private network that has a set of nodes whose IP addresses are `192.168.2.*`. Other nodes are in a public network whose IP addresses are `123.123.123.*`. A GCB broker for the 192 network is running on IP address `123.123.123.123`. In this case, the routing table for both the public and private nodes should be:

```
123.123.123.123/32 GCB
```

This rule states that for IP addresses where all 32 bits exactly match the address `123.123.123.123`, first communicate with the GCB broker.

Since the default is to directly connect when no rule in the routing table matches a given target IP, this single rule is all that is required. However, to illustrate how the routing table syntax works, the following routing table is equivalent:

```
123.123.123.123/32 GCB
*/0 direct
```

Any attempt to connect to `123.123.123.123` uses GCB, as it is the first rule in the file. All other IP addresses will connect directly. This table explicitly defines GCB's default behavior.

**More complex GCB routing table example (2 private, 1 public)**

As a more complicated case, consider a single Condor pool that spans one public network and two private networks. The two separate private networks each have machines with private addresses like `192.168.2.*`. Identify one of these private networks as `A`, and the other one as `B`. The public network has nodes with IP addresses like `123.123.123.*`. Assume that the GCB broker for nodes in the `A` network has IP address `123.123.123.65`, and the GCB broker for the nodes in the `B` network has IP address `123.123.123.66`. All of the nodes need to be able to talk to each other. In this case, nodes in private network `A` advertise themselves as `123.123.123.65`, so any node, regardless of being in A, B, or the public network, must treat that IP address as a GCB broker. Similarly, nodes in private network `B` advertise themselves as `123.123.123.66`, so any node, regardless of being in A, B, or the public network, must treat that IP address as a GCB broker. All other connections from any node can be made directly. Therefore, here is the appropriate routing table for all nodes:

```
123.123.123.65/32 GCB
123.123.123.66/32 GCB
```

**Implications of GCB on Condor's Host/IP-based Security Configuration**

When a message is received at a Condor daemon's command socket, Condor authenticates based on the IP address of the incoming socket. For more information about this host-based security in Condor, see section 3.6.10 on page 282. Because of the way GCB changes the IP addresses that are used and advertised by GCB-enabled clients, and since all nodes being represented by a GCB broker are represented by different ports on the broker node (a process known as *address leasing*), using GCB has implications for this process.

Depending on the communication pathway used by a GCB-enabled Condor client (either a tool or another Condor daemon) to connect to a given Condor server daemon, and where in the network each side of the connection resides, the IP address of the resulting socket actually used will be very different. In the case of a private client (that is, a client behind a firewall, which may or may not be using NAT and a fully private, non-routable IP address) attempting to connect to a server, there are three possibilities:

- For a direct connection to another node within the private network, the server will see the private IP address of the client.
- For a direct outbound connection to a public node: if NAT is being used, the server will see the IP address of the NAT server for the private network. If there is no NAT, and the firewall is blocking connections in only one direction, but not re-writing IP addresses, the server will see the client's real IP address.
- For a connection to a host in a different private network that must be relayed through the GCB broker, the server will see the IP address of the GCB broker representing the server. This is an instance of the private server case, as described below.

Therefore, any public server that wants to allow a command from a specific client must have any or all of the various IP addresses mentioned above within the appropriate `HOSTALLOW` settings. In practice, that means opening up the `HOSTALLOW` settings to include not just the actual IP addresses of each node, but also the IP address of the various GCB brokers in use, and potentially, the public IP address of the NAT host for each private network.

However, given that all private nodes which are represented by a given GCB broker could potentially make connections to any other host using the GCB broker's IP address (whenever proxy socket forwarding is being used), if a single private node is being granted a certain level of permission within the Condor pool, all of the private nodes using the same GCB broker will have the same level of permission. This is particularly important in the consideration of granting `HOSTALLOW_ADMINISTRATOR` or `HOSTALLOW_CONFIG` privileges to a private node represented by a GCB broker.

In the case of a public client attempting to connect to a private server, there are only two possible cases:

- the GCB broker can arrange a direct socket from the private server. The private server will see the real public IP address of the client.

- the GCB broker must forward packets from a proxy socket. This may happen because of a non-GCB aware public client, a misconfigured or missing GCB routing table, or a client in a different private network. The private server will see the IP address of its own GCB broker. In the case where the GCB broker runs on a node on the network boundary, the private server will see the GCB broker's private IP address (even if the GCB broker is also listening on the public interface and the leased addresses it provides use the public IP addresses). If the GCB broker is running entirely in the public network and cannot directly connect to the private nodes, the private server will see the remote connection as coming from the broker's public IP address.

This second case is particularly troubling. Since there are legitimate circumstances where a private server would need to use a forwarded proxy socket from its GCB broker, in general, the server should allow requests originating from its GCB broker. But, precisely because of the proxy forwarding, that implies that *any* client that can connect to the GCB broker would be allowed into the private server (if IP-based authorization was the only defense).

The final host-based security setting that requires special mention is `HOSTALLOW_NEGOTIATOR` . If the *condor_negotiator* for the pool is running on a private node being represented by a GCB broker, there must be modifications to the default value. For the purposes of Condor's host-based security, the *condor_negotiator* acts as a client when communicating with each *condor_schedd* in the pool which has idle jobs that need to be matched with available resources. Therefore, all the possible cases of a private client attempting to connect to a given server apply to a private *condor_negotiator*. In practice, that means adding the public IP address of the broker, the real private IP address of the negotiator host, and possibly the public IP address of the NAT host for this private network to the `HOSTALLOW_NEGOTIATOR` setting. Unfortunately, this implies that *any* host behind the same NAT host or using the same GCB broker will be authorized as if it was the *condor_negotiator*.

Future versions of GCB and Condor will hopefully add some form of authentication and authorization to the GCB broker itself, to help alleviate these problems. Until then, sites using GCB are encouraged to use GSI strong authentication (since Kerberos also depends on IP addresses and is therefore incompatible with GCB) to rely on an authorization system that is not affected by address leasing. This is especially true for sites that (foolishly) choose to run their central manager on a private node.

#### Implications of GCB for Other Condor Configuration

Using GCB and address leasing has implications for Condor configuration settings outside of the Host/IP-based security settings. Each is described.

**`COLLECTOR_HOST`** If the *condor_collector* for the pool is running on a private node being represented by a GCB broker, `COLLECTOR_HOST` must be set to the host name or IP address of the GCB broker machine, *not* the real host name/IP address of the private node where the daemons are actually running. When the *condor_collector* on the private node attempts to `bind()` to its command port (9618 by default), it will request port 9618 on the GCB broker

node, instead. The port is not a worry, but the host name or IP address is a worry. When public nodes want to communicate with the *condor_collector*, they must go through the GCB broker. In theory, other nodes inside the same private network could be told to directly use the private IP address of the *condor_collector* host, but that is unnecessary, and would probably lead to other confusion and configuration problems.

However, because the *condor_collector* is listening on a fixed port, and that single port is reserved on the GCB broker node, no two private nodes using the same broker can attempt to use the same port for their *condor_collector*. Therefore, any site that is attempting to set up multiple pools within the same private network is strongly encouraged to set up separate GCB brokers for each pool. Otherwise, one or both of the pools must use a non-standard port for the *condor_collector*, which adds yet more complication to an already complicated situation.

**CKPT_SERVER_HOST** Much like the case for COLLECTOR_HOST described above, a checkpoint server on a private node will have to lease a port on the GCB broker node. However, the checkpoint server also uses a fixed port, and unlike the *condor_collector*, there is no way to configure an alternate value. Therefore, only a single checkpoint server can be run behind a given GCB broker. The same solution works: if multiple checkpoint servers are required, multiple GCB brokers are deployed and configured. Furthermore, the host name of the GCB broker should be used as the value for CKPT_SERVER_HOST, not the real IP address or host name of the private node where the *condor_ckpt_server* is running.

**SEC_DEFAULT_AUTHENTICATION_METHODS** KERBEROS may not be used for authentication on a GCB-enabled pool. The IP addresses used in various circumstances will not be the real IP addresses of the machines. Since Kerberos stores the IP address of each host as part of the Kerberos ticket, authentication will fail on a GCB-enabled pool.

Due to the complications and security limitations that arise from running a central manager on a private node represented by GCB (both regarding the COLLECTOR_HOST and HOSTALLOW_NEGOTIATOR), we recommend that sites avoid locating a central manager on a private host whenever possible.

### 3.7.4 Using TCP to Send Updates to the *condor_collector*

TCP sockets are reliable, connection-based sockets that guarantee the delivery of any data sent. However, TCP sockets are fairly expensive to establish, and there is more network overhead involved in sending and receiving messages.

UDP sockets are datagrams, and are not reliable. There is very little overhead in establishing or using a UDP socket, but there is also no guarantee that the data will be delivered. All previous Condor versions used UDP sockets to send updates to the *condor_collector*, and this did not cause problems.

Beginning with version 6.5.0, Condor can be configured to use TCP sockets to send updates to the *condor_collector* instead of UDP datagrams. It is *not* intended for most sites. This feature is targeted at sites where UDP updates are lost because of the underlying network. Most Condor

administrators that believe this is a good idea for their site are wrong. Do not enable this feature just because it sounds like a good idea. The only cases where an administrator would want this feature are if the ClassAd updates are consistently not getting to the *condor_collector*. An example where this may happen is if the pool is comprised of machines across a wide area network (WAN) where UDP packets are frequently dropped.

Configuration variables are set to enable the use of TCP sockets. There are two variables that an administrator must define to enable this feature:

**UPDATE_COLLECTOR_WITH_TCP** When set to `True`, the Condor daemons to use TCP to update the *condor_collector*, instead of the default UDP. Defaults to `False`.

**COLLECTOR_SOCKET_CACHE_SIZE** Specifies the number of TCP sockets cached at the *condor_collector*. The default value for this setting is 0, with no cache enabled.

The use of a cache allows Condor to leave established TCP sockets open, facilitating much better performance. Subsequent updates can reuse an already open socket. The work to establish a TCP connection may be lengthy, including authentication and setting up encryption. Therefore, Condor requires that a socket cache be defined if TCP updates are to be used. TCP updates will be refused by the *condor_collector* daemon if a cache is not enabled.

Each Condor daemon will have 1 socket open to the *condor_collector*. So, in a pool with N machines, each of them running a *condor_master*, *condor_schedd*, and *condor_startd*, the *condor_collector* would need a socket cache that has at least 3*N entries. Machines running Personal Condor in the pool need an additional two entries (for the *condor_master* and *condor_schedd*) for each Personal Condor installation.

Every cache entry utilizes a file descriptor within the *condor_collector* daemon. Therefore, be careful not to define a cache that is larger than the number of file descriptors the underlying operating system allocates for a single process.

NOTE: At this time, `UPDATE_COLLECTOR_WITH_TCP`, only affects the main *condor_collector* for the site, not any sites that a *condor_schedd* might flock to.

## 3.8 The Checkpoint Server

A Checkpoint Server maintains a repository for checkpoint files. Using checkpoint servers reduces the disk requirements of submitting machines in the pool, since the submitting machines no longer need to store checkpoint files locally. Checkpoint server machines should have a large amount of disk space available, and they should have a fast connection to machines in the Condor pool.

If your spool directories are on a network file system, then checkpoint files will make two trips over the network: one between the submitting machine and the execution machine, and a second between the submitting machine and the network file server. If you install a checkpoint server and configure it to use the server's local disk, the checkpoint will travel only once over the network,

between the execution machine and the checkpoint server. You may also obtain checkpointing network performance benefits by using multiple checkpoint servers, as discussed below.

NOTE: It is a good idea to pick very stable machines for your checkpoint servers. If individual checkpoint servers crash, the Condor system will continue to operate, although poorly. While the Condor system will recover from a checkpoint server crash as best it can, there are two problems that can (and will) occur:

1. A checkpoint cannot be sent to a checkpoint server that is not functioning. Jobs will keep trying to contact the checkpoint server, backing off exponentially in the time they wait between attempts. Normally, jobs only have a limited time to checkpoint before they are kicked off the machine. So, if the server is down for a long period of time, chances are that a lot of work will be lost by jobs being killed without writing a checkpoint.
2. If a checkpoint is not available from the checkpoint server, a job cannot be retrieved, and it will either have to be restarted from the beginning, or the job will wait for the server to come back online. This behavior is controlled with the `MAX_DISCARDED_RUN_TIME` parameter in the config file (see section 3.3.8 on page 162 for details). This parameter represents the maximum amount of CPU time you are willing to discard by starting a job over from scratch if the checkpoint server is not responding to requests.

### 3.8.1 Preparing to Install a Checkpoint Server

The location of checkpoints changes upon the installation of a checkpoint server. A configuration change would cause currently queued jobs with checkpoints to not be able to find their checkpoints. This results in the jobs with checkpoints remaining indefinitely queued (never running) due to the lack of finding their checkpoints. It is therefore best to either remove jobs from the queues or let them complete before installing a checkpoint server. It is advisable to shut your pool down before doing any maintenance on your checkpoint server. See section **??** on page **??** for details on shutting down your pool.

A graduated installation of the checkpoint server may be accomplished by configuring submit machines as their queues empty.

### 3.8.2 Installing the Checkpoint Server Module

Files relevant to a checkpoint server are

```
sbin/condor_ckpt_server
sbin/condor_cleanckpts
etc/examples/condor_config.local.ckpt.server
```

`condor_ckpt_server` is the checkpoint server binary. `condor_cleanckpts` is a script that can be periodically run to remove stale checkpoint files from your server. The checkpoint server

normally cleans all old files itself. However, in certain error situations, stale files can be left that are no longer needed. You may set up a cron job that calls *condor_cleanckpts* every week or so to automate the cleaning up of any stale files. The example configuration file give with the module is described below.

There are three steps necessary towards running a checkpoint server:

1. Configure the checkpoint server.
2. Start the checkpoint server.
3. Configure your pool to use the checkpoint server.

**Configure the Checkpoint Server** Place settings in the local configuration file of the checkpoint server. The file `etc/examples/condor_config.local.ckpt.server` contains the needed settings. Insert these into the local configuration file of your checkpoint server machine.

The `CKPT_SERVER_DIR` must be customized. The `CKPT_SERVER_DIR` attribute defines where your checkpoint files are to be located. It is better if this is on a very fast local file system (preferably a RAID). The speed of this file system will have a direct impact on the speed at which your checkpoint files can be retrieved from the remote machines.

The other optional settings are:

**`DAEMON_LIST`** (Described in section 3.3.9). To have the checkpoint server managed by the *condor_master*, the `DAEMON_LIST` entry must have `MASTER` and `CKPT_SERVER`. Add `STARTD` if you want to allow jobs to run on your checkpoint server. Similarly, add `SCHEDD` if you would like to submit jobs from your checkpoint server.

The rest of these settings are the checkpoint server-specific versions of the Condor logging entries, as described in section 3.3.4 on page 151.

**`CKPT_SERVER_LOG`** The `CKPT_SERVER_LOG` is where the checkpoint server log is placed.

**`MAX_CKPT_SERVER_LOG`** Sets the maximum size of the checkpoint server log before it is saved and the log file restarted.

**`CKPT_SERVER_DEBUG`** Regulates the amount of information printed in the log file. Currently, the only debug level supported is `D_ALWAYS`.

**Start the Checkpoint Server** To start the newly configured checkpoint server, restart Condor on that host to enable the *condor_master* to notice the new configuration. Do this by sending a *condor_restart* command from any machine with administrator access to your pool. See section 3.6.10 on page 282 for full details about IP/host-based security in Condor.

**Configure the Pool to Use the Checkpoint Server** After the checkpoint server is running, you change a few settings in your configuration files to let your pool know about your new server:

**`USE_CKPT_SERVER`** This parameter should be set to TRUE (the default).

**CKPT_SERVER_HOST** This parameter should be set to the full hostname of the machine that is now running your checkpoint server.

It is most convenient to set these parameters in your global configuration file, so they affect all submission machines. However, you may configure each submission machine separately (using local configuration files) if you do not want all of your submission machines to start using the checkpoint server at one time. If USE_CKPT_SERVER is set to FALSE, the submission machine will not use a checkpoint server.

Once these settings are in place, send a *condor_reconfig* to all machines in your pool so the changes take effect. This is described in section **??** on page **??**.

### 3.8.3 Configuring your Pool to Use Multiple Checkpoint Servers

It is possible to configure a Condor pool to use multiple checkpoint servers. The deployment of checkpoint servers across the network improves checkpointing performance. In this case, Condor machines are configured to checkpoint to the *nearest* checkpoint server. There are two main performance benefits to deploying multiple checkpoint servers:

- Checkpoint-related network traffic is localized by intelligent placement of checkpoint servers.
- Faster checkpointing implies that jobs spend less time checkpointing, more time doing useful work, jobs have a better chance of checkpointing successfully before returning a machine to its owner, and workstation owners see Condor jobs leave their machines quicker.

Once you have multiple checkpoint servers running in your pool, the following configuration changes are required to make them active.

First, USE_CKPT_SERVER should be set to TRUE (the default) on all submitting machines where Condor jobs should use a checkpoint server. Additionally, STARTER_CHOOSES_CKPT_SERVER should be set to TRUE (the default) on these submitting machines. When TRUE, this parameter specifies that the checkpoint server specified by the machine running the job should be used instead of the checkpoint server specified by the submitting machine. See section 3.3.8 on page 162 for more details. This allows the job to use the checkpoint server closest to the machine on which it is running, instead of the server closest to the submitting machine. For convenience, set these parameters in the global configuration file.

Second, set CKPT_SERVER_HOST on each machine. As described, this is set to the full hostname of the checkpoint server machine. In the case of multiple checkpoint servers, set this in the local configuraton file. It is the hostname of the nearest server to the machine.

Third, send a *condor_reconfig* to all machines in the pool so the changes take effect. This is described in section **??** on page **??**.

After completing these three steps, the jobs in your pool will send checkpoints to the nearest checkpoint server. On restart, a job will remember where its checkpoint was stored and get it from

the appropriate server. After a job successfully writes a checkpoint to a new server, it will remove any previous checkpoints left on other servers.

NOTE: If the configured checkpoint server is unavailable, the job will keep trying to contact that server as described above. It will not use alternate checkpoint servers. This may change in future versions of Condor.

### 3.8.4 Checkpoint Server Domains

The configuration described in the previous section ensures that jobs will always write checkpoints to their nearest checkpoint server. In some circumstances, it is also useful to configure Condor to localize checkpoint read transfers, which occur when the job restarts from its last checkpoint on a new machine. To localize these transfers, we want to schedule the job on a machine which is near the checkpoint server on which the job's checkpoint is stored.

We can say that all of the machines configured to use checkpoint server "A" are in "checkpoint server domain A." To localize checkpoint transfers, we want jobs which run on machines in a given checkpoint server domain to continue running on machines in that domain, transferring checkpoint files in a single local area of the network. There are two possible configurations which specify what a job should do when there are no available machines in its checkpoint server domain:

- The job can remain idle until a workstation in its checkpoint server domain becomes available.
- The job can try to immediately begin executing on a machine in another checkpoint server domain. In this case, the job transfers to a new checkpoint server domain.

These two configurations are described below.

The first step in implementing checkpoint server domains is to include the name of the nearest checkpoint server in the machine ClassAd, so this information can be used in job scheduling decisions. To do this, add the following configuration to each machine:

```
CkptServer = "$(CKPT_SERVER_HOST)"
STARTD_EXPRS = $(STARTD_EXPRS), CkptServer
```

For convenience, we suggest that you set these parameters in the global config file. Note that this example assumes that `STARTD_EXPRS` is defined previously in your configuration. If not, then you should use the following configuration instead:

```
CkptServer = "$(CKPT_SERVER_HOST)"
STARTD_EXPRS = CkptServer
```

Now, all machine ClassAds will include a `CkptServer` attribute, which is the name of the checkpoint server closest to this machine. So, the `CkptServer` attribute defines the checkpoint server domain of each machine.

To restrict jobs to one checkpoint server domain, we need to modify the jobs' `Requirements` expression as follows:

```
Requirements = ((LastCkptServer == TARGET.CkptServer) || (LastCkptServer =?= UNDEFI
```

This `Requirements` expression uses the `LastCkptServer` attribute in the job's ClassAd, which specifies where the job last wrote a checkpoint, and the `CkptServer` attribute in the machine ClassAd, which specifies the checkpoint server domain. If the job has not written a checkpoint yet, the `LastCkptServer` attribute will be UNDEFINED, and the job will be able to execute in any checkpoint server domain. However, once the job performs a checkpoint, `LastCkptServer` will be defined and the job will be restricted to the checkpoint server domain where it started running.

If instead we want to allow jobs to transfer to other checkpoint server domains when there are no available machines in the current checkpoint server domain, we need to modify the jobs' `Rank` expression as follows:

```
Rank = ((LastCkptServer == TARGET.CkptServer) || (LastCkptServer =?= UNDEFINED))
```

This `Rank` expression will evaluate to 1 for machines in the job's checkpoint server domain and 0 for other machines. So, the job will prefer to run on machines in its checkpoint server domain, but if no such machines are available, the job will run in a new checkpoint server domain.

You can automatically append the checkpoint server domain `Requirements` or `Rank` expressions to all STANDARD universe jobs submitted in your pool using `APPEND_REQ_STANDARD` or `APPEND_RANK_STANDARD` . See section 3.3.15 on page 190 for more details.

## 3.9 DaemonCore

This section is a brief description of *DaemonCore*. DaemonCore is a library that is shared among most of the Condor daemons which provides common functionality. Currently, the following daemons use DaemonCore:

- *condor_master*
- *condor_startd*
- *condor_schedd*
- *condor_collector*
- *condor_negotiator*
- *condor_kbdd*

Most of DaemonCore's details are not interesting for administrators. However, DaemonCore does provide a uniform interface for the daemons to various Unix signals, and provides a common set of command-line options that can be used to start up each daemon.

### 3.9.1 DaemonCore and Unix signals

One of the most visible features DaemonCore provides for administrators is that all daemons which use it behave the same way on certain Unix signals. The signals and the behavior DaemonCore provides are listed below:

**SIGHUP** Causes the daemon to reconfigure itself.

**SIGTERM** Causes the daemon to gracefully shutdown.

**SIGQUIT** Causes the daemon to quickly shutdown.

Exactly what "gracefully" and "quickly" means varies from daemon to daemon. For daemons with little or no state (the kbdd, collector and negotiator) there's no difference and both signals result in the daemon shutting itself down basically right away. For the master, graceful shutdown just means it asks all of its children to perform their own graceful shutdown methods, while fast shutdown means it asks its children to perform their own fast shutdown methods. In both cases, the master only exits once all its children have exited. In the startd, if the machine is not claimed and running a job, both result in an immediate exit. However, if the startd is running a job, graceful shutdown results in that job being checkpointed, while fast shutdown does not. In the schedd, if there are no jobs currently running (i.e. no *condor_shadow* processes), both signals result in an immediate exit. With jobs running, however, graceful shutdown means that the schedd asks each shadow to gracefully vacate whatever job it is serving, while fast shutdown results in a hard kill of every shadow with no chance of checkpointing.

For all daemons, "reconfigure" just means that the daemon re-reads its configuration file(s) and any settings that have changed take effect. For example, changing the level of debugging output, the value of timers that determine how often daemons perform certain actions, the paths to the binaries you want the *condor_master* to spawn, etc. See section 3.3 on page 138, "Configuring Condor" for full details on what settings are in the configuration files and what they do.

### 3.9.2 DaemonCore and Command-line Arguments

The other visible feature that DaemonCore provides to administrators is a common set of command-line arguments that all daemons understand. These arguments and what they do are described below:

**-b** Causes the daemon to start up in the background. When a DaemonCore process starts up with this option, it disassociates itself from the terminal and forks itself, so that it runs in the background. This is the default behavior for Condor daemons.

**-f** Causes the daemon to start up in the foreground. Instead of forking, the daemon runs in the foreground.

NOTE: When the *condor_master* starts up daemons, it does so with the **-f** option, as it has already forked a process for the new daemon. There will be a **-f** in the argument list for all Condor daemons that the *condor_master* spawns.

**-c filename** Causes the daemon to use the specified **filename** (use a full path name) as its global configuration file. This overrides the `CONDOR_CONFIG` environment variable and the regular locations that Condor checks for its configuration file: the user condor's home directory and file `/etc/condor/condor_config`.

**-p port** Causes the daemon to bind to the specified port as its command socket. The *condor_master* daemon uses this option to ensure that the *condor_collector* and *condor_negotiator* start up using well-known ports that the rest of Condor depends upon them using.

**-t** Causes the daemon to print out its error message to `stderr` instead of its specified log file. This option forces the **-f** option.

**-v** Causes the daemon to print out version information and exit.

**-l directory** Overrides the value of `LOG` as specified in the configuration files. Primarily, this option is used with the *condor_kbdd* when it needs to run as the individual user logged into the machine, instead of running as root. Regular users would not normally have permission to write files into Condor's log directory. Using this option, they can override the value of `LOG` and have the *condor_kbdd* write its log file into a directory that the user has permission to write to.

**-a string** Append a period (".") concatenated with **string** to the file name of the log for this daemon, as specified in the configuration file.

**-pidfile filename** Causes the daemon to write out its PID (process id number) to the specified **filename**. This file can be used to help shutdown the daemon without first searching through the output of the Unix *ps* command.

Since daemons run with their current working directory set to the value of `LOG`, if a full path (one that begins with a "/"), the file will be placed in the `LOG` directory. If you leave the **filename** in the `LOG` directory, add this **filename** to the `VALID_LOG_FILES` configuration variable, described in section 3.3.16 on page 192, so that *condor_preen* does not remove it.

**-k filename** For non-Windows operating systems, causes the daemon to read out a PID from the specified **filename**, and send a SIGTERM to that process. The daemon started with this optional argument waits until the daemon it is attempting to kill has exited.

**-r minutes** Causes the daemon to set a timer, upon expiration of which, it sends itself a SIGTERM for graceful shutdown.

**-q** Quiet output; write less verbose error messages to `stderr` when something goes wrong, and before regular logging can be initialized.

**-d** Use dynamic directories. The LOG, SPOOL, and EXECUTE directories are all created by the daemon at run time, and they are named by appending the parent's IP address and PID to the value in the configuration file. These settings are then inherited by all children of the daemon invoked with this *-d* argument. For the *condor_master*, all Condor processes will use the new directories. If a *condor_schedd* is invoked with the *-d* argument, then only the *condor_schedd* daemon and any *condor_shadow* daemons it spawns will use the dynamic directories (named with the *condor_schedd* daemon's PID).

Note that by using a dynamically-created spool directory named by the IP address and PID, upon restarting daemons, jobs submitted to the original *condor_schedd* daemon that were stored in the old spool directory will not be noticed by the new *condor_schedd* daemon, unless you manually specify the old, dyanmically-generated SPOOL directory path in the configuration of the new *condor_schedd* daemon.

## 3.10 The High Availability of Daemons

In the case that a key machine no longer functions, Condor can be configured such that another machine takes on the key functions. This is called *High Availability*. While high availability is generally applicable, there are currently two specialized cases for its use: when the central manager (running the *condor_negotiator* and *condor_collector* daemons) becomes unavailable, and when the machine running the *condor_schedd* daemon (maintaining the job queue) becomes unavailable.

### 3.10.1 High Availability of the Job Queue

For a pool where all jobs are submitted through a single machine in the pool, and there are lots of jobs, this machine becoming nonfunctional means that jobs stop running. The *condor_schedd* daemon maintains the job queue. No job queue due to having a nonfunctional machine implies that no jobs can be run. This situation is worsened by using one machine as the single submission point. For each Condor job (taken from the queue) that is executed, a *condor_shadow* process runs on the machine where submitted to handle input/output functionality. If this machine becomes nonfunctional, none of the jobs can continue. The entire pool stops running jobs.

The goal of *High Availability* in this special case is to transfer the *condor_schedd* daemon to run on another designated machine. Jobs caused to stop without finishing can be restarted from the beginning, or can continue execution using the most recent checkpoint. New jobs can enter the job queue. Without *High Availability*, the job queue would remain intact, but further progress on jobs would wait until the machine running the *condor_schedd* daemon became available (after fixing whatever caused it to become unavailable).

Condor uses its flexible configuration mechanisms to allow the transfer of the *condor_schedd* daemon from one machine to another. The configuration specifies which machines are chosen to run the *condor_schedd* daemon. To prevent multiple *condor_schedd* daemons from running at the same time, a lock (semaphore-like) is held over the job queue. This synchronizes the situation in which

control is transferred to a secondary machine, and the primary machine returns to functionality. Configuration variables also determine time intervals at which the lock expires, and periods of time that pass between polling to check for expired locks.

To specify a single machine that would take over, if the machine running the *condor_schedd* daemon stops working, the following additions are made to the local configuration of any and all machines that are able to run the *condor_schedd* daemon (becoming the single pool submission point):

```
MASTER_HA_LIST = SCHEDD
SPOOL = /share/spool
HA_LOCK_URL = file:/share/spool
```

Configuration macro `MASTER_HA_LIST` identifies the *condor_schedd* daemon as the daemon that is to be watched to make sure that it is running. Each machine with this configuration must have access to the lock (the job queue) which synchronizes which single machine does run the *condor_schedd* daemon. This lock and the job queue must both be located in a shared file space, and is currently specified only with a file URL. The configuration specifies the shared space (`SPOOL`), and the URL of the lock.

As Condor starts on machines that are configured to run the single *condor_schedd* daemon, the *condor_master* daemon of the first machine that looks at (polls) the lock and notices that no lock is held. This implies that no *condor_schedd* daemon is running. This *condor_master* daemon acquires the lock and runs the *condor_schedd* daemon. Other machines with this same capability to run the *condor_schedd* daemon look at (poll) the lock, but do not run the daemon, as the lock is held. The machine running the *condor_schedd* daemon renews the lock periodically.

If the machine running the *condor_schedd* daemon fails to renew the lock (because the machine is not functioning), the lock times out (becomes stale). The lock is released by the *condor_master* daemon if *condor_off* or *condor_off -schedd* is executed, or when the *condor_master* daemon knows that the *condor_schedd* daemon is no longer running. As other machines capable of running the *condor_schedd* daemon look at the lock (poll), one machine will be the first to notice that the lock has timed out or been released. This machine (correctly) interprets this situation as the *condor_schedd* daemon is no longer running. This machine's *condor_master* daemon then acquires the lock and runs the *condor_schedd* daemon.

See section 3.3.9, in the section on *condor_master* Configuration File Macros for details relating to the configuration variables used to set timing and polling intervals.

### 3.10.2 High Availability of the Central Manager

#### Interaction with Flocking

The Condor high availability mechanisms discussed in this section currently do not work well in configurations involving flocking. The individual problems listed interact to make the situation worse.

Because of these problems, we advice against the use of flocking to pools with high availability mechanisms enabled.

- The *condor_schedd* has a hard configured list of *condor_collector* and *condor_negotiator* daemons, and does not query redundant collectors to get the current *condor_negotiator*, as it does when communicating with its local pool. As a result, if the default *condor_negotiator* fails, the *condor_schedd* does not learn of the failure, and thus, talk to the new *condor_negotiator*.
- When the *condor_negotiator* is unable to communicate with a *condor_collector*, it utilizes the next *condor_collector* within the list. Unfortunately, it does not start over at the top of the list. When combined with the previous problem, a backup *condor_negotiator* will never get jobs from a flocked *condor_schedd*.

**Introduction**

The *condor_negotiator* and *condor_collector* daemons are the heart of the Condor matchmaking system. The availability of these daemons is critical to a Condor pool's functionality. Both daemons usually run on the same machine, most often known as the central manager. The failure of a central manager machine prevents Condor from matching new jobs and allocating new resources. High availability of the *condor_negotiator* and *condor_collector* daemons eliminates this problem.

Configuration allows one of multiple machines within the pool to function as the central manager. While there are may be many active *condor_collector* daemons, only a single, active *condor_negotiator* daemon will be running. The machine with the *condor_negotiator* daemon running is the active central manager. The other potential central managers each have a *condor_collector* daemon running; these are the idle central managers.

All submit and execute machines are configured to report to all potential central manager machines.

Each potential central manager machine runs the high availability daemon, *condor_had*. These daemons communicate with each other, constantly monitoring the pool to ensure that one active central manager is available. If the active central manager machine crashes or is shut down, these daemons detect the failure, and they agree on which of the idle central managers is to become the active one. A protocol determines this.

In the case of a network partition, idle *condor_had* daemons within each partition detect (by the lack of communication) a partitioning, and then use the protocol to chose an active central manager. As long as the partition remains, and there exists an idle central manager within the partition, there will be one active central manager within each partition. When the network is repaired, the protocol returns to having one central manager.

Through configuration, a specific central manager machine may act as the primary central manager. While this machine is up and running, it functions as the central manager. After a failure of this primary central manager, another idle central manager becomes the active one. When the primary recovers, it again becomes the central manager. This is a recommended configuration, if one of the central managers is a reliable machine, which is expected to have very short periods of instability.

An alternative configuration allows the promoted active central manager (in the case that the central manager fails) to stay active after the failed central manager machine returns.

This high availability mechanism operates by monitoring communication between machines. Note that there is a significant difference in communications between machines when

1. a machine is down
2. a specific daemon (the *condor_had* daemon in this case) is not running, yet the machine is functioning

The high availability mechanism distinguishes between these two, and it operates based only on first (when a central manager machine is down). A lack of executing daemons does *not* cause the protocol to choose or use a new active central manager.

The central manager machine contains state information, and this includes information about user priorities. The information is kept in a single file, and is used by the central manager machine. Should the primary central manager fail, a pool with high availability enabled would lose this information (and continue operation, but with re-initialized priorities). Therefore, the *condor_replication* daemon exists to replicate this file on all potential central manager machines. This daemon promulgates the file in a way that is safe from error, and more secure than dependence on a shared file system copy.

The *condor_replication* daemon runs on each potential central manager machine as well as on the active central manager machine. There is a unidirectional communication between the *condor_had* daemon and the *condor_replication* daemon on each machine.

### Configuration

The high availability of central manager machines is enabled through configuration. It is disabled by default. All machines in a pool must be configured appropriately in order to make the high availability mechanism work. See section 3.3.25, for definitions of these configuration variables.

The stabilization period is the time it takes for the *condor_had* daemons to detect a change in the pool state such as an active central manager failure or network partition, and recover from this change. It may be computed using the following formula:

```
stabilization period = 12 * (number of central managers) *
                       $(HAD_CONNECTION_TIMEOUT)
```

To disable the high availability of central managers mechanism, it is sufficient to remove `HAD`, `REPLICATION`, and `NEGOTIATOR` from the `DAEMON_LIST` configuration variable on all machines, leaving only one *condor_negotiator* in the pool.

To shut down a currently operating high availability mechanism, follow the given steps. All commands must be invoked from a host which has administrative permissions on all central managers. The first three commands kill all *condor_had*, *condor_replication*, and all running *condor_negotiator*

daemons. The last command is invoked on the host where the single *condor_negotiator* daemon is to run.

1. `condor_off -all -neg`
2. `condor_off -all -subsystem -replication`
3. `condor_off -all -subsystem -had`
4. `condor_on -neg`

When configuring *condor_had* to control the *condor_negotiator*, if the default backoff constant value is too small, it can result in a churning of the *condor_negotiator*, especially in cases in which the primary negotiator is unable to run due to misconfiguration. In these cases, the *condor_master* will kill the *condor_had* after the *condor_negotiator* exists, wait a short period, then restart *condor_had*. The *condor_had* will then win the election, so the secondary *condor_negotiator* will be killed, and the primary will be restarted, only to exit again. If this happens to quickly, neither *condor_negotiator* will run long enough to complete a negotiation cycle, resulting in no jobs getting started. Increasing this value via `MASTER_HAD_BACKOFF_CONSTANT` to be larger than a typical negotiation cycle can help solve this problem.

To run a high availability pool without the replication feature, do the following operations:

1. Set the `HAD_USE_REPLICATION` configuration variable to `false`, and thus disable the replication on configuration level.
2. Remove `REPLICATION` from both `DAEMON_LIST` and `DC_DAEMON_LIST` in the configuration file.

### Sample Configuration

This section provides sample configurations for high availability. The two parts to this are the configuration for the potential central manager machines, and the configuration for the machines within the pool that will *not* be central managers.

This is a sample configuration relating to the high availability of central managers. This is for the potential central manager machines.

```
##############################################################################
# A sample configuration file for central managers, to enable the        #
# the high availability  mechanism.                                      #
##############################################################################

# unset these two macros
NEGOTIATOR_HOST=
CONDOR_HOST=
```

```
##############################################################################
## THE FOLLOWING MUST BE IDENTICAL ON ALL POTENTIAL CENTRAL MANAGERS.   #
##############################################################################
## For simplicity in writing other expressions, define a variable
## for each potential central manager in the pool.
## These are samples.
CENTRAL_MANAGER1 = cm1.cs.technion.ac.il
CENTRAL_MANAGER2 = cm2.cs.technion.ac.il
## A list of all potential central managers in the pool.
COLLECTOR_HOST = $(CENTRAL_MANAGER1),$(CENTRAL_MANAGER2)

## Define the port number on which the condor_had daemon will
## listen.  The port must match the port number used
## for when defining HAD_LIST.  This port number is
## arbitrary; make sure that there is no port number collision
## with other applications.
HAD_PORT = 51450
HAD_ARGS = -p $(HAD_PORT)

## The following macro defines the port number condor_replication will listen
## on on this machine. This port should match the port number specified
## for that replication daemon in the REPLICATION_LIST
## Port number is arbitrary (make sure no collision with other applications)
## This is a sample port number
REPLICATION_PORT = 41450
REPLICATION_ARGS = -p $(REPLICATION_PORT)

## The following list must contain the same addresses
## as HAD_LIST. In addition, for each hostname, it should specify
## the port number of condor_replication daemon running on that host.
## This parameter is mandatory and has no default value
REPLICATION_LIST = $(CENTRAL_MANAGER1):$(REPLICATION_PORT),$(CENTRAL_MANAGER2):$( REPLICATION_PORT)

## The following list must contain the same addresses in the same order
## as COLLECTOR_HOST. In addition, for each hostname, it should specify
## the port number of condor_had daemon running on that host.
## The first machine in the list will be the PRIMARY central manager
## machine, in case HAD_USE_PRIMARY is set to true.
HAD_LIST = $(CENTRAL_MANAGER1):$(HAD_PORT),$(CENTRAL_MANAGER2):$(HAD_PORT)


## HAD connection time.
## Recommended value is 2 if the central managers are on the same subnet.
## Recommended value is 5 if Condor security is enabled.
## Recommended value is 10 if the network is very slow, or
## to reduce the sensitivity of HA daemons to network failures.
HAD_CONNECTION_TIMEOUT = 2

##If true, the first central manager in HAD_LIST is a primary.
HAD_USE_PRIMARY = true

##--------------------------------------------------------------------
##  Host/IP access levels
##--------------------------------------------------------------------

##  What machines have administrative rights for your pool?  This
##  defaults to your central manager.  You should set it to the
##  machine(s) where whoever is the condor administrator(s) works
```

```
##  (assuming you trust all the users who log into that/those
##  machine(s), since this is machine-wide access you're granting).
HOSTALLOW_ADMINISTRATOR = $(COLLECTOR_HOST)


##  Negotiator access.  Machines listed here are trusted central
##  managers.  You should normally not have to change this.
HOSTALLOW_NEGOTIATOR = $(COLLECTOR_HOST)

##########################################################################
## THE PARAMETERS BELOW ARE ALLOWED TO BE DIFFERENT ON EACH       #
## CENTRAL MANAGERS                                              #
## THESE ARE MASTER SPECIFIC PARAMETERS
##########################################################################

## The location of executable files
HAD = $(SBIN)/condor_had
REPLICATION = $(SBIN)/condor_replication

## the master should start at least these five daemons
DAEMON_LIST = MASTER, COLLECTOR, NEGOTIATOR, HAD, REPLICATION
## DC_Daemon list should contain at least these five
DC_DAEMON_LIST = MASTER, COLLECTOR, NEGOTIATOR, HAD, REPLICATION

## Enables/disables the replication feature of HAD daemon
## Default: no
HAD_USE_REPLICATION = true

## Name of the file from the SPOOL directory that will be replicated
## Default: $(SPOOL)/Accountantnew.log
STATE_FILE = $(SPOOL)/Accountantnew.log

## Period of time between two successive awakenings of the replication daemon
## Default: 300
REPLICATION_INTERVAL = 300

# Period of time, in which transferer daemons have to accomplish the
## downloading/uploading process
## Default: 300
MAX_TRANSFERER_LIFETIME = 300

## Period of time between two successive sends of classads to the collector by HAD
## Default: 300
HAD_UPDATE_INTERVAL = 300


## The HAD controlls the negotiator, and should have a larger
## backoff constant
MASTER_NEGOTIATOR_CONTROLLER = HAD
MASTER_HAD_BACKOFF_CONSTANT = 360

## The size of the log file
MAX_HAD_LOG = 640000
## debug level
HAD_DEBUG = D_COMMAND
## location of the condor_had log file
HAD_LOG = $(LOG)/HADLog
```

```
## The size of replication log file
MAX_REPLICATION_LOG = 640000
## Replication debug level
REPLICATION_DEBUG = D_COMMAND
## Replication log file
REPLICATION_LOG = $(LOG)/ReplicationLog
```

Machines that are not potential central managers also require configuration. The following is a sample configuration relating to high availability for machines that will *not* be central managers.

```
##############################################################################
# Sample configuration relating to high availability for machines        #
# that DO NOT run the condor_had daemon.                                 #
##############################################################################

#unset these variables
NEGOTIATOR_HOST =
CONDOR_HOST =

## For simplicity define a variable for each potential central manager
## in the pool.
CENTRAL_MANAGER1 = cm1.cs.technion.ac.il
CENTRAL_MANAGER2 = cm2.cs.technion.ac.il
## List of all potential central managers in the pool
COLLECTOR_HOST = $(CENTRAL_MANAGER1),$(CENTRAL_MANAGER2)

##--------------------------------------------------------------------
##  Host/IP access levels
##--------------------------------------------------------------------

##  Negotiator access.  Machines listed here are trusted central
##  managers.  You should normally not need to change this.
HOSTALLOW_NEGOTIATOR = $(COLLECTOR_HOST)

##  Now, with flocking (and HA) we need to let the SCHEDD trust the other
##  negotiators we are flocking with as well.  You should normally
##  not need to change this.
HOSTALLOW_NEGOTIATOR_SCHEDD = $(COLLECTOR_HOST)
```

## 3.11 Quill

Quill builds and maintains a mirror database of a Condor job queue. The *condor_quill* daemon implements it, and the *condor_q* and *condor_history* tools use it.

### 3.11.1 Installation and Configuration

Quill uses the *PostgreSQL* database management system. Quill uses the *PostgreSQL* server as its back end and client library, *libpq* to talk to the server. While Quill works and has been tested with

*PostgreSQL* versions 7.4 and beyond, we **strongly recommend** the use of version 8.1 or later due to its integrated facilities of certain key database maintenance tasks.

Obtain *PostgreSQL* from

http://www.postgresql.org/ftp/source/

Installation instructions are detailed in: http://www.postgresql.org/docs/8.1/static/installation.html

Configure *PostgreSQL* after installation:

1. Configure to accept TCP/IP connections. For *PostgreSQL* version 8, use the `listen_addresses` variable in `postgresql.conf` file as a guide. For example, `listen_addresses = '*'` means listen on any IP interface. In *PostgreSQL* version 7, this was accomplished by setting `tcpip_socket=true` in the `postgresql.conf` file.
2. Configure automatic vacuuming. For *PostgreSQL* version 8.1 or later, ensure that these variables with these defaults are commented in and/or set properly in the `postgresql.conf` configuration file:

```
# Turn on/off automatic vacuuming
autovacuum = on

# time between autovacuum runs, in secs
autovacuum_naptime = 60

# min # of tuple updates before vacuum
autovacuum_vacuum_threshold = 1000

# min # of tuple updates before analyze
autovacuum_analyze_threshold = 500

# fraction of rel size before vacuum
autovacuum_vacuum_scale_factor = 0.4

# fraction of rel size before analyze
autovacuum_analyze_scale_factor = 0.2

# default vacuum cost delay for
   # autovac, -1 means use
   # vacuum_cost_delay
autovacuum_vacuum_cost_delay = -1

# default vacuum cost limit for
   # autovac, -1 means use
   # vacuum_cost_limit
autovacuum_vacuum_cost_limit = -1
```

3. Configure *PostgreSQL* to accept TCP/IP connections from specific hosts. Modify the `pg_hba.conf` file (which usually resides in the *PostgreSQL* server's data directory). Access is required by the *condor_quill* daemon, as well as the database users "**quillreader**" and "**quillwriter**". For example, to give database users "**quillreader**" and "**quillwriter**" password-enabled access to all databases on current machine from any other machine in the network, add the following:

   ```
   host  all  quillreader  128.105.0.0  255.255.0.0  password
   host  all  quillwriter  128.105.0.0  255.255.0.0  password
   ```

   Note that in addition to the database specified by the configuration variable `QUILL_DB_NAME`, the *condor_quill* daemon also needs access to the database "template1". In order to create the database in the first place, the *condor_quill* daemon needs to connect to the database.

4. Start the *PostgreSQL* server service. See the URL for the installation instructions for the appropriate method to start the service.

5. The *condor_quill* daemon and client tools connect to the database as users "**quillreader**" and "**quillwriter**". These are database users, not operating system users. The two types of users are quite different from each other. If these data base users do not exist, add them using the *createuser* command supplied with the installation. Assign them with appropriate passwords; these passwords will be used by the Quill tools to connect to the database in a secure way. User "**quillreader**" should not be allowed to create more databases nor create more users. User "**quillwriter**" should not be allowed to create more users, however it should be allowed to create more databases. The following commands create the two users with the appropriate permissions, and be ready to enter the corresponding passwords when prompted.

   ```
   /path/to/postgreSQL/bin/directory/createuser quillreader \
   --no-createdb --no-adduser --pwprompt

   /path/to/postgreSQL/bin/directory/createuser quillwriter \
   --createdb --no-adduser --pwprompt
   ```

   In the case of *PostgreSQL* 8.1 or later, answer "no" to the question about the ability for role creation.

6. The *condor_quill* daemon needs read and write access to the database. It connects as user "**quillwriter**", who has owner privileges to the database. Since this gives all access to the "**quillwriter**" user, this password cannot be stored in a public place (such as the *condor_collector*). For this reason, the "**quillwriter**" password is stored in a file named `.quillwritepassword` in the Condor spool directory. Appropriate protections on this file guarantee secure access to the database. This file must be created and protected by the site administrator; if this file does not exist as and where expected, the *condor_quill* daemon logs an error and exits.

After *PostgreSQL* is configured and running, Condor must also be configured to use Quill since by default Quill is configured to be off.

Add the file `.quillwritepassword` to the `VALID_SPOOL_FILES` variable, since *condor_preen* must be told not to delete this file.

Set up configuration variables that are specific to the installation.

```
QUILL_ENABLED           = TRUE
QUILL_NAME              = some-unique-quill-name.cs.wisc.edu
QUILL_DB_NAME           = database-for-some-unique-quill-name
QUILL_DB_IP_ADDR        = databaseipaddress:port
# the following parameter's units is in seconds
QUILL_POLLING_PERIOD    = 10
# the following parameter's units is in hours
QUILL_HISTORY_CLEANING_INTERVAL = 24
# the following parameter's units is in days
QUILL_HISTORY_DURATION  = 30
QUILL_MANAGE_VACUUM = FALSE
QUILL_IS_REMOTELY_QUERYABLE = TRUE
QUILL_DB_QUERY_PASSWORD =  password-for-database-user-quillreader
QUILL_ADDRESS_FILE      = $(LOG)/.quill_address
```

Descriptions of these and other configuration variables are in section 3.3.26. Here are further brief details:

**QUILL_DB_NAME and QUILL_DB_IP_ADDR** These two variables are used to determine the location of the database server that this Quill would talk to, and the name of the database that it creates. More than one Quill server can talk to the same database server. This can be done by simply letting all the QUILL_DB_IP_ADDR point to the same database server.

If more than one Quill server are sharing the same database server, then the QUILL_DB_NAME variable for all of them should be unique. Otherwise, there would be record overwriting and corruption of job queue information.

**QUILL_NAME** Each quill daemon in the pool has to be uniquely named.

**QUILL_POLLING_PERIOD** This controls the frequency with which Quill polls the job_queue.log file. By default, it is 10 seconds. Since Quill works by periodically sniffing the log file for updates and then sending those updates to the database, this variable controls the trade off between the currency of query results and Quill's load on the system, which is usually negligible.

**QUILL_HISTORY_CLEANING_INTERVAL and QUILL_HISTORY_DURATION** These two variables control the deletion of historical jobs from the database. QUILL_HISTORY_DURATION is the number of days after completion that a given job will stay in the database. A more precise definition is the number of days since the history ad got into the history database; those two might be different, if a job is completed but stays in the queue for a while. All jobs beyond QUILL_HISTORY_DURATION will be deleted. As scanning the entire database for old jobs can be expensive, the other variable QUILL_HISTORY_CLEANING_INTERVAL is the number of hours between two successive scans. By default, QUILL_HISTORY_DURATION is set to 180 days and QUILL_HISTORY_CLEANING_INTERVAL is set to 24 hours.

**QUILL_MANAGE_VACUUM** Set to False by default, this variable determines whether Quill is to perform vacuuming tasks on its tables or not. Vacuuming is a maintenance task that needs to

be performed on tables in *PostgreSQL*. The frequency with which a table is vacuumed typically depends on the number of updates (inserts/deletes) performed on the table. Fortunately, with *PostgreSQL* version 8.1, vacuuming tasks can be configured to be performed automatically by the database server. We recommend that users upgrade to 8.1 and use the integrated vacuuming facilities of the database server, instead of having Quill do them. If the user does prefer having Quill perform those vacuuming tasks, it can be achieved by setting this variable to ExprTrue. However, it cannot be overstated that Quill's vacuuming policy is quite rudimentary as compared to the integrated facilities of the database server, and under high update workloads, can prove to be a bottleneck on the Quill daemon. As such, setting this variable to ExprTrue results in some warning messages in the log file regarding this issue.

**QUILL_IS_REMOTELY_QUERYABLE** Thanks to *PostgreSQL*, one can now remotely query both the job queue and the history tables. This variable controls whether this remote querying feature should be enabled. By default it is `True`. Note that even if this is `False`, one can still query the job queue at the remote *condor_schedd* daemon. This variable only controls whether the database tables are remotely queryable.

**QUILL_DB_QUERY_PASSWORD** In order for the query tools to connect to a database, they need to provide the password that is assigned to the database user "**quillreader**". This variable is then advertised by the *condor_quill* daemon to the *condor_collector*. This facility enables remote querying: remote *condor_q* query tools first ask the *condor_collector* for the password associated with a particular Quill database, and then query that database. Users who do not have access to the *condor_collector* cannot view the password, and as such cannot query the database. Again, this password only provides read access to the database.

**QUILL_ADDRESS_FILE** When Quill starts up, it can place its address (IP and port) into a file. This way, tools running on the local machine do not need to query the central manager to find Quill. This feature can be turned off by commenting out the variable.

### 3.11.2 Four Usage Examples

1. Query a remote Quill daemon on `regular.cs.wisc.edu` for all the jobs in the queue

```
condor_q -name quill@regular.cs.wisc.edu
condor_q -name schedd@regular.cs.wisc.edu
```

There are two ways to get to a Quill daemon: directly using its name as specified in the `QUILL_NAME` configuration variable, or indirectly by querying the *condor_schedd* daemon using its name. In the latter case, *condor_q* will detect if that *condor_schedd* daemon is being serviced by a database, and if so, directly query it. In both cases, the IP address and port of the database server hosting the data of this particular remote Quill daemon can be figured out by the `QUILL_DB_IP_ADDR` and `QUILL_DB_NAME` variables specified in the `QUILL_AD` sent by the quill daemon to the collector and in the `SCHEDD_AD` sent by the *condor_schedd* daemon.

2. Query a remote Quill daemon on `regular.cs.wisc.edu` for all historical jobs belonging to owner einstein.

   ```
   condor_history -name quill@regular.cs.wisc.edu einstein
   ```

3. Query the local Quill daemon for the average time spent in the queue for all non-completed jobs.

   ```
   condor_q -avgqueuetime
   ```

   The average queue time is defined as the average of `(currenttime - jobsubmissiontime)` over all jobs which are neither completed `(JobStatus == 4)` or removed `(JobStatus == 3)`.

4. Query the local Quill daemon for all historical jobs completed since Apr 1, 2005 at 13h 00m.

   ```
   condor_history -completedsince '04/01/2005 13:00'
   ```

   It fetches all jobs which got into the 'Completed' state on or after the specified time stamp. It use the *PostgreSQL* date/time syntax rules, as it encompasses most format options. See http://www.postgresql.org/docs/8.0/static/datatype-datetime.html#AEN4516 for the various time stamp formats.

### 3.11.3 Quill and Its RDBMS Schema

With only 7 tables and 2 views, Quill uses a relatively simple database schema. These can be broadly divided into tables used to store job queue information and those used to store historical information.

The job queue part of the schema closely follows Condor's ClassAd data model. For example, each row in these tables describe an ¡attribute,value¿ pair of the classad. Additionally, just as how Condor distinguishes a ClusterAd from a ProcAd where the former stores attributes common to all jobs within a cluster whereas the latter stores attributes specific to each job, the schema also makes this distinction. Finally, numerical and string valued attributes are stored separately.

Thus, there are four tables:

- **ClusterAds_Str (**cid int, attr text, val text, primary key (cid, attr)**)**
- **ClusterAds_Num (**cid int, attr text, val double precision, primary key (cid, attr)**)**
- **ProcAds_Str (**cid int, pid int, attr text, val text, primary key (cid, pid, attr)**)**
- **ProcAds_Num (**cid int, pid int, attr text, val double precision, primary key (cid, pid, attr)**)**

In addition to the ¡attribute, value¿, each row contains the cluster-id (cid) and in the case of the ProcAd tables, also the proc-id (pid).

Since each ClassAd would be split into potentially two tables (string and numeric), there are views that unify them into a single entity in order to simplify queries.

Here are the view definitions:

- Definition of ClusterAds view
  `CREATE VIEW` **ClusterAds** `as` select cid, attr, val from ClusterAds_Str UNION ALL select cid, attr, cast(val as text) from ClusterAds_Num**;**
- Definition of ProcAds view
  `CREATE VIEW` **ProcAds** `as` select cid, pid, attr, val from ProcAds_Str UNION ALL select cid, pid, attr, cast(val as text) from ProcAds_Num**;**

Finally, the job queue part of the schema also contains a table that stores metadata information related to the `job_queue.log` file.

- **JobQueuePollingInfo (**last_file_mtime BIGINT, last_file_size BIGINT, last_next_cmd_offset BIGINT, last_cmd_offset BIGINT, last_cmd_type SMALLINT, last_cmd_key text, last_cmd_mytype text, last_cmd_targettype text, last_cmd_name text, last_cmd_value text, log_seq_num BIGINT, log_creation_time BIGINT**)**

At all times, there is only 1 row in this table and it describes information related to the last time Quill polled the `job_queue.log` file.

- **last_file_mtime** and **last_file_size**
  The last modified time and size of the file.
- **last_cmd_offset** and **last_next_cmd_offset**
  The offsets of the record last read from the file and its successive record.
- **last_cmd_type**
  The command type (101, 102, etc.) of the record.
- **last_cmd_key**, **last_cmd_mytype**, **last_cmd_targettype**, **last_cmd_name**, and **last_cmd_value**
  Together, these attributes define the record itself. The key refers to the combined ”cid.pid” pair, mytype and target usually contains Job and Machine respectively, and finally the name and value contains the ¡attribute,value¿ pair.
- **log_seq_num** and **log_creation_time**
  Together, these form the first record of the job queue log file. They describe the sequence number of the next historical job queue log file and the creation time of the current job queue log file respectively. Together, they help Quill detect whether the schedd has rotated the job queue log file and, if so, take appropriate actions.

The historical information on the other hand is slightly differently designed. Instead of a purely vertical data model (each row is a <attribute,value> pair), we have two tables that together represent the complete job classad. Their schema is as follows:

1. **History_Horizontal (**cid int, pid int, EnteredHistoryTable timestamp with time zone, Owner text, QDate int, RemoteWallClockTime int, RemoteUserCpu float, RemoteSysCpu float, ImageSize int, JobStatus int, JobPrio int, Cmd text, CompletionDate int, LastRemoteHost text, primary key(cid,pid)**)**
2. **History_Vertical (**cid int, pid int, attr text, val text, primary key (cid, pid, attr)**)**

Each historical job ad is divided into its horizontal and vertical counterparts. This division was made because of query performance reasons. While its easier to store ClassAds in a vertical table, queries on vertical tables generally perform worse than those on horizontal tables since the latter has lot fewer records. However, in Condor, since job ads do not have a fixed schema (users can define their own attributes), a purely horizontal schema would end up having a lot of null values. As such, we have a hybrid schema where attributes on which queries are frequently performed (via *condor_history*) are put in the **History_Horizontal** table and the other attributes are stored vertically (just as in the Cluster/Proc tables above) in the **History_Vertical** table. Also **History_Horizontal** contains all the attributes needed to service the short form of the *condor_history* command (that is, without the -l option).

The resulting hybrid schema has proven to be the most efficient in servicing *condor_history* queries. The job queue tables (Cluster and Proc) were not designed in this hybrid manner because job queues aren't as large as history; just a vertical schema worked great.

### 3.11.4 Quill and Security

There are several layers of security in Quill, some provided by Condor and others provided by the database. First, all accesses to the database are password-protected.

1. The query tools, *condor_q* and *condor_history* connect to the database as user "**quillreader**". The password for this user can vary from one database to another and as such, each Quill daemon advertises this password to the collector. The query tools then obtain this password from the collector and connect successfully to the database. Access to the database by the "**quillreader**" user is read-only, as this is sufficient for the query tools. The *condor_quill* daemon ensures this protected access using the sql GRANT command when it first creates the tables in the database. Note that access to the "**quillreader**" password itself can be blocked by blocking access to the collector, a feature already supported in Condor.
2. The *condor_quill* daemon, on the other hand, needs read and write access to the database. As such, it connects as user "**quillwriter**", who has owner privileges to the database. Since this gives all access to the "**quillwriter**" user, this password cannot be stored in a public place (such as the collector). For this reason, the "**quillwriter**" password is stored in a file called

`.quillwritepassword` in the Condor spool directory. Appropriate protections on this file guarantee secure access to the database. This file must be created and protected by the site administrator; if this file does not exist as and where expected, the *condor_quill* daemon logs an error and exits.

3. The `IsRemotelyQueryable` attribute in the Quill ClassAd advertised by the Quill daemon to the collector can be used by site administrators to disallow the database from being read by all remote Condor query tools.

## 3.12 Setting Up for Special Environments

The following sections describe how to set up Condor for use in special environments or configurations. See section **??** on page **??** for installation instructions on the various Contrib modules that can be optionally downloaded and installed.

### 3.12.1 Using Condor with AFS

If you are using AFS at your site, be sure to read section 3.3.7 on "Shared Filesystem Config Files Entries" for details on configuring your machines to interact with and use shared filesystems, AFS in particular.

Condor does not currently have a way to authenticate itself to AFS. This is true of the Condor daemons that would like to authenticate as AFS user Condor, and the *condor_shadow*, which would like to authenticate as the user who submitted the job it is serving. Since neither of these things can happen yet, there are a number of special things people who use AFS with Condor must do. Some of this must be done by the administrator(s) installing Condor. Some of this must be done by Condor users who submit jobs.

#### AFS and Condor for Administrators

The most important thing is that since the Condor daemons can't authenticate to AFS, the `LOCAL_DIR` (and it's subdirectories like "log" and "spool") for each machine must be either writable to unauthenticated users, or must not be on AFS. The first option is a **VERY** bad security hole so you should **NOT** have your local directory on AFS. If you've got NFS installed as well and want to have your `LOCAL_DIR` for each machine on a shared file system, use NFS. Otherwise, you should put the `LOCAL_DIR` on a local partition on each machine in your pool. This means that you should run *condor_install* to install your release directory and configure your pool, setting the `LOCAL_DIR` parameter to some local partition. When that's complete, log into each machine in your pool and run *condor_init* to set up the local Condor directory.

The `RELEASE_DIR`, which holds all the Condor binaries, libraries and scripts can and probably should be on AFS. None of the Condor daemons need to write to these files, they just need to read them. So, you just have to make your `RELEASE_DIR` world readable and Condor will work just

fine. This makes it easier to upgrade your binaries at a later date, which means that your users can find the Condor tools in a consistent location on all the machines in your pool, and that you can have the Condor config files in a centralized location. This is what we do at UW-Madison's CS department Condor pool and it works quite well.

Finally, you might want to setup some special AFS groups to help your users deal with Condor and AFS better (you'll want to read the section below anyway, since you're probably going to have to explain this stuff to your users). Basically, if you can, create an AFS group that contains all unauthenticated users but that is restricted to a given host or subnet. You're supposed to be able to make these host-based ACLs with AFS, but we've had some trouble getting that working here at UW-Madison. What we have instead is a special group for all machines in our department. So, the users here just have to make their output directories on AFS writable to any process running on any of our machines, instead of any process on any machine with AFS on the Internet.

#### AFS and Condor for Users

The *condor_shadow* process runs on the machine where you submitted your Condor jobs and performs all file system access for your jobs. Because this process isn't authenticated to AFS as the user who submitted the job, it will not normally be able to write any output. So, when you submit jobs, any directories where your job will be creating output files will need to be world writable (to non-authenticated AFS users). In addition, if your program writes to `stdout` or `stderr`, or you're using a user log for your jobs, those files will need to be in a directory that's world-writable.

Any input for your job, either the file you specify as input in your submit file, or any files your program opens explicitly, needs to be world-readable.

Some sites may have special AFS groups set up that can make this unauthenticated access to your files less scary. For example, there's supposed to be a way with AFS to grant access to any unauthenticated process on a given host. That way, you only have to grant write access to unauthenticated processes on your submit machine, instead of any unauthenticated process on the Internet. Similarly, unauthenticated read access could be granted only to processes running on your submit machine. Ask your AFS administrators about the existence of such AFS groups and details of how to use them.

The other solution to this problem is to just not use AFS at all. If you have disk space on your submit machine in a partition that is not on AFS, you can submit your jobs from there. While the *condor_shadow* is not authenticated to AFS, it does run with the effective UID of the user who submitted the jobs. So, on a local (or NFS) file system, the *condor_shadow* will be able to access your files normally, and you won't have to grant any special permissions to anyone other than yourself. If the Condor daemons are not started as root however, the shadow will not be able to run with your effective UID, and you'll have a similar problem as you would with files on AFS. See the section on "Running Condor as Non-Root" for details.

### 3.12.2 Configuring Condor for Multiple Platforms

Beginning with Condor version 6.0.1, a single, global configuration file may be used for all platforms in a Condor pool, with only platform-specific settings placed in separate files. This greatly simplifies administration of a heterogeneous pool by allowing changes of platform-independent, global settings in one place, instead of separately for each platform. This is made possible by treating the LOCAL_CONFIG_FILE configuration variable as a list of files, instead of a single file. Of course, this only helps when using a shared file system for the machines in the pool, so that multiple machines can actually share a single set of configuration files.

With multiple platforms, put all platform-independent settings (the vast majority) into the regular condor_config file, which would be shared by all platforms. This global file would be the one that is found with the CONDOR_CONFIG environment variable, the user condor's home directory, or /etc/condor/condor_config.

Then set the LOCAL_CONFIG_FILE configuration variable from that global configuration file to specify both a platform-specific configuration file and optionally, a local, machine-specific configuration file (this parameter is described in section 3.3.3 on "Condor-wide Configuration File Entries").

The order of file specification in the LOCAL_CONFIG_FILE configuration variable is important, because settings in files at the beginning of the list are overridden if the same settings occur in files later within the list. So, if specifying the platform-specific file and then the machine-specific file, settings in the machine-specific file would override those in the platform-specific file (as is likely desired).

#### Utilizing a Platform-Specific Configuration File

The name of platform-specific configuration files may be specified by using the ARCH and OPSYS parameters, as are defined automatically by Condor. For example, for Intel Linux machines, and Sparc Solaris 2.6 machines, the files ought to be named:

```
condor_config.INTEL.LINUX
condor_config.SUN4x.SOLARIS26
```

Then, assuming these three files are in the directory defined by the ETC configuration macro, and machine-specific configuration files are in the same directory, named by each machine's host name, the LOCAL_CONFIG_FILE configuration macro should be:

```
LOCAL_CONFIG_FILE = $(ETC)/condor_config.$(ARCH).$(OPSYS), \
                    $(ETC)/$(HOSTNAME).local
```

Alternatively, when using AFS, an "@sys link" may be used to specify the platform-specific configuration file, and let AFS resolve this link differently on different systems. For example, consider a soft link named condor_config.platform that points to condor_config.@sys. In this case, the files might be named:

```
condor_config.i386_linux2
condor_config.sun4x_56
condor_config.sgi_64
condor_config.platform -> condor_config.@sys
```

and the LOCAL_CONFIG_FILE configuration variable would be set to:

```
LOCAL_CONFIG_FILE = $(ETC)/condor_config.platform, \
                    $(ETC)/$(HOSTNAME).local
```

**Platform-Specific Configuration File Settings**

The configuration variables that are truly platform-specific are:

**RELEASE_DIR** Full path to to the installed Condor binaries. While the configuration files may be shared among different platforms, the binaries certainly cannot. Therefore, maintain separate release directories for each platform in the pool. See section 3.3.3 on "Condor-wide Configuration File Entries" for details.

**MAIL** The full path to the mail program. See section 3.3.3 on "Condor-wide Configuration File Entries" for details.

**CONSOLE_DEVICES** Which devices in /dev should be treated as console devices. See section 3.3.10 on "condor_startd Configuration File Entries" for details.

**DAEMON_LIST** Which daemons the *condor_master* should start up. The reason this setting is platform-specific is to distinguish the *condor_kbdd*. On Alphas running Digital Unix, it is needed, and it is not needed on other platforms. See section 3.3.9 on for details.

Reasonable defaults for all of these configuration variables will be found in the default configuration files inside a given platform's binary distribution (except the RELEASE_DIR, since the location of the Condor binaries and libraries is installation specific). With multiple platforms, use one of the condor_config files from either running *condor_install* or from the <release_dir>/etc/examples/condor_config.generic file, take these settings out, save them into a platform-specific file, and install the resulting platform-independent file as the global configuration file. Then, find the same settings from the configuration files for any other platforms to be set up, and put them in their own platform-specific files. Finally, set the LOCAL_CONFIG_FILE configuration variable to point to the appropriate platform-specific file, as described above.

Not even all of these configuration variables are necessarily going to be different. For example, if an installed mail program understands the **-s** option in /usr/local/bin/mail on all platforms, the MAIL macro may be set to that in the global configuration file, and not define it anywhere else. For a pool with only Digital Unix, the DAEMON_LIST will be the same for each, so there is no reason not to put that in the global configuration file.

**Other Uses for Platform-Specific Configuration Files**

It is certainly possible that an installation may want other configuration variables to be platform-specific as well. Perhaps a different policy is desired for one of the platforms. Perhaps different people should get the e-mail about problems with the different platforms. There is nothing hard-coded about any of this. What is shared and what should not shared is entirely configurable.

Since the `LOCAL_CONFIG_FILE` macro can be an arbitrary list of files, an installation can even break up the global, platform-independent settings into separate files. In fact, the global configuration file might only contain a definition for `LOCAL_CONFIG_FILE`, and all other configuration variables would be placed in separate files.

Different people may be given different permissions to change different Condor settings. For example, if a user is to be able to change certain settings, but nothing else, those settings may be placed in a file which was early in the `LOCAL_CONFIG_FILE` list, to give that user write permission on that file, then include all the other files after that one. In this way, if the user was trying to change settings she/he should not, they would simply be overridden.

This mechanism is quite flexible and powerful. For very specific configuration needs, they can probably be met by using file permissions, the `LOCAL_CONFIG_FILE` configuration variable, and imagination.

### 3.12.3 Full Installation of condor_compile

In order to take advantage of two major Condor features: checkpointing and remote system calls, users of the Condor system need to relink their binaries. Programs that are not relinked for Condor can run in Condor's "vanilla" universe just fine, however, they cannot checkpoint and migrate, or run on machines without a shared filesystem.

To relink your programs with Condor, we provide a special tool, *condor_compile*. As installed by default, *condor_compile* works with the following commands: *gcc*, *g++*, *g77*, *cc*, *acc*, *c89*, *CC*, *f77*, *fort77*, *ld*. On Solaris and Digital Unix, *f90* is also supported. See the *condor_compile*(1) man page for details on using *condor_compile*.

However, you can make *condor_compile* work transparently with all commands on your system whatsoever, including *make*.

The basic idea here is to replace the system linker (*ld*) with the Condor linker. Then, when a program is to be linked, the condor linker figures out whether this binary will be for Condor, or for a normal binary. If it is to be a normal compile, the old *ld* is called. If this binary is to be linked for condor, the script performs the necessary operations in order to prepare a binary that can be used with condor. In order to differentiate between normal builds and condor builds, the user simply places *condor_compile* before their build command, which sets the appropriate environment variable that lets the condor linker script know it needs to do its magic.

In order to perform this full installation of *condor_compile*, the following steps need to be taken:

1. Rename the system linker from ld to ld.real.
2. Copy the condor linker to the location of the previous ld.
3. Set the owner of the linker to root.
4. Set the permissions on the new linker to 755.

The actual commands that you must execute depend upon the system that you are on. The location of the system linker (*ld*), is as follows:

| Operating System | Location of ld (ld-path) |
|---|---|
| Linux | /usr/bin |
| Solaris 2.X | /usr/ccs/bin |
| OSF/1 (Digital Unix) | /usr/lib/cmplrs/cc |

On these platforms, issue the following commands (as root), where *ld-path* is replaced by the path to your system's *ld*.

```
mv /[ld-path]/ld /[ld-path]/ld.real
cp /usr/local/condor/lib/ld /[ld-path]/ld
chown root /[ld-path]/ld
chmod 755 /[ld-path]/ld
```

If you remove Condor from your system latter on, linking will continue to work, since the condor linker will always default to compiling normal binaries and simply call the real ld. In the interest of simplicity, it is recommended that you reverse the above changes by moving your ld.real linker back to it's former position as ld, overwriting the condor linker.

NOTE: If you ever upgrade your operating system after performing a full installation of *condor_compile*, you will probably have to re-do all the steps outlined above. Generally speaking, new versions or patches of an operating system might replace the system ld binary, which would undo the full installation of *condor_compile*.

### 3.12.4 Installing the *condor_kbdd*

The condor keyboard daemon (*condor_kbdd*) monitors X events on machines where the operating system does not provide a way of monitoring the idle time of the keyboard or mouse. In particular, this is necessary on Digital Unix machines.

NOTE: If you are running on Solaris, Linux, or HP/UX, you do not need to use the keyboard daemon.

Although great measures have been taken to make this daemon as robust as possible, the X window system was not designed to facilitate such a need, and thus is less then optimal on machines where many users log in and out on the console frequently.

In order to work with X authority, the system by which X authorizes processes to connect to X servers, the condor keyboard daemon needs to run with super user privileges. Currently, the daemon assumes that X uses the `HOME` environment variable in order to locate a file named `.Xauthority`, which contains keys necessary to connect to an X server. The keyboard daemon attempts to set this environment variable to various users home directories in order to gain a connection to the X server and monitor events. This may fail to work on your system, if you are using a non-standard approach. If the keyboard daemon is not allowed to attach to the X server, the state of a machine may be incorrectly set to idle when a user is, in fact, using the machine.

In some environments, the keyboard daemon will not be able to connect to the X server because the user currently logged into the system keeps their authentication token for using the X server in a place that no local user on the current machine can get to. This may be the case if you are running AFS and have the user's X authority file in an AFS home directory. There may also be cases where you cannot run the daemon with super user privileges because of political reasons, but you would still like to be able to monitor X activity. In these cases, you will need to change your XDM configuration in order to start up the keyboard daemon with the permissions of the currently logging in user. Although your situation may differ, if you are running X11R6.3, you will probably want to edit the files in /usr/X11R6/lib/X11/xdm. The Xsession file should have the keyboard daemon startup at the end, and the Xreset file should have the keyboard daemon shutdown. As of patch level 4 of Condor version 6.0, the keyboard daemon has some additional command line options to facilitate this. The -l option can be used to write the daemons log file to a place where the user running the daemon has permission to write a file. We recommend something akin to $HOME/.kbdd.log since this is a place where every user can write and won't get in the way. The -pidfile and -k options allow for easy shutdown of the daemon by storing the process id in a file. You will need to add lines to your XDM config that look something like this:

```
condor_kbdd -l $HOME/.kbdd.log -pidfile $HOME/.kbdd.pid
```

This will start the keyboard daemon as the user who is currently logging in and write the log to a file in the directory $HOME/.kbdd.log/. Also, this will save the process id of the daemon to /.kbdd.pid, so that when the user logs out, XDM can simply do a:

```
condor_kbdd -k $HOME/.kbdd.pid
```

This will shutdown the process recorded in /.kbdd.pid and exit.

To see how well the keyboard daemon is working on your system, review the log for the daemon and look for successful connections to the X server. If you see none, you may have a situation where the keyboard daemon is unable to connect to your machines X server. If this happens, please send mail to condor-admin@cs.wisc.edu and let us know about your situation.

### 3.12.5 Configuring The CondorView Server

The CondorView server is an alternate use of the *condor_collector* that logs information on disk, providing a persistent, historical database of pool state. This includes machine state, as well as the

state of jobs submitted by users. Historical information logging can be turned on or off, so you can install the CondorView collector without using up disk space for historical information if you do not want it.

The CondorView collector is a *condor_collector* that has been specially configured *and is running on a different machine from the main condor_collector*. Unfortunately, installing the CondorView collector on a separate host generates more network traffic (from all the duplicate updates that are sent from each machine in your pool to both collectors).

The following sections describe how to configure a machine to run a CondorView server and to configure your pool to send updates to it.

### Configuring a Machine to be a CondorView Server

To configure the CondorView collector, you have to add a few settings to the *local* configuration file of the chosen machine (a separate machine from the main *condor_collector*) to enable historical data collection. These settings are described in detail in the Condor Version 6.8.6 Administrator's Manual, in section 3.3.17 on page 192. A short explanation of the entries you must customize is provided below.

**POOL_HISTORY_DIR** This is the directory where historical data will be stored. This directory must be writable by whatever user the CondorView collector is running as (usually the user condor). There is a configurable limit to the maximum space required for all the files created by the CondorView server called (`POOL_HISTORY_MAX_STORAGE` ).

NOTE: This directory should be separate and different from the `spool` or `log` directories already set up for Condor. There are a few problems putting these files into either of those directories.

**KEEP_POOL_HISTORY** This is a boolean value that determines if the CondorView collector should store the historical information. It is false by default, which is why you must specify it as true in your local configuration file to enable data collection.

Once these settings are in place in the local configuration file for your CondorView server host, you must to create the directory you specified in `POOL_HISTORY_DIR` and make it writable by the user your CondorView collector is running as. This is the same user that owns the `CollectorLog` file in your `log` directory. The user is usually condor.

After you've configured the CondorView attributes, you must configure Condor to automatically start the CondorView server. You do this by adding `VIEW_SERVER` to the `DAEMON_LIST` on this machine and defining what `VIEW_SERVER` means. For example:

```
VIEW_SERVER = $(SBIN)/condor_collector
DAEMON_LIST = MASTER, STARTD, SCHEDD, VIEW_SERVER
```

For this change to take effect, you must re-start the *condor_master* on this host (which you can do with the *condor_restart* command, if you run the command from a machine with administrator

access to your pool. (See section 3.6.10 on page 282 for full details of IP/host-based security in Condor).

NOTE: Before you spawn the CondorView server by restarting your *condor_master*, you should make sure `CONDOR_VIEW_HOST` is defined in your configuration (as described in the following section).

#### Configuring a Pool to Report to the CondorView Server

For the CondorView server to function, you must configure your pool to send updates to it. You do this by configuring your existing *condor_collector* to forward its updates to the CondorView server. All the Condor daemons in the pool send their ClassAd updates to the regular *condor_collector*, which in turn will forward them onto the CondorView server.

You do this by defining the following setting in your configuration file:

```
CONDOR_VIEW_HOST = full.hostname
```

where `full.hostname` is the full hostname of the machine where you are running your CondorView collector.

You should place this setting in your global configuration file, since it should be the same for both the main *condor_collector* and the CondorView server. If you do not have a shared global configuration file for Condor, you should put the same value in the configuration files on both the main *condor_collector* and the CondorView server host.

Once this setting is in place, you can finally restart the *condor_master* at your CondorView server host (or spawn the *condor_master* if it is not yet running). Once the CondorView server is running, you can finally send a *condor_reconfig* to your main *condor_collector* for the change to take effect so it will begin forwarding updates.

### 3.12.6 Running Condor Jobs within a VMware or Xen Virtual Machine Environment

Condor jobs are formed from executables that are compiled to execute on specific platforms. This in turn restricts the machines within a Condor pool where a job may be executed. A Condor job may now be executed on a virtual machine system running VMware or Xen. This allows Windows executables to run on a Linux machine, and Linux executables to run on a Windows machine. These virtual machine systems exist for the Intel x86 architecture.

The term *virtual machine* used in this section is different than use of the term in other parts of Condor, due to the historical evolution of Condor. A virtual machine here describes the environment in which the outside operating system (called the host) emulates an inner operating system (called the inner virtual machine), such that an executable appears to run directly on the inner virtual machine.

In other parts of Condor, a virtual machine refers to the multiple CPUs of an SMP machine, or refers to a Java virtual machine.

Under Xen or VMware, Condor has the flexibility to run a job on either the host or the inner virtual machine, hence two platforms appear to exist on a single machine. Since two platforms are an illusion, Condor understands the illusion, allowing a Condor job to be execute on only one at a time.

**Installation and Configuration**

Condor must be separately installed, separately configured, and separately running on both the host and the inner virtual machine.

The configuration for the host specifies `VMP_VM_LIST` . This specifies host names or IP addresses of all inner virtual machines running on this host. An example configuration on the host machine:

```
VMP_VM_LIST = vmware1.domain.com, vmware2.domain.com
```

The configuration for each separate inner virtual machine specifies `VMP_HOST_MACHINE` . This specifies the host for the inner virtual machine. An example configuration on an inner virtual machine:

```
VMP_HOST_MACHINE = host.domain.com
```

Given this configuration, as well as communication between Condor daemons running on the host and on the inner virtual machine, the policy for when jobs may execute is set by Condor. While the host is executing a Condor job, the `START` policy on the inner virtual machine is overridden with `False`, so no Condor jobs will be started on the inner virtual machine. Conversely, while the inner virtual machine is executing a Condor job, the `START` policy on the host is overridden with `False`, so no Condor jobs will be started on the host.

The inner virtual machine is further provided with a new syntax for referring to the machine ClassAd attributes of its host. Any machine ClassAd attribute with a prefix of the string `HOST_` explicitly refers to the host's ClassAd attributes. The `START` policy on the inner virtual machine ought to use this syntax to avoid starting jobs when its host is too busy processing other items. An example configuration for `START` on an inner virtual machine:

```
START = ( (KeyboardIdle > 150 ) && ( HOST_KeyboardIdle > 150 ) \
        && ( LoadAvg <= 0.3 ) && ( HOST_TotalLoadAvg <= 0.3 ) )
```

### 3.12.7 Configuring The Startd for SMP Machines

This section describes how to configure the *condor_startd* for SMP (Symmetric Multi-Processor) machines. Beginning with Condor version 6.1, machines with more than one CPU can be configured

to run more than one job at a time. As always, owners of the resources have great flexibility in defining the policy under which multiple jobs may run, suspend, vacate, etc.

#### How Shared Resources are Represented to Condor

The way SMP machines are represented to the Condor system is that the shared resources are broken up into individual *virtual machines* (each virtual machine is called a VM). Each VM can be matched and claimed by users. Each VM is represented by an individual ClassAd (see the ClassAd reference, section 4.1, for details). In this way, each SMP machine will appear to the Condor system as a collection of separate VMs. As an example, an SMP machine named vulture.cs.wisc.edu would appear to Condor as the multiple machines, named vm1@vulture.cs.wisc.edu, vm2@vulture.cs.wisc.edu, vm3@vulture.cs.wisc.edu, and so on.

The way that the *condor_startd* breaks up the shared system resources into the different virtual machines is configurable. All shared system resources (like RAM, disk space, swap space, etc.) can either be divided evenly among all the virtual machines, with each CPU getting its own virtual machine, or you can define your own *virtual machine types*, so that resources can be unevenly partitioned. Regardless of the partioning scheme used, is important to remember the goal is to create a representative virtual machine ClassAd, to be used for matchmaking with jobs. Condor does not directly enforce virtual machine shared resource allocations, and jobs are free to oversubscribe to shared resources.

Consider an example where two VMs are each defined with 50%of available RAM. The resultant ClassAd for each VM will advertise one half the available RAM. Users may submit jobs with RAM requirements that match these VMs. However, jobs run on either VM are free to consume more than 50%of available RAM. Condor will not directly enforce a RAM utilization limit on either VM. If a shared resource enforcement capability is needed, it is possible to write a Startd policy that will evict a job that oversubscribes to shared resources, see section 3.12.7.

The following section gives details on how to configure Condor to divide the resources on an SMP machine into separate virtual machines.

#### Dividing System Resources in SMP Machines

This section describes the settings that allow you to define your own virtual machine types and to control how many virtual machines of each type are reported to Condor.

There are two main ways to go about partitioning an SMP machine:

**Define your own virtual machine types.** By defining your own types, you can specify what fraction of shared system resources (CPU, RAM, swap space and disk space) go to each virtual machine. Once you define your own types, you can control how many of each type are reported at any given time.

**Evenly divide all resources.** If you do not define your own types, the *condor_startd* will automat-

ically partition your machine into virtual machines for you. It will do so by placing a single CPU in each VM, and evenly dividing all shared resources among the VMs. With this default partitioning, you only specify how many VMs are reported at a time. By default, all VMs are reported to Condor.

Beginning with Condor version 6.1.6, the number of each type being reported can be changed at run-time, by issuing a reconfiguration command to the *condor_startd* daemon (sending a SIGHUP or using *condor_reconfig*). However, the definitions for the types themselves cannot be changed with reconfiguration. If you change any VM type definitions, you must use *condor_restart*

```
condor_restart -startd
```

for that change to take effect.

**Defining Virtual Machine Types**

To define your own virtual machine types, add configuration file parameters that list how much of each system resource you want in the given VM type. Do this by defining configuration variables of the form `VIRTUAL_MACHINE_TYPE_<N>` . The `<N>` represents an integer (for example, `VIRTUAL_MACHINE_TYPE_1`), which specifies the virtual machine type defined. This number is used to configure how many VMs of this type are advertised.

A type describes what share of the total system resources a given virtual machine has available to it.

The type can be defined by:

- A simple fraction, such as 1/4
- A simple percentage, such as 25%
- A comma-separated list of attributes, with a percentage, fraction, or value for each one.

A simple fraction or percentage causes an allocation of the total system resources. This includes the number of CPUs. A comma-separated list allows a fine-tuning of the amounts for specific attributes.

The attributes that specify the number of CPUs and the total amount of RAM in the SMP machine do not change. For these attributes, specify either absolute values or percentages of the total available amount. For example, in a machine with 128 Mbytes of RAM, all the following definitions result in the same allocation amount.

```
mem=64
mem=1/2
mem=50%
```

Other attributes are dynamic, such as disk space and swap space. For these, specify a percentage or fraction of the total value that is allocated to each VM, instead of specifying absolute values. As the total values of these resources change on your machine, each VM will take its fraction of the total and report that as its available amount.

The four attribute names are case insensitive when defining VM types. The first letter of the attribute name distinguishes between the attributes. The four attributes, with several examples of acceptable names for each are

- Cpus, C, c, cpu
- ram, RAM, MEMORY, memory, Mem, R, r, M, m
- disk, Disk, D, d
- swap, SWAP, S, s, VirtualMemory, V, v

As an example, consider a host of 4 CPUs and 256 megs of RAM. Here are valid example VM type definitions. Types 1-3 are all equivalent to each other, as are types 4-6

`VIRTUAL_MACHINE_TYPE_1` = cpus=2, ram=128, swap=25%, disk=1/2

`VIRTUAL_MACHINE_TYPE_2` = cpus=1/2, memory=128, virt=25%, disk=50%

`VIRTUAL_MACHINE_TYPE_3` = c=1/2, m=50%, v=1/4, disk=1/2

`VIRTUAL_MACHINE_TYPE_4` = c=25%, m=64, v=1/4, d=25%

`VIRTUAL_MACHINE_TYPE_5` = 25%

`VIRTUAL_MACHINE_TYPE_6` = 1/4

The number of virtual machines of each type is set with the configuration variable `NUM_VIRTUAL_MACHINES_TYPE_<N>` , where N is the type as given in the `VIRTUAL_MACHINE_TYPE_<N>`variable.

Note that it is possible to set the configuration variables such that they specify an impossible configuration. If this occurs, the *condor_startd* daemon fails after writing a message to its log attempting to indicate the configuration requirements that it could not implement.

**Evenly Divided Resources**

If you are not defining your own VM types, then all resources are divided equally among the VMs. The number of VMs within the SMP machine is the only attribute that needs to be defined. Its definition is accomplished by setting the configuration variable `NUM_VIRTUAL_MACHINES` to the integer number of machines desired. If variable `NUM_VIRTUAL_MACHINES` is not defined, it defaults to the number of CPUs within the SMP machine.

### Configuring Startd Policy for SMP Machines

Section 3.5 details the Startd Policy Configuration. This section continues the discussion with respect to SMP machines.

Each virtual machine within an SMP machine is treated as an independent machine, each with its own view of its machine state. There is a single set of policy expressions for the SMP machine as a whole. This policy may consider the VM state(s) in its expressions. This makes some policies easy to set, but it makes other policies difficult or impossible to set.

An easy policy to set configures how many of the virtual machines notice console or tty activity on the SMP as a whole. VMs that are not configured to notice any activity will report ConsoleIdle and KeyboardIdle times from when the *condor_startd* daemon was started, (plus a configurable number of seconds). With this, you can set up a multiple CPU machine with the default policy settings plus add that the keyboard and console noticed by only one virtual machine. Assuming a reasonable load average (see section 3.12.7 below on "Load Average for SMP Machines"), only the one virtual machine will suspend or vacate its job when the owner starts typing at their machine again. The rest of the virtual machines could be matched with jobs and leave them running, even while the user was interactively using the machine. If the default policy is used, all virtual machines notice tty and console activity and currently running jobs would suspend or preempt.

This example policy is controlled with the following configuration variables.

- `VIRTUAL_MACHINES_CONNECTED_TO_CONSOLE`
- `VIRTUAL_MACHINES_CONNECTED_TO_KEYBOARD`
- `DISCONNECTED_KEYBOARD_IDLE_BOOST`

These configuration variables are fully described in section 3.3.10 on page 168 which lists all the configuration file settings for the *condor_startd*.

The configuration of virtual machines allows each VM to advertise its own machine ClassAd. Yet, there is only one set of policy expressions for the SMP machine as a whole. This makes the implementation of certain types of policies impossible. While evaluating the state of one VM (within the SMP machine), the state of other VMs (again within the SMP machine) are not available. Decisions for one VM cannot be based on what other machines within the SMP are doing.

Specifically, the evaluation of a VM policy expression works in the following way.

1. The configuration file specifies policy expressions that are shared among all of the VMs on the SMP machine.
2. Each VM reads the configuration file and sets up its own machine ClassAd.
3. Each VM is now separate from the others. It has a different state, a different machine ClassAd, and if there is a job running, a separate job ad. Each VM periodically evaluates the policy expressions, changing its own state as necessary. This occurs independently of the other VMs

on the machine. So, if the *condor_startd* daemon is evaluating a policy expression on a specific VM, and the policy expression refers to `ProcID`, `Owner`, or any attribute from a job ad, it *always* refers to the ClassAd of the job running on the specific VM.

To set a different policy for the VMs within an SMP machine, a (`SUSPEND`) policy will be of the form

```
SUSPEND = ( (VirtualMachineID == 1) && (PolicyForVM1) ) || \
            ( (VirtualMachineID == 2) && (PolicyForVM2) )
```

where `(PolicyForVM1)` and `(PolicyForVM2)` are the desired expressions for each VM.

#### Load Average for SMP Machines

Most operating systems define the load average for an SMP machine as the total load on all CPUs. For example, if you have a 4-CPU machine with 3 CPU-bound processes running at the same time, the load would be 3.0 In Condor, we maintain this view of the total load average and publish it in all resource ClassAds as `TotalLoadAvg`.

Condor also provides a per-CPU load average for SMP machines. This nicely represents the model that each node on an SMP is a virtual machine, separate from the other nodes. All of the default, single-CPU policy expressions can be used directly on SMP machines, without modification, since the `LoadAvg` and `CondorLoadAvg` attributes are the per-virtual machine versions, not the total, SMP-wide versions.

The per-CPU load average on SMP machines is a Condor invention. No system call exists to ask the operating system for this value. Condor already computes the load average generated by Condor on each virtual machine. It does this by close monitoring of all processes spawned by any of the Condor daemons, even ones that are orphaned and then inherited by *init*. This Condor load average per virtual machine is reported as the attribute `CondorLoadAvg` in all resource ClassAds, and the total Condor load average for the entire machine is reported as `TotalCondorLoadAvg`. The total, system-wide load average for the entire machine is reported as `TotalLoadAvg`. Basically, Condor walks through all the virtual machines and assigns out portions of the total load average to each one. First, Condor assigns the known Condor load average to each node that is generating load. If there's any load average left in the total system load, it is considered an owner load. Any virtual machines Condor believes are in the Owner state (like ones that have keyboard activity), are the first to get assigned this owner load. Condor hands out owner load in increments of at most 1.0, so generally speaking, no virtual machine has a load average above 1.0. If Condor runs out of total load average before it runs out of virtual machines, all the remaining machines believe that they have no load average at all. If, instead, Condor runs out of virtual machines and it still has owner load remaining, Condor starts assigning that load to Condor nodes as well, giving individual nodes with a load average higher than 1.0.

**Debug logging in the SMP Startd**

This section describes how the *condor_startd* daemon handles its debugging messages for SMP machines. In general, a given log message will either be something that is machine-wide (like reporting the total system load average), or it will be specific to a given virtual machine. Any log entrees specific to a virtual machine have an extra header printed out in the entry: `vm#:`. So, for example, here's the output about system resources that are being gathered (with `D_FULLDEBUG` and `D_LOAD` turned on) on a 2-CPU machine with no Condor activity, and the keyboard connected to both virtual machines:

```
11/25 18:15 Swap space: 131064
11/25 18:15 number of kbytes available for (/home/condor/execute): 1345063
11/25 18:15 Looking up RESERVED_DISK parameter
11/25 18:15 Reserving 5120 kbytes for file system
11/25 18:15 Disk space: 1339943
11/25 18:15 Load avg: 0.340000 0.800000 1.170000
11/25 18:15 Idle Time: user= 0 , console= 4 seconds
11/25 18:15 SystemLoad: 0.340   TotalCondorLoad: 0.000  TotalOwnerLoad: 0.340
11/25 18:15 vm1: Idle time: Keyboard: 0        Console: 4
11/25 18:15 vm1: SystemLoad: 0.340  CondorLoad: 0.000  OwnerLoad: 0.340
11/25 18:15 vm2: Idle time: Keyboard: 0        Console: 4
11/25 18:15 vm2: SystemLoad: 0.000  CondorLoad: 0.000  OwnerLoad: 0.000
11/25 18:15 vm1: State: Owner          Activity: Idle
11/25 18:15 vm2: State: Owner          Activity: Idle
```

If, on the other hand, this machine only had one virtual machine connected to the keyboard and console, and the other VM was running a job, it might look something like this:

```
11/25 18:19 Load avg: 1.250000 0.910000 1.090000
11/25 18:19 Idle Time: user= 0 , console= 0 seconds
11/25 18:19 SystemLoad: 1.250   TotalCondorLoad: 0.996  TotalOwnerLoad: 0.254
11/25 18:19 vm1: Idle time: Keyboard: 0        Console: 0
11/25 18:19 vm1: SystemLoad: 0.254  CondorLoad: 0.000  OwnerLoad: 0.254
11/25 18:19 vm2: Idle time: Keyboard: 1496     Console: 1496
11/25 18:19 vm2: SystemLoad: 0.996  CondorLoad: 0.996  OwnerLoad: 0.000
11/25 18:19 vm1: State: Owner          Activity: Idle
11/25 18:19 vm2: State: Claimed        Activity: Busy
```

As you can see, shared system resources are printed without the header (like total swap space), and VM-specific messages (like the load average or state of each VM) get the special header appended.

**Configuring STARTD_EXPRS on a per-VM basis**

The STARTD_ATTRS (and legacy STARTD_EXPRS) settings can be configured on a per-VM basis. The *condor_startd* daemon builds the list of items to advertise by combining the lists in this order:

1. STARTD_ATTRS
2. STARTD_EXPRS
3. VMx_STARTD_ATTRS
4. VMx_STARTD_EXPRS

For example, consider the following configuration:

```
STARTD_EXPRS = favorite_color, favorite_season
VM1_STARTD_EXRS = favorite_movie
VM2_STARTD_EXPRS = favorite_song
```

This will result in the *condor_startd* ClassAd for VM1 defining values for `favorite_color`, `favorite_season`, and `favorite_movie`. VM2 will have values for `favorite_color`, `favorite_season`, and `favorite_song`.

Attributes themselves in the STARTD_EXPRS and STARTD_ATTRS list can also be defined on a per-VM basis. Here is another example:

```
favorite_color = "blue"
favorite_season = "spring"
STARTD_EXPRS = favorite_color, favorite_season
VM2_favorite_color = "green"
VM3_favorite_season = "summer"
```

For this example, the *condor_startd* ClassAds are

VM1:

```
favorite_color = "blue"
favorite_season = "spring"
```

VM2:

```
favorite_color = "green"
favorite_season = "spring"
```

VM3:

```
favorite_color = "blue"
favorite_season = "summer"
```

### 3.12.8 Condor's Dedicated Scheduling

Applications that require multiple resources, yet must not be preempted, are handled gracefully by Condor. Condor combines *opportunistic scheduling* and *dedicated scheduling* within a single system. *Opportunistic scheduling* involves placing a job on a non-dedicated resource under the assumption that the resource may not be available for the entire duration of the job. *Dedicated scheduling* assumes the constant availability of resources; it is assumed that the job will run to completion, without interruption.

To support applications needing dedicated resources, an administrator configures resources to be dedicated. These resources are controlled by a dedicated scheduler, a single machine within the pool that runs a *condor_schedd* daemon. There is no limit on the number of dedicated schedulers within a Condor pool. However, each dedicated resource may only be managed by a single dedicated scheduler. Running multiple dedicated schedulers within a single pool results in a fragmentation of dedicated resources. This can create a situation where jobs cannot run, because there are too few resource that may be allocated.

After a *condor_schedd* daemon has been selected as the dedicated scheduler for the pool and resources are configured to be dedicated, users submit parallel universe jobs (including MPI applications) through that *condor_schedd* daemon. When an idle parallel universe job is found in the queue, this dedicated scheduler performs its own scheduling algorithm to find and claim appropriate resources for the job. When a resource can no longer be used to serve a job that must not be preempted, the resource is allowed to run opportunistic jobs.

#### Selecting and Setting Up a Dedicated Scheduler

We recommend that you select a single machine within a Condor pool to act as the dedicated scheduler. This becomes the machine from upon which all users submit their parallel universe jobs. The perfect choice for the dedicated scheduler is the single, front-end machine for a dedicated cluster of compute nodes. For the pool without an obvious choice for a submit machine, choose a machine that all users can log into, as well as one that is likely to be up and running all the time. All of Condor's other resource requirements for a submit machine apply to this machine, such as having enough disk space in the spool directory to hold jobs. See section 3.2.2 on page 109 for details on these issues.

#### Configuration Examples for Dedicated Resources

Each machine may have its own policy for the execution of jobs. This policy is set by configuration. Each machine with aspects of its configuration that are dedicated identifies the dedicated scheduler. And, the ClassAd representing a job to be executed on one or more of these dedicated machines includes an identifying attribute. An example configuration file with the following various policy settings is `/etc/condor_config.local.dedicated.resource`.

Each dedicated machine defines the configuration variable `DedicatedScheduler` , which identifies the dedicated scheduler it is managed by. The local configuration file for any dedicated

resource contains a modified form of

```
DedicatedScheduler = "DedicatedScheduler@full.host.name"
STARTD_EXPRS = $(STARTD_EXPRS), DedicatedScheduler
```

Substitute the host name of the dedicated scheduler machine for the string "`full.host.name`".

If running personal Condor, the name of the scheduler includes the user name it was started as, so the configuration appears as:

```
DedicatedScheduler = "DedicatedScheduler@username@full.host.name"
STARTD_EXPRS = $(STARTD_EXPRS), DedicatedScheduler
```

All dedicated resources must have policy expressions which allow for jobs to always run, but not be preempted. The resource must also be configured to prefer jobs from the dedicated scheduler over all other jobs. Therefore, configuration gives the dedicated scheduler of choice the highest rank. It is worth noting that Condor puts no other requirements on a resource for it to be considered dedicated.

Job ClassAds from the dedicated scheduler contain the attribute `Scheduler`. dedicated scheduler. The attribute is defined by a string of the form

```
Scheduler = "DedicatedScheduler@full.host.name"
```

The host name of the dedicated scheduler substitutes for the string "`full.host.name`".

Different resources in the pool may have different dedicated policies by varying the local configuration.

**Policy Scenario: Machine Runs Only Jobs That Require Dedicated Resources** One possible scenario for the use of a dedicated resource is to only run jobs that require the dedicated resource. To enact this policy, the configure with the following expressions:

```
START     = Scheduler =?= $(DedicatedScheduler)
SUSPEND   = False
CONTINUE  = True
PREEMPT   = False
KILL      = False
WANT_SUSPEND   = False
WANT_VACATE    = False
RANK      = Scheduler =?= $(DedicatedScheduler)
```

The `START` expression specifies that a job with the `Scheduler` attribute must match the string corresponding `DedicatedScheduler` attribute in the machine ClassAd. The `RANK`

expression specifies that this same job (with the Scheduler attribute) has the highest rank. This prevents other jobs from preempting it based on user priorities. The rest of the expressions disable all of the *condor_startd* daemon's regular policies for evicting jobs when keyboard and CPU activity is discovered on the machine.

**Policy Scenario: Run Both Jobs That Do and Do Not Require Dedicated Resources** While the first example works nicely for jobs requiring dedicated resources, it can lead to poor utilization of the dedicated machines. A more sophisticated strategy allows the machines to run other jobs, when no jobs that require dedicated resources exist. The machine is configured to prefer jobs that require dedicated resources, but not prevent others from running.

To implement this, configure the machine as a dedicated resource (as above) modifying only the START expression:

```
START = True
```

**Policy Scenario: Adding Desk-Top Resources To The Mix** A third policy example allows all jobs. These desk-top machines use a preexisting START expression that takes the machine owner's usage into account for some jobs. The machine does not preempt jobs that must run on dedicated resources, while it will preempt other jobs based on a previously set policy. So, the default pool policy is used for starting and stopping jobs, while jobs that require a dedicated resource always start and are not preempted.

The START, SUSPEND, PREEMPT, and RANK policies are set in the global configuration. Locally, the configuration is modified to this hybrid policy by adding a second case.

```
SUSPEND    = Scheduler =!= $(DedicatedScheduler) && ($(SUSPEND))
PREEMPT    = Scheduler =!= $(DedicatedScheduler) && ($(PREEMPT))
RANK_FACTOR    = 1000000
RANK   = (Scheduler =?= $(DedicatedScheduler) * $(RANK_FACTOR)) \
               + $(RANK)
START  = (Scheduler =?= $(DedicatedScheduler)) || ($(START))
```

Define RANK_FACTOR to be a larger value than the maximum value possible for the existing rank expression. RANK is just a floating point value, so there is no harm in having a value that is very large.

**Policy Scenario: Parallel Scheduling Groups** In some parallel environments, machines are divided into groups, and jobs should not cross groups of machines – that is, all the nodes of a parallel job should be allocated to machines within the same group. The most common example is a pool of machines using infiniband switches. Each switch might connect 16 machines, and a pool might have 160 machines on 10 switches. If the infiniband switches are not routed to each other, each job must run on machines connected to the same switch.

The dedicated scheduler's parallel scheduling groups features supports jobs that must not cross group boundaries. Define a group by having each machine within a group set the configuration variable ParallelSchedulingGroup with a string that is a unique name for the group. The submit description file for a parallel universe job which must not cross group boundaries contains

```
+WantParallelSchedulingGroups = True
```

The dedicated scheduler enforces the allocation to within a group.

#### Preemption with Dedicated Jobs

The dedicated scheduler can optionally preempt running MPI jobs in favor of higher priority MPI jobs in its queue. Note that this is different from preemption in non-parallel universes, and MPI jobs cannot be preempted either by a machine's user pressing a key or by other means.

By default, the dedicated scheduler will never preempt running MPI jobs. Two configuration file items control dedicated preemption: `SCHEDD_PREEMPTION_REQUIREMENTS` and `SCHEDD_PREEMPTION_RANK` . These have no default value, so if either are not defined, preemption will never occur. `SCHEDD_PREEMPTION_REQUIREMENTS` must evaluate to `True` for a machine to be a candidate for this kind of preemption. If more machines are candidates for preemption than needed to satisfy a higher priority job, the machines are sorted by `SCHEDD_PREEMPTION_RANK`, and only the highest ranked machines are taken.

Note that preempting one node of a running MPI job requires killing the entire job on all of its nodes. So, when preemption happens, it may end up freeing more machines than strictly speaking are needed. Also, as Condor cannot produce checkpoints for MPI jobs, preempted jobs will be rerun, starting again from the beginning. Thus, the administrator should be careful when enabling dedicated preemption. The following example shows how to enable dedicated preemption.

```
STARTD_JOB_EXPRS = JobPrio
SCHEDD_PREEMPTION_REQUIREMENTS = (My.JobPrio < Target.JobPrio)
SCHEDD_PREEMPTION_RANK = 0.0
```

In this case, preemption is enabled by the user job priority. If a set of machines is running a job at user priority 5, and the user submits a new job at user priority 10, the running job will be preempted for the new job. The old job is put back in the queue, and will begin again from the beginning when assigned to a new set of machines.

#### Grouping dedicated nodes into parallel scheduling groups

In some parallel environments, machines are divided into groups, and jobs should not cross groups of machines – that is, all the nodes of a parallel job should be allocated to machines in the same group. The most common example is a pool of machine using infiniband switches. Each switch might connect 16 machines, and a pool might have 160 machines on 10 switches. If the infiniband switches are not routed to each other, each job must run on machines connected to the same switch. The dedicated scheduler's parallel scheduling groups features supports this operation.

Each startd must define which group it belongs to by setting the `ParallelSchedulingGroup` property in the config file, and advertising it into the machine classad. The value of this property is simply a string, which should be the same for all

startds in a given group. The property must be advertised in the startd job ad by appending `ParallelSchedulingGroup` into the `STARTD_EXPRS` configuration variable. Then, parallel jobs which want to be scheduled by group, declare this in their submit file by setting +WantParallelSchedulingGroups=True.

### 3.12.9 Configuring Condor for Running Backfill Jobs

Beginning with Condor version 6.7.17, Condor can be configured to run backfill jobs whenever the *condor_startd* has no other work to perform. These jobs are considered the lowest possible priority, but when machines would otherwise be idle, the resources can be put to good use.

Currently, Condor only supports using the Berkeley Open Infrastructure for Network Computing (BOINC) to provide the backfill jobs. More information about BOINC is available at http://boinc.berkeley.edu. Furthermore, Condor currently does not support backfill jobs on windows machines.

The rest of this section will provide an overview of how backfill jobs work in Condor, details for configuring the policy for when backfill jobs are started or killed, and details on how to configure Condor to spawn the BOINC client to perform the work.

#### Overview of Backfill jobs in Condor

Whenever a resource controlled by Condor is in the Unclaimed/Idle state, it is totally idle: neither the interactive user nor a Condor job is performing any work. Machines in this state can be configured to enter the *Backfill* state, which means the resource will attempt to perform a background computation to keep itself busy until other work arrives (either a user returning to use the machine interactively, or a normal Condor job). Once a resource enters the Backfill state, the *condor_startd* will attempt to spawn another program, called a *backfill client*, to launch and manage the backfill computation. When other work arrives, the *condor_startd* will kill the backfill client and clean up any processes it has spawned, freeing the machine resources for the new, higher priority task. More details about the different states a Condor resource can enter and all of the possible transitions between them are described in section 3.5 beginning on page 221, especially sections 3.5.6, 3.5.7, and 3.5.8.

At this point, the only backfill system supported by Condor is BOINC. The *condor_startd* has the ability to start and stop the BOINC client program at the appropriate times, but otherwise provides no additional services to configure the BOINC computations themselves. Future versions of Condor might provide additional functionality to make it easier to manage BOINC computations from within the Condor configuration settings. For now, the BOINC client must be manually installed and configured outside of Condor on each backfill-enabled machine.

### Defining the Backfill Policy

There are a small set of policy expressions that determine if a *condor_startd* will attempt to spawn backfill jobs at all, and if so, to control the transitions in to and out of the Backfill state. This section briefly lists these expressions. More detail can be found in section 3.3.10 on page 168.

**ENABLE_BACKFILL** A boolean value to determine if any backfill functionality should be used. The default is `False`.

**BACKFILL_SYSTEM** A string that defines what backfill system to use for spawning and managing backfill computations. Currently, the only supported value for this is `"BOINC"`.

**START_BACKFILL** A boolean expression to control if a Condor resource should start a backfill computation. This is only evaluated when the machine is in the Unclaimed/Idle state and the `ENABLE_BACKFILL` expression is `True`.

**EVICT_BACKFILL** A boolean expression that is evaluated whenever a Condor resource is in the Backfill state which, when `True`, indicates the machine should immediately kill the currently running backfill computation and return to the Owner state.

The following examples show some possible uses of these settings:

```
# Turn on backfill functionality, and use BOINC
ENABLE_BACKFILL = TRUE
BACKFILL_SYSTEM = BOINC

# Spawn a backfill job if we've been Unclaimed for more than 5
# minutes
START_BACKFILL = $(StateTimer) > (5 * $(MINUTE))

# Evict a backfill job if the machine is busy (based on keyboard
# activity or cpu load)
EVICT_BACKFILL = $(MachineBusy)
```

### Overview of the BOINC system

The BOINC system is a distributed computing environment for solving large scale scientific problems. A detailed explanation of this system is beyond the scope of this manual. Thorough documentation about BOINC is available at their website: http://boinc.berkeley.edu. However, a brief overview is provided here for sites interested in using BOINC with Condor to manage backfill jobs.

BOINC grew out of the relatively famous SETI@home computation, where volunteers would install special client software (in the form of a screen saver) that would contact a centralized server to download work units. Each work unit contained a set of radio telescope data and the computation tried to find patterns in the data, a sign of intelligent life elsewhere in the universe (hence the name: "Search for Extra Terrestrial Intelligence at home"). BOINC is developed by the Space Sciences Lab at the University of California, Berkeley, by the same people who created SETI@home. However,

instead of being tied to the specific radio telescope application, BOINC is a generic infrastructure where many different kinds of scientific computations can be solved. The current generation of SETI@home now runs on top of BOINC, along with various physics, biology, climatology, and other applications.

The basic computational model for BOINC and the original SETI@home is the same: volunteers install BOINC client software which will run whenever the machine would otherwise be idle. However, the BOINC installation on any given machine must be configured so that it knows what computations to work for (each computation is referred to as a *project* using BOINC's terminology), instead of always working on a hard coded computation. A given BOINC client can be configured to donate all of its cycles to a single project, or to split the cycles between projects so that, on average, the desired percentage of the computational power is allocated to each project. Once the client software (a program called the *boinc_client*) starts running, it will attempt to contact a centralized server for each project it has been configured to work for. The BOINC software will download the appropriate platform-specific application binary and some work units from the central server for each project. Whenever the client software completes a given work unit, it will once again attempt to connect to that project's central server to upload the results and download more work.

BOINC participants must register at the centralized server for each project they wish to donate cycles to. The process produces a unique identifier so that the work performed by a given client can be credited to a specific user. BOINC keeps track of the work units completed by each user, so that users providing the most cycles get the highest rankings (and therefore, bragging rights).

Because BOINC already handles the problems of distributing the application binaries for each scientific computation, the work units, and compiling the results, it is a perfect system for managing backfill computations in Condor. Many of the applications that run on top of BOINC do their own application-specific checkpointing, so even if the *boinc_client* is killed (for example, when a Condor job arrives at a machine, or if the interactive user returns) an entire work unit won't necessarily be lost.

#### Installing the BOINC client software

If a working installation of BOINC currently exists on machines where backfill is desired, skip the remainder of this section. Continue reading with the section titled "Configuring the BOINC client under Condor".

In Condor Version 6.8.6, the BOINC client software that actually spawns and manages the backfill computations (the *boinc_client*) must be manually downloaded, installed and configured outside of Condor. Hopefully in future versions, the Condor package will include the *boinc_client*, and there will be a way to automatically install and configure the BOINC software together with Condor.

The *boinc_client* executables can be obtained at one of the following locations:

**http://boinc.berkeley.edu/download.php** This is the official BOINC download site, which provides binaries for MacOS 10.3 or higher, Linux/x86, Solaris/SPARC and Windows/x86. From the download table, use the "Recommended version", and use the "Core client only

(command-line)" package when available.

**http://boinc.berkeley.edu/download_other.php** This page contains links to sites that distribute *boinc_client* binaries for other platforms beyond the officially supported ones.

Once the BOINC client software has been downloaded, the *boinc_client* binary should be placed in a location where the Condor daemons can use it. The path will be specified via a Condor configuration setting, `BOINC_Executable` , described below.

Additionally, a local directory on each machine should be created where the BOINC system can write files it needs. This directory must not be shared by multiple instances of the BOINC software, just like the `spool` or `execute` directories used by Condor. This location of this directory is defined using the `BOINC_InitialDir` macro, described below. The directory must be writable by whatever user the *boinc_client* will run as. This user is either the same as the user the Condor daemons are running as (if Condor is not running as root), or a user defined via the `BOINC_Owner` setting described below.

Finally, Condor administrators wishing to use BOINC for backfill jobs must create accounts at the various BOINC projects they want to donate cycles to. The details of this process vary from project to project. Beware that this step must be done manually, as the BOINC software spawned by Condor (the *boinc_client*) can not automatically register a user at a given project (unlike the more fancy GUI version of the BOINC client software which many users run as a screen saver). For example, to configure machines to perform work for the Einstein@home project (a physics experiment run by the University of Wisconsin at Milwaukee) Condor administrators should go to http://einstein.phys.uwm.edu/create_account_form.php, fill in the web form, and generate a new Einstein@home identity. This identity takes the form of a project URL (such as http://einstein.phys.uwm.edu) followed by an *account key*, which is a long string of letters and numbers that is used as a unique identifier. This URL and account key will be needed when configuring Condor to use BOINC for backfill computations (described in the next section).

#### Configuring the BOINC client under Condor

This section assumes that the BOINC client software has already been installed on a given machine, that the BOINC projects to join have been selected, and that a unique project account key has been created for each project. If any of these steps has not been completed, please read the previous section titled "Installing the BOINC client software"

Whenever the *condor_startd* decides to spawn the *boinc_client* to perform backfill computations (when `ENABLE_BACKFILL` is `True`, when the resource is in Unclaimed/Idle, and when the `START_BACKFILL` expression evaluates to `True`), it will spawn a *condor_starter* to directly launch and monitor the *boinc_client* program. This *condor_starter* is just like the one used to spawn normal Condor jobs. In fact, the argv[0] of the *boinc_client* will be renamed to "condor_exec", as described in section 2.15.1 on page 103.

The *condor_starter* for spawning the *boinc_client* reads values out of the Condor configuration files to define the job it should run, as opposed to getting these values from a job classified ad in the

case of a normal Condor job. All of the configuration settings to control things like the path to the *boinc_client* binary to use, the command-line arguments, the initial working directory, and so on, are prefixed with the string `"BOINC_"`. Each possible setting is described below:

Required settings:

**`BOINC_Executable`** The full path to the *boinc_client* binary to use.

**`BOINC_InitialDir`** The full path to the local directory where BOINC should run.

**`BOINC_Universe`** The Condor universe used for running the *boinc_client* program. This **must** be set to `"vanilla"` for BOINC to work under Condor.

**`BOINC_Owner`** What user the *boinc_client* program should be run as. This macro is only used if the Condor daemons are running as root. In this case, the *condor_starter* must be told what user identity to switch to before spawning the *boinc_client*. This can be any valid user on the local system, but it must have write permission in whatever directory is specified in `BOINC_InitialDir`).

Optional settings:

**`BOINC_Arguments`** Command-line arguments that should be passed to the *boinc_client* program. For example, one way to specify the BOINC project to join is to use the **–attach_project** argument to specify a project URL and account key. For example:

```
BOINC_Arguments = --attach_project http://einstein.phys.uwm.edu [account_key]
```

**`BOINC_Environment`** Environment variables that should be set for the *boinc_client*.

**`BOINC_Output`** Full path to the file where STDOUT from the *boinc_client* should be written. If this macro is not defined, STDOUT will be discarded.

**`BOINC_Error`** Full path to the file where STDERR from the *boinc_client* should be written. If this macro is not defined, STDERR will be discarded.

The following example shows one possible usage of these settings:

```
# Define a shared macro that can be used to define other settings.
# This directory must be manually created before attempting to run
# any backfill jobs.
BOINC_HOME = $(LOCAL_DIR)/boinc

# Path to the boinc_client to use, and required universe setting
BOINC_Executable = /usr/local/bin/boinc_client
BOINC_Universe = vanilla

# What initial working directory should BOINC use?
BOINC_InitialDir = $(BOINC_HOME)
```

```
# Save STDOUT and STDERR
BOINC_Output = $(BOINC_HOME)/boinc.out
BOINC_Error = $(BOINC_HOME)/boinc.err
```

If the Condor daemons reading this configuration are running as root, an additional macro must be defined:

```
# Specify the user that the boinc_client should run as:
BOINC_Owner = nobody
```

In this case, Condor would spawn the *boinc_client* as "**nobody**", so the directory specified in `$(BOINC_HOME)` would have to be writable by the "**nobody**" user.

A better choice would probably be to create a separate user account just for running BOINC jobs, so that the local BOINC installation is not writable by other processes running as "**nobody**". Alternatively, the `BOINC_Owner` could be set to "**daemon**".

**Attaching to a specific BOINC project**

There are a few ways to attach a Condor/BOINC installation to a given BOINC project:

- The **–attach_project** argument to the *boinc_client* program, defined via the `BOINC_Arguments` setting (described above). The *boinc_client* will only accept a single **–attach_project** argument, so this method can only be used to attach to one project.
- The *boinc_cmd* command-line tool can perform various BOINC administrative tasks, including attaching to a BOINC project. Using *boinc_cmd*, the appropriate argument to use is called **–project_attach**. Unfortunately, the *boinc_client* must be running for *boinc_cmd* to work, so this method can only be used once the Condor resource has entered the Backfill state and has spawned the *boinc_client*.
- Manually create account files in the local BOINC directory. Upon startup, the *boinc_client* will scan its local directory (the directory specified with `BOINC_InitialDir` ) for files of the form `account_[URL].xml`, for example, `account_einstein.phys.uwm.edu.xml`. Any files with a name that matches this convention will be read and processed. The contents of the file define the project URL and the authentication key. The format is:

  ```
  <account>
    <master_url>[URL]</master_url>
    <authenticator>[key]</authenticator>
  </account>
  ```

  For example:

  ```
  <account>
    <master_url>http://einstein.phys.uwm.edu</master_url>
    <authenticator>aaaa1111bbbb2222cccc3333</authenticator>
  </account>
  ```

(Of course, the `<authenticator>` tag would use the real authentication key returned when the account was created at a given project).

These account files can be copied to the local BOINC directory on all machines in a Condor pool, so administrators can either distribute them manually, or use symbolic links to point to a shared file system.

In the first two cases (using command-line arguments for *boinc_client* or running the *boinc_cmd* tool), BOINC will write out the resulting account file to the local BOINC directory on the machine, and then future invocations of the *boinc_client* will already be attached to the appropriate project(s). More information about participating in multiple BOINC projects can be found at http://boinc.berkeley.edu/multiple_projects.php.

## 3.13 Java Support Installation

Compiled Java programs may be executed (under Condor) on any execution site with a Java Virtual Machine (JVM). To do this, Condor must be informed of some details of the JVM installation.

Begin by installing a Java distribution according to the vendor's instructions. We have successfully used the Sun Java Developer's Kit, but any distribution should suffice. Your machine may have been delivered with a JVM already installed – installed code is frequently found in `/usr/bin/java`.

Condor's configuration includes the location of the installed JVM. Edit the configuration file. Modify the `JAVA` entry to point to the JVM binary, typically `/usr/bin/java`. Restart the *condor_startd* daemon on that host. For example,

```
% condor_restart -startd bluejay
```

The *condor_startd* daemon takes a few moments to exercise the Java capabilites of the *condor_starter*, query its properties, and then advertise the machine to the pool as Java-capable. If the set up succeeded, then *condor_status* will tell you the host is now Java-capable by printing the Java vendor and the version number:

```
% condor_status -java bluejay
```

After a suitable amount of time, if this command does not give any output, then the *condor_starter* is having difficulty executing the JVM. The exact cause of the problem depends on the details of the JVM, the local installation, and a variety of other factors. We can offer only limited advice on these matters, but here is an approach to solving the problem.

To reproduce the test that the *condor_starter* is attempting, try running the Java *condor_starter* directly. To find where the *condor_starter* is installed, run this command:

```
% condor_config_val STARTER
```

This command prints out the path to the *condor_starter*, perhaps something like this:

```
/usr/condor/sbin/condor_starter
```

Use this path to execute the *condor_starter* directly with the *-classad* argument. This tells the starter to run its tests and display its properties.

```
/usr/condor/sbin/condor_starter -classad
```

This command will display a short list of cryptic properties, such as:

```
IsDaemonCore = True
HasFileTransfer = True
HasMPI = True
CondorVersion = "$CondorVersion: 6.2$"
```

If the Java configuration is correct, there will also be a short list of Java properties, such as:

```
JavaVendor = "Sun Microsystems Inc."
JavaVersion = "1.2.2"
JavaMFlops = 9.279696
HasJava = True
```

If the Java installation is incorrect, then any error messages from the shell or Java will be printed on the error stream instead.

One identified difficulty occurs when the machine has a large quantity of physical RAM, and this quantity exceeds the Java limitations. This is a known problem for the Sun JVM. Condor appends the maximum amount of system RAM to the Java Maxheap Argument, and sometimes this value is larger than the JVM allows. The end result is that Condor believes that the JVM on the machine is faulty, resulting in nothing showing up as a result of executing the command `condor_status -java`.

The way to work around this particular problem is to modify the configuration file for those machines that may execute Java universe jobs. The `JAVA_MAXHEAP_ARGUMENT` macro is explicitly set to null in the configuration, to prevent Condor from appending the machine-specific, but too-big value. Then the Java Maxheap Argument is set (again, in the configuration) to the maximum value allowed for the JVM on that platform, using the `JAVA_EXTRA_ARGUMENTS` configuration variable. Note that the name of the switch that regulates the Java Maxheap Argument is different for different vendors' JVM.

The following is an example of the configuration fix for the Sun JVM:

```
# First set JAVA_MAXHEAP_ARGUMENT to null, to disable the default of max RAM
JAVA_MAXHEAP_ARGUMENT =
# Now set the argument with the Sun-specific maximum allowable value
JAVA_EXTRA_ARGUMENTS = -Xmx1906m
```

CHAPTER

# FOUR

# Miscellaneous Concepts

This chapter contains sections describing a variety of key Condor concepts that do not belong in other chapters.

ClassAds and the ClassAd language are presented.

Details of checkpoints are presented.

Description and useage of COD (Computing on Demand) extensions to Condor are presented.

The various APIs that Condor implements are described.

## 4.1 Condor's ClassAd Mechanism

ClassAds are a flexible mechanism for representing the characteristics and constraints of machines and jobs in the Condor system. ClassAds are used extensively in the Condor system to represent jobs, resources, submitters and other Condor daemons. An understanding of this mechanism is required to harness the full flexibility of the Condor system.

A ClassAd is is a set of uniquely named expressions. Each named expression is called an *attribute*. Figure 4.1 shows an example of a ClassAd with ten attributes.

ClassAd expressions look very much like expressions in C, and are composed of literals and attribute references composed with operators and functions. The difference between ClassAd expressions and C expressions arise from the fact that ClassAd expressions operate in a much more dynamic environment. For example, an expression from a machine's ClassAd may refer to an attribute in a job's ClassAd, such as `TARGET.Owner` in the above example. The value and type of the attribute is not known until the expression is evaluated in an environment which pairs a specific

```
MyType       = "Machine"
TargetType   = "Job"
Machine      = "froth.cs.wisc.edu"
Arch         = "INTEL"
OpSys        = "SOLARIS251"
Disk         = 35882
Memory       = 128
KeyboardIdle = 173
LoadAvg      = 0.1000
Requirements = TARGET.Owner=="smith" || LoadAvg<=0.3 && KeyboardIdle>15*60
```

Figure 4.1: An example ClassAd

job ClassAd with the machine ClassAd.

ClassAd expressions handle these uncertainties by defining all operators to be *total* operators, which means that they have well defined behavior regardless of supplied operands. This functionality is provided through two distinguished values, `UNDEFINED` and `ERROR`, and defining all operators so that they can operate on all possible values in the ClassAd system. For example, the multiplication operator which usually only operates on numbers, has a well defined behavior if supplied with values which are not meaningful to multiply. Thus, the expression `10 * "A string"` evaluates to the value `ERROR`. Most operators are *strict* with respect to `ERROR`, which means that they evaluate to `ERROR` if any of their operands are `ERROR`. Similarly, most operators are strict with respect to `UNDEFINED`.

### 4.1.1 Syntax

ClassAd expressions are formed by composing literals, attribute references and other subexpressions with operators and functions.

#### Literals

Literals in the ClassAd language may be of integer, real, string, undefined or error types. The syntax of these literals is as follows:

**Integer** A sequence of continuous digits (i.e., `[0-9]`). Additionally, the keywords `TRUE` and `FALSE` (case insensitive) are syntactic representations of the integers 1 and 0 respectively.

**Real** Two sequences of continuous digits separated by a period (i.e., `[0-9]+.[0-9]+`).

**String** A double quote character, followed by an list of characters terminated by a double quote character. A backslash character inside the string causes the following character to be considered as part of the string, irrespective of what that character is.

**Undefined** The keyword `UNDEFINED` (case insensitive) represents the `UNDEFINED` value.

**Error** The keyword `ERROR` (case insensitive) represents the `ERROR` value.

### Attributes

Every expression in a ClassAd is named by an *attribute name*. Together, the (name,expression) pair is called an *attribute*. An attributes may be referred to in other expressions through its attribute name.

Attribute names are sequences of alphabetic characters, digits and underscores, and may not begin with a digit. All characters in the name are significant, but case is *not* significant. Thus, `Memory`, `memory` and `MeMoRy` all refer to the same attribute.

An *attribute reference* consists of the name of the attribute being referenced, and an optional *scope resolution prefix*. The prefixes that may be used are `MY.` and `TARGET.`. The case used for these prefixes is *not* significant. The semantics of supplying a prefix are discussed in Section 4.1.2.

### Operators

The operators that may be used in ClassAd expressions are similar to those available in C. The available operators and their relative precedence is shown in figure 4.2. The operator with the highest

```
- (unary negation)   (high precedence)
*   /
+   - (addition, subtraction)
<   <=   >=   >
==  !=  =?=  =!=
&&
||                  (low precedence)
```

Figure 4.2: Relative precedence of ClassAd expression operators

precedence is the unary minus operator. The only operators which are unfamiliar are the `=?=` and `=!=` operators, which are discussed in Section 4.1.2.

### Predefined Functions

Any ClassAd expression may utilize predefined functions. Function names are case insensitive. Parameters to functions and a return value from a function may be typed (as given) or not. Nested or recursive function calls are allowed.

Here are descriptions of each of these predefined functions. The possible types are the same as itemized in in Section 4.1.1. Where the type may be any of these literal types, it is called out as `AnyType`. Where the type is `Integer`, but only returns the value 1 or 0 (implying `True` or `False`), it is called out as `Boolean`. The format of each function is given as

```
ReturnType FunctionName(ParameterType parameter1, ParameterType parameter2, ...)
```

Optional parameters are given within square brackets.

**AnyType ifThenElse(AnyType IfExpr,AnyType ThenExpr, AnyType ElseExpr)**
A conditional expression is described by `IfExpr`. The following defines return values, when `IfExpr` evaluates to

- `True`. Evaluate and return the value as given by `ThenExpr`.
- `False`. Evaluate and return the value as given by `ElseExpr`.
- `UNDEFINED`. Return the value `UNDEFINED`.
- `ERROR`. Return the value `ERROR`.
- `0.0`. Evaluate, and return the value as given by `ElseExpr`.
- non-`0.0` Real values. Evaluate, and return the value as given by `ThenExpr`.

Where `IfExpr` evaluates to give a value of type `String`, the function returns the value `ERROR`. The implementation uses lazy evaluation, so expressions are only evaluated as defined.

This function returns `ERROR` if other than exactly 3 arguments are given.

**Boolean isUndefined(AnyType Expr)** Returns `True`, if `Expr` evaluates to `UNDEFINED`. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

**Boolean isError(AnyType Expr)** Returns `True`, if `Expr` evaluates to `ERROR`. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

**Boolean isString(AnyType Expr)** Returns `True`, if the evaluation of `Expr` gives a value of type `String`. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

**Boolean isInteger(AnyType Expr)** Returns `True`, if the evaluation of `Expr` gives a value of type `Integer`. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

**Boolean isReal(AnyType Expr)** Returns `True`, if the evaluation of `Expr` gives a value of type `Real`. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

**Boolean isBoolean(AnyType Expr)** Returns `True`, if the evaluation of `Expr` gives the integer value 0 or 1. Returns `False` in all other cases.

This function returns `ERROR` if other than exactly 1 argument is given.

**Integer int(AnyType Expr)** Returns the integer value as defined by Expr. Where the type of the evaluated Expr is Real, the value is truncated (round towards zero) to an integer. Where the type of the evaluated Expr is String, the string is converted to an integer using a C-like atoi() function. When this result is not an integer, ERROR is returned. Where the evaluated Expr is ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

**Real real(AnyType Expr)** Returns the real value as defined by Expr. Where the type of the evaluated Expr is Integer, the return value is the converted integer. Where the type of the evaluated Expr is String, the string is converted to a real value using a C-like atof() function. When this result is not a real, ERROR is returned. Where the evaluated Expr is ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

**String string(AnyType Expr)** Returns the string that results from the evaluation of Expr. Converts a non-string value to a string. Where the evaluated Expr is ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

**Integer floor(AnyType Expr)** Returns the integer that results from the evaluation of Expr, where the type of the evaluated Expr is Integer. Where the type of the evaluated Expr is *not* Integer, function real(Expr) is called. Its return value is then used to return the largest magnitude integer that is not larger than the returned value. Where real(Expr) returns ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

**Integer ceiling(AnyType Expr)** Returns the integer that results from the evaluation of Expr, where the type of the evaluated Expr is Integer. Where the type of the evaluated Expr is *not* Integer, function real(Expr) is called. Its return value is then used to return the smallest magnitude integer that is not less than the returned value. Where real(Expr) returns ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

**Integer round(AnyType Expr)** Returns the integer that results from the evaluation of Expr, where the type of the evaluated Expr is Integer. Where the type of the evaluated Expr is *not* Integer, function real(Expr) is called. Its return value is then used to return the integer that results from a round-to-nearest rounding method. The nearest integer value to the return value is returned, except in the case of the value at the exact midpoint between two integer values. In this case, the even valued integer is returned. Where real(Expr) returns ERROR or UNDEFINED, or the integer value does not fit into 32 bits, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

**Integer random([ AnyType Expr ])** Where the optional argument Expr evaluates to type Integer or type Real (and called x), the return value is the integer or real r randomly

chosen from the interval `0 <= r < x`. With no argument, the return value is chosen with `random(1.0)`. Returns ERROR in all other cases.

This function returns ERROR if greater than 1 argument is given.

**String strcat(AnyType Expr1 [ , AnyType Expr2 ...])** Returns the string which is the concatenation of all arguments, where all arguments are converted to type `String` by function `string(Expr)`. Returns ERROR if any argument evaluates to UNDEFINED or ERROR.

**String substr(String s, Integer offset [ , Integer length ])** Returns the substring of `s`, from the position indicated by `offset`, with (optional) `length` characters. The first character within `s` is at offset 0. If the optional `length` argument is not present, the substring extends to the end of the string. If `offset` is negative, the value `(length - offset)` is used for the offset. If `length` is negative, an initial substring is computed, from the offset to the end of the string. Then, the absolute value of `length` characters are deleted from the right end of the initial substring. Further, where characters of this resulting substring lie outside the original string, the part that lies within the original string is returned. If the substring lies completely outside of the original string, the null string is returned.

This function returns ERROR if greater than 3 or less than 2 arguments are given.

**Integer strcmp(AnyType Expr1, AnyType Expr2)** Both arguments are converted to type `String` by function `string(Expr)`. The return value is an integer that will be

- less than 0, if `Expr1` is lexicographically less than `Expr2`
- equal to 0, if `Expr1` is lexicographically equal to `Expr2`
- greater than 0, if `Expr1` is lexicographically greater than `Expr2`

Case is significant in the comparison. Where either argument evaluates to ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than 2 arguments are given.

**Integer stricmp(AnyType Expr1, AnyType Expr2)** This function is the same as `strcmp`, except that letter case is *not* significant.

**String toUpper(AnyType Expr)** The single argument is converted to type `String` by function `string(Expr)`. The return value is this string, with all lower case letters converted to upper case. If the argument evaluates to ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if greater than 1 argument is given.

**String toLower(AnyType Expr)** The single argument is converted to type `String` by function `string(Expr)`. The return value is this string, with all upper case letters converted to lower case. If the argument evaluates to ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

**Integer size(AnyType Expr)** Returns the number of characters in the string, after calling function `string(Expr)`. If the argument evaluates to ERROR or UNDEFINED, ERROR is returned.

This function returns ERROR if other than exactly 1 argument is given.

For the following functions, a delimiter is represented by a string. Each character within the delimiter string delimits individual strings within a list of strings that is given by a single string. The default delimiter contains the comma and space characters. A string within the list is ended (delimited) by one or more characters within the delimiter string.

**Integer stringListSize(String list [ , String delimiter ])** Returns the number of elements in the string `list`, as delimited by the optional `delimiter` string. Returns ERROR if either argument is not a string.

This function returns ERROR if other than 1 or 2 arguments are given.

**Integer stringListSum(String list [ , String delimiter ])OR Real stringListSum(String** Sums and returns the sum of all items in the string `list`, as delimited by the optional `delimiter` string. If all items in the list are integers, the return value is also an integer. If any item in the list is a real value (noninteger), the return value is a real. If any item does not represent an integer or real value, the return value is ERROR.

**Real stringListAve(String list [ , String delimiter ])** Sums and returns the real-valued average of all items in the string `list`, as delimited by the optional `delimiter` string. If any item does not represent an integer or real value, the return value is ERROR. A list with 0 items (the empty list) returns the value 0.0.

**Integer stringListMin(String list [ , String delimiter ])OR Real stringListMin(String** Finds and returns the minimum value from all items in the string `list`, as delimited by the optional `delimiter` string. If all items in the list are integers, the return value is also an integer. If any item in the list is a real value (noninteger), the return value is a real. If any item does not represent an integer or real value, the return value is ERROR. A list with 0 items (the empty list) returns the value UNDEFINED.

**Integer stringListMax(String list [ , String delimiter ])OR Real stringListMax(String** Finds and returns the maximum value from all items in the string `list`, as delimited by the optional `delimiter` string. If all items in the list are integers, the return value is also an integer. If any item in the list is a real value (noninteger), the return value is a real. If any item does not represent an integer or real value, the return value is ERROR. A list with 0 items (the empty list) returns the value UNDEFINED.

**Boolean stringListMember(String x, String list [ , String delimiter ])** Returns TRUE if item `x` is in the string `list`, as delimited by the optional `delimiter` string. Returns FALSE if item `x` is not in the string `list`. Comparison is done with `strcmp()`. The return value is ERROR, if any of the arguments are not strings.

**Boolean stringListIMember(String x, String list [ , String delimiter ])** Same as `stringListMember()`, but comparison is done with `stricmp()`, so letter case is not relevant.

The following three functions utilize regular expressions as defined and supported by the PCRE library. See http://www.pcre.org for complete documentation of regular expressions.

The `options` argument to these functions is a string of special characters that modify the use of the regular expressions. Inclusion of characters other than these as options are ignored.

**`I` or `i`** Ignore letter case.

**`M` or `m`** Modifies the interpretation of the carat (^) and dollar sign ($) characters. The carat character matches the start of a string, as well as after each newline character. The dollar sign character matches before a newline character.

**`S` or `s`** The period matches any character, including the newline character.

**`X` or `x`** Ignore both white space and comments within the pattern. A comment is defined by starting with the pound sign (#) character, and continuing until the newline character.

**`Boolean regexp(String pattern, String target [ , String options ])`**
Returns `TRUE` if the string `target` is a regular expression as described by `pattern`. Returns `FALSE` otherwise. If any argument is not a string, or if `pattern` does not describe a valid regular expression, returns `ERROR`.

**`String regexps(String pattern, String target, String substitute, [ String options ])`**
The regular expression `pattern` is applied to `target`. If the string `target` is a regular expression as described by `pattern`, the string `substitute` is returned, with backslash expansion performed. The return value is `ERROR`, if any of the arguments are not strings.

**`Boolean stringListRegexpMember(String pattern, String list [ , String delimiter ] [ ,`**
Returns `TRUE` if any of the strings within the `list` is a regular expression as described by `pattern`. Returns `FALSE` otherwise. If any argument is not a string, or if `pattern` does not describe a valid regular expression, returns `ERROR`. To include the fourth (optional) argument `options`, a third argument of `delimiter` is required. A default value for a delimiter is " ,".

**`Integer time()`** Returns the current coordinated universal time, which is the same as the ClassAd attribute `CurrentTime`. This is the time, in seconds, since midnight of January 1, 1970.

**`String interval(Integer seconds)`** Uses `seconds` to return a string of the form `days+hh:mm:ss`. This represents an interval of time. Leading values that are zero are omitted from the string. For example, `seconds` of 67 becomes "1:07". A second example, `seconds` of 1472523 = 17*24*60*60 + 1*60*60 + 2*60 + 3, results in the string "17+1:02:03".

### 4.1.2 Evaluation Semantics

The ClassAd mechanism's primary purpose is for matching entities that supply constraints on candidate matches. The mechanism is therefore defined to carry out expression evaluations in the context of two ClassAds that are testing each other for a potential match. For example, the *condor_negotiator* evaluates the `Requirements` expressions of machine and job ClassAds to test if they can be matched. The semantics of evaluating such constraints is defined below.

#### Literals

Literals are self-evaluating, Thus, integer, string, real, undefined and error values evaluate to themselves.

#### Attribute References

Since the expression evaluation is being carried out in the context of two ClassAds, there is a potential for name space ambiguities. The following rules define the semantics of attribute references made by ad $A$ that is being evaluated in a context with another ad $B$:

1. If the reference is prefixed by a scope resolution prefix,
   - If the prefix is `MY.`, the attribute is looked up in ClassAd $A$. If the named attribute does not exist in $A$, the value of the reference is `UNDEFINED`. Otherwise, the value of the reference is the value of the expression bound to the attribute name.
   - Similarly, if the prefix is `TARGET.`, the attribute is looked up in ClassAd $B$. If the named attribute does not exist in $B$, the value of the reference is `UNDEFINED`. Otherwise, the value of the reference is the value of the expression bound to the attribute name.
2. If the reference is not prefixed by a scope resolution prefix,
   - If the attribute is defined in $A$, the value of the reference is the value of the expression bound to the attribute name in $A$.
   - Otherwise, if the attribute is defined in $B$, the value of the reference is the value of the expression bound to the attribute name in $B$.
   - Otherwise, if the attribute is defined in the ClassAd environment, the value from the environment is returned. This is a special environment, to be distinguished from the Unix environment. Currently, the only attribute of the environment is `CurrentTime`, which evaluates to the integer value returned by the system call `time(2)`.
   - Otherwise, the value of the reference is `UNDEFINED`.
3. Finally, if the reference refers to an expression that is itself in the process of being evaluated, there is a circular dependency in the evaluation. The value of the reference is `ERROR`.

### Operators

All operators in the ClassAd language are *total*, and thus have well defined behavior regardless of the supplied operands. Furthermore, most operators are *strict* with respect to ERROR and UNDEFINED, and thus evaluate to ERROR (or UNDEFINED) if either of their operands have these exceptional values.

- **Arithmetic operators:**
  1. The operators *, /, + and - operate arithmetically only on integers and reals.
  2. Arithmetic is carried out in the same type as both operands, and type promotions from integers to reals are performed if one operand is an integer and the other real.
  3. The operators are strict with respect to both UNDEFINED and ERROR.
  4. If either operand is not a numerical type, the value of the operation is ERROR.
- **Comparison operators:**
  1. The comparison operators ==, !=, <=, <, >= and > operate on integers, reals and strings.
  2. String comparisons are case insensitive for most operators. The only exceptions are the operators =?= and =!=, which do case sensitive comparisons assuming both sides are strings.
  3. Comparisons are carried out in the same type as both operands, and type promotions from integers to reals are performed if one operand is a real, and the other an integer. Strings may not be converted to any other type, so comparing a string and an integer or a string and a real results in ERROR.
  4. The operators ==, !=, <=, < and >= > are strict with respect to both UNDEFINED and ERROR.
  5. In addition, the operators =?= and =!= behave similar to == and !=, but are not strict. Semantically, the =?= tests if its operands are "identical," i.e., have the same type and the same value. For example, `10 == UNDEFINED` and `UNDEFINED == UNDEFINED` both evaluate to UNDEFINED, but `10 =?= UNDEFINED` and `UNDEFINED =?= UNDEFINED` evaluate to FALSE and TRUE respectively. The =!= operator test for the "is not identical to" condition.
- **Logical operators:**
  1. The logical operators && and || operate on integers and reals. The zero value of these types are considered FALSE and non-zero values TRUE.
  2. The operators are *not* strict, and exploit the "don't care" properties of the operators to squash UNDEFINED and ERROR values when possible. For example, `UNDEFINED && FALSE` evaluates to FALSE, but `UNDEFINED || FALSE` evaluates to UNDEFINED.
  3. Any string operand is equivalent to an ERROR operand for a logical operator. In other words, `TRUE && "foobar"` evaluates to ERROR.

### 4.1.3 ClassAds in the Condor System

The simplicity and flexibility of ClassAds is heavily exploited in the Condor system. ClassAds are not only used to represent machines and jobs in the Condor pool, but also other entities that exist in the pool such as checkpoint servers, submitters of jobs and master daemons. Since arbitrary expressions may be supplied and evaluated over these ads, users have a uniform and powerful mechanism to specify constraints over these ads. These constraints can take the form of `Requirements` expressions in resource and job ads, or queries over other ads.

#### Constraints and Preferences

The `requirements` and `rank` expressions within the submit description file are the mechanism by which users specify the constraints and preferences of jobs. For machines, the configuration determines both constraints and preferences of the machines.

For both machine and job, the `rank` expression specifies the desirability of the match (where higher numbers mean better matches). For example, a job ad may contain the following expressions:

```
Requirements = Arch=="SUN4u" && OpSys == "SOLARIS251"
Rank         = TARGET.Memory + TARGET.Mips
```

In this case, the job requires an UltraSparc computer running the Solaris 2.5.1 operating system. Among all such computers, the customer prefers those with large physical memories and high MIPS ratings. Since the `Rank` is a user-specified metric, *any* expression may be used to specify the perceived desirability of the match. The *condor_negotiator* daemon runs algorithms to deliver the best resource (as defined by the `rank` expression) while satisfying other required criteria.

Similarly, the machine may place constraints and preferences on the jobs that it will run by setting the machine's configuration. For example,

```
Friend        = Owner == "tannenba" || Owner == "wright"
ResearchGroup = Owner == "jbasney" || Owner == "raman"
Trusted       = Owner != "rival" && Owner != "riffraff"
START         = Trusted && ( ResearchGroup || LoadAvg < 0.3 &&
                      KeyboardIdle > 15*60 )
RANK          = Friend + ResearchGroup*10
```

The above policy states that the computer will never run jobs owned by users rival and riffraff, while the computer will always run a job submitted by members of the research group. Furthermore, jobs submitted by friends are preferred to other foreign jobs, and jobs submitted by the research group are preferred to jobs submitted by friends.

**Note:** Because of the dynamic nature of ClassAd expressions, there is no *a priori* notion of an integer-valued expression, a real-valued expression, etc. However, it is intuitive to think of the `Requirements` and `Rank` expressions as integer-valued and real-valued expressions, respectively. If the actual type of the expression is not of the expected type, the value is assumed to be zero.

### Querying with ClassAd Expressions

The flexibility of this system may also be used when querying ClassAds through the *condor_status* and *condor_q* tools which allow users to supply ClassAd constraint expressions from the command line.

For example, to find all computers which have had their keyboards idle for more than 20 minutes and have more than 100 MB of memory:

```
% condor_status -const 'KeyboardIdle > 20*60 && Memory > 100'

Name       Arch     OpSys        State      Activity   LoadAv Mem  ActvtyTime

amul.cs.wi SUN4u    SOLARIS251   Claimed    Busy       1.000  128   0+03:45:01
aura.cs.wi SUN4u    SOLARIS251   Claimed    Busy       1.000  128   0+00:15:01
balder.cs. INTEL    SOLARIS251   Claimed    Busy       1.000  1024  0+01:05:00
beatrice.c INTEL    SOLARIS251   Claimed    Busy       1.000  128   0+01:30:02
...
...
                     Machines Owner Claimed Unclaimed Matched Preempting

    SUN4u/SOLARIS251        3     0       3         0       0          0
    INTEL/SOLARIS251       21     0      21         0       0          0
    SUN4x/SOLARIS251        3     0       3         0       0          0
       INTEL/WINNT51        1     0       0         1       0          0
         INTEL/LINUX        1     0       1         0       0          0

               Total       29     0      28         1       0          0
```

Here is an example that utilizes a regular expression ClassAd function to list specific information. A file contains ClassAd information. *condor_advertise* is used to inject this information, and *condor_status* constrains the search with an expression that contains a ClassAd function.

```
% cat ad
MyType = "Generic"
FauxType = "DBMS"
Name = "random-test"
Machine = "f05.cs.wisc.edu"
MyAddress = "<128.105.149.105:34000>"
DaemonStartTime = 1153192799
UpdateSequenceNumber = 1

% condor_advertise UPDATE_AD_GENERIC ad

% condor_status -any -constraint 'FauxType=="DBMS" && regexp("random.*", Name, "i")'

MyType              TargetType          Name

Generic             None                random-test
```

Similar flexibility exists in querying job queues in the Condor system.

## 4.2 Condor's Checkpoint Mechanism

Checkpointing is taking a snapshot of the current state of a program in such a way that the program can be restarted from that state at a later time. Checkpointing gives the Condor scheduler the freedom to reconsider scheduling decisions through preemptive-resume scheduling. If the scheduler decides to no longer allocate a machine to a job (for example, when the owner of that machine returns), it can checkpoint the job and preempt it without losing the work the job has already accomplished. The job can be resumed later when the scheduler allocates it a new machine. Additionally, periodic checkpointing provides fault tolerance in Condor. Snapshots are taken periodically, and after an interruption in service the program can continue from the most recent snapshot.

Condor provides checkpointing services to single process jobs on a number of Unix platforms. To enable checkpointing, the user must link the program with the Condor system call library (`libcondorsyscall.a`), using the *condor_compile* command. This means that the user must have the object files or source code of the program to use Condor checkpointing. However, the checkpointing services provided by Condor are strictly optional. So, while there are some classes of jobs for which Condor does not provide checkpointing services, these jobs may still be submitted to Condor to take advantage of Condor's resource management functionality. (See section 2.4.1 on page 17 for a description of the classes of jobs for which Condor does not provide checkpointing services.)

Process checkpointing is implemented in the Condor system call library as a signal handler. When Condor sends a checkpoint signal to a process linked with this library, the provided signal handler writes the state of the process out to a file or a network socket. This state includes the contents of the process stack and data segments, all shared library code and data mapped into the process's address space, the state of all open files, and any signal handlers and pending signals. On restart, the process reads this state from the file, restoring the stack, shared library and data segments, file state, signal handlers, and pending signals. The checkpoint signal handler then returns to user code, which continues from where it left off when the checkpoint signal arrived.

Condor processes for which checkpointing is enabled perform a checkpoint when preempted from a machine. When a suitable replacement execution machine is found (of the same architecture and operating system), the process is restored on this new machine from the checkpoint, and computation is resumed from where it left off. Jobs that can not be checkpointed are preempted and restarted from the beginning.

Condor's periodic checkpointing provides fault tolerance. Condor pools are each configured with the `PERIODIC_CHECKPOINT` expression which controls when and how often jobs which can be checkpointed do periodic checkpoints (examples: never, every three hours, etc.). When the time for a periodic checkpoint occurs, the job suspends processing, performs the checkpoint, and immediately continues from where it left off. There is also a *condor_ckpt* command which allows the user to request that a Condor job immediately perform a periodic checkpoint.

In all cases, Condor jobs continue execution from the most recent complete checkpoint. If service is interrupted while a checkpoint is being performed, causing that checkpoint to fail, the process will restart from the previous checkpoint. Condor uses a commit style algorithm for writing checkpoints: a previous checkpoint is deleted only after a new complete checkpoint has been written

successfully.

In certain cases, checkpointing may be delayed until a more appropriate time. For example, a Condor job will defer a checkpoint request if it is communicating with another process over the network. When the network connection is closed, the checkpoint will occur.

The Condor checkpointing facility can also be used for any Unix process outside of the Condor batch environment. Standalone checkpointing is described in section 4.2.1.

Condor can produce and use compressed checkpoints. Configuration variables (detailed in section 3.3.12 control whether compression is used. The default is to not compress.

By default, a checkpoint is written to a file on the local disk of the machine where the job was submitted. A Condor pool can also be configured with a checkpoint server or servers that serve as a repository for checkpoints. (See section 3.8 on page 315.) When a host is configured to use a checkpoint server, jobs submitted on that machine write and read checkpoints to and from the server rather than the local disk of the submitting machine, taking the burden of storing checkpoint files off of the submitting machines and placing it instead on server machines (with disk space dedicated to the purpose of storing checkpoints).

### 4.2.1 Standalone Checkpointing

Using the Condor checkpoint library without the remote system call functionality and outside of the Condor system is known as standalone mode checkpointing.

To prepare a program for standalone checkpointing, simply use the *condor_compile* utility as for a standard Condor job, but do not use *condor_submit* – run the program normally from the command line. The checkpointing library will print a message to let you know that checkpointing is enabled and to inform you of the default name for the checkpoint image. The message is of the form:

```
Condor: Notice: Will checkpoint to program_name.ckpt
Condor: Notice: Remote system calls disabled.
```

To force the program to write a checkpoint image and stop, send it the SIGTSTP signal or press control-Z. To force the program to write a checkpoint image and continue executing, send it the SIGUSR2 signal.

To restart a program using a checkpoint, run the program with the argument *-_condor_restart* followed by the name of the checkpoint image file. As an example, if the program is called *P1* and the checkpoint is called `P1.ckpt`, use

```
P1 -_condor_restart P1.ckpt
```

### 4.2.2 Checkpoint Safety

Some programs have fundamental limitations that make them unsafe for checkpointing. For example, a program that both reads and writes a single file may enter an unexpected state. Here is an example of how this might happen.

1. Record a checkpoint image.
2. Read data from a file.
3. Write data to the same file.
4. Execution failure, so roll back to step 2.

In this example, the program would re-read data from the file, but instead of finding the original data, would see data created in the future, and yield unexpected results.

To prevent this sort of accident, Condor displays a warning if a file is used for both reading and writing. You can ignore or disable these warnings if you choose (see section 4.2.3,) but please understand that your program may compute incorrect results.

### 4.2.3 Checkpoint Warnings

Condor has warning messages in the case unexpected behaviors in your program. For example, if file `x` is opened for reading and writing, you will see:

```
Condor: Warning: READWRITE: File '/tmp/x' used for both reading and writing.
```

You may control how these messages are displayed with the `-_condor_warning` command-line argument. This argument accepts a warning category and a mode. The category describes a certain class of messages, such as READWRITE or ALL. The mode describes what to do with the category. It may be ON, OFF, or ONCE. If a category is ON, it is always displayed. If a category is OFF, it is never displayed. If a category is ONCE, it is displayed only once. To show all the available categories and modes, just use `-_condor_warning` with no arguments.

For example, to limit read/write warnings to one instance:

```
-_condor_warning READWRITE ONCE
```

To turn all ordinary notices off:

```
-_condor_warning NOTICE OFF
```

The same effect can be accomplished within a program by using the function `_condor_warning_config`, described in section 4.2.4.

### 4.2.4 Checkpoint Library Interface

A program need not be rewritten to take advantage of checkpointing. However, the checkpointing library provides several C entry points that allow for a program to control its own checkpointing behavior if needed.

- `void init_image_with_file_name( char *ckpt_file_name )`
  This function explicitly sets a file name to use when producing or using a checkpoint. `ckpt()` or `ckpt_and_exit()` must be called to produce the checkpoint, and `restart()` must be called to perform the actual restart.
- `void init_image_with_file_descriptor( int fd )`
  This function explicitly sets a file descriptor to use when producing or using a checkpoint. `ckpt()` or `ckpt_and_exit()` must be called to produce the checkpoint, and `restart()` must be called to perform the actual restart.
- `void ckpt()`
  This function causes a checkpoint image to be written to disk. The program will continue to execute. This is identical to sending the program a SIGUSR2 signal.
- `void ckpt_and_exit()`
  This function causes a checkpoint image to be writtent to disk. The program will then exit. This is identical to sending the program a SIGTSTP signal.
- `void restart()`
  This function causes the program to read the checkpoint image and to resume execution of the program from the point where the checkpoint was taken. This function does not return.
- `void _condor_ckpt_disable()`
  This function temporarily disables checkpointing. This can be handy if your program does something that is not checkpoint-safe. For example, if a program must not be interrupted while accessing a special file, call `_condor_ckpt_disable()`, access the file, and then call `_condor_ckpt_enable()`. Some program actions, such as opening a socket or a pipe, implicitly cause checkpointing to be disabled.
- `void _condor_ckpt_enable()`
  This function re-enables checkpointing after a call to `_condor_ckpt_disable()`. If a checkpointing signal arrived while checkpointing was disabled, the checkpoint will occur when this function is called. Disabling and enabling of checkpointing must occur in matched pairs. `_condor_ckpt_enable()` must be called once for every time that `_condor_ckpt_disable()` is called.
- `int _condor_warning_config( const char *kind, const char *mode )`
  This function controls what warnings are displayed by Condor. The `kind` and `mode` arguments are the same as for the `-_condor_warning` option described in section 4.2.3. This function returns true if the arguments are understood and accepted. Otherwise, it returns false.

- `extern int condor_compress_ckpt`
  Setting this variable to one causes checkpoint images to be compressed. Setting it to zero disables compression.

## 4.3 Computing On Demand (COD)

Computing On Demand (COD) extends Condor's high throughput computing abilities to include a method for running short-term jobs on instantly-available resources. Support for COD was added to Condor in version 6.5.2.

The motivation for COD extends Condor's job management to include interactive, compute-intensive jobs, giving these jobs immediate access to the compute power they need over a relatively short period of time. COD provides computing power *on demand*, switching predefined resources from working on Condor jobs to working on the COD jobs. These COD jobs (applications) cannot use the batch scheduling functionality of Condor, since the COD jobs require interactive response-time. Many of the applications that are well-suited to Condor's COD capabilities involve a cycle: application blocked on user input, computation burst to compute results, block again on user input, computation burst, etc. When the resources are not being used for the bursts of computation to service the application, they should continue to execute long-running batch jobs.

Here are examples of applications that may benefit from COD capability:

- A giant spreadsheet with a large number of highly complex formulas which take a lot of compute power to recalculate. The spreadsheet application (as a COD application) predefines a claim on resources within the Condor pool. When the user presses a `recalculate` button, the predefined Condor resources (nodes) work on the computation and send the results back to the master application providing the user interface and displaying the data. Ideally, while the user is entering new data or modifying formulas, these nodes work on non-COD jobs.
- A graphics rendering application that waits for user input to select an image to render. The rendering requires a huge burst of computation to produce the image. Examples are various Computer-Aided Design (CAD) tools, fractal rendering programs, and ray-tracing tools.
- Visualization tools for data mining.

The way Condor helps these kinds of applications is to provide an infrastructure to use Condor batch resources for the types of compute nodes described above. Condor does *NOT* provide tools to parallelize existing GUI applications. The COD functionality is an interface to allow these compute nodes to interact with long-running Condor batch jobs. The user provides both the compute node applications and the interactive master application that controls them. Condor only provides a mechanism to allow these interactive (and often parallelized) applications to seamlessly interact with the Condor batch system.

### 4.3.1 Overview of How COD Works

The resources of a Condor pool (nodes) run jobs. When a high-priority COD job appears at a node, the lower-priority (currently running) batch job is suspended. The COD job runs immediately, while the batch job remains suspended. When the COD job completes, the batch job instantly resumes execution.

Administratively, an interactive COD application puts claims on nodes. While the COD application does not need the nodes (to run the COD jobs), the claims are suspended, allowing batch jobs to run.

### 4.3.2 Authorizing Users to Create and Manage COD Claims

Claims on nodes are assigned to users. A user with a claim on a resource can then suspend and resume a COD job at will. This gives the user a great deal of power on the claimed resource, even if it is owned by another user. Because of this, it is essential that users allowed to claim COD resources can be trusted not to abuse this power. Users are authorized to have access to the privilege of creating and using a COD claim on a machine. This privilege is granted when the Condor administrator places a given user name in the `VALID_COD_USERS` list in the Condor configuration for the machine (usually in a local configuration file).

In addition, the tools to request and manage COD claims require that the user issuing the commands be authenticated. Use one of the strong authentication methods described in section 3.6.1 "Security Configuration" on page 261. If one of these methods cannot be used, then file system authentication may be used when directly logging in to that machine (to be claimed) and issuing the command locally.

### 4.3.3 Defining a COD Application

To run an application on a claimed COD resource, an authorized user defines characteristics of the application. Examples of characteristics are the executable or script to use, the directory to run the application in, command-line arguments, and files to use for standard input and output. COD users specify a ClassAd that describes these characteristics for their application. There are two ways for a user to define a COD application's ClassAd:

1. in the Condor configuration files of the COD resources
2. when they use the *condor_cod* command-line tool to launch the application itself

These two methods for defining the ClassAd can be used together. For example, the user can define some attributes in the configuration file, and only provide a few dynamically defined attributes with the *condor_cod* tool.

Regardless of how the COD application's ClassAd is defined, the application's executable and input data must be pre-staged at the node. This is a current limitation of Condor's support for COD that will eventually go away. For now, there is no mechanism to transfer files for a COD application, and all I/O must be performed locally or onto a network file system that is accessible by a node.

The following three sections detail defining the attributes. The first lists the attributes that can be used to define a COD application. The second describes how to define these attributes in a Condor configuration file. The third explains how to define these attributes using the *condor_cod* tool.

### COD Application Attributes

Attributes for a COD application are either required or optional. The following attributes are *required*:

**Cmd** This attribute defines the full path to the executable program to be run as a COD application. Since Condor does not currently provide any mechanism to transfer files on behalf of COD applications, this path should be a valid path on the machine where the application will be run. It is a string attribute, and must therefore be enclosed in quotation marks (`"`). There is no default.

**IWD** IWD is an acronym for Initial Working Directory. It defines the full path to the directory where a given COD application are to be run. Unless the application changes its current working directory, any relative path names used by the application will be relative to the IWD. If any other attributes that define file names (for example, `In`, `Out`, and so on) do not contain a full path, the `IWD` will automatically be pre-pended to those filenames. It is a string attribute, and must therefore be enclosed in quotation marks (`"`). There is no default.

**Owner** If the *condor_startd* daemon is executing as root on the resource where a COD application will run, the user must also define `Owner` to specify what user name the application will run as. (On Windows, the *condor_startd* daemon always runs as an Administrator service, which is equivalent to running as root on UNIX platforms). If the user specifies any COD application attributes with the *condor_cod_activate* command-line tool, the `Owner` attribute will be defined as the user name that ran *condor_cod_activate*. However, if the user defines all attributes of their COD application in the Condor configuration files, and does not define any attributes with the *condor_cod_activate* command-line tool (both methods are described below in more detail), there is no default and `Owner` must be specified in the configuration file. `Owner` must contain a valid user name on the given COD resource. It is a string attribute, and must therefore be enclosed in quotation marks (`"`).

The following list of attributes are *optional*:

**In** This string defines the path to the file on the COD resource that should be used as standard input (`stdin`) for the COD application. This file (and all parent directories) must be readable by whatever user the COD application will run as. If not specified, the default is `/dev/null`.

**Out** This string defines the path to the file on the COD resource that should be used as standard output (`stdout`) for the COD application. This file must be writable (and all parent directories readable) by whatever user the COD application will run as. If not specified, the default is `/dev/null`. It is a string attribute, and must therefore be enclosed in quotation marks (").

**Err** This string defines the path to the file on the COD resource that should be used as standard error (`stderr`) for the COD application. This file must be writable (and all parent directories readable) by whatever user the COD application will run as. If not specified, the default is `/dev/null`. It is a string attribute, and must therefore be enclosed in quotation marks (").

**Env** This string defines environment variables to set for a given COD application. Each environment variable has the form `NAME=value`. Multiple variables are delimited with a semicolon. An example: `Env = "PATH=/usr/local/bin:/usr/bin;TERM=vt100"` It is a string attribute, and must therefore be enclosed in quotation marks (").

**Args** This string attribute defines the list of arguments to be supplied to the program on the command-line. The arguments are delimited (separated) by space characters. There is no default. If the `JobUniverse` corresponds to the Java universe, the first argument must be the name of the class containing `main`. It is a string attribute, and must therefore be enclosed in quotation marks (").

**JobUniverse** This attribute defines what Condor job universe to use for the given COD application. At this point, the only supported universes are vanilla and Java. This attribute must be an integer, with vanilla using the value 5, and Java the value 10. If `JobUniverse` is not specified, the vanilla universe is used by default. For more information about the Condor job universes, see section 2.4.1 on page 16.

**JarFiles** This string attribute is only used if `JobUniverse` is 10 (the Java universe). If a given COD application is a Java program, specify the JAR files that the program requires with this attribute. There is no default. It is a string attribute, and must therefore be enclosed in quotation marks ("). Multiple file names may be delimited with either commas or whitespace characters, and therefore, file names can not contain spaces.

**KillSig** This attribute specifies what signal should be sent whenever the Condor system needs to gracefully shutdown the COD application. It can either be specified as a string containing the signal name (for example `KillSig = "SIGQUIT"`), or as an integer (`KillSig = 3`) The default is to use SIGTERM.

**StarterUserLog** This string specifies a file name for a log file that the *condor_starter* daemon can write with entries for relevant events in the life of a given COD application. It is similar to the UserLog file specified for regular Condor jobs with the `Log` setting in a submit description file. However, certain attributes that are placed in the regular UserLog file do not make sense in the COD environment, and are therefore omitted. The default is not to write this log file. It is a string attribute, and must therefore be enclosed in quotation marks (").

**StarterUserLogUseXML** If the `StarterUserLog` attribute is defined, the default format is a human-readable format. However, Condor can write out this log in an XML representation, instead. To enable the XML format for this UserLog, the `StarterUserLogUseXML` boolean is set to `TRUE`. The default if not specified is `FALSE`.

NOTE: If any path attribute (`Cmd`, `In`, `Out`,`Err`, `StarterUserLog`) is not a full path name, Condor automatically prepends the value of `IWD`.

The final set of attributes define an identification for a COD application. The job ID is made up of both the `ClusterId` and `ProcId` attributes (as described below). This job ID is similar to the job ID that is created whenever a regular Condor batch job is submitted. For regular Condor batch jobs, the job ID is assigned automatically by the *condor_schedd* whenever a new job is submitted into the persistent job queue. However, since there is no persistent job queue for COD, the usual mechanism to identify the jobs does not exist. Moreover, commands that require the job ID for batch jobs such as *condor_q* and *condor_rm* do not exist for COD. Instead, the claim ID is the unique identifier for COD jobs and COD-related commands.

When using COD, the job ID is only used to identify the job in various log messages and in the COD-specific output of *condor_status*. The COD job ID is part of the information included in all events written to the `StarterUserLog` regarding a given job. The COD job ID is also used in the Condor debugging logs described in section 3.3.4 on page 151 For example, in the *condor_starter* daemon's log file for COD jobs (called `StarterLog.cod` by default) or in the *condor_startd* daemon's log file (called `StartLog` by default).

These COD IDs are optional. The job ID is useful to define where it helps a user with accounting or debugging of their own application. In this case, it is the user's responsibility to ensure uniqueness, if so desired.

**`ClusterId`** This integer defines the cluster identifier for a COD job. The default value is 1. The `ClusterId` can also be defined with the *condor_cod_activate* command-line tool using the **-cluster** option.

**`ProcId`** This integer defines the process identifier (within a cluster) for a COD job. The default value is 0. The `ProcId` can also be defined with the *condor_cod_activate* command-line tool using the **-cluster** option.

NOTE: The cluster and proc identifiers can also be specified as command-line arguments to the *condor_cod_activate* tool when spawning a given COD application. See section 4.3.4 below for details on using *condor_cod_activate*.

#### Defining Attributes in the Condor Configuration Files

To define COD attributes in the Condor configuration file for a given application, the user selects a keyword to uniquely name ClassAd attributes of the application. This case-insensitive keyword is used as a prefix for the various configuration file attribute names. When a user wishes to spawn a given application, the keyword is given as an argument to the *condor_cod* tool and the keyword is used at the remote COD resource to find attributes which define the application.

Any of the ClassAd attributes described in the previous section can be specified in the configuration file with the keyword prefix followed by an underscore character (`"_"`).

For example, if the user's keyword for a given fractal generation application is "FractGen", the resulting entries in the Condor configuration file may appear as:

```
FractGen_Cmd = "/usr/local/bin/fractgen"
FractGen_Iwd = "/tmp/cod-fractgen"
FractGen_Out = "/tmp/cod-fractgen/output"
FractGen_Err = "/tmp/cod-fractgen/error"
FractGen_Args = "mandelbrot -0.65865,-0.56254 -0.45865,-0.71254"
```

In this example, the executable may create other files. The `Out` and `Err` attributes specified in the configuration file are only for standard output and standard error redirection.

When the user wishes to spawn an instance of this application, they use the **-keyword** option of **FractGen** *i*n the command-line of the *condor_cod_activate* command.

NOTE: If a user is defining all attributes of their COD application in the Condor configuration files, and the *condor_startd* daemon on the COD resource they are using is running as root, the user must also define `Owner` to be the user that the COD application should run as (see section 4.3.3 above).

#### Defining Attributes with the *condor_cod* Tool

COD users may define attributes dynamically (at the time they spawn a COD application). In this case, the user writes the ClassAd attributes into a file, and the file name is passed to the *condor_cod_activate* tool using the **-jobad** command-line option. These attributes are read by the *condor_cod* tool and passed through the system onto the *condor_starter* daemon which spawns the COD application. If the file name given is `-`, the *condor_cod* tool will read from standard input (`stdin`).

Users should not add a keyword prefix when defining attributes with the *condor_cod_activate* tool. The attribute names can be used in the file directly.

WARNING: The current syntax for this file is not the same as the syntax in the file used with *condor_submit*.

NOTE: Users should not define the `Owner` attribute when using *condor_cod_activate* on the command line, since Condor will automatically insert the correct value based on what user runs the *condor_cod_activate* command and how that user authenticates to the COD resource. If a user defines an attribute that does not match the authenticated identity, Condor treats this case as an error, and it will fail to launch the application.

### 4.3.4 Managing COD Resource Claims

Separate commands are provided by Condor to manage COD claims on batch resources. Once created, each COD claim has a unique identifying string, called the claim ID. Most commands

require a claim ID to specify which claim you wish to act on. These commands are the means by which COD applications interact with the rest of the Condor system. They should be issued by the controller application to manage its compute nodes. Here is a list of the commands:

**Request** Create a new COD claim on a given resource.

**Activate** Spawn a specific application on a specific COD claim.

**Suspend** Suspend a running application within a specific COD claim.

**Resume** Resume a suspended application on a specific COD claim.

**Deactivate** Shut down an application, but hold onto the COD claim for future use.

**Release** Destroy a specific COD claim, and shut down any job that is currently running on it.

To issue these commands, a user or application invokes the *condor_cod* tool. A command may be specified as the first argument to this tool, as

```
condor_cod request -name c02.cs.wisc.edu
```

or the *condor_cod* tool can be installed in such a way that the same binary is used for a set of names, as

```
condor_cod_request -name c02.cs.wisc.edu
```

Other than the command name itself (which must be included in full) additional options supported by each tool can be abbreviated to the shortest unambiguous value. For example, **-name** can also be specified as **-n**. However, for a command like *condor_cod_activate* that supports both **-classad** and **-cluster**, the user must use at least **-cla** or **-clu**. If the user specifies an ambiguous option, the *condor_cod* tool will exit with an error message.

In addition, there is now a **-cod** option to *condor_status*.

The following sections describe each option in greater detail.

#### Request

A user must be granted authorization to a create COD claims on a specific machine. In addition, when the user uses these COD claims, the application binary or script they wish to run (and any input data) must be pre-staged on the machine. Therefore, a user cannot simply request a COD claim at random.

The user specifies the resource on which to make a COD claim. This is accomplished by specifying the name of the *condor_startd* daemon desired by invoking *condor_cod_request* with the **-name** option and the resource name (usually the host name). For example:

```
condor_cod_request -name c02.cs.wisc.edu
```

If the *condor_startd* daemon desired belongs to a different Condor pool than the one where executing the COD commands, use the **-pool** option to provide the name of the central manager machine of the other pool. For example:

```
condor_cod_request -name c02.cs.wisc.edu -pool condor.cs.wisc.edu
```

An alternative is to provide the IP address and port number where the *condor_startd* daemon is listening with the **-addr** option. This information can be found in the *condor_startd* ClassAd as the attribute `StartdIpAddr` or by reading the log file when the *condor_startd* first starts up. For example:

```
condor_cod_request -addr "<128.105.146.102:40967>"
```

If neither **-name** or **-addr** are specified, *condor_cod_request* attempts to connect to the *condor_startd* daemon running on the local machine (where the request command was issued).

If the *condor_startd* daemon to be used for the COD claim is an SMP machine and has multiple virtual machines, specify which resource on the machine to use for COD by providing the full name of the resource, not just the host name. For example:

```
condor_cod_request -name vm2@c02.cs.wisc.edu
```

A constraint on what virtual machine is desired may be provided, instead of specifying it by name. For example, to run on machine c02.cs.wisc.edu, not caring which virtual machine is used, so long as it the machine is not currently running a job, use something like:

```
condor_cod_request -name c02.cs.wisc.edu -requirements 'State!="Claimed"'
```

In general, be careful with shell quoting issues, so that your shell is not confused by the ClassAd expression syntax (in particular if the expression includes a string). The safest method is to enclose any requirement expression within single quote marks (as shown above).

Once a given *condor_startd* daemon has been contacted to request a new COD claim, the *condor_startd* daemon checks for proper authorization of the user issuing the command. If the user has the authority, and the *condor_startd* daemon finds a resource that matches any given requirements, the *condor_startd* daemon creates a new COD claim and gives it a unique identifier, the claim ID. This ID is used to identify COD claims when using other commands. If *condor_cod_request* succeeds, the claim ID for the new claim is printed out to the screen. All other commands to manage this claim require the claim ID to be provided as a command-line option.

When the *condor_startd* daemon assigns a COD claim, the ClassAd describing the resource is returned to the user that requested the claim. This ClassAd is a snap-shot of the output of `condor_status -long` for the given machine. If *condor_cod_request* is invoked with the

**-classad** option (which takes a file name as an argument), this ClassAd will be written out to the given file. Otherwise, the ClassAd is printed to the screen. The only essential piece of information in this ClassAd is the Claim ID, so that is printed to the screen, even if the whole ClassAd is also being written to a file.

The claim ID as given after listing the machine ClassAd appears as this example:

```
ID of new claim is: "<128.105.121.21:49973>#1073352104#4"
```

When using this claim ID in further commands, include the quote marks as well as all the characters in between the quote marks.

NOTE: Once a COD claim is created, there is no persistent record of it kept by the *condor_startd* daemon. So, if the *condor_startd* daemon is restarted for any reason, all existing COD claims will be destroyed and the new *condor_startd* daemon will not recognize any attempts to use the previous claims.

### Activate

Once a user has created a valid COD claim and has the claim ID, the next step is to spawn a COD job using the claim. The way to do this is to activate the claim, using the *condor_cod_activate* command. Once a COD application is active on a COD claim, the COD claim will move into the `Running` state, and any batch Condor job on the same resource will be suspended. Whenever the COD application is inactive (either suspended, removed from the machine, or if it exits on its own), the state of the COD claim changes. The new state depends on why the application became inactive. The batch Condor job then resumes.

To activate a COD claim, first define attributes about the job to be run in either the local configuration of the COD resource, or in a separate file as described in this manual section. Invoke the *condor_cod_activate* command to launch a specific instance of the job on a given COD claim ID. The options given to *condor_cod_activate* vary depending on if the job attributes are defined in the configuration file or are passed via a file to the *condor_cod_activate* tool itself. However, the **-id** option is always required by *condor_cod_activate*, and this option should be followed by a COD claim ID that the user acquired via *condor_cod_request*.

If the application is defined in the configuration files for the COD resource, the user provides the keyword (described in section 4.3.3) that uniquely identifies the application's configuration attributes. To continue the example from that section, the user would spawn their job by specifying `-keyword FractGen`, for example:

```
condor_cod_activate -id "<claim_id>" -keyword FractGen
```

Substitute the `<claim_id>` with the valid Cod Claim Id. Using the same example as given above, this example would be:

```
condor_cod_activate -id "<128.105.121.21:49973>#1073352104#4" -keyword FractGen
```

If the job attributes are placed into a file to be passed to the *condor_cod_activate* tool, the user must provide the name of the file using the **-jobad** option. For example, if the job attributes were defined in a file named `cod-fractgen.txt`, the user spawns the job using the command:

```
condor_cod_activate -id "<claim_id>" -jobad cod-fractgen.txt
```

Alternatively, if the filename specified with **-jobad** is `-`, the *condor_cod_activate* tool reads the job ClassAd from standard input (`stdin`).

Regardless of how the job attributes are defined, there are other options that *condor_cod_activate* accepts. These options specify the job ID for the application to be run. The job ID can either be specified in the job's ClassAd, or it can be specified on the command line to *condor_cod_activate*. These options are **-cluster** and **-proc**. For example, to launch a COD job with keyword `foo` as cluster 23, proc 5, or 23.5, the user invokes:

```
condor_cod_activate -id "<claim_id>" -key foo -cluster 23 -proc 5
```

The **-cluster** and **-proc** arguments are optional, since the job ID is not required for COD. If not specified, the job ID defaults to `1.0`.

#### Suspend

Once a COD application has been activated with *condor_cod_activate* and is running on a COD resource, it may be temporarily suspended using *condor_cod_suspend*. In this case, the claim state becomes `Suspended`. Once a given COD job is suspended, if there are no other running COD jobs on the resource, a Condor batch job can use the resource. By suspending the COD application, the batch job is allowed to run. If a resource is idle when a COD application is first spawned, suspension of the COD job makes the batch resource available for use in the Condor system. Therefore, whenever a COD application has no work to perform, it should be suspended to prevent the resource from being wasted.

The interface of *condor_cod_suspend* supports the single option **-id**, to specify the COD claim ID to be suspended. For example:

```
condor_cod_suspend -id "<claim_id>"
```

If the user attempts to suspend a COD job that is not running, *condor_cod_suspend* exits with an error message. The COD job may not be running because it is already suspended or because the job was never spawned on the given COD claim in the first place.

#### Resume

Once a COD application has been suspended with *condor_cod_suspend*, it can be resumed using *condor_cod_resume*. In this case, the claim state returns to `Running`. If there is a regular batch job

running on the same resource, it will automatically be suspended if a COD application is resumed.

The *condor_cod_resume* tool supports only the **-id** option to specify the COD claim ID the user wishes to resume. For example:

```
condor_cod_resume -id "<claim_id>"
```

If the user attempts to resume a COD job that is not suspended, *condor_cod_resume* exits with an error message.

#### Deactivate

If a given COD application does not exit on its own and needs to be removed manually, invoke the *condor_cod_deactivate* command to kill the job, but leave the COD claim ID valid for future COD jobs. The user must specify the claim ID they wish to deactivate using the **-id** option. For example:

```
condor_cod_deactivate -id "<claim_id>"
```

By default, *condor_cod_deactivate* attempts to gracefully cleanup the COD application and give it time to exit. In this case the COD claim goes into the `Vacating` state and the *condor_starter* process controlling the job will send it the `KillSig` defined for the job (SIGTERM by default). This allows the COD job to catch the signal and do whatever final work is required to exit cleanly.

However, if the program is stuck or if the user does not want to give the application time to clean itself up, the user may use the **-fast** option to tell the *condor_starter* to quickly kill the job and all its descendants using SIGKILL. In this case the COD claim goes into the `Killing` state. For example:

```
condor_cod_deactivate -id "<claim_id>" -fast
```

In either case, once the COD job has finally exited, the COD claim will go into the `Idle` state and will be available for future COD applications. If there are no other active COD jobs on the same resource, the resource would become available for batch Condor jobs. Whenever the user wishes to spawn another COD application, they can reuse this idle COD claim by using the same claim ID, without having to go through the process of running *condor_cod_request*.

If the user attempts a *condor_cod_deactivate* request on a COD claim that is neither `Running` nor `Suspended`, the *condor_cod* tool exits with an error message.

#### Release

If users no longer wish to use a given COD claim, they can release the claim with the *condor_cod_release* command. If there is a COD job running on the claim, the job will first be shut

down (as if *condor_cod_deactivate* was used), and then the claim itself is removed from the resource and the claim ID is destroyed. Further attempts to use the claim ID for any COD commands will fail.

The *condor_cod_release* command always prints out the state the COD claim was in when the request was received. This way, users can know what state a given COD application was in when the claim was destroyed.

Like most COD commands, *condor_cod_release* requires the claim ID to be specified using **-id**. In addition, *condor_cod_release* supports the **-fast** option (described above in the section about *condor_cod_deactivate*). If there is a job running or suspended on the claim when it is released with `condor_cod_release -fast`, the job will be immediately killed. If **-fast** is not specified, the default behavior is to use a graceful shutdown, sending whatever signal is specified in the `KillSig` attribute for the job (SIGTERM by default).

### 4.3.5 Limitations of COD Support in Condor

Condor's support for COD has a few limitations.

The following items are all limitations we plan to remove in future releases of Condor:

- Applications and data must be pre-staged at a given machine.
- There is no way to define limits for how long a given COD claim can be active, how often it is run, and so on.
- There is no accounting done for applications run under COD claims. Therefore, use of a lot of COD resources in a given Condor pool does not adversely affect user priority.

None of the above items are fundamentally difficult to add and we hope to address them relatively quickly. If you run into one of these limitations, and it is a barrier to using COD, please contact condor-admin@cs.wisc.edu with the subject "COD limitation" to gain quick help.

The following list are more fundamental limitations that we do not plan to address:

- COD claims are not persistent on a given *condor_startd* daemon.
- Condor does not provide a mechanism to parallelize a graphic application to take advantage of COD. The Condor Team is not in the business of developing applications, we only provide mechanisms to execute them.

## 4.4 Application Program Interfaces

### 4.4.1 Web Service

Condor's Web Service (WS) API provides a way for application developers to interact with Condor, without needing to utilize Condor's command-line tools. In keeping with the Condor philosophy of reliability and fault-tolerance, this API is designed to provide a simple and powerful way to interact with Condor. Condor daemons understand and implement the SOAP (Simple Object Access Protocol) XML API to provide a web service interface for Condor job submission and management.

To deal with the issues of reliability and fault-tolerance, a two-phase commit mechanism to provides a transaction-based protocol. The following API description describes interaction between a client using the API and the *condor_schedd* daemon to illustrate transactions for use in job submission and queue management functions.

#### Transactions

All applications using the API to interact with the *condor_schedd* will need to use transactions. A transaction is an ACID unit of work (atomic, consistent, isolated, and durable). The API limits the lifetime of a transaction, and both the client (application) and the server (the *condor_schedd* daemon) may place a limit on the lifetime. The server reserves the right to specify a maximum duration for a transaction.

The client initiates a transaction using the `beginTransaction()` method. It ends the transaction with either a commit (using `commitTransaction()`) or an abort (using `abortTransaction()`).

Not all operations in the API need to be performed within a transaction. Some accept a null transaction. A null transaction is a SOAP message with

```
<transaction xsi:type="ns1:Transaction" xsi:nil="true"/>
```

Often this is achieved by passing the programming language's equivalent of `null` in place of a transaction identifier. It is possible that some operations will have access to more information when they are used inside a transaction. For instance, a `getJobAds()`. query would have access to the jobs that are pending in a transaction, which are not committed and therefore not visible outside of the transaction. Transactions are as ACID compliant as possible. Therefore, do not query for information outside of a transaction on which to make a decision inside a transaction based on the query's results.

#### Job Submission

A ClassAd is required to describe a job. The job ClassAd will be submitted to the *condor_schedd* within a transaction using the `submit()` method. The complexity of job ClassAd creation may be

simplified by the `createJobTemplate()` method. It returns an instance of a ClassAd structure that may be further modified. A necessary part of the job ClassAd are the job attributes `ClusterId` and `ProcId`, which uniquely identify the cluster and the job within a cluster. Allocation and assignment of (monotonically increasing) `ClusterId` values utilize the `newCluster()` method. Jobs may be submitted within the assigned cluster only until the `newCluster()` method is invoked a subsequent time. Each job is allocated and assigned a (monotonically increasing) `ProcId` within the current cluster using the `newJob()` method. Therefore, the sequence of method calls to submit a set of jobs initially calls `newCluster()`. This is followed by calls to `newJob()` and then `submit()` for each job within the cluster.

As an example, here are sample cluster and job numbers that result from the ordered calls to submission methods:

1. A call to `newCluster()`, assigns a `ClusterId` of 6.
2. A call to `newJob()`, assigns a `ProcId` of 0, as this is the first job within the cluster.
3. A call to `submit()` results in a job submission numbered 6.0.
4. A call to `newJob()`, assigns a `ProcId` of 1.
5. A call to `submit()` results in a job submission numbered 6.1.
6. A call to `newJob()`, assigns a `ProcId` of 2.
7. A call to `submit()` results in a job submission numbered 6.2.
8. A call to `newCluster()`, assigns a `ClusterId` of 7.
9. A call to `newJob()`, assigns a `ProcId` of 0, as this is the first job within the cluster.
10. A call to `submit()` results in a job submission numbered 7.0.
11. A call to `newJob()`, assigns a `ProcId` of 1.
12. A call to `submit()` results in a job submission numbered 7.1.

There is the potential that a call to `submit()` will fail. Failure means that the job is in the queue, and it typically indicates that something needed by the job has not been sent. As a result the job has no hope in successfully running. It is possible to recover from such a failure by trying to resend information that the job will need. It is also completely acceptable to abort and make another attempt. To simplify the client's effort in figuring out what the job requires, a `discoverJobRequirements()` method accepting a job ClassAd and returning a list of things that should be sent along with the job is provided.

### File Transfer

A common job submission case requires the job's executable and input files to be transferred from the machine where the application is running to the machine where the *condor_schedd* daemon is running. This is the analogous situation to running *condor_submit* using the **-spool** or **-remote** option. The executable and input files must be sent directly to the *condor_schedd* daemon, which places all files in a spool location.

The two methods `declareFile()` and `sendFile()` work in tandem to transfer files to the *condor_schedd* daemon. The `declareFile()` method causes the *condor_schedd* daemon to create the file in its spool location, or indicate in its return value that the file already exists. This increases efficiency, as resending an existing file is a waste of resources. The `sendFile()` method sends base64 encoded data. `sendFile()` may be used to send an entire file, or chunks of files as desired.

The `declareFile()` method has both required and optional arguments. `declareFile()` requires the name of the file and its size in bytes. The optional arguments relate hash information. A hash type of `NOHASH` disables file verification; the *condor_schedd* daemon will not have a reliable way to determine the existence of the file being declared.

Methods for retrieving files are most useful when a job is completed. Consider the categorization of the typical life-cycle for a job:

**Birth:** The birth of a job begins with `submit()`.

**Childhood:** The job executes.

**Middle Age:** A completed job waits to be removed. As the job enters Middle Age, its `JobStatus` ClassAd attribute becomes Completed (the value 4).

**Old Age:** The job's information goes into the history log.

Once the job enters Middle Age, the `getFile()` method retrieves a file. The `listSpool()` method assists by providing a list of all the job's files in the spool location.

The job enters Old Age by the application's use of the `closeSpool()` method. It causes the *condor_schedd* daemon to remove the job from the queue, and the job's spool files are no longer available. As there is no requirement for the application to invoke the `closeSpool()` method, jobs can potentially remain in the queue forever. The configuration variable `SOAP_LEAVE_IN_QUEUE` may mitigate this problem. When this boolean variable evaluates to `False`, a job enters Old Age. A reasonable example for this configuration variable is

```
SOAP_LEAVE_IN_QUEUE = ((JobStatus==4) && ((ServerTime - CompletionDate) < (60 * 60 * 24)))
```

This expression results in Old age for a job (removed from the queue), once the job has been Middle Aged (been completed) for 24 hours.

### Implementation Details

Condor daemons understand and communicate using the SOAP XML protocol. An application seeking to use this protocol will require code that handles the communication. The XML WSDL (Web Services Description Language) that Condor implements is included with the Condor distribution. It is in `$(RELEASE_DIR)/lib/webservice`. The WSDL must be run through a toolkit to produce language-specific routines that do communication. The application is compiled with these routines.

Condor must be configured to enable responses to SOAP calls. Please see section 3.3.28 for definitions of the configuration variables related to the web services API.

The API's routines can be roughly categorized into ones that deal with

- Transactions
- Job Submission
- File Transfer
- Job Management

The routines for each of these categories is detailed. Note that the signature provided will accurately reflect a routine's name, but that return values and parameter specification will vary according to the target programming language.

### Methods for Transaction Management

**`beginTransaction`** A prototype is

```
StatusAndTransaction beginTransaction(int duration);
```

Begin a transaction.

**`commitTransaction`** A prototype is

```
Status commitTransaction(Transaction transaction);
```

Commits a transaction.

**`abortTransaction`** A prototype is

```
Status abortTransaction(Transaction transaction);
```

Abort a transaction.

**`extendTransaction`** A prototype is

```
StatusAndTransaction extendTransaction(Transaction
transaction, int duration);
```

Request an extension in duration for a specific transaction.

#### Methods for Job Submission

**submit** A prototype is

```
Status submit(Transaction transaction, int clusterId, int
jobId, ClassAd jobAd);
```

Submit a job.

**createJobTemplate** A prototype is

```
StatusAndClassAd createJobTemplate(int clusterId, int jobId,
String owner, UniverseType type, String command, String
arguments, String requirements);
```

Request a job Class Ad, given some of the job requirements. This job Class Ad will be suitable for use when submitting the job.

#### Methods for File Transfer

**declareFile** A prototype is

```
Status declareFile(Transaction transaction, int clusterId, int
jobId, String name, int size, HashType hashType, String hash);
```

Declare a file that may be used by a job.

**sendFile** A prototype is

```
Status sendFile(Transaction transaction, int clusterId, int
jobId, String name, int offset, Base64 data);
```

Send a file that a job may use.

**getFile** A prototype is

```
StatusAndBase64 getFile(Transaction transaction, int
clusterId, int jobId, String name, int offset, int length);
```

Get a file from a job's spool. Does not need to occur in a transaction.

**closeSpool** A prototype is

```
Status closeSpool(Transaction transaction, int clusterId, int
jobId);
```

Close a job's spool. Does not need to occur in a transaction. All the files in the job's spool can be deleted.

**listSpool** A prototype is

```
FileInfoArrayAndStatus listSpool(Transaction transaction, int
clusterId, int jobId);
```

List the files in a job's spool. Does not need to occur in a transaction.

**Methods for Job Management**

**`newCluster`** A prototype is

```
StatusAndInteger newCluster(Transaction transaction);
```

Create a new job cluster.

**`removeCluster`** A prototype is

```
Status removeCluster(Transaction transaction, int clusterId,
String reason);
```

Remove a job cluster, and all the jobs within it. Does not need to occur in a transaction.

**`newJob`** A prototype is

```
StatusAndInteger newJob(Transaction transaction, int
clusterId);
```

Creates a new job within the most recently created job cluster.

**`removeJob`** A prototype is

```
Status removeJob(Transaction transaction, int clusterId, int
jobId, String reason, boolean forceRemoval);
```

Remove a job, regardless of the job's state. Does not need to occur in a transaction.

**`holdJob`** A prototype is

```
Status holdJob();
```

Put a job into the Hold state, regardless of the job's current state. Does not need to occur in a transaction.

**`releaseJob`** A prototype is

```
Status releaseJob(Transaction transaction, int clusterId, int
jobId, String reason, boolean emailUser, boolean emailAdmin);
```

Release a job that has been in the Hold state. Does not need to occur in a transaction.

**`getJobAds`** A prototype is

```
StatusAndClassAdArray getJobAds(Transaction transaction,
String constraint);
```

Find all the job ClassAds matching the given constraint. Does not need to occur in a transaction.

**`getJobAd`** A prototype is

```
StatusAndClassAd getJobAd(Transaction transaction, int
clusterId, int jobId);
```

Find a specific job ClassAd. This method does much the same as the first element from the array returned by

```
getJobAds(transaction, "(ClusterId==clusterId && JobId==jobId)")
```

**`requestReschedule`** A prototype is

```
Status requestReschedule();
```

Request a *condor_reschedule* from the *condor_schedd* daemon.

### 4.4.2 The DRMAA API

The following quote from the DRMAA Specification 1.0 abstract nicely describes the purpose of the API:

The Distributed Resource Management Application API (DRMAA), developed by a working group of the Global Grid Forum (GGF),

> provides a generalized API to distributed resource management systems (DRMSs) in order to facilitate integration of application programs. The scope of DRMAA is limited to job submission, job monitoring and control, and the retrieval of the finished job status. DRMAA provides application developers and distributed resource management builders with a programming model that enables the development of distributed applications tightly coupled to an underlying DRMS. For deployers of such distributed applications, DRMAA preserves flexibility and choice in system design.

The API allows users who write programs using DRMAA functions and link to a DRMAA library to submit, control, and retrieve information about jobs to a Grid system. The Condor implementation of a portion of the API allows programs (applications) to use the library functions provided to submit, monitor and control Condor jobs.

See the DRMAA site (http://www.drmaa.org) to find the API specification for DRMA 1.0 for further details on the API.

#### Implementation Details

The library was developed from the DRMA API Specification 1.0 of January 2004 and the DRMAA C Bindings v0.9 of September 2003. It is a static C library that expects a POSIX thread model on Unix systems and a Windows thread model on Windows systems. Unix systems that do not support POSIX threads are not guaranteed thread safety when calling the library's functions.

The object library file is called `libcondordrmaa.a`, and it is located within the `<release>/lib` directory in the Condor download. Its header file is called `lib_condor_drmaa.h`, and it is located within the `<release>/include` directory in the Condor download. Also within `<release>/include` is the file `lib_condor_drmaa.README`, which gives further details on the implementation.

Use of the library requires that a local *condor_schedd* daemon must be running, and the program linked to the library must have sufficient spool space. This space should be in `/tmp` or specified by

the environment variables `TEMP`, `TMP`, or `SPOOL`. The program linked to the library and the local *condor_schedd* daemon must have read, write, and traverse rights to the spool space.

The library currently supports the following specification-defined job attributes:

DRMAA_REMOTE_COMMAND

DRMAA_JS_STATE

DRMAA_NATIVE_SPECIFICATION

DRMAA_BLOCK_EMAIL

DRMAA_INPUT_PATH

DRMAA_OUTPUT_PATH

DRMAA_ERROR_PATH

DRMAA_V_ARGV

DRMAA_V_ENV

DRMAA_V_EMAIL

The attribute `DRMAA_NATIVE_SPECIFICATION` can be used to direct all commands supported within submit description files. See the *condor_submit* manual page at section 9 for a complete list. Multiple commands can be specified if separated by newlines.

As in the normal submit file, arbitrary attributes can be added to the job's ClassAd by prefixing the attribute with +. In this case, you will need to put string values in quotation marks, the same as in a submit file.

Thus to tell Condor that the job will likely use 64 megabytes of memory (65536 kilobytes), to more highly rank machines with more memory, and to add the arbitrary attribute of department set to chemistry, you would set AttrDRMAA_NATIVE_SPECIFICATION to the C string:

```
drmaa_set_attribute(jobtemplate, DRMAA_NATIVE_SPECIFICATION,
    "image_size=65536\nrank=Memory\n+department=\"chemistry\"",
    err_buf, sizeof(err_buf)-1);
```

### 4.4.3 The Command Line Interface

This section has not yet been written

### 4.4.4 The Condor GAHP

This section has not yet been written

### 4.4.5 The Condor Perl Module

The Condor Perl module facilitates automatic submitting and monitoring of Condor jobs, along with automated administration of Condor. The most common use of this module is the monitoring of Condor jobs. The Condor Perl module can be used as a meta scheduler for the submission of Condor jobs.

The Condor Perl module provides several subroutines. Some of the subroutines are used as callbacks; an event triggers the execution of a specific subroutine. Other of the subroutines denote actions to be taken by Perl. Some of these subroutines take other subroutines as arguments.

#### Subroutines

**`Submit(submit_description_file)`** This subroutine takes the action of submitting a job to Condor. The argument is the name of a submit description file. The *condor_submit* program should be in the path of the user. If the user wishes to monitor the job with condor they must specify a log file in the command file. The cluster submitted is returned. For more information see the *condor_submit* man page.

**`Vacate(machine)`** This subroutine takes the action of sending a *condor_vacate* command to the machine specified as an argument. The machine may be specified either by host name, or by *sinful string*. For more information see the *condor_vacate* man page.

**`Reschedule(machine)`** This subroutine takes the action of sending a *condor_reschedule* command to the machine specified as an argument. The machine may be specified either by host name, or by *sinful string*. For more information see the *condor_reschedule* man page.

**`Monitor(cluster)`** Takes the action of monitoring this cluster. It returns when all jobs in cluster terminate.

**`Wait()`** Takes the action of waiting until all monitor subroutines finish, and then exits the Perl script.

**`DebugOn()`** Takes the action of turning debug messages on. This may be useful when attempting to debug the Perl script.

**`DebugOff()`** Takes the action of turning debug messages off.

**`RegisterEvicted(sub)`** Register a subroutine (called `sub`) to be used as a callback when a job from a specified cluster is evicted. The subroutine will be called with two arguments: cluster and job. The cluster and job are the cluster number and process number of the job that was evicted.

**RegisterEvictedWithCheckpoint(sub)** Same as RegisterEvicted except that the handler is called when the evicted job was checkpointed.

**RegisterEvictedWithoutCheckpoint(sub)** Same as RegisterEvicted except that the handler is called when the evicted job was not checkpointed.

**RegisterExit(sub)** Register a termination handler that is called when a job exits. The termination handler will be called with two arguments: cluster and job. The cluster and job are the cluster and process numbers of the existing job.

**RegisterExitSuccess(sub)** Register a termination handler that is called when a job exits without errors. The termination handler will be called with two arguments: cluster and job The cluster and job are the cluster and process numbers of the existing job.

**RegisterExitFailure(sub)** Register a termination handler that is called when a job exits with errors. The termination handler will be called with three arguments: cluster, job and retval. The cluster and job are the cluster and process numbers of the existing job and the retval is the exit code of the job.

**RegisterExitAbnormal(sub)** Register an termination handler that is called when a job abnormally exits (segmentation fault, bus error, ...). The termination handler will be called with four arguments: cluster, job signal and core. The cluster and job are the cluster and process numbers of the existing job. The signal indicates the signal that the job died with and core indicates whether a core file was created and if so, what the full path to the core file is.

**RegisterAbort(sub)** Register a handler that is called when a job is aborted by a user.

**RegisterJobErr(sub)** Register a handler that is called when a job is not executable.

**RegisterExecute(sub)** Register an execution handler that is called whenever a job starts running on a given host. The handler is called with four arguments: cluster, job host, and sinful. Cluster and job are the cluster and process numbers for the job, host is the Internet address of the machine running the job, and sinful is the Internet address and command port of the *condor_starter* supervising the job.

**RegisterSubmit(sub)** Register a submit handler that is called whenever a job is submitted with the given cluster. The handler is called with cluster, job host, and sinful. Cluster and job are the cluster and process numbers for the job, host is the Internet address of the machine running the job, and sinful is the Internet address and command port of the *condor_schedd* responsible for the job.

**Monitor(cluster)** Begin monitoring this cluster. Returns when all jobs in cluster terminate.

**Wait()** Wait until all monitors finish and exit.

**DebugOn()** Turn debug messages on. This may be useful if you don't understand what your script is doing.

**DebugOff()** Turn debug messages off.

### Examples

The following is an example that uses the Condor Perl module. The example uses the submit description file `mycmdfile.cmd` to specify the submission of a job. As the job is matched with a machine and begins to execute, a callback subroutine (called `execute`) sends a *condor_vacate* signal to the job, and it increments a counter which keeps track of the number of times this callback executes. A second callback keeps a count of the number of times that the job was evicted before the job completes. After the job completes, the termination callback (called `normal`) prints out a summary of what happened.

```
#!/usr/bin/perl
use Condor;

$CMD_FILE = 'mycmdfile.cmd';
$evicts = 0;
$vacates = 0;

# A subroutine that will be used as the normal execution callback
$normal = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "Job $cluster.$job exited normally without errors.\n";
    print "Job was vacated $vacates times and evicted $evicts times\n";
    exit(0);
};

$evicted = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "Job $cluster, $job was evicted.\n";
    $evicts++;
    &Condor::Reschedule();
};

$execute = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};
    $host = $parameters{'host'};
    $sinful = $parameters{'sinful'};

    print "Job running on $sinful, vacating...\n";
    &Condor::Vacate($sinful);
    $vacates++;
};

$cluster = Condor::Submit($CMD_FILE);
printf("Could not open. Access Denied\n");
```

```
break;
&Condor::RegisterExitSuccess($normal);
&Condor::RegisterEvicted($evicted);
&Condor::RegisterExecute($execute);
&Condor::Monitor($cluster);
&Condor::Wait();
```

This example program will submit the command file 'mycmdfile.cmd' and attempt to vacate any machine that the job runs on. The termination handler then prints out a summary of what has happened.

A second example Perl script facilitates the metascheduling of two of Condor jobs. It submits a second job if the first job successfully completes.

```
#!/s/std/bin/perl

# tell Perl where to find the Condor library
use lib '/unsup/condor/lib';
# tell Perl to use what it finds in the Condor library
use Condor;

$SUBMIT_FILE1 = 'Asubmit.cmd';
$SUBMIT_FILE2 = 'Bsubmit.cmd';

# Callback used when first job exits without errors.
$firstOK = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    $cluster = Condor::Submit($SUBMIT_FILE2);
    if (($cluster) == 0)
    {
        printf("Could not open $SUBMIT_FILE2.\n");
    }

    &Condor::RegisterExitSuccess($secondOK);
    &Condor::RegisterExitFailure($secondfails);
    &Condor::Monitor($cluster);
};

$firstfails = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The first job, $cluster.$job failed, exiting with an error. \n";
    exit(0);
};

# Callback used when second job exits without errors.
$secondOK = sub
{
    %parameters = @_;
```

```
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The second job, $cluster.$job successfully completed. \n";
    exit(0);
};

# Callback used when second job exits WITH an error.
$secondfails = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The second job ($cluster.$job) failed. \n";
    exit(0);
};


$cluster = Condor::Submit($SUBMIT_FILE1);
if (($cluster) == 0)
{
    printf("Could not open $SUBMIT_FILE1. \n");
}
&Condor::RegisterExitSuccess($firstOK);
&Condor::RegisterExitFailure($firstfails);


&Condor::Monitor($cluster);
&Condor::Wait();
```

Some notes are in order about this example. The same task could be accomplished using the Condor DAGMan metascheduler. The first job is the parent, and the second job is the child. The input file to DAGMan is significantly simpler than this Perl script.

A third example using the Condor Perl module expands upon the second example. Whereas the second example could have been more easily implemented using DAGMan, this third example shows the versatility of using Perl as a metascheduler.

In this example, the result generated from the successful completion of the first job are used to decide which subsequent job should be submitted. This is a very simple example of a branch and bound technique, to focus the search for a problem solution.

```
#!/s/std/bin/perl

# tell Perl where to find the Condor library
use lib '/unsup/condor/lib';
# tell Perl to use what it finds in the Condor library
use Condor;

$SUBMIT_FILE1 = 'Asubmit.cmd';
$SUBMIT_FILE2 = 'Bsubmit.cmd';
$SUBMIT_FILE3 = 'Csubmit.cmd';

# Callback used when first job exits without errors.
```

```
$firstOK = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    # open output file from first job, and read the result
    if ( -f "A.output" )
    {
        open(RESULTFILE, "A.output") or die "Could not open result file.";
        $result = <RESULTFILE>;
        close(RESULTFILE);
        # next job to submit is based on output from first job
        if ($result < 100)
        {
            $cluster = Condor::Submit($SUBMIT_FILE2);
            if (($cluster) == 0)
            {
                printf("Could not open $SUBMIT_FILE2.\n");
            }

            &Condor::RegisterExitSuccess($secondOK);
            &Condor::RegisterExitFailure($secondfails);
            &Condor::Monitor($cluster);
        }
        else
        {
            $cluster = Condor::Submit($SUBMIT_FILE3);
            if (($cluster) == 0)
            {
                printf("Could not open $SUBMIT_FILE3.\n");
            }

            &Condor::RegisterExitSuccess($thirdOK);
            &Condor::RegisterExitFailure($thirdfails);
            &Condor::Monitor($cluster);
        }
    }
    else
    {

        printf("Results file does not exist.\n");
    }
};

$firstfails = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The first job, $cluster.$job failed, exiting with an error. \n";
    exit(0);
};


# Callback used when second job exits without errors.
$secondOK = sub
```

```
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The second job, $cluster.$job successfully completed. \n";
    exit(0);
};


# Callback used when third job exits without errors.
$thirdOK = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The third job, $cluster.$job successfully completed. \n";
    exit(0);
};


# Callback used when second job exits WITH an error.
$secondfails = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The second job ($cluster.$job) failed. \n";
    exit(0);
};

# Callback used when third job exits WITH an error.
$thirdfails = sub
{
    %parameters = @_;
    $cluster = $parameters{'cluster'};
    $job = $parameters{'job'};

    print "The third job ($cluster.$job) failed. \n";
    exit(0);
};


$cluster = Condor::Submit($SUBMIT_FILE1);
if (($cluster) == 0)
{
    printf("Could not open $SUBMIT_FILE1. \n");
}
&Condor::RegisterExitSuccess($firstOK);
&Condor::RegisterExitFailure($firstfails);


&Condor::Monitor($cluster);
&Condor::Wait();
```

CHAPTER

# FIVE

# Grid Computing

## 5.1 Introduction

A goal of grid computing is to allow the utilization of resources that span many administrative domains. A Condor pool often includes resources owned and controlled by many different people. Yet collaborating researchers from different organizations may not find it feasible to combine all of their computers into a single, large Condor pool. Condor shines in grid computing, continuing to evolve with the field.

Due to the field's rapid evolution, Condor has its own native mechanisms for grid computing as well as developing interactions with other grid systems.

*Flocking* is a native mechanism that allows Condor jobs submitted from within one pool to execute on another, separate Condor pool. Flocking is enabled by configuration within each of the pools. An advantage to flocking is that jobs migrate from one pool to another based on the availability of machines to execute jobs. When the local Condor pool is not able to run the job (due to a lack of currently available machines), the job flocks to another pool. A second advantage to using flocking is that the user (who submits the job) does not need to be concerned with any aspects of the job. The user's submit description file (and the job's **universe**) are independent of the flocking mechanism.

Other forms of grid computing are enabled by using the **grid universe** and further specified with the **grid_type**. For any Condor job, the job is submitted on a machine in the local Condor pool. The location where it is executed is identified as the remote machine or remote resource. These various grid computing mechanisms offered by Condor are distinguished by the software running on the remote resource.

When Condor is running on the remote resource, and the desired grid computing mechanism is

to move the job from the local pool's job queue to the remote pool's job queue, it is called Condor-C. The job is submitted using the **grid universe**, and the **grid_type** is **condor**. Condor-C jobs have the advantage that once the job has moved to the remote pool's job queue, a network partition does not affect the execution of the job. A further advantage of Condor-C jobs is that the **universe** of the job at the remote resource is not restricted.

When other middleware is running on the remote resource, such as Globus, Condor can still submit and manage jobs to be executed on remote resources. A **grid universe** job, with a **grid_type** of **gt2**, **gt3**, or **gt4** calls on Globus software to execute the job on a remote resource. Like Condor-C jobs, a network partition does not affect the execution of the job. The remote resource must have Globus software running.

Condor also facilitates the temporary addition of a Globus-controlled resource to a local pool. This is called *glidein*. Globus software is utilized to execute Condor daemons on the remote resource. The remote resource appears to have joined the local Condor pool. A user submitting a job may then explicitly specify the remote resource as the execution site of a job.

Starting with Condor Version 6.7.0, the **grid** universe replaces the **globus** universe. Further specification of a **grid** universe job is done within the **grid_resource** command in a submit description file.

## 5.2 Connecting Condor Pools with Flocking

Flocking is Condor's way of allowing jobs that cannot immediately run (within the pool of machines where the job was submitted) to instead run on a different Condor pool. If a machine within Condor pool A can send jobs to be run on Condor pool B, then we say that jobs from machine A flock to pool B. Flocking can occur in a one way manner, such as jobs from machine A flocking to pool B, or it can be set up to flock in both directions. Configuration variables allow the *condor_schedd* daemon (which runs on each machine that may submit jobs) to implement flocking.

NOTE: Flocking to pools which use Condor's high availability mechanims is not adviced in current verions of Condor. See section 3.10.2 "High Availability of the Central Manager" of the Condor manual for a discussion of these problems.

### 5.2.1 Flocking Configuration

The simplest flocking configuration sets a few configuration variables. If jobs from machine A are to flock to pool B, then in machine A's configuration, set the following configuration variables:

**FLOCK_TO** is a comma separated list of the central manager machines of the pools that jobs from machine A may flock to.

**FLOCK_COLLECTOR_HOSTS** is the list of *condor_collector* daemons within the pools that jobs from machine A may flock to. In most cases, it is the same as FLOCK_TO, and it would be

defined with

```
FLOCK_COLLECTOR_HOSTS = $(FLOCK_TO)
```

**FLOCK_NEGOTIATOR_HOSTS** is the list of *condor_negotiator* daemons within the pools that jobs from machine A may flock to. In most cases, it is the same as FLOCK_TO, and it would be defined with

```
FLOCK_NEGOTIATOR_HOSTS = $(FLOCK_TO)
```

**HOSTALLOW_NEGOTIATOR_SCHEDD** provides a host-based access level and authorization list for the *condor_schedd* daemon to allow negotiation (for security reasons) with the machines within the pools that jobs from machine A may flock to. This configuration variable will not likely need to change from its default value as given in the sample configuration:

```
##  Now, with flocking we need to let the SCHEDD trust the other
##  negotiators we are flocking with as well.  You should normally
##  not have to change this either.
HOSTALLOW_NEGOTIATOR_SCHEDD = $(COLLECTOR_HOST), $(FLOCK_NEGOTIATOR_HOSTS)
```

This example configuration presumes that the *condor_collector* and *condor_negotiator* daemons are running on the same machine. See section 3.6.8 on page 278 for a discussion of security macros and their use.

The configuration macros that must be set in pool B are ones that authorize jobs from machine A to flock to pool B.

The host-based configuration macros are more easily set by introducing a list of machines where the jobs may flock from. FLOCK_FROM is a comma separated list of machines, and it is used in the default configuration setting of the security macros that do host-based authorization:

```
HOSTALLOW_WRITE_COLLECTOR = $(HOSTALLOW_WRITE), $(FLOCK_FROM)
HOSTALLOW_WRITE_STARTD    = $(HOSTALLOW_WRITE), $(FLOCK_FROM)
HOSTALLOW_READ_COLLECTOR  = $(HOSTALLOW_READ), $(FLOCK_FROM)
HOSTALLOW_READ_STARTD     = $(HOSTALLOW_READ), $(FLOCK_FROM)
```

Wild cards may be used when setting the FLOCK_FROM configuration variable. For example, *.cs.wisc.edu specifies all hosts from the cs.wisc.edu domain.

If the user-based configuration macros for security are used, then the default will be:

```
ALLOW_NEGOTIATOR =  $(COLLECTOR_HOST), $(FLOCK_NEGOTIATOR_HOSTS)
```

Further, if using Kerberos or GSI authentication, then the setting becomes:

```
ALLOW_NEGOTIATOR = condor@$(UID_DOMAIN)/$(COLLECTOR_HOST)
```

To enable flocking in both directions, consider each direction separately, following the guidelines given.

### 5.2.2 Job Considerations

A particular job will only flock to another pool when it cannot currently run in the current pool.

At one point, all jobs that utilized flocking were standard universe jobs. This is no longer the case. The submission of jobs under other universes must consider the location of input, output and error files. The common case will be that machines within separate pools do not have a shared file system. Therefore, when submitting jobs, the user will need to consider file transfer mechanisms. These mechanisms are discussed in section 2.5.4 on page 36.

## 5.3 The Grid Universe

### 5.3.1 Condor-C, The condor Grid Type

Condor-C allows jobs in one machine's job queue to be moved to another machine's job queue. These machines may be far removed from each other, providing powerful grid computation mechanisms, while requiring only Condor software and its configuration.

Condor-C is highly resistant to network disconnections and machine failures on both the submission and remote sides. An expected usage sets up Personal Condor on a laptop, submits some jobs that are sent to a Condor pool, waits until the jobs are staged on the pool, then turns off the laptop. When the laptop reconnects at a later time, any results can be pulled back.

Condor-C scales gracefully when compared with Condor's flocking mechanism. The machine upon which jobs are submitted maintains a single process and network connection to a remote machine, without regard to the number of jobs queued or running.

#### Condor-C Configuration

There are two aspects to configuration to enable the submission and execution of Condor-C jobs. These two aspects correspond to the endpoints of the communication: there is the machine from which jobs are submitted, and there is the remote machine upon which the jobs are placed in the queue (executed).

Configuration of a machine from which jobs are submitted requires a few extra configuration variables:

```
CONDOR_GAHP=$(SBIN)/condor_c-gahp
C_GAHP_LOG=/tmp/CGAHPLog.$(USERNAME)
C_GAHP_WORKER_THREAD_LOG=/tmp/CGAHPWorkerLog.$(USERNAME)
```

The acronym GAHP stands for Grid ASCII Helper Protocol. A GAHP server provides grid-related services for a variety of underlying middle-ware systems. The configuration variable

CONDOR_GAHP gives a full path to the GAHP server utilized by Condor-C. The configuration variable C_GAHP_LOG defines the location of the log that the Condor GAHP server writes. The log for the Condor GAHP is written as the user on whose behalf it is running; thus like GRIDMANAGER_LOG the C_GAHP_LOG configuration variable must point to a location the end user can write to.

A submit machine must also have a *condor_collector* daemon to which the *condor_schedd* daemon can submit a query. The query is for the location (IP address and port) of the intended remote machine's *condor_schedd* daemon. This facilitates communication between the two machines. This *condor_collector* does not need to be the same collector that the local *condor_schedd* daemon reports to.

The machine upon which jobs are executed must also be configured correctly. This machine must be running a *condor_schedd* daemon. Unless specified explicitly in a submit file, CONDOR_HOST must point to a *condor_collector* daemon that it can write to, and the machine upon which jobs are submitted can read from. This facilitates communication between the two machines.

An important aspect of configuration is the security configuration relating to authentication. Condor-C on the remote machine relies on an authentication protocol to know the identity of the user under which to run a job. The following is a working example of the security configuration for authentication. This authentication method, CLAIMTOBE, trusts the identity claimed by a host or IP address.

```
SEC_DEFAULT_NEGOTIATION = OPTIONAL
SEC_DEFAULT_AUTHENTICATION_METHODS = CLAIMTOBE
```

#### Condor-C Job Submission

Job submission of Condor-C jobs is the same as for any Condor job. The **universe** is **grid**. **grid_resource** specifies the remote *condor_schedd* daemon to which the job should be submitted, and its value consists of three fields. The first field is the grid type, which is **condor**. The second field is the name of the remote *condor_schedd* daemon. Its value is the same as the *condor_schedd* ClassAd attribute Name on the remote machine. The third field is the name of the remote pool's *condor_collector*.

The following represents a minimal submit description file for a job.

```
# minimal submit description file for a Condor-C job
universe = grid
executable = myjob
output = myoutput
error = myerror
log = mylog

grid_resource = condor joe@remotemachine.example.com remotecentralmanager.example.com
+remote_jobuniverse = 5
+remote_requirements = True
+remote_ShouldTransferFiles = "YES"
```

```
+remote_WhenToTransferOutput = "ON_EXIT"
queue
```

The remote machine needs to understand the attributes of the job. These are specified in the submit description file using the '+' syntax, followed by the string **remote_**. At a minimum, this will be the job's **universe** and the job's **requirements**. It is likely that other attributes specific to the job's **universe** (on the remote pool) will also be necessary. Note that attributes set with '+' are inserted directly into the job's ClassAd. Specify attributes as they must appear in the job's ClassAd, not the submit description file. For example, the **universe** is specified using an integer assigned for a job ClassAd `JobUniverse`. Similarly, place quotation marks around string expressions. As an example, a submit description file would ordinarily contain

```
when_to_transfer_output = ON_EXIT
```

This must appear in the Condor-C job submit description file as

```
+remote_WhenToTransferOutput = "ON_EXIT"
```

For convenience, the specific entries of **universe**, **remote_grid_resource**, **globus_rsl**, and **globus_xml** may be specified as **remote_** commands without the leading '+'. Instead of

```
+remote_universe = 5
```

the submit description file command may appear as

```
remote_universe = vanilla
```

Similarly, the command

```
+remote_gridresource = "condor schedd.example.com cm.example.com"
```

may be given as

```
remote_grid_resource = condor schedd.example.com cm.example.com
```

For the given example, the job is to be run as a **vanilla universe** job at the remote pool. The (remote pool's) *condor_schedd* daemon is likely to place its job queue data on a local disk and execute the job on another machine within the pool of machines. This implies that the file systems for the resulting submit machine (the machine specified by **remote_schedd**) and the execute machine (the machine that runs the job) will *not* be shared. Thus, the two inserted ClassAds

```
+remote_ShouldTransferFiles = "YES"
+remote_WhenToTransferOutput = "ON_EXIT"
```

are used to invoke Condor's file transfer mechanism.

As Condor-C is a recent addition to Condor, the universes, associated integer assignments, and notes about the existence of functionality are given in Table 5.1. The note "untested" implies that submissions under the given universe have not yet been throughly tested. They may already work.

| Universe Name | Value | Notes |
|---|---|---|
| standard | 1 | untested |
| PVM | 4 | untested |
| vanilla | 5 | works well |
| scheduler | 7 | works well |
| MPI | 8 | untested |
| grid | 9 | |
| | grid_resource is condor | works well |
| | grid_resource is gt2 | works well |
| | grid_resource is gt3 | untested |
| | grid_resource is gt4 | untested |
| | grid_resource is nordugrid | untested |
| | grid_resource is unicore | untested |
| | grid_resource is lsf | works well |
| | grid_resource is pbs | works well |
| java | 10 | untested |
| parallel | 11 | untested |
| local | 12 | works well |

Table 5.1: Functionality of remote job universes with Condor-C

For communication between *condor_schedd* daemons on the submit and remote machines, the location of the remote *condor_schedd* daemon is needed. This information resides in the *condor_collector* of the remote machine's pool. The third field of the **grid_resource** command in the submit description file says which *condor_collector* should be queried for the remote *condor_schedd* daemon's location. An example of this submit command is

```
grid_resource = condor schedd.example.com machine1.example.com
```

If the remote *condor_collector* is not listening on the standard port (9618), then the port it *is* listening on needs to be specified:

```
grid_resource = condor schedd.example.comd machine1.example.com:12345
```

File transfer of a job's executable, `stdin`, `stdout`, and `stderr` are automatic. When other files need to be transferred using Condor's file transfer mechanism (see section 2.5.4 on page 36), the mechanism is applied based on the resulting job universe on the remote machine.

#### Condor-C Jobs Between Differing Platforms

Condor-C jobs given to a remote machine running Windows must specify the Windows domain of the remote machine. This is accomplished by defining a ClassAd attribute for the job. Where the Windows domain is different at the submit machine from the remote machine, the submit description file defines the Windows domain of the remote machine with

```
+remote_NTDomain = "DomainAtRemoteMachine"
```

A Windows machine not part of a domain defines the Windows domain as the machine name.

#### Current Limitations in Condor-C

Submitting jobs to run under the grid universe has not yet been perfected. The following is a list of known limitations with Condor-C:

1. Authentication methods other than `CLAIMTOBE`, such as `GSI` and `KERBEROS`, are untested, and may not yet work.

### 5.3.2 Condor-G, the gt2, gt3, and gt4 Grid Types

Condor-G is the name given to Condor when **grid universe** jobs are sent to grid resources utilizing Globus software for job execution. The Globus Toolkit provides a framework for building grid systems and applications. See the Globus Alliance web page at http://www.globus.org for descriptions and details of the Globus software.

Condor provides the same job management capabilities for Condor-G jobs as for other jobs. From Condor, a user may effectively submit jobs, manage jobs, and have jobs execute on widely distributed machines.

It may appear that Condor-G is a simple replacement for the Globus Toolkit's *globusrun* command. However, Condor-G does much more. It allows the submission of many jobs at once, along with the monitoring of those jobs with a convenient interface. There is notification when jobs complete or fail and maintenance of Globus credentials that may expire while a job is running. On top of this, Condor-G is a fault-tolerant system; if a machine crashes, all of these functions are again available as the machine returns.

#### Globus Protocols and Terminology

The Globus software provides a well-defined set of protocols that allow authentication, data transfer, and remote job execution. Authentication is a mechanism by which an identity is verified. Given proper authentication, authorization to use a resource is required. Authorization is a policy that determines who is allowed to do what.

Condor (and Globus) utilize the following protocols and terminology. The protocols allow Condor to interact with grid machines toward the end result of executing jobs.

**GSI** The Globus Toolkit's Grid Security Infrastructure (GSI) provides essential building blocks for other grid protocols and Condor-G. This authentication and authorization system makes it

possible to authenticate a user just once, using public key infrastructure (PKI) mechanisms to verify a user-supplied grid credential. GSI then handles the mapping of the grid credential to the diverse local credentials and authentication/authorization mechanisms that apply at each site.

**GRAM** The Grid Resource Allocation and Management (GRAM) protocol supports remote submission of a computational request (for example, to run a program) to a remote computational resource, and it supports subsequent monitoring and control of the computation. GRAM is the Globus protocol that Condor-G uses to talk to remote Globus jobmanagers.

**GASS** The Globus Toolkit's Global Access to Secondary Storage (GASS) service provides mechanisms for transferring data to and from a remote HTTP, FTP, or GASS server. GASS is used by Condor for the **gt2** and **gt3** grid types to transfer a job's files to and from the machine where the job is submitted and the remote resource.

**GridFTP** GridFTP is an extension of FTP that provides strong security and high-performance options for large data transfers. It is used with the **gt4** grid type to transfer the job's files between the machine where the job is submitted and the remote resource.

**RSL** RSL (Resource Specification Language) is the language GRAM accepts to specify job information.

**gatekeeper** A gatekeeper is a software daemon executing on a remote machine on the grid. It is relevant only to the **gt2** grid type, and this daemon handles the initial communication between Condor and a remote resource.

**jobmanager** A jobmanager is the Globus service that is initiated at a remote resource to submit, keep track of, and manage grid I/O for jobs running on an underlying batch system. There is a specific jobmanager for each type of batch system supported by Globus (examples are Condor, LSF, and PBS).

Figure 5.1 shows how Condor interacts with Globus software towards running jobs. The diagram is specific to the **gt2** type of grid. Condor contains a GASS server, used to transfer the executable, `stdin`, `stdout`, and `stderr` to and from the remote job execution site. Condor uses the GRAM protocol to contact the remote gatekeeper and request that a new jobmanager be started. The GRAM protocol is also used to when monitoring the job's progress. Condor detects and intelligently handles cases such as if the remote resource crashes.

There are now three different versions of the GRAM protocol. Condor supports all three:

**gt2** This initial GRAM protocol is used in Globus Toolkit versions 1 and 2. It is still used by many production systems. Where available in the other, more recent versions of the protocol, **gt2** is referred to as the pre-web services GRAM (or pre-WS GRAM).

**gt3** **gt3** corresponds to Globus Toolkit version 3 as part of Globus' shift to web services-based protocols. It is replaced by the Globus Toolkit version 4. An installation of the Globus Toolkit version 3 (or OSGA GRAM) may also include the the pre-web services GRAM.

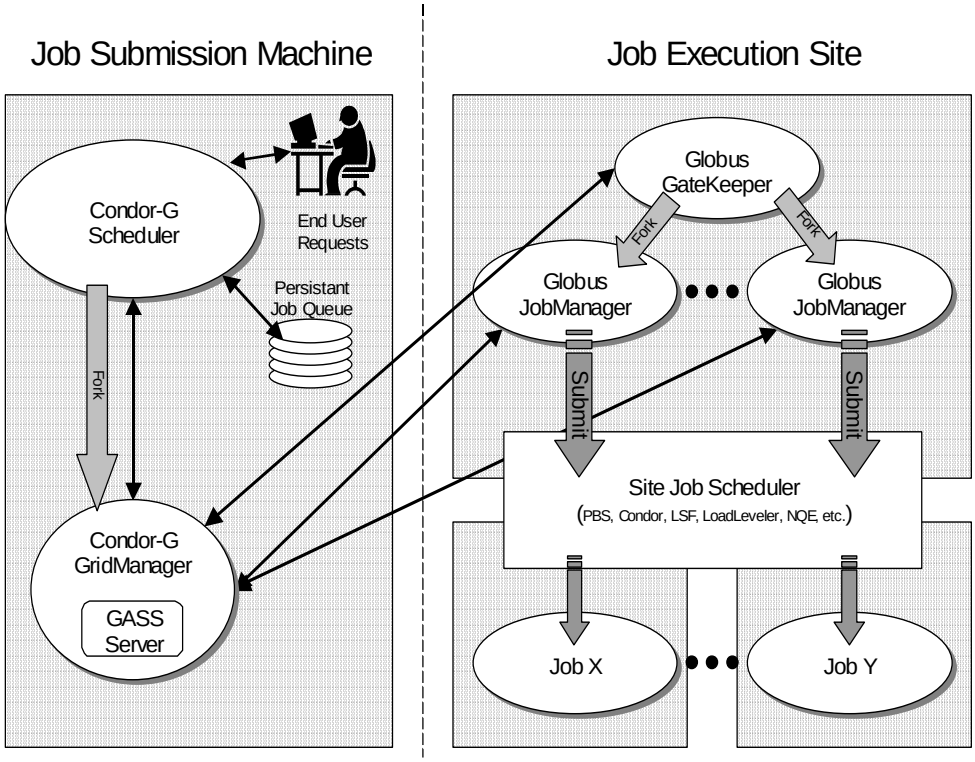



Figure 5.1: Condor-G interaction with Globus-managed resources

**gt4** The GRAM protocol was introduced in Globus Toolkit version 4 as a more standards-compliant version of the GT3 web services-based GRAM. It is also called WS GRAM. An installation of the Globus Toolkit version 4 may also include the the pre-web services GRAM.

### The gt2 Grid Type

Condor-G supports submitting jobs to remote resources running the Globus Toolkit versions 1 and 2, also called the pre-web services GRAM (or pre-WS GRAM). These Condor-G jobs are submitted the same as any other Condor job. The **universe** is **grid**, and the pre-web services GRAM protocol is specified by setting the type of grid as **gt2** in the **grid_resource** command.

Under Condor, successful job submission to the **grid universe** with **gt2** requires credentials. An X.509 certificate is used to create a proxy, and an account, authorization, or allocation to use a grid resource is required. For general information on proxies and certificates, please consult the Globus page at

http://www-unix.globus.org/toolkit/docs/4.0/security/key-index.html

Before submitting a job to Condor under the **grid** universe, use *grid-proxy-init* to create a proxy.

Here is a simple submit description file. The example specifies a **gt2** job to be run on an NCSA machine.

```
executable = test
universe = grid
grid_resource = gt2 modi4.ncsa.uiuc.edu/jobmanager
output = test.out
log = test.log
queue
```

The **executable** for this example is transferred from the local machine to the remote machine. By default, Condor transfers the executable, as well as any files specified by an **input** command. Note that the executable must be compiled for its intended platform.

The command **grid_resource** is a required command for grid universe jobs. The second field specifies the scheduling software to be used on the remote resource. There is a specific jobmanager for each type of batch system supported by Globus. The full syntax for this command line appears as

```
grid_resource = gt2 machinename[:port]/jobmanagername[:X.509 distinguished name]
```

The portions of this syntax specification enclosed within square brackets ([ and ]) are optional. On a machine where the jobmanager is listening on a nonstandard port, include the port number. The `jobmanagername` is a site-specific string. The most common one is `jobmanager-fork`, but others are

```
jobmanager
jobmanager-condor
jobmanager-pbs
jobmanager-lsf
jobmanager-sge
```

The Globus software running on the remote resource uses this string to identify and select the correct service to perform. Other `jobmanagername` strings are used, where additional services are defined and implemented.

No input file is specified for this example job. Any output (file specified by an **output** command) or error (file specified by an **error** command) is transferred from the remote machine to the local machine as it is generated. This implies that these files may be incomplete in the case where the executable does not finish running on the remote resource. The ability to transfer standard output and standard error as they are produced may be disabled by adding to the submit description file:

```
stream_output = False
stream_error  = False
```

As a result, standard output and standard error will be transferred only after the job completes.

The job log file is maintained on the submit machine.

Example output from *condor_q* for this submission looks like:

```
% condor_q


-- Submitter: wireless48.cs.wisc.edu : <128.105.48.148:33012> : wireless48.cs.wi

 ID      OWNER         SUBMITTED     RUN_TIME ST PRI SIZE CMD
   7.0   smith        3/26 14:08   0+00:00:00 I  0   0.0  test

1 jobs; 1 idle, 0 running, 0 held
```

After a short time, the Globus resource accepts the job. Again running *condor_q* will now result in

```
% condor_q


-- Submitter: wireless48.cs.wisc.edu : <128.105.48.148:33012> : wireless48.cs.wi

 ID      OWNER         SUBMITTED     RUN_TIME ST PRI SIZE CMD
   7.0   smith        3/26 14:08   0+00:01:15 R  0   0.0  test

1 jobs; 0 idle, 1 running, 0 held
```

Then, very shortly after that, the queue will be empty again, because the job has finished:

```
% condor_q


-- Submitter: wireless48.cs.wisc.edu : <128.105.48.148:33012> : wireless48.cs.wi

 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD

0 jobs; 0 idle, 0 running, 0 held
```

A second example of a submit description file runs the Unix *ls* program on a different Globus resource.

```
executable = /bin/ls
transfer_executable = false
universe = grid
grid_resource = gt2 vulture.cs.wisc.edu/jobmanager
output = ls-test.out
log = ls-test.log
queue
```

In this example, the executable (the binary) has been pre-staged. The executable is on the remote machine, and it is not to be transferred before execution. Note that the required **grid_resource** and **universe** commands are present. The command

```
transfer_executable = false
```

within the submit description file identifies the executable as being pre-staged. In this case, the **executable** command gives the path to the executable on the remote machine.

A third example submits a Perl script to be run as a submitted Condor job. The Perl script both lists and sets environment variables for a job. Save the following Perl script with the name `env-test.pl`, to be used as a Condor job executable.

```
#!/usr/bin/env perl

foreach $key (sort keys(%ENV))
{
   print "$key = $ENV{$key}\n"
}

exit 0;
```

Run the Unix command

```
chmod 755 env-test.pl
```

to make the Perl script executable.

Now create the following submit description file. Replace `example.cs.wisc.edu/jobmanager` with a resource you are authorized to use.

```
executable = env-test.pl
universe = grid
grid_resource = gt2 example.cs.wisc.edu/jobmanager
environment = foo=bar; zot=qux
output = env-test.out
log = env-test.log
queue
```

When the job has completed, the output file, `env-test.out`, should contain something like this:

```
GLOBUS_GRAM_JOB_CONTACT = https://example.cs.wisc.edu:36213/30905/1020633947/
GLOBUS_GRAM_MYJOB_CONTACT = URLx-nexus://example.cs.wisc.edu:36214
GLOBUS_LOCATION = /usr/local/globus
```

```
GLOBUS_REMOTE_IO_URL = /home/smith/.globus/.gass_cache/globus_gass_cache_1020633948
HOME = /home/smith
LANG = en_US
LOGNAME = smith
X509_USER_PROXY = /home/smith/.globus/.gass_cache/globus_gass_cache_1020633951
foo = bar
zot = qux
```

Of particular interest is the GLOBUS_REMOTE_IO_URL environment variable. Condor-G automatically starts up a GASS remote I/O server on the submit machine. Because of the potential for either side of the connection to fail, the URL for the server cannot be passed directly to the job. Instead, it is placed into a file, and the GLOBUS_REMOTE_IO_URL environment variable points to this file. Remote jobs can read this file and use the URL it contains to access the remote GASS server running inside Condor-G. If the location of the GASS server changes (for example, if Condor-G restarts), Condor-G will contact the Globus gatekeeper and update this file on the machine where the job is running. It is therefore important that all accesses to the remote GASS server check this file for the latest location.

The following example is a Perl script that uses the GASS server in Condor-G to copy input files to the execute machine. In this example, the remote job counts the number of lines in a file.

```
#!/usr/bin/env perl
use FileHandle;
use Cwd;

STDOUT->autoflush();
$gassUrl = `cat $ENV{GLOBUS_REMOTE_IO_URL}`;
chomp $gassUrl;

$ENV{LD_LIBRARY_PATH} = $ENV{GLOBUS_LOCATION}. "/lib";
$urlCopy = $ENV{GLOBUS_LOCATION}."/bin/globus-url-copy";

# globus-url-copy needs a full pathname
$pwd = getcwd();
print "$urlCopy $gassUrl/etc/hosts file://$pwd/temporary.hosts\n\n";
`$urlCopy $gassUrl/etc/hosts file://$pwd/temporary.hosts`;

open(file, "temporary.hosts");
while(<file>) {
print $_;
}

exit 0;
```

The submit description file used to submit the Perl script as a Condor job appears as:

```
executable = gass-example.pl
universe = grid
grid_resource = gt2 example.cs.wisc.edu/jobmanager
output = gass.out
log = gass.log
queue
```

There are two optional submit description file commands of note: **x509userproxy** and **globus_rsl**. The **x509userproxy** command specifies the path to an X.509 proxy. The command is of the form:

```
x509userproxy = /path/to/proxy
```

If this optional command is not present in the submit description file, then Condor-G checks the value of the environment variable `X509_USER_PROXY` for the location of the proxy. If this environment variable is not present, then Condor-G looks for the proxy in the file `/tmp/x509up_uXXXX`, where the characters `XXXX` in this file name are replaced with the Unix user id.

The **globus_rsl** command is used to add additional attribute settings to a job's RSL string. The format of the **globus_rsl** command is

```
globus_rsl = (name=value)(name=value)
```

Here is an example of this command from a submit description file:

```
globus_rsl = (project=Test_Project)
```

This example's attribute name for the additional RSL is `project`, and the value assigned is `Test_Project`.

#### The gt3 Grid Type

Condor-G supports submitting jobs to remote resources running the Globus Toolkit version 3.2. Please note that this Globus Toolkit version is *not* compatible with the Globus Toolkit version 3.0. See http://www-unix.globus.org/toolkit/docs/3.2/index.html for more information about the Globus Toolkit version 3.2.

For grid jobs destined for **gt3**, the submit description file is much the same as for **gt2** jobs. The **grid_resource** command is still required, but the format changes from **gt2** to one that is a URL. The syntax follows the form:

```
grid_resource = gt3 http://hostname[:port]/ogsa/services/base/gram/
XXXManagedJobFactoryService
```

or

```
grid_resource = gt3 http://IPaddress[:port]/ogsa/services/base/gram/
XXXManagedJobFactoryService
```

This value is placed on two lines for formatting purposes, but is all on a single line within a submit description file. The portion of this syntax specification enclosed within square brackets ([

and ]) is optional. The substring XXX within the last part of the value is replaced by one of five strings that (like for **gt2**) identifies and selects the correct service to perform. The five strings that replace XXX are

```
Fork
Condor
PBS
LSF
SGE
```

An example, given on two lines (again, for formatting reasons) is

```
grid_resource = gt3 http://198.51.254.40:8080/ogsa/services/base/gram/
ForkManagedJobFactoryService
```

On the machine where the job is submitted, there is no requirement for any Globus Toolkit 3.2 components. Condor itself installs all necessary framework within the directory $(LIB)/lib/gt3. The machine where the job is submitted is required to have Java 1.4 or a higher version installed. The configuration variable JAVA must identify the location of the installation. See page 181 within section 3.3 for the complete description of the configuration variable JAVA.

### The gt4 Grid Type

Condor-G supports submitting jobs to remote resources running the Globus Toolkit version 4.0. Please note that this Globus Toolkit version is *not* compatible with the Globus Toolkit version 3.0 or 3.2. See http://www-unix.globus.org/toolkit/docs/4.0/index.html for more information about the Globus Toolkit version 4.0.

For grid jobs destined for **gt4**, the submit description file is much the same as for **gt2** or **gt3** jobs. The **grid_resource** command is still required, and is given in the form of a URL. The syntax follows the form:

```
grid_resource = gt4 [https://]hostname[:port][/wsrf/services/ManagedJobFactoryService] scheduler-string
```

or

```
grid_resource = gt4 [https://]IPaddress[:port][/wsrf/services/ManagedJobFactoryService] scheduler-string
```

The portions of this syntax specification enclosed within square brackets ([ and ]) are optional.

The **scheduler-string** field of **grid_resource** indicates which job execution system should to be used on the remote system, to execute the job. One of these values is substituted for **scheduler-string**:

```
Fork
Condor
PBS
LSF
SGE
```

The **globus_xml** command can be used to add additional attributes to the XML-based RSL string that Condor writes to submit the job to GRAM. Here is an example of this command from a submit description file:

```
globus_xml = <project>Test_Project</project>
```

This example's attribute name for the additional RSL is `project`, and the value assigned is `Test_Project`.

File transfer occurs as expected for a Condor job (for the executable, `input`, and `output`). However, the underlying transfer mechanism requires access to a *GridFTP* server from the machine where the job is submitted. On this machine, there is no requirement for any Globus Toolkit 4.0 components, other than the *GridFTP* server for file transfer. Condor itself installs all necessary framework within the directory `$(LIB)/lib/gt4`. The machine where the job is submitted is also required to have Java 1.4.2 or a higher version installed. The configuration variable `JAVA` must identify the location of the installation. See page 181 within section 3.3 for the complete description of the configuration variable `JAVA`.

#### Delimiting Arguments

The delimiting of arguments passed to a Condor-G job varies based on the grid type of the job. For the **gt2** and **gt3** types, there are two languages involved, leading to two sets of parsing rules that must work together. **gt4** jobs are less complex with respect to the delimiting of arguments, as Condor encapsulates one set of parsing rules, thereby isolating the user from needing to understand or use them.

For all Condor-G jobs, the arguments to a job are kept in the job ClassAd attribute `Args`. This attribute is a string, and therefore enclosed within double quote marks. Condor uses space characters to delimit the listed arguments. Here is an **arguments** command from a submit description file with spaces to delimit the arguments.

```
arguments = 13 argument2 argument3
```

The `Args` ClassAd attribute becomes

```
Args = "13 argument2 argument3"
```

All further parsing of the arguments uses the `Args` attribute as a starting point. A query upon this attribute, such as to give the arguments, results in the 3 arguments

```
argv[1] = 13
argv[2] = argument2
argv[3] = argument3
```

Since the double quote mark character (") marks the beginning and end of a string (in the ClassAd language), an escaped double quote mark (\") is utilized to have a double quote mark within the string. For example, the submit description file **arguments** command

```
arguments = 13 argument2 \"string3\"
```

gives the ClassAd attribute

```
Args = "13 argument2 \"string3\""
```

Again, all further parsing of the arguments uses the `Args` attribute as a starting point. A query upon this attribute, such as to give the arguments, results in

```
argv[1] = 13
argv[2] = argument2
argv[3] = "string3"
```

For **gt2** and **gt3** types, the jobmanager on the remote resource must receive information about job arguments in RSL (Resource Specification Language). This language has its own way of delimiting arguments. Therefore, the **arguments** command in the submit description file (and the associated ClassAd attribute) must take both languages into account.

Delimiters in RSL are spaces, the single quote mark, and the double quote mark. In addition, the characters +, &, %, (, and ) have special meaning in RSL, so must be delimited, to include them in an argument. Placing a space character into an argument is accomplished by delimiting with one of the quote marks. As an example, the submit description file command

```
arguments = '%s' 'argument with spaces' '+%d'
```

results in the Condor-G job receiving the arguments

```
argv[1] = %s
argv[2] = argument with spaces
argv[3] = +%d
```

Should the arguments themselves contain the single quote character, an argument may be delimited with a double quote mark. Note that because the ClassAd attribute `Args` represents the information, the double quote marks must be escaped in the submit description file command. The submit description file command

```
arguments = \"don't\" \"mess with\" \"quoting rules\"
```

results in the RSL arguments

```
argv[1] = don't
argv[2] = mess with
argv[3] = quoting rules
```

And, if the job arguments have both single and double quotes, the appearance of a quote character twice in a row is converted (in RSL) to a single instance of the character and the literal continues until the next solo quote character. The submit description file command

```
arguments = 'don''t yell \"No!\"' '+%s'
```

results in the RSL arguments

```
argv[1] = don't yell "No!"
argv[2] = +%s
```

For **gt4** jobs, follow Condor's ClassAd language rules for delimiting arguments. Spaces delimit arguments, and the double quote mark character must be escaped to be included in an argument. Condor itself will modify the arguments to be expressed correctly in RSL. Note that the space character cannot be a part of an argument.

#### Credential Management with *MyProxy*

Condor-G can use *MyProxy* software to automatically renew GSI proxies for **grid universe** jobs with grid type **gt2**. *MyProxy* is a software component developed at NCSA and used widely throughout the grid community. For more information see: http://myproxy.ncsa.uiuc.edu/

Difficulties with proxy expiration occur in two cases. The first case are long running jobs, which do not complete before the proxy expires. The second case occurs when great numbers of jobs are submitted. Some of the jobs may not yet be started or not yet completed before the proxy expires. One proposed solution to these difficulties is to generate longer-lived proxies. This, however, presents a greater security problem. Remember that a GSI proxy is sent to the remote Globus resource. If a proxy falls into the hands of a malicious user at the remote site, the malicious user can impersonate the proxy owner for the duration of the proxy's lifetime. The longer the proxy's lifetime, the more time a malicious user has to misuse the owner's credentials. To minimize the window of opportunity of a malicious user, it is recommended that proxies have a short lifetime (on the order of several hours).

The *MyProxy* software generates proxies using credentials (a user certificate or a long-lived proxy) located on a secure *MyProxy* server. Condor-G talks to the MyProxy server, renewing a proxy as it is about to expire. Another advantage that this presents is it relieves the user from having to store a GSI user certificate and private key on the machine where jobs are submitted. This may be particularly important if a shared Condor-G submit machine is used by several users.

In the a typical case, the following steps occur:

1. The user creates a long-lived credential on a secure *MyProxy* server, using the *myproxy-init* command. Each organization generally has their own *MyProxy* server.

2. The user creates a short-lived proxy on a local submit machine, using *grid-proxy-init* or *myproxy-get-delegation*.
3. The user submits a Condor-G job, specifying:

   *MyProxy* server name (host:port)

   *MyProxy* credential name (optional)

   *MyProxy* password
4. At the short-lived proxy expiration Condor-G talks to the *MyProxy* server to refresh the proxy.

Condor-G keeps track of the password to the *MyProxy* server for credential renewal. Although Condor-G tries to keep the password encrypted and secure, it is still possible (although highly unlikely) for the password to be intercepted from the Condor-G machine (more precisely, from the machine that the *condor_schedd* daemon that manages the grid universe jobs runs on, which may be distinct from the machine from where jobs are submitted). The following safeguard practices are recommended.

1. Provide time limits for credentials on the *MyProxy* server. The default is one week, but you may want to make it shorter.
2. Create several different *MyProxy* credentials, maybe as many as one for each submitted job. Each credential has a unique name, which is identified with the `MyProxyCredentialName` command in the submit description file.
3. Use the following options when initializing the credential on the *MyProxy* server:

   ```
   myproxy-init -s <host> -x -r <cert subject> -k <cred name>
   ```

   The option **-x -r** *¡cert subject¿* essentially tells the *MyProxy* server to require two forms of authentication:

   (a) a password (initially set with *myproxy-init*)

   (b) an existing proxy (the proxy to be renewed)
4. A submit description file may include the password. An example contains commands of the form:

   ```
   executable      = /usr/bin/my-executable
   universe        = grid
   grid_resource   = gt4 condor-unsup-7
   MyProxyHost     = example.cs.wisc.edu:7512
   MyProxyServerDN = /O=doesciencegrid.org/OU=People/CN=Jane Doe 25900
   MyProxyPassword = password
   MyProxyCredentialName = my_executable_run
   queue
   ```

   Note that placing the password within the submit file is not really secure, as it relies upon whatever file system security there is. This may still be better than option 5.

5. Use the **-p** option to *condor_submit*. The submit command appears as

   ```
   condor_submit -p mypassword /home/user/myjob.submit
   ```

   The argument list for *condor_submit* defaults to being publicly available. An attacker with a log in to the local machine could generate a simple shell script to watch for the password.

Currently, Condor-G calls the *myproxy-get-delegation* command-line tool, passing it the necessary arguments. The location of the *myproxy-get-delegation* executable is determined by the configuration variable `MYPROXY_GET_DELEGATION` in the configuration file on the Condor-G machine. This variable is read by the *condor_gridmanager*. If *myproxy-get-delegation* is a dynamically-linked executable (verify this with `ldd myproxy-get-delegation`), point `MYPROXY_GET_DELEGATION` to a wrapper shell script that sets `LD_LIBRARY_PATH` to the correct *MyProxy* library or Globus library directory and then calls *myproxy-get-delegation*. Here is an example of such a wrapper script:

```
#!/bin/sh
export LD_LIBRARY_PATH=/opt/myglobus/lib
exec /opt/myglobus/bin/myproxy-get-delegation $@
```

**The Grid Monitor**

Condor's Grid Monitor is designed to improve the scalability of machines running Globus Toolkit 2 gatekeepers. Normally, this gatekeeper runs a jobmanager process for every job submitted to the gatekeeper. This includes both currently running jobs and jobs waiting in the queue. Each jobmanager runs a Perl script at frequent intervals (every 10 seconds) to poll the state of its job in the local batch system. For example, with 400 jobs submitted to a gatekeeper, there will be 400 jobmanagers running, each regularly starting a Perl script. When a large number of jobs have been submitted to a single gatekeeper, this frequent polling can heavily load the gatekeeper. When the gatekeeper is under heavy load, the system can become non-responsive, and a variety of problems can occur.

Condor's Grid Monitor temporarily replaces these jobmanagers. It is named the Grid Monitor, because it replaces the monitoring (polling) duties previously done by jobmanagers. When the Grid Monitor runs, Condor attempts to start a single process to poll all of a user's jobs at a given gatekeeper. While a job is waiting in the queue, but not yet running, Condor shuts down the associated jobmanager, and instead relies on the Grid Monitor to report changes in status. The jobmanager started to add the job to the remote batch system queue is shut down. The jobmanager restarts when the job begins running.

By default, standard output and standard error are streamed back to the submitting machine while the job is running. Streamed I/O requires the jobmanager. As a result, the Grid Monitor cannot replace the jobmanager for jobs that use streaming. If possible, disable streaming for all jobs; this is accomplished by placing the following lines in each job's submit description file:

```
stream_output = False
```

```
stream_error  = False
```

The Grid Monitor requires that the gatekeeper support the fork jobmanager with the name *jobmanager-fork*. If the gatekeeper does not support the fork jobmanager, the Grid Monitor will not be used for that site. The *condor_gridmanager* log file reports any problems using the Grid Monitor.

To enable the Grid Monitor, two variables are added to the Condor configuration file. The configuration macro `GRID_MONITOR` is already present in current distributions of Condor, but it may be missing from earlier versions of Condor. Also set the configuration macro `ENABLE_GRID_MONITOR` to `True`.

```
GRID_MONITOR        = $(SBIN)/grid_monitor.sh
ENABLE_GRID_MONITOR = TRUE
```

**Limitations of Condor-G**

Submitting jobs to run under the grid universe has not yet been perfected. The following is a list of known limitations:

1. No checkpoints.
2. No job exit codes. Job exit codes are not available (when using **gt2** and **gt3**).
3. Limited platform availability. Windows support is not yet available.

### 5.3.3 The nordugrid Grid Type

NorduGrid is a project to develop free grid middleware named the Advanced Resource Connector (ARC). See the NorduGrid web page (http://www.nordugrid.org) for more information about NorduGrid software.

Condor jobs may be submitted to NorduGrid resources using the **grid** universe. The **grid_resource** command specifies the name of the NorduGrid resource as follows:

```
grid_resource = nordugrid ng.example.com
```

NorduGrid uses X.509 credentials for authentication, usually in the form a proxy certificate. For more information about proxies and certificates, please consult the Alliance PKI pages at http://archive.ncsa.uiuc.edu/SCD/Alliance/GridSecurity/. *condor_submit* looks in default locations for the proxy. The submit description file command **x509userproxy** is used to give the full path name to the directory containing the proxy, when the proxy is not in a default location. If this

optional command is not present in the submit description file, then the value of the environment variable `X509_USER_PROXY` is checked for the location of the proxy. If this environment variable is not present, then the proxy in the file `/tmp/x509up_uXXXX` is used, where the characters `XXXX` in this file name are replaced with the Unix user id.

NorduGrid uses RSL syntax to describe jobs. The submit description file command **nordugrid_rsl** adds additional attributes to the job RSL that Condor constructs. The format this submit description file command is

```
nordugrid_rsl = (name=value)(name=value)
```

### 5.3.4 The unicore Grid Type

Unicore is a Java-based grid scheduling system. See http://unicore.sourceforge.net for more information about Unicore.

Condor jobs may be submitted to Unicore resources using the **grid** universe. The **grid_resource** command specifies the name of the Unicore resource as follows:

```
grid_resource = unicore usite.example.com vsite
```

**usite.example.com** is the host name of the Unicore gateway machine to which the Condor job is to be submitted. **vsite** is the name of the Unicore virtual resource to which the Condor job is to be submitted.

Unicore uses certificates stored in a Java keystore file for authentication. The following submit description file commands are required to properly use the keystore file.

**keystore_file** Specifies the complete path and file name of the Java keystore file to use.

**keystore_alias** A string that specifies which certificate in the Java keystore file to use.

**keystore_passphrase_file** Specifies the complete path and file name of the file containing the passphrase protecting the certificate in the Java keystore file.

### 5.3.5 The pbs Grid Type

The popular PBS (Portable Batch System) comes in several varieties: OpenPBS (http://www.openpbs.org), PBS Pro (http://www.altair.com/software/pbspro.htm), and Torque (http://www.clusterresources.com/pages/products/torque-resource-manager.php).

Condor jobs are submitted to a local PBS system using the **grid** universe and the **grid_resource** command by placing the following into the submit description file.

```
grid_resource = pbs
```

The pbs grid type requires two variables to be set in the Condor configuration file. PBS_GAHP is the path to the PBS GAHP server binary that is to be used to submit PBS jobs. GLITE_LOCATION is the path to the directory containing the GAHP's configuration file and auxillary binaries. In the Condor distribution, these files are located in $(LIB)/glite. The PBS GAHP's configuration file is in $(GLITE_LOCATION)/etc/batch_gahp.config. The PBS GAHP's auxillary binaries are to be in the directory $(GLITE_LOCATION)/bin. The Condor configuration file appears

```
GLITE_LOCATION = $(LIB)/glite
PBS_GAHP       = $(GLITE_LOCATION)/bin/batch_gahp
```

The PBS GAHP's configuration file contains two variables that must be modified to tell it where to find PBS on the local system. pbs_binpath is the directory that contains the PBS binaries. pbs_spoolpath is the PBS spool directory.

### 5.3.6 The lsf Grid Type

Condor jobs may be submitted to the Platform LSF batch system. See the Products page of the Platform web page at http://www.platform.com/Products/ for more information about Platform LSF.

Condor jobs are submitted to a local Platform LSF system using the **grid** universe and the **grid_resource** command by placing the following into the submit description file.

```
grid_resource = lsf
```

The lsf grid type requires two variables to be set in the Condor configuration file. LSF_GAHP is the path to the LSF GAHP server binary that is to be used to submit Platform LSF jobs. GLITE_LOCATION is the path to the directory containing the GAHP's configuration file and auxillary binaries. In the Condor distribution, these files are located in $(LIB)/glite. The LSF GAHP's configuration file is in $(GLITE_LOCATION)/etc/batch_gahp.config. The LSF GAHP's auxillary binaries are to be in the directory $(GLITE_LOCATION)/bin. The Condor configuration file appears

```
GLITE_LOCATION = $(LIB)/glite
LSF_GAHP       = $(GLITE_LOCATION)/bin/batch_gahp
```

The LSF GAHP's configuration file contains two variables that must be modified to tell it where to find LSF on the local system. lsf_binpath is the directory that contains the LSF binaries. lsf_confpath is the location of the LSF configuration file.

### 5.3.7 Matchmaking in the Grid Universe

In a simple usage, the grid universe allows users to specify a single grid site as a destination for jobs. This is sufficient when a user knows exactly which grid site they wish to use, or a higher-level

resource broker (such as the European Data Grid's resource broker) has decided which grid site should be used.

When a user has a variety of grid sites to choose from, Condor allows matchmaking of grid universe jobs to decide which grid resource a job should run on. Please note that this form of matchmaking is relatively new. There are some rough edges as continual improvement occurs.

To facilitate Condor's matching of jobs with grid resources, both the jobs and the grid resources are involved. The job's submit description file provides all commands needed to make the job work on a matched grid resource. The grid resource identifies itself to Condor by advertising a ClassAd. This ClassAd specifies all necessary attributes, such that Condor can properly make matches. The grid resource identification is accomplished by using *condor_advertise* to send a ClassAd representing the grid resource, which is then used by Condor to make matches.

#### Job Submission

To submit a grid universe job intended for a single, specific **gt2** resource, the submit description file for the job explicitly specifies the resource:

```
grid_resource = gt2 grid.example.com/jobmanager-pbs
```

If there were multiple **gt2** resources that might be matched to the job, the submit description file changes:

```
grid_resource   = $$(resource_name)
requirements    = TARGET.resource_name =!= UNDEFINED
```

The **grid_resource** command uses a substitution macro. The substitution macro defines the value of `resource_name` using attributes as specified by the matched grid resource. The **requirements** command further restricts that the job may only run on a machine (grid resource) that defines `grid_resource`. Note that this attribute name is invented for this example. To make matchmaking work in this way, both the job (as used here within the submit description file) and the grid resource (in its created and advertised ClassAd) must agree upon the name of the attribute.

As a more complex example, consider a job that wants to run not only on a **gt2** resource, but on one that has the Bamboozle software installed. The complete submit description file might appear:

```
universe        = grid
executable      = analyze_bamboozle_data
output          = aaa.$(Cluster).out
error           = aaa.$(Cluster).err
log             = aaa.log
grid_resource   = $$(resource_name)
requirements    = (TARGET.HaveBamboozle == True) && (TARGET.resource_name =!= UNDEFINED)
queue
```

Any grid resource which has the `HaveBamboozle` attribute defined as well as set to `True` is further checked to have the `resource_name` attribute defined. Where this occurs, a match may be made (from the job's point of view). A grid resource that has one of these attributes defined, but not the other results in no match being made.

Note that the entire value of **grid_resource** comes from the grid resource's ad. This means that the job can be matched with a resource of any type, not just **gt2**.

### Advertising Grid Resources to Condor

Any grid resource that wishes to be matched by Condor with a job must advertise itself to Condor using a ClassAd. To properly advertise, a ClassAd is sent periodically to the *condor_collector* daemon. A ClassAd is a list of pairs, where each pair consists of an attribute name and value that describes an entity. There are two entities relevant to Condor: a job, and a machine. A grid resource is a machine. The ClassAd describes the grid resource, as well as identifying the capabilities of the grid resource. It may also state both requirements and preferences (called **rank**) for the jobs it will run. See Section 2.3 for an overview of the interaction between matchmaking and ClassAds. A list of common machine ClassAd attributes is given in Section 2.5.2.

To advertise a grid site, place the attributes in a file. Here is a sample ClassAd that describes a grid resource that is capable of running a **gt2** job.

```
# example grid resource ClassAd for a gt2 job
MyType         = "Machine"
TargetType     = "Job"
Name           = "Example1_Gatekeeper"
Machine        = "Example1_Gatekeeper"
resource_name  = "gt2 grid.example.com/jobmanager-pbs"
UpdateSequenceNumber  = 4
Requirements   = (TARGET.JobUniverse == 9)
Rank           = 0.000000
CurrentRank    = 0.000000
```

Some attributes are defined as expressions, while others are integers, floating point values, or strings. The type is important, and must be correct for the ClassAd to be effective. The attributes

```
MyType         = "Machine"
TargetType     = "Job"
```

identify the grid resource as a machine, and that the machine is to be matched with a job. In Condor, machines are matched with jobs, and jobs are matched with machines. These attributes are strings. Strings are surrounded by double quote marks.

The attributes `Name` and `Machine` are likely to be defined to be the same string value as in the example:

```
Name           = "Example1_Gatekeeper"
Machine        = "Example1_Gatekeeper"
```

Both give the fully qualified host name for the resource. The `Name` may be different on an SMP machine, where the individual CPUs are given names that can be distiguished from each other. Each separate grid resource must have a unique name.

Where the job depends on the resource to specify the value of the **grid_resource** command by the use of the substitution macro, the ClassAd for the grid resource (machine) defines this value. The example given as

```
grid_resource = "gt2 grid.example.com/jobmanager-pbs"
```

defines this value. Note that the invented name of this variable must match the one utilized within the submit description file. To make the matchmaking work, both the job (as used within the submit description file) and the grid resource (in this created and advertised ClassAd) must agree upon the name of the attribute.

A machine's ClassAd information can be time sensitive, and may change over time. Therefore, ClassAds expire and are thrown away. In addition, the communication method by which ClassAds are sent implies that entire ads may be lost without notice or may arrive out of order. Out of order arrival leads to the definition of an attribute which provides an ordering. This positive integer value is given in the example ClassAd as

```
UpdateSequenceNumber  = 4
```

This value must increase for each subsequent ClassAd. If state information for the ClassAd is kept in a file, a script executed each time the ClassAd is to be sent may use a counter for this value. An alternative for a stateless implementation sends the current time in seconds (since the epoch, as given by the C `time()` function call).

The requirements that the grid resource sets for any job that it will accept are given as

```
Requirements     = (TARGET.JobUniverse == 9)
```

This set of requirments state that any job is required to be for the **grid** universe.

The attributes

```
Rank             = 0.000000
CurrentRank      = 0.000000
```

are both necessary for Condor's negotiation to procede, but are not relevant to grid matchmaking. Set both to the floating point value 0.0.

The example machine ClassAd becomes more complex for the case where the grid resource allows matches with more than one job:

```
# example grid resource ClassAd for a gt2 job
MyType         = "Machine"
TargetType     = "Job"
Name           = "Example1_Gatekeeper"
Machine        = "Example1_Gatekeeper"
```

```
resource_name  = "gt2 grid.example.com/jobmanager-pbs"
UpdateSequenceNumber  = 4
Requirements   = (CurMatches < 10) && (TARGET.JobUniverse == 9)
Rank           = 0.000000
CurrentRank    = 0.000000
WantAdRevaluate = True
CurMatches     = 1
```

In this example, the two attributes `WantAdRevaluate` and `CurMatches` appear, and the `Requirements` expression has changed.

`WantAdRevaluate` is a boolean value, and may be set to either `True` or `False`. When `True` in the ClassAd and a match is made (of a job to the grid resource), the machine (grid resource) is not removed from the set of machines to be considered for further matches. This implements the ability for a single grid resource to be matched to more than one job at a time. Note that the spelling of this attribute is incorrect, and remains incorrect to maintain backward compatibility.

To limit the number of matches made to the single grid resource, the resource must have the ability to keep track of the number of Condor jobs it has. This integer value is given as the `CurMatches` attribute in the advertised ClassAd. It is then compared in order to limit the number of jobs matched with the grid resource.

```
Requirements   = (CurMatches < 10) && (TARGET.JobUniverse == 9)
CurMatches     = 1
```

This example assumes that the grid resource already has one job, and is willing to accept a maximum of 9 jobs. If `CurMatches` does not appear in the ClassAd, Condor uses a default value of 0.

This ClassAd (likely in a file) is to be periodically sent to the *condor_collector* daemon using *condor_advertise*. A recommended implementation uses a script to create or modify the ClassAd together with *cron* to send the ClassAd every five minutes. The *condor_advertise* program must be installed on the machine sending the ClassAd, but the remainder of Condor does not need to be installed. The required argument for the *condor_advertise* command is *UPDATE_STARTD_AD*.

*condor_advertise* uses UDP to transmit the ClassAd. Where this is insufficient, specify the **-tcp** option to *condor_advertise* to use TCP for communication.

#### Advanced usage

What if a job fails to run at a grid site due to an error? It will be returned to the queue, and Condor will attempt to match it and re-run it at another site. Condor isn't very clever about avoiding sites that may be bad, but you can give it some assistance. Let's say that you want to avoid running at the last grid site you ran at. You could add this to your job description:

```
match_list_length = 1
Rank              = TARGET.Name != LastMatchName0
```

This will prefer to run at a grid site that was not just tried, but it will allow the job to be run there if there is no other option.

When you specify **match_list_length**, you provide an integer N, and Condor will keep track of the last N matches. The oldest match will be LastMatchName0, and next oldest will be LastMatchName1, and so on. (See the *condor_submit* manual page for more details.) The Rank expression allows you to specify a numerical ranking for different matches. When combined with **match_list_length**, you can prefer to avoid sites that you have already run at.

In addition, *condor_submit* has two options to help you control grid universe job resubmissions and rematching. See **globus_resubmit** and **globus_rematch** in the *condor_submit* manual page. These options are independent of **match_list_length**.

There are some new attributes that will be added to the Job ClassAd, and may be useful to you when you write your rank, requirements, globus_resubmit or globus_rematch option. Please refer to Section 2.5.2 and read about the following option:

- NumJobMatches
- NumGlobusSubmits
- NumSystemHolds
- HoldReason
- ReleaseReason
- EnteredCurrentStatus
- LastMatchTime
- LastRejMatchTime
- LastRejMatchReason

The following example of a command within the submit description file releases jobs 5 minutes after being held, increasing the time between releases by 5 minutes each time. It will continue to retry up to 4 times per Globus submission, plus 4. The plus 4 is necessary in case the job goes on hold before being submitted to Globus, although this is unlikely.

```
periodic_release = ( NumSystemHolds <= ((NumGlobusSubmits * 4) + 4) ) \
   && (NumGlobusSubmits < 4) && \
   ( HoldReason != "via condor_hold (by user $ENV(USER))" ) && \
   ((CurrentTime - EnteredCurrentStatus) > ( NumSystemHolds *60*5 ))
```

The following example forces Globus resubmission after a job has been held 4 times per Globus submission.

```
globus_resubmit = NumSystemHolds == (NumGlobusSubmits + 1) * 4
```

If you are concerned about unknown or malicious grid sites reporting to your *condor_collector*, you should use Condor's security options, documented in Section 3.6.

## 5.4 Glidein

Glidein is a mechanism by which one or more grid resources (remote machines) temporarily join a local Condor pool. The program *condor_glidein* is used to add a machine to a Condor pool. During the period of time when the added resource is part of the local pool, the resource is visible to users of the pool. But, by default, the resource is only available for use by the user that added the resource to the pool.

After glidein, the user may submit jobs for execution on the added resource the same way that all Condor jobs are submitted. To force a submitted job to run on the added resource, the submit description file could contain a requirement that the job run specifically on the added resource.

### 5.4.1 What *condor_glidein* Does

*condor_glidein* works by installing and executing necessary Condor daemons and configuration on the remote resource, such that the resource reports to and joins the local pool. *condor_glidein* accomplishes two separate tasks towards having a remote grid resource join the local Condor pool. They are the set up task and the execution task.

The set up task generates necessary configuration files and locates proper platform-dependent binaries for the Condor daemons. A script is also generated that can be used during the execution task to invoke the proper Condor daemons. These files are copied to the remote resource as necessary. The configuration variable `GLIDEIN_SERVER_URLS` defines a list of locations from which the necessary binaries are obtained. Default values cause binaries to be downloaded from the UW site. See section 3.3.22 on page 203 for a full definition of this configuration variable.

When the files are correctly in place, the execution task starts the Condor daemons. *condor_glidein* does this by submitting a Condor job to run under the grid universe. The job runs the *condor_master* on the remote grid resource. The *condor_master* invokes other daemons, which contact the local pool's *condor_collector* to join the pool. The Condor daemons exit gracefully when no jobs run on the daemons for a preset period of time.

Here is an example of how a glidein resource appears, similar to how any other machine appears. The name has a slightly different form, in order to handle the possibility of multiple instances of glidein daemons inhabiting a multi-processor machine.

```
% condor_status | grep denal
7591386@denal LINUX       INTEL  Unclaimed  Idle       3.700  24064  0+00:06:35
```

### 5.4.2 Configuration Requirements in the Local Pool

As remote grid resources join the local pool, these resources must report to the local pool's *condor_collector* daemon. Security demands that the local pool's *condor_collector* list all hosts from which they will accept communication. Therefore, all remote grid resources accepted for glidein must be given `HOSTALLOW_WRITE` permission. An expected way to do this is to modify the empty variable (within the sample configuration file) `GLIDEIN_SITES` to list all remote grid resources accepted for glidein. The list is a space or comma separated list of hosts. This list is then given the proper permissions by an additional redefinition of the `HOSTALLOW_WRITE` configuration variable, to also include the list of hosts as in the following example.

```
GLIDEIN_SITES = A.example.com, B.example.com, C.example.com
HOSTALLOW_WRITE = $(HOSTALLOW_WRITE) $(GLIDEIN_SITES)
```

Recall that for configuration file changes to take effect, *condor_reconfig* must be run.

If this configuration change to the security settings on the local Condor pool cannot be made, an additional Condor pool that utilizes personal Condor may be defined. The single machine pool may coexist with other instances of Condor. *condor_glidein* is executed to have the remote grid resources join this personal Condor pool.

### 5.4.3 Running Jobs on the Remote Grid Resource After Glidein

Once the Globus resource has been added to the local Condor pool with *condor_glidein*, job(s) may be submitted. To force a job to run on the Globus resource, specify that Globus resource as a machine requirement in the submit description file. Here is an example from within the submit description file that forces submission to the Globus resource denali.mcs.anl.gov:

```
requirements = ( machine == "denali.mcs.anl.gov" ) \
   && FileSystemDomain != "" \
   && Arch != "" && OpSys != ""
```

This example requires that the job run only on denali.mcs.anl.gov, and it prevents Condor from inserting the file system domain, architecture, and operating system attributes as requirements in the matchmaking process. Condor must be told not to use the submission machine's attributes in those cases where the Globus resource's attributes do not match the submission machine's attributes and your job really is capable of running on the target machine. You may want to use Condor's file-transfer capabilities in order to copy input and output files back and forth between the submission and execution machine.

## 5.5 Dynamic Deployment

See section 3.2.10 for a complete description of Condor's dynamic deployment tools.

Condor's dynamic deployment tools (*condor_cold_start* and *condor_glidein*) allow new pools of resources to be incorporated on the fly. While Condor is able to manage compute jobs remotely through Globus and other grid-computing protocols, dynamic deployment of Condor makes it possible to go one step further. Condor remotely installs and runs portions of itself. This process of Condor gliding in to inhabit computing resources on demand leverages the lowest common denominator of grid middleware systems, simple program execution, to bind together resources in a heterogeneous computing grid, with different management policies and different job execution methods, into a full-fledged Condor system.

The mobility of Condor services also benefits from the development of Condor-C, which provides a richer tool set for interlinking Condor-managed computers. Condor-C is a protocol that allows one Condor scheduler to delegate jobs to another Condor scheduler. The second scheduler could be at a remote site and/or an entry point into a restricted network. Delegating details of managing a job achieves greater flexibility with respect to network architecture, as well as fault tolerance and scalability. In the context of glide in deployments, the beach-head for each compute site is a dynamically deployed Condor scheduler which then serves as a target for Condor-C traffic.

In general, the mobility of the Condor scheduler and job execution agents, and the flexibility in how these are interconnected provide a uniform and feature-rich platform that can expand onto diverse resources and environments when the user requires it.

CHAPTER

# SIX

# Platform-Specific Information

The Condor Team strives to make Condor work the same way across all supported platforms. However, because Condor is a very low-level system which interacts closely with the internals of the operating systems on which it runs, this goal is not always possible to achieve. The following sections provide detailed information about using Condor on different computing platforms and operating systems.

## 6.1 Linux

This section provides information specific to the Linux port of Condor. Linux is a difficult platform to support. It changes very frequently, and Condor has some extremely system-dependent code (for example, the checkpointing library).

Condor is sensitive to changes in the following elements of the system:

- The kernel version
- The version of the GNU C library (glibc)
- the version of GNU C Compiler (GCC) used to build and link Condor jobs (this only matters for Condor's Standard universe which provides checkpointing and remote system calls)

The Condor Team tries to provide support for various releases of the distribution of Linux. Red Hat is probably the most popular Linux distribution, and it provides a common set of versions for the above system components at which Condor can aim support. Condor will often work with Linux distributions other than Red Hat (for example, Debian or SuSE) that have the same versions of the

above components. However, we do not usually test Condor on other Linux distributions and we do not provide any guarantees about this.

New releases of Red Hat usually change the versions of some or all of the above system-level components. A version of Condor that works with one release of Red Hat might not work with newer releases. The following sections describe the details of Condor's support for the currently available versions of Red Hat Linux on x86 architecture machines.

### 6.1.1 Linux Kernel-specific Information

Distributions that rely on the Linux 2.4.x and all Linux 2.6.x kernels through version 2.6.10 do not modify the `atime` of the input device file. This leads to difficulty when Condor is run using one of these kernels. The problem manifests itself in that Condor cannot properly detect keyboard or mouse activity. Therefore, using the activity in policy setting cannot signal that Condor should stop running a job on a machine.

Condor version 6.6.8 implements a workaround for PS/2 devices. A better fix is the Linux 2.6.10 kernel patch linked to from the directions posted at http://www.cs.wisc.edu/condor/kernel.patch.html. This patch works better for PS/2 devices, and may also work for USB devices. A future version of Condor will implement better recognition of USB devices, such that the kernel patch will also definitively work for USB devices.

### 6.1.2 Red Hat Version 6.x

Red Hat version 6.x is an older release of Red Hat, but it is still used by a number of sites. Since Condor has worked on this platform since it was first released, we still provide binaries that support it. Red Hat 6.2 uses the 2.2.x Linux kernel series, glibc version 2.1.x, and GCC version egcs-1.1.2. To run Condor on this platform and be able to use *condor_compile*, you should download the Condor Version 6.8.6 binaries listed for use with "Linux 2.0.x and 2.2.x (glibc 2.1)".

### 6.1.3 Red Hat Version 7.x

Red Hat version 7.x is fully supported in Condor Version 6.8.6. *condor_compile* works to link user jobs for the Standard universe with the versions of GCC and glibc that comes with Red Hat 7.x. Both the statically linked and dynamically linked versions of the Condor binaries listed for use with "Linux 2.4.x (glibc 2.2) - Red Hat 7.1, 7.2, 7.3" will work with no additional effort.

### 6.1.4 Red Hat Version 8.x

Red Hat version 8.x is fully supported in Condor Version 6.8.6. *condor_compile* works to link user jobs for the Standard universe with the versions of gcc and glibc that come with Red Hat 8.x.

### 6.1.5 Red Hat Version 9.x

Red Hat version 9.x is fully supported in Condor Version 6.8.6. *condor_compile* works to link user jobs for the Standard universe with the versions of gcc and glibc that come with Red Hat 9.x.

### 6.1.6 Red Hat Fedora 1, 2, and 3

Redhat Fedora Core 1, 2, and 3 now support the checkpointing of statically linked executables just like previous revisions of Condor for Red Hat. *condor_compile* works to link user jobs for the Standard universe with the versions of gcc that are distributed with Red Hat Fedora Core 1, 2, and 3.

However, there are some caveats: A) You must install and use the dynamic Red Hat 9.x binaries on the Fedora machine and B) if you wish to do run a *condor_compile*d binary in standalone mode(either initially or in resumption mode), then you must prepend the execution of said binary with **setarch i386**. Here is an example: suppose we have a Condor-linked binary called **myapp**, running this application as a standalone executable will result in this command: **setarch i386 myapp**. The subsequent resumption command will be: **setarch i386 myapp -_condor_restart myapp.ckpt**.

When standard universe executables *condor_compile*d under any currently supported Linux architecture of the same kind (including Fedora 1, 2, and 3) are running inside Condor, they will automatically execute in the i386 execution domain. This means that the **exec_shield** functionality (if available) will be turned off and the shared segment layout will default to Red Hat 9 style. There is no need to do the above instructions concerning **setarch** if the executables are being submitted directly into Condor via *condor_submit*.

## 6.2 Microsoft Windows

Windows is a strategic platform for Condor, and therefore we have been working toward a complete port to Windows. Our goal is to make Condor every bit as capable on Windows as it is on Unix – or even more capable.

Porting Condor from Unix to Windows is a formidable task, because many components of Condor must interact closely with the underlying operating system. Instead of waiting until all components of Condor are running and stabilized on Windows, we have decided to make a clipped version of Condor for Windows. A clipped version is one in which there is no checkpointing and there are no remote system calls.

This section contains additional information specific to running Condor on Windows. Eventually this information will be integrated into the Condor Manual as a whole, and this section will disappear. In order to effectively use Condor, first read the overview chapter (section 1.1) and the user's manual (section 2.1). If you will also be administrating or customizing the policy and set up of Condor, also read the administrator's manual chapter (section 3.1). After reading these chapters, review the information in this chapter for important information and differences when using

and administrating Condor on Windows. For information on installing Condor for Windows, see section 3.2.6.

### 6.2.1 Limitations under Windows

In general, this release for Windows works the same as the release of Condor for Unix. However, the following items are not supported in this version:

- The Standard and PVM job universes are not present. This means transparent process checkpoint/migration and remote system calls are not supported.
- For **grid** universe jobs, the only supported grid type is **condor**.
- Accessing files via a network share that requires a kerberos ticket (such as AFS) is not yet supported.

### 6.2.2 Supported Features under Windows

Except for those items listed above, most everything works the same way in Condor as it does in the Unix release. This release is based on the Condor Version 6.8.6 source tree, and thus the feature set is the same as Condor Version 6.8.6 for Unix. For instance, all of the following work in Condor:

- The ability to submit, run, and manage queues of jobs running on a cluster of Windows machines.
- All tools such as *condor_q*, *condor_status*, *condor_userprio*, are included. Only *condor_compile* is *not* included.
- The ability to customize job policy using ClassAds. The machine ClassAds contain all the information included in the Unix version, including current load average, RAM and virtual memory sizes, integer and floating-point performance, keyboard/mouse idle time, etc. Likewise, job ClassAds contain a full complement of information, including system dependent entries such as dynamic updates of the job's image size and CPU usage.
- Everything necessary to run a Condor central manager on Windows.
- Security mechanisms.
- Support for SMP machines.
- Condor for Windows can run jobs at a lower operating system priority level. Jobs can be suspended, soft-killed by using a WM_CLOSE message, or hard-killed automatically based upon policy expressions. For example, Condor can automatically suspend a job whenever keyboard/mouse or non-Condor created CPU activity is detected, and continue the job after the the machine has been idle for a specified amount of time.

- Condor correctly manages jobs which create multiple processes. For instance, if a Condor job spawns multiple processes and Condor needs to kill the job, all processes created by the job will be terminated.
- In addition to interactive tools, users and administrators can receive information from Condor by e-mail (standard SMTP) and/or by log files.
- Condor includes a friendly GUI installation and set up program, which can perform a full install or deinstall of Condor. Information specified by the user in the set up program is stored in the system registry. The set up program can update a current installation with a new release using a minimal amount of effort.

### 6.2.3 Secure Password Storage

In order for Condor to operate properly, it must at times be able to act on behalf of users who submit jobs. In particular, this is required on submit machines so that Condor can access a job's input files, create and access the job's output files, and write to the job's log file from within the appropriate security context. It may also be desirable for Condor to execute the job itself under the security context of its submitting user (see 6.2.4 for details on running jobs as the submitting user on Windows).

On Unix systems, arbitrarily changing what user Condor performs its actions as is easily done when Condor is started with root privileges. On Windows, however, performing an action as a particular user requires knowledge of that user's password, even when running at the maximum privilege level.

Condor on Windows supports the notion of *user privilege switching* through the use of a secure password store. Users can provide Condor with their passwords using the *condor_store_cred* tool. Passwords managed by Condor are encrypted and stored at a secure location within the Windows registry. When Condor needs to perform an action as a particular user, it can then use the securely stored password to do so.

The secure password store can be managed by the *condor_schedd*. This is Condor's default behavior, and is usually a good approach in environments where the user's password is only needed on the submit machine. This occurs when users are are not allowed to submit jobs that run under the security context of the submitting user.

In environments where users can submit Condor jobs that run using their Windows accounts, it is necessary to configure a centralized *condor_credd* daemon to manage the secure password store. This makes a user's password available, via an encrypted connection to the *condor_credd*, to any execute machine that may need to execute a job under the user's Windows account.

The `condor_config.local.credd` example file, included in the `etc` subdirectory of the Condor distribution, demonstrates how to configure a Condor pool to use the *condor_credd* for password managment.

The following configuration macros are needed for all hosts that share a *condor_credd* daemon for password management. These will typically be placed in the global Condor configuration file.

- CREDD_HOST - This is the name of the machine that runs the *condor_credd*.
- CREDD_CACHE_LOCALLY - This affects Condor's behavior when a daemon does a password fetch operation to the *condor_credd*. If CREDD_CACHE_LOCALLY is True, the first successful fetch of a user's password will result in the password being stashed in a local secure password store. Subsequent uses of that user's password will not require communication with the *condor_credd*. If not defined, the default value is False.

Careful attention must be given to the *condor_credd* daemon's security configuration. All communication with the *condor_credd* daemon should be strongly authenticated and encrypted. The condor_config.local.credd file configures the *condor_credd* daemon to only accept password store requests from users authenticated using the NTSSPI authentication method. Password fetch requests must come from Condor daemons authenticated using a shared secret via the password authentication method. Both types of traffic are required to be encrypted. Please refer to section 3.6.1 for details on configuring security in Condor.

### 6.2.4 Executing Jobs as the Submitting User

By default, Condor executes jobs on Windows using a dedicated "run account" that has minimal access rights and privileges. As an alternative, Condor can be configured to run a user's jobs using their own account if the job owner wishes. This may be useful if the job needs to access files on a network share, or access other resources that aren't available to a low-privilege run account. To enable this feature, the following steps must be taken.

- Execute machines must have access to users' passwords so they may log into a user's account before running jobs on their behalf. This can be accomplished through the use of a central *condor_credd*. Please refer to section 6.2.3 for more information on password storage and the *condor_credd*.
- The boolean configuration parameter STARTER_ALLOW_RUNAS_OWNER must be set to True on all execute machines.

A user that then wants a job to run using their own account can simply use the **run_as_owner** command in the job's submit file as follows:

```
run_as_owner = true
```

### 6.2.5 Details on how Condor for Windows starts/stops a job

This section provides some details on how Condor starts and stops jobs. This discussion is geared for the Condor administrator or advanced user who is already familiar with the material in the Administrator's Manual and wishes to know detailed information on what Condor does when starting and stopping jobs.

When Condor is about to start a job, the *condor_startd* on the execute machine spawns a *condor_starter* process. The *condor_starter* then creates:

1. a run account on the machine with a login name of "condor-reuse-vmX", where X is the Virtual Machine number of the *condor_starter*. This account is added to group Users. This step is skipped if the job is to be run using the submitting user's account (see section 6.2.4).
2. a new temporary working directory for the job on the execute machine. This directory is named "dir_XXX", where XXX is the process ID of the *condor_starter*. The directory is created in the `$(EXECUTE)` directory as specified in Condor's configuration file. Condor then grants write permission to this directory for the user account newly created for the job.
3. a new, non-visible Window Station and Desktop for the job. Permissions are set so that only the account that will run the job has access rights to this Desktop. Any windows created by this job are not seen by anyone; the job is run in the background. (Note: Setting `USE_VISIBLE_DESKTOP` to True will allow the job to access the default desktop instead of a newly created one.)

Next, the *condor_starter* (called the starter) contacts the *condor_shadow* (called the shadow) process, which is running on the submitting machine, and pulls over the job's executable and input files. These files are placed into the temporary working directory for the job. After all files have been received, the starter spawns the user's executable. Its current working directory set to the temporary working directory (that is, `$(EXECUTE)`/dir_XXX, where XXX is the process id of the *condor_starter* daemon).

While the job is running, the starter closely monitors the CPU usage and image size of all processes started by the job. Every 20 minutes the starter sends this information, along with the total size of all files contained in the job's temporary working directory, to the shadow. The shadow then inserts this information into the job's ClassAd so that policy and scheduling expressions can make use of this dynamic information.

If the job exits of its own accord (that is, the job completes), the starter first terminates any processes started by the job which could still be around if the job did not clean up after itself. The starter examines the job's temporary working directory for any files which have been created or modified and sends these files back to the shadow running on the submit machine. The shadow places these files into the **initialdir** specified in the submit description file; if no **initialdir** was specified, the files go into the directory where the user invoked *condor_submit*. Once all the output files are safely transferred back, the job is removed from the queue. If, however, the *condor_startd* forcibly kills the job before all output files could be transferred, the job is not removed from the queue but instead switches back to the Idle state.

If the *condor_startd* decides to vacate a job prematurely, the starter sends a WM_CLOSE message to the job. If the job spawned multiple child processes, the WM_CLOSE message is only sent to the parent process (that is, the one started by the starter). The WM_CLOSE message is the preferred way to terminate a process on Windows, since this method allows the job to cleanup and free any resources it may have allocated. When the job exits, the starter cleans up any processes left behind. At this point, if **transfer_files** is set to *ONEXIT* (the default) in the job's submit description file, the

job switches from states, from Running to Idle, and no files are transferred back. If **transfer_files** is set to *ALWAYS*, then any files in the job's temporary working directory which were changed or modified are first sent back to the submitting machine. But this time, the shadow places these so-called intermediate files into a subdirectory created in the `$(SPOOL)` directory on the submitting machine (`$(SPOOL)` is specified in Condor's configuration file). The job is then switched back to the Idle state until Condor finds a different machine on which to run. When the job is started again, Condor places into the job's temporary working directory the executable and input files as before, *plus* any files stored in the submit machine's `$(SPOOL)` directory for that job.

NOTE: A Windows console process can intercept a WM_CLOSE message via the Win32 SetConsoleCtrlHandler() function if it needs to do special cleanup work at vacate time; a WM_CLOSE message generates a CTRL_CLOSE_EVENT. See SetConsoleCtrlHandler() in the Win32 documentation for more info.

NOTE: The default handler in Windows for a WM_CLOSE message is for the process to exit. Of course, the job could be coded to ignore it and not exit, but eventually the *condor_startd* will become impatient and hard-kill the job (if that is the policy desired by the administrator).

Finally, after the job has left and any files transferred back, the starter deletes the temporary working directory, the temporary account (if one was created), the WindowStation, and the Desktop before exiting. If the starter should terminate abnormally, the *condor_startd* attempts the clean up. If for some reason the *condor_startd* should disappear as well (that is, if the entire machine was power-cycled hard), the *condor_startd* will clean up when Condor is restarted.

### 6.2.6 Security Considerations in Condor for Windows

On the execute machine (by default), the user job is run using the access token of an account dynamically created by Condor which has bare-bones access rights and privileges. For instance, if your machines are configured so that only Administrators have write access to `C:\WINNT`, then certainly no Condor job run on that machine would be able to write anything there. The only files the job should be able to access on the execute machine are files accessible by the Users and Everyone groups, and files in the job's temporary working directory. Of course, if the job is configured to run using the account of the submitting user (as described in section 6.2.4), it will be able to do anything that the user is able to do on the execute machine it runs on.

On the submit machine, Condor impersonates the submitting user, therefore the File Transfer mechanism has the same access rights as the submitting user. For example, say only Administrators can write to `C:\WINNT` on the submit machine, and a user gives the following to *condor_submit* :

```
executable = mytrojan.exe
initialdir = c:\winnt
output = explorer.exe
queue
```

Unless that user is in group Administrators, Condor will not permit `explorer.exe` to be overwritten.

If for some reason the submitting user's account disappears between the time *condor_submit* was run and when the job runs, Condor is not able to check and see if the now-defunct submitting user has read/write access to a given file. In this case, Condor will ensure that group "Everyone" has read or write access to any file the job subsequently tries to read or write. This is in consideration for some network setups, where the user account only exists for as long as the user is logged in.

Condor also provides protection to the job queue. It would be bad if the integrity of the job queue is compromised, because a malicious user could remove other user's jobs or even change what executable a user's job will run. To guard against this, in Condor's default configuration all connections to the *condor_schedd* (the process which manages the job queue on a given machine) are authenticated using Windows' SSPI security layer. The user is then authenticated using the same challenge-response protocol that Windows uses to authenticate users to Windows file servers. Once authenticated, the only users allowed to edit job entry in the queue are:

1. the user who originally submitted that job (i.e. Condor allows users to remove or edit their own jobs)
2. users listed in the `condor_config` file parameter `QUEUE_SUPER_USERS`. In the default configuration, only the "SYSTEM" (LocalSystem) account is listed here.

WARNING: Do not remove "SYSTEM" from `QUEUE_SUPER_USERS`, or Condor itself will not be able to access the job queue when needed. If the LocalSystem account on your machine is compromised, you have all sorts of problems!

To protect the actual job queue files themselves, the Condor installation program will automatically set permissions on the entire Condor release directory so that only Administrators have write access.

Finally, Condor has all the IP/Host-based security mechanisms present in the full-blown version of Condor. See section 3.6.10 starting on page 282 for complete information on how to allow/deny access to Condor based upon machine host name or IP address.

### 6.2.7 Network files and Condor

Condor can work well with a network file server. The recommended approach to having jobs access files on network shares is to configure jobs to run using the security context of the submitting user (see section 6.2.4). If this is done, the job will be able to access resources on the network in the same way as the user can when logged in interactively.

In some environments, running jobs as their submitting users is not a feasible option. This section outlines some possible alternatives. The heart of the difficulty in this case is that on the execute machine, Condor creates a temporary user that will run the job. The file server has never heard of this user before.

Choose one of these methods to make it work:

- METHOD A: access the file server as a different user via a net use command with a login and password
- METHOD B: access the file server as guest
- METHOD C: access the file server with a ”NULL” descriptor
- METHOD D: create and have Condor use a special account
- METHOD E: use the contrib module from the folks at Bristol University

All of these methods have advantages and disadvantages.

Here are the methods in more detail:

METHOD A - access the file server as a different user via a net use command with a login and password

Example: you want to copy a file off of a server before running it....

```
@echo off
net use \\myserver\someshare MYPASSWORD /USER:MYLOGIN
copy \\myserver\someshare\my-program.exe
my-program.exe
```

The idea here is to simply authenticate to the file server with a different login than the temporary Condor login. This is easy with the ”net use” command as shown above. Of course, the obvious disadvantage is this user’s password is stored and transferred as clear text.

METHOD B - access the file server as guest

Example: you want to copy a file off of a server before running it as GUEST

```
@echo off
net use \\myserver\someshare
copy \\myserver\someshare\my-program.exe
my-program.exe
```

In this example, you’d contact the server MYSERVER as the Condor temporary user. However, if you have the GUEST account enabled on MYSERVER, you will be authenticated to the server as user ”GUEST”. If your file permissions (ACLs) are setup so that either user GUEST (or group EVERYONE) has access the share ”someshare” and the directories/files that live there, you can use this method. The downside of this method is you need to enable the GUEST account on your file server. WARNING: This should be done *with extreme caution* and only if your file server is well protected behind a firewall that blocks SMB traffic.

METHOD C - access the file server with a ”NULL” descriptor

One more option is to use NULL Security Descriptors. In this way, you can specify which shares are accessible by NULL Descriptor by adding them to your registry. You can then use the batch file wrapper like:

```
net use z: \\myserver\someshare /USER:""
z:\my-program.exe
```

so long as 'someshare' is in the list of allowed NULL session shares. To edit this list, run regedit.exe and navigate to the key:

```
HKEY_LOCAL_MACHINE\
   SYSTEM\
     CurrentControlSet\
       Services\
         LanmanServer\
           Parameters\
             NullSessionShares
```

and edit it. unfortunately it is a binary value, so you'll then need to type in the hex ASCII codes to spell out your share. each share is separated by a null (0x00) and the last in the list is terminated with two nulls.

although a little more difficult to set up, this method of sharing is a relatively safe way to have one quasi-public share without opening the whole guest account. you can control specifically which shares can be accessed or not via the registry value mentioned above.

METHOD D - create and have Condor use a special account

Create a permanent account (called condor-guest in this description) under which Condor will run jobs. On all Windows machines, and on the file server, create the condor-guest account.

On the network file server, give the condor-guest user permissions to access files needed to run Condor jobs.

Securely store the password of the condor-guest user in the Windows registry using *condor_store_cred* on all Windows machines.

Tell Condor to use the condor-guest user as the owner of jobs, when required. Details for this are in section 3.6.12.

METHOD E - access with the contrib module from Bristol

Another option: some hardcore Condor users at Bristol University developed their own module for starting jobs under Condor NT to access file servers. It involves storing submitting user's passwords on a centralized server. Below I have included the README from this contrib module, which will soon appear on our website within a week or two. If you want it before that, let me know, and I could e-mail it to you.

Here is the README from the Bristol Condor contrib module:

```
README
Compilation Instructions
Build the projects in the following order

CondorCredSvc
CondorAuthSvc
Crun
Carun
AfsEncrypt
RegisterService
DeleteService
Only the first 3 need to be built in order. This just makes sure that the
RPC stubs are correctly rebuilt if required. The last 2 are only helper
applications to install/remove the services. All projects are Visual Studio
6 projects. The nmakefiles have been exported for each. Only the project
for Carun should need to be modified to change the location of the AFS
libraries if needed.

Details
CondorCredSvc
CondorCredSvc is a simple RPC service that serves the domain account
credentials. It reads the account name and password from the registry of
the machine it's running on. At the moment these details are stored in
clear text under the key

HKEY_LOCAL_MACHINE\Software\Condor\CredService

The account name and password are held in REG_SZ values "Account" and
"Password" respectively. In addition there is an optional REG_SZ value
"Port" which holds the clear text port number (e.g. "1234"). If this value
is not present the service defaults to using port 3654.

At the moment there is no attempt to encrypt the username/password when it
is sent over the wire - but this should be reasonably straightforward to
change. This service can sit on any machine so keeping the registry entries
secure ought to be fine. Certainly the ACL on the key could be set to only
allow administrators and SYSTEM access.

CondorAuthSvc and Crun
These two programs do the hard work of getting the job authenticated and
running in the right place. CondorAuthSvc actually handles the process
creation while Crun deals with getting the winstation/desktop/working
directory and grabbing the console output from the job so that Condor's
output handling mechanisms still work as advertised. Probably the easiest
```

way to see how the two interact is to run through the job creation process:

The first thing to realize is that condor itself only runs Crun.exe. Crun treats its command line parameters as the program to really run. e.g. "Crun \\mymachine\myshare\myjob.exe" actually causes \\mymachine\myshare\myjob.exe to be executed in the context of the domain account served by CondorCredSvc. This is how it works:

When Crun starts up it gets its window station and desktop - these are the ones created by condor. It also gets its current directory - again already created by condor. It then makes sure that SYSTEM has permission to modify the DACL on the window station, desktop and directory. Next it creates a shared memory section and copies its environment variable block into it. Then, so that it can get hold of STDOUT and STDERR from the job it makes two named pipes on the machine it's running on and attaches a thread to each which just prints out anything that comes in on the pipe to the appropriate stream. These pipes currently have a NULL DACL, but only one instance of each is allowed so there shouldn't be any issues involving malicious people putting garbage into them. The shared memory section and both named pipes are tagged with the ID of Crun's process in case we're on a multi-processor machine that might be running more than one job. Crun then makes an RPC call to CondorAuthSvc to actually start the job, passing the names of the window station, desktop, executable to run, current directory, pipes and shared memory section (it only attempts to call CondorAuthSvc on the same machine as it is running on). If the jobs starts successfully it gets the process ID back from the RPC call and then just waits for the new process to finish before closing the pipes and exiting. Technically, it does this by synchronizing on a handle to the process and waiting for it to exit. CondorAuthSvc sets the ACL on the process to allow EVERYONE to synchronize on it.

[ Technical note: Crun adds "C:\WINNT\SYSTEM32\CMD.EXE /C" to the start of the command line. This is because the process is created with the network context of the caller i.e. LOCALSYSTEM. Pre-pending cmd.exe gets round any unexpected "Access Denied" errors. ]

If Crun gets a WM_CLOSE (CTRL_CLOSE_EVENT) while the job is running it attempts to stop the job, again with an RPC call to CondorAuthSvc passing the job's process ID.

CondorAuthSvc runs as a service under the LOCALSYSTEM account and does the work of starting the job. By default it listens on port 3655, but this can be changed by setting the optional REG_SZ value "Port" under the registry key

HKEY_LOCAL_MACHINE\Software\Condor\AuthService

(Crun also checks this registry key when attempting to contact CondorAuthSvc.) When it gets the RPC to start a job CondorAuthSvc first connects to the pipes for STDOUT and STDERR to prevent anyone else sending data to them. It also opens the shared memory section with the environment stored by Crun. It then makes an RPC call to CondorCredSvc (to get the name and password of the domain account) which is most likely running on another system. The location information is stored in the registry under the key

HKEY_LOCAL_MACHINE\Software\Condor\CredService

The name of the machine running CondorCredSvc must be held in the REG_SZ value "Host". This should be the fully qualified domain name of the machine. You can also specify the optional "Port" REG_SZ value in case you are running CondorCredSvc on a different port.

Once the domain account credentials have been received the account is logged on through a call to LogonUser. The DACLs on the window station, desktop and current directory are then modified to allow the domain account access to them and the job is started in that window station and desktop with a call to CreateProcessAsUser. The starting directory is set to the same as sent by Crun, STDOUT and STDERR handles are set to the named pipes and the environment sent by Crun is used. CondorAuthSvc also starts a thread which waits on the new process handle until it terminates to close the named pipes. If the process starts correctly the process ID is returned to Crun.

If Crun requests that the job be stopped (again via RPC), CondorAuthSvc loops over all windows on the window station and desktop specified until it finds the one associated with the required process ID. It then sends that window a WM_CLOSE message, so any termination handling built in to the job should work correctly.

[Security Note: CondorAuthSvc currently makes no attempt to verify the origin of the call starting the job. This is, in principal, a bad thing since if the format of the RPC call is known it could let anyone start a job on the machine in the context of the domain user. If sensible security practices have been followed and the ACLs on sensitive system directories (such as C:\WINNT) do not allow write access to anyone other than trusted users the problem should not be too serious.]

Carun and AFSEncrypt
Carun and AFSEncrypt are a couple of utilities to allow jobs to access AFS without any special recompilation. AFSEncrypt encrypts an AFS username/password into a file (called .afs.xxx) using a simple XOR algorithm. It's not a particularly secure way to do it, but it's simple and

```
self-inverse. Carun reads this file and gets an AFS token before running
whatever job is on its command line as a child process. It waits on the
process handle and a 24 hour timer. If the timer expires first it briefly
suspends the primary thread of the child process and attempts to get a new
AFS token before restarting the job, the idea being that the job should
have uninterrupted access to AFS if it runs for more than 25 hours (the
default token lifetime). As a security measure, the AFS credentials are
cached by Carun in memory and the .afs.xxx file deleted as soon as the
username/password have been read for the first time.

Carun needs the machine to be running either the IBM AFS client or the
OpenAFS client to work. It also needs the client libraries if you want to
rebuild it.

For example, if you wanted to get a list of your AFS tokens under Condor
you would run the following:

Crun \\mymachine\myshare\Carun tokens.exe

Running a job
To run a job using this mechanism specify the following in your job
submission (assuming Crun is in C:\CondorAuth):

Executable= c:\CondorAuth\Crun.exe
Arguments = \\mymachine\myshare\carun.exe
\\anothermachine\anothershare\myjob.exe
Transfer_Input_Files = .afs.xxx

along with your usual settings.

Installation
A basic installation script for use with the Inno Setup installation
package compiler can be found in the Install folder.
```

### 6.2.8 Interoperability between Condor for Unix and Condor for Windows

Unix machines and Windows machines running Condor can happily co-exist in the same Condor pool without any problems. Jobs submitted on Windows can run on Windows or Unix, and jobs submitted on Unix can run on Unix or Windows. Without any specification (using the `requirements` expression in the submit description file), the default behavior will be to require the execute machine to be of the same architecture and operating system as the submit machine.

There is absolutely no need to run more than one Condor central manager, even if you have both Unix and Windows machines. The Condor central manager itself can run on either Unix or Windows; there is no advantage to choosing one over the other. Here at University of Wisconsin-

Madison, for instance, we have hundreds of Unix (Solaris, Linux, etc) and Windows machines in our Computer Science Department Condor pool. Our central manager is running on Linux. All is happy.

### 6.2.9 Some differences between Condor for Unix -vs- Condor for Windows

- On Unix, we recommend the creation of a "*condor*" account when installing Condor. On Windows, this is not necessary, as Condor is designed to run as a system service as user LocalSystem.
- On Unix, Condor finds the `condor_config` main configuration file by looking in ~condor, in /etc, or via an environment variable. On NT, the location of `condor_config` file is determined via the registry key `HKEY_LOCAL_MACHINE/Software/Condor`. You can override this value by setting an environment variable named `CONDOR_CONFIG`.
- On Unix, in the VANILLA universe at job vacate time Condor sends the job a softkill signal defined in the submit-description file (defaults to SIGTERM). On NT, Condor sends a WM_CLOSE message to the job at vacate time.
- On Unix, if one of the Condor daemons has a fault, a core file will be created in the `$(Log)` directory. On Condor NT, a "core" file will also be created, but instead of a memory dump of the process it will be a very short ASCII text file which describes what fault occurred and where it happened. This information can be used by the Condor developers to fix the problem.

## 6.3 Macintosh OS X

This section provides information specific to the Macintosh OS X port of Condor. The Macintosh port of Condor is more accurately a port of Condor to Darwin, the BSD core of OS X. Condor uses the Carbon library only to detect keyboard activity, and it does not use Cocoa at all. Condor on the Macintosh is a relatively new port, and it is not yet well-integrated into the Macintosh environment.

Condor on the Macintosh has a few shortcomings:

- Users connected to the Macintosh via *ssh* are not noticed for console activity.
- The memory size of threaded programs is reported incorrectly.
- No Macintosh-based installer is provided.
- The example start up scripts do not follow Macintosh conventions.
- Kerberos is not supported.

### 6.3.1 Macintosh OS X Panther (10.3)

Condor Version 6.8.6is built on a machine running Panther (10.3). Therefore, these binaries work on any PPC machine running MacOS 10.3.

### 6.3.2 Macintosh OS X Tiger (10.4)

Even though Condor Version 6.8.6is built on a machine running Panther (10.3), the resulting binaries are known to work on PPC machines running MacOS 10.4.

### 6.3.3 Macintosh OS X Tiger (10.4) for Intel CPUs

Condor does not yet provide Universal binaries or natively-built binaries for Intel MacOS 10.4. Given MacOS's support for running PPC binaries in emulation (their "Rosetta" system), the existing Condor binaries built for PPC MacOS 10.3 might work. Very few parts of the Condor system are performance bottle-necks, so running in emulation will hopefully cause no problems. The end-user applications (the jobs submitted to Condor), should probably be Universal binaries or x86-specific, to provide the most throughput on these machines.

## 6.4 AIX

This section provides information specific to the AIX ports of Condor.

### 6.4.1 AIX 5.2L

The version of Condor for AIX 5.2L has the same shortcommings as Condor for the AIX5.1L platform.

In addition the Condor binaries for AIX 5.2L will NOT execute on an AIX 5.1L machine.

### 6.4.2 AIX 5.1L

This is a relatively new port of Condor to the AIX architecture and as such there are a few things that aren't completely finished. Over time, these will be fixed.

Condor on AIX 5.1L has a few shortcomings:

- Keyboard Idle and Mouse Idle is wrong and will be fixed in a future release of Condor.

- The memory size of threaded programs is reported incorrectly.
- The memory/usage statistics of completed jobs is sometimes wrong.
- The Standard Universe is not supported.

In addition, Condor for the AIX 5.1L machine WILL execute correctly on an AIX 5.2L machine.

CHAPTER

# SEVEN

# Frequently Asked Questions (FAQ)

This is where you can find quick answers to some commonly asked questions about Condor.

## 7.1 Obtaining & Installing Condor

### Where can I download Condor?

Condor can be downloaded from http://www.cs.wisc.edu/condor/downloads (Madison, Wisconsin, USA) or http://www.bo.infn.it/condor-mirror/downloads (a mirror site at the Istituto Nazionale di Fisica Nucleare in Bologna, Italy).

### When I click to download Condor, it sends me back to the downloads page!

If you are trying to download Condor through a web proxy, try disabling it. Our web site uses the "referring page" as you navigate through our download menus in order to give you the right version of Condor, but sometimes proxies block this information from reaching our web site.

### What platforms do you support?

See Section 1.5, on page 5. Also, you might want to read the platform-specific information in Chapter 6 on page 443.

### What versions of Red Hat Linux does Condor support?

See Section 6.1 on page 443.

### Do you distribute source code?

At this time we do **not** distribute source code publicly, but instead consider requests on a case-by-case basis. If you need the source code, please e-mail us at condor-admin@cs.wisc.edu explaining why, and we'll get back to you.

### How do I upgrade the Unix machines in my pool from 6.4.x to 6.6.x?

This series of steps explains how to upgrade a pool of machines from running Condor version 6.4.x to version 6.6.x. Read through the entire set of directions before following them.

Briefly, the steps are to download the new version in order to replace your current binaries with the new binaries. Condor will notice that there are new binaries, since it checks for this every few minutes. The next time it checks, the new binaries will be used.

**Step 1: (Optional) Place test jobs in queue** This optional first step safeguards jobs currently in the queue when you upgrade. By completing this extra step, you will not lose any partially completed jobs, even if something goes wrong with your upgrade.

Manufacture test jobs that utilize each universe you use in your Condor pool. Submit each job, and put the job in the hold state, using *condor_hold*.

**Step 2: Place all jobs on hold** Place all jobs into the hold state while replacing binaries.

**Step 3: Download Condor 6.6.x** To ensure that both new and current binaries are within the same volume, make a new directory within your current release directory where 6.6.x will go. Unix commands will be of the form

```
cd <release-dir>
mkdir new
cd new
```

Locate the correct version of the Condor binary, and download into this `new` directory.

Do *not* install the downloaded version. Do uncompress and then untar the downloaded version. Further untar the release directory (called `release.tar`). This will create the directories

```
bin
etc
include
```

```
sbin
libexec
lib
man
```

From this list of created directories, `bin`, `include`, `sbin`, `libexec`, and `lib` will be used to replace current directories. Note that older versions of Condor do not have a libexec directory.

**Step 4: Configuration files** The downloaded version 6.6.x configuration file will have extra, new suggestions for configuration macro settings, to go with new features in Condor. These extra configuration macros are not be required in order to run version Condor 6.6.x.

Make a backup copy of the current configuration, to safeguard backing out of the upgrade, if something goes wrong.

Work through the new example configuration file to see if there is anything useful and merge with your site-specific (current) configuration file.

Note that starting in Condor 6.6.x, security sessions are turned on by default. If you will be retaining some 6.4.x series Condor installations in your pool, you must turn security sessions off in your 6.6.x configuration files. This can be accomplished by setting

```
SEC_DEFAULT_NEGOTIATION = NEVER
```

Also in 6.6.x, the definition of Hawkeye / Startd Cron jobs has changed. The old syntax allowed the following

```
HAWKEYE_JOBS =\
job1:job1_:/path/to/job1:1h \
job2:job2_:/path/to/job2:5m \
...
```

This is no longer supported, and must be replaced with the following

```
HAWKEYE_JOBS = job1:job1_:/path/to/job1:1h
HAWKEYE_JOBS = $(HAWKEYE_JOBS) job2:job2_:/path/to/job2:5m
HAWKEYE_JOBS = $(HAWKEYE_JOBS) ...
```

It should also be noted that in 6.6.x, the *condor_collector* and *condor_negotiator* can be set to run on non-standard ports. This will cause older (6.4.x and earlier) Condor installations in that pool to no longer function.

**Step 5: Replace release directories** For each of the directories that is to be replaced, move the current one aside, and put the new one in its place. The Unix commands to do this will be of the form

```
cd <release-dir>

mv bin bin.v64
mv new/bin bin

mv include include.v64
mv new/include include

mv sbin sbin.v64
mv new/sbin sbin

mv lib lib.v64
mv new/lib lib
```

Do this series of directory moves at one sitting, especially avoiding a long time lag between the moves relating to the `sbin` directory. Condor imposes a delay by design, but it does not idly wait for the new binaries to be in place.

**Step 6: Observe propagation of new binaries** Use *condor_status* to observe the propagation of the upgrade through the pool. As the machines notice and use the new binaries, their version number will change. Complete propagation should occur in five to ten minutes.

The command

```
condor_status -format "%s" Machine -format " %s\n" CondorVersion
```

gives a single line of information about each machine in the pool, containing only the machine name and version of Condor it is running.

**Step 7: (Optional) Release test jobs** Release the test jobs that were placed into the hold state in Step 1. If these test jobs complete successfully, then the upgrade is successful. If these test jobs fail (possibly by leaving the queue before finishing), then the upgrade is unsuccessful. If unsuccessful, back out of the upgrade by replacing the new configuration file with the backup copy and moving the Version 6.4.x release directories back to their previous location. Also send e-mail to condor-admin@cs.wisc.edu, explaining the situation and we'll help you work through it.

**Step 8: Release all jobs** Release all jobs in the queue, but running *condor_release*.

**Step 9: (Optional) Install manual pages** The `man` directory was new with Condor version 6.4.x. It contains manual pages. Note that installation of manual pages is optional; the chapter containing manual pages are in section 9.

To install the manual pages, move the `man` directory from `<release-dir>/new` to the desired location. Add the path name to this directory to the `MANPATH`.

## What is Personal Condor?

Personal Condor is a term used to describe a specific style of Condor installation suited for individual users who do not have their own pool of machines, but want to submit Condor jobs to run elsewhere.

A Personal Condor is essentially a one-machine, self-contained Condor pool which can use *flocking* to access resources in other Condor pools. See Section 5.2, on page 412 for more information on flocking.

### What do I do now? My installation of Condor does not work.

What to do to get Condor running properly depends on what sort of error occurs. One common error category are communication errors. Condor daemon log files report a failure to bind. For example:

```
(date and time) Failed to bind to command ReliSock
```

Or, the errors in the various log files may be of the form:

```
(date and time) Error sending update to collector(s)
(date and time) Can't send end_of_message
(date and time) Error sending UDP update to the collector

(date and time) failed to update central manager

(date and time) Can't send EOM to the collector
```

This problem can also be observed by running *condor_status*. It will give a message of the form:

```
Error:  Could not fetch ads --- error communication error
```

To solve this problem, understand that Condor uses the first network interface it sees on the machine. Since machines often have more than one interface, this problem usually implies that the wrong network interface is being used. It also may be the case that the system simply has the wrong IP address configured.

It is incorrect to use the localhost network interface. This has IP address 127.0.0.1 on all machines. To check if this incorrect IP address is being used, look at the contents of the CollectorLog file on the pool's your central manager right after it is started. The contents will be of the form:

```
5/25 15:39:33 ***************************************************
5/25 15:39:33 ** condor_collector (CONDOR_COLLECTOR) STARTING UP
5/25 15:39:33 ** $CondorVersion: 6.2.0 Mar 16 2001 $
5/25 15:39:33 ** $CondorPlatform: INTEL-LINUX-GLIBC21 $
5/25 15:39:33 ** PID = 18658
5/25 15:39:33 ***************************************************
5/25 15:39:33 DaemonCore: Command Socket at <128.105.101.15:9618>
```

The last line tells the IP address and port the collector has bound to and is listening on. If the IP address is 127.0.0.1, then Condor is definitely using the wrong network interface.

There are two solutions to this problem. One solution changes the order of the network interfaces. The preferred solution sets which network interface Condor should use by adding the following parameter to the local Condor configuration file:

```
NETWORK_INTERFACE = machine-ip-address
```

Where `machine-ip-address` is the IP address of the interface you wish Condor to use.

### After an installation of Condor, why do the daemons refuse to start, placing this message in the log files?

```
ERROR "The following configuration macros appear to contain default values
that must be changed before Condor will run.  These macros are:
hostallow_write
(found on line 1853 of /scratch/adesmet/TRUNK/work/src/localdir/condor_config)"
at line 217 in file condor_config.C
```

As of Condor 6.8.0, if Condor sees the bare key word: `YOU_MUST_CHANGE_THIS_INVALID_CONDOR_CONFIGURATION_VALUE` as the value of a configuration file entry, Condor daemons will log the given error message and exit.

By default, an installation of Condor 6.8.0 and later releases will have the configuration file entry `HOSTALLOW_WRITE` set to the above sentinel value. The Condor administrator must alter this value to be the correct domain or IP addresses that the administrator desires. The wildcard character (`*`) may be used to define this entry, but that allows anyone, from anywhere, to submit jobs into your pool. A better value will be of the form `*.domainname.com`.

### Why do standard universe jobs never run after an upgrade?

Standard universe jobs that remain in the job queue across an upgrade from any Condor release previous to 6.7.15 to any Condor release of 6.7.15 or more recent cannot run. They are missing a required ClassAd attribute (`LastCheckpointPlatform`) added for all standard universe jobs as of Condor version 6.7.15. This new attribute describes the platform where a job was running when it produced a checkpoint. The attribute is utilized to identify platforms capable of continuing the job (using the checkpoint).

This attribute becomes necessary due to bugs in some Linux kernels. A standard universe job may be continued on some, but not all Linux machines. And, the `CkptOpSys` attribute is not specific enough to be utilized.

There are two possible solutions for these standard universe jobs that cannot run, yet are in the queue:

1. Remove and resubmit the standard universe jobs that remain in the queue across the upgrade. This includes all standard universe jobs that have flocked in to the pool. Note that the resubmitted jobs will start over again from the beginning.

2. For each standard universe job in the queue, modify its job ClassAd such that it can possibly run within the upgraded pool. If the job has already run and produced a checkpoint on a machine before the upgrade, determine the machine that produced the checkpoint using the `LastRemoteHost` attribute in the job's ClassAd. Then look at that machine's ClassAd (after the upgrade) to determine and extract the value of the `CheckpointPlatform` attribute. Add this (using *condor_qedit*) as the value of the new attribute `LastCheckpointPlatform` in the job's ClassAd. Note that this operation must also have to be performed on standard universe jobs flocking in to an upgraded pool. It is recommended that pools that flock between each other upgrade to a post 6.7.15 version of Condor.

Note that if the upgrade to Condor takes place at the same time as a platform change (such as booting an upgraded kernel), there is no way to properly set the `LastCheckpointPlatform` attribute. The only option is to remove and resubmit the standard universe jobs.

## 7.2 Setting up Condor

### How do I set up a central manager on a machine with multiple network interfaces?

Please see section 3.7.2 on page 298.

### How do I get more than one job to run on my SMP machine?

Condor will automatically recognize a SMP machine and advertise each CPU of the machine separately. For more details, see section 3.12.7 on page 347.

### How do I configure a separate policy for the CPUs of an SMP machine?

Please see section 3.12.7 on page 347 for a lengthy discussion on this topic.

### How do I set up my machines so that only specific users' jobs will run on them?

Restrictions on what jobs will run on a given resource are enforced by only starting jobs that meet specific constraints, and these constraints are specified as part of the configuration.

To specify that a given machine should only run certain users' jobs, and always run the jobs regardless of other activity on the machine, load average, etc., place the following entry in the machine's Condor configuration file:

```
START = ( (RemoteUser == "userfoo@baz.edu") || \
          (RemoteUser == "userbar@baz.edu") )
```

A more likely scenario is that the machine is restricted to run only specific users' jobs, contingent on the machine not having other interactive activity and not being heavily loaded. The following entries are in the machine's Condor configuration file. Note that extra configuration variables are defined to make the START variable easier to read.

```
# Only start jobs if:
# 1) the job is owned by the allowed users, AND
# 2) the keyboard has been idle long enough, AND
# 3) the load average is low enough OR the machine is currently
#    running a Condor job, and would therefore accept running
#    a different one
AllowedUser    = ( (RemoteUser == "userfoo@baz.edu") || \
                   (RemoteUser == "userbar@baz.edu") )
KeyboardUnused = (KeyboardIdle > $(StartIdleTime))
NoOwnerLoad    = ($(CPUIdle) || (State != "Unclaimed" && State != "Owner"))
START          = $(AllowedUser) && $(KeyboardUnused) && $(NoOwnerLoad)
```

To configure multiple machines to do so, create a common configuration file containing this entry for them to share.

## How do I configure Condor to run my jobs only on machines that have the right packages installed?

This is a two-step process. First, you need to tell the machines to report that they have special software installed, and second, you need to tell the jobs to require machines that have that software.

To tell the machines to report the presence of special software, first add a parameter to their configuration files like so:

```
HAS_MY_SOFTWARE = True
```

And then, if there are already STARTD_EXPRS defined in that file, add HAS_MY_SOFTWARE to them, or, if not, add the line:

```
STARTD_EXPRS = HAS_MY_SOFTWARE, $(STARTD_EXPRS)
```

NOTE: For these changes to take effect, each *condor_startd* you update needs to be reconfigured with *condor_reconfig* -startd.

Next, to tell your jobs to only run on machines that have this software, add a requirements statement to their submit files like so:

```
Requirements = (HAS_MY_SOFTWARE =?= True)
```

NOTE: Be sure to use =?= instead of == so that if a machine doesn't have the HAS_MY_SOFTWARE parameter defined, the job's Requirements expression will not evaluate to "undefined", preventing it from running anywhere!

## How do I configure Condor to only run jobs at night?

A commonly requested policy for running batch jobs is to only allow them to run at night, or at other pre-specified times of the day. Condor allows you to configure this policy with the use of the `ClockMin` and `ClockDay` *condor_startd* attributes. A complete example of how to use these attributes for this kind of policy is discussed in subsubsection 3.5.10 on page 254.

## How do I configure Condor such that all machines do not produce checkpoints at the same time?

If machines are configured to produce checkpoints at fixed intervals, a large number of jobs are queued (submitted) at the same time, and these jobs start on machines at about the same time, then all these jobs will be trying to write out their checkpoints at the same time. It is likely to cause rather poor performance during this burst of writing.

The `RANDOM_INTEGER()` macro can help in this instance. Instead of defining `PERIODIC_CHECKPOINT` to be a fixed interval, each machine is configured to randomly choose one of a set of intervals. For example, to set a machine's interval for producing checkpoints to within the range of two to three hours, use the following configuration:

```
PERIODIC_CHECKPOINT = $(LastCkpt) > ( 2 * $(HOUR) + \
      $RANDOM_INTEGER(0,60,10) * $(MINUTE) )
```

The interval used is set at configuration time. Each machine is randomly assigned a different interval (2 hours, 2 hours and 10 minutes, 2 hours and 20 minutes, etc.) at which to produce checkpoints. Therefore, the various machines will not all attempt to produce checkpoints at the same time.

## Why will the *condor_master* not run when a local configuration file is missing?

If a `LOCAL_CONFIG_FILE` is specified in the global configuration file, but the specified file does not exist, the *condor_master* will not start up, and it prints a variation of the following example message.

```
ERROR: Can't read config file /mnt/condor/hosts/bagel/condor_config.local
```

This is not a bug; it is a feature! Condor has always worked this way on purpose. There is a potentially large security hole if Condor is configured to read from a file that does not exist. By creating that file, a malicious user could change all sorts of Condor settings. This is an easy way to gain root access to a machine, where the daemons are running as root.

The intent is that if you've set up your global configuration file to read from a local configuration file, and the local file is not there, then something is wrong. It is better for the *condor_master* to exit right away and log an error message than to start up.

If the *condor_master* continued with the local configuration file missing, either A) someone could breach security or B) you will have potentially important configuration information missing. Consider the example where the local configuration file was on an NFS partition and the server was down. There would be all sorts of really important stuff in the local configuration file, and Condor might do bad things if it started without those settings.

If supplied it with an empty file, the *condor_master* works fine.

## 7.3 Running Condor Jobs

### I'm at the University of Wisconsin-Madison Computer Science Dept., and I am having problems!

Please see the web page http://www.cs.wisc.edu/condor/uwcs. As it explains, your home directory is in AFS, which by default has access control restrictions which can prevent Condor jobs from running properly. The above URL will explain how to solve the problem.

### I'm getting a lot of e-mail from Condor. Can I just delete it all?

Generally you shouldn't ignore **all** of the mail Condor sends, but you can reduce the amount you get by telling Condor that you don't want to be notified every time a job successfully completes, only when a job experiences an error. To do this, include a line in your submit file like the following:

```
Notification = Error
```

See the Notification parameter in the *condor_q* man page on page 720 of this manual for more information.

### Why will my vanilla jobs only run on the machine where I submitted them from?

Check the following:

1. Did you submit the job from a local file system that other computers can't access?

   See Section 3.3.7, on page 159.

2. Did you set a special requirements expression for vanilla jobs that's preventing them from running but not other jobs?

   See Section 3.3.7, on page 159.

3. Is Condor running as a non-root user?

   See Section 3.6.12, on page 290.

### My job starts but exits right away with signal 9.

This can occur when the machine your job is running on is missing a shared library required by your program. One solution is to install the shared library on all machines the job may execute on. Another, easier, solution is to try to re-link your program statically so it contains all the routines it needs.

### Why aren't any or all of my jobs running?

Problems like the following are often reported to us:

```
I have submitted 100 jobs to my pool, and only 18 appear to be
running, but there are plenty of machines available.  What should I
do to investigate the reason why this happens?
```

Start by following these steps to understand the problem:

1. Run *condor_q* -analyze and see what it says.
2. Look at the User Log file (whatever you specified as "log = XXX" in the submit file).

   See if the jobs are starting to run but then exiting right away, or if they never even start.
3. Look at the SchedLog on the submit machine after it negotiates for this user. If a user doesn't have enough priority to get more machines the SchedLog will contain a message like "lost priority, no more jobs".
4. If jobs are successfully being matched with machines, they still might be dying when they try to execute due to file permission problems or the like. Check the ShadowLog on the submit machine for warnings or errors.
5. Look at the NegotiatorLog during the negotiation for the user. Look for messages about priority, "no more machines", or similar.

Another problem shows itself with statements within the log file produced by the *condor_schedd* daemon (given by `$(SCHEDD_LOG)`) that say the following:

```
2/3 17:46:53 Swap space estimate reached! No more jobs can be run!
12/3 17:46:53     Solution: get more swap space, or set RESERVED_SWAP = 0
12/3 17:46:53     0 jobs matched, 1 jobs idle
```

Condor computes the total swap space on your submit machine. It then tries to limit the total number of jobs it will spawn based on an estimate of the size of the *condor_shadow* daemon's memory footprint and a configurable amount of swap space that should be reserved. This is done to avoid the situation within a very large pool in which all the jobs are submitted from a single host. The huge number of *condor_shadow* processes would overwhelm the submit machine, it would run out of swap space, and thrash.

Things can go wrong if a machine has a lot of physical memory and little or no swap space. Condor does not consider the physical memory size, so the situation occurs where Condor thinks it has no swap space to work with, and it will not run the submitted jobs.

To see how much swap space Condor thinks a given machine has, use the output of a *condor_status* command of the following form:

```
condor_status -schedd [hostname] -long | grep VirtualMemory
```

If the value listed is 0, then this is what is confusing Condor. There are two ways to fix the problem:

1. Configure your machine with some real swap space.
2. Disable this check within Condor. Define the amount of reserved swap space for the submit machine to 0. Set `RESERVED_SWAP` to 0 in the configuration file:

   ```
   RESERVED_SWAP = 0
   ```

   and then send a *condor_restart* to the submit machine.

## Why does the `requirements` expression for the job I submitted have extra things that I did not put in my submit description file?

There are several extensions to the submitted `requirements` that are automatically added by Condor. Here is a list:

- Condor automatically adds `arch` and `opsys` if not specified in the submit description file. It is assumed that the executable needs to execute on the same platform as the machine on which the job is submitted.

- Condor automatically adds the expression `(Memory * 1024 > ImageSize)`. This ensures that the job will run on a machine with at least as much physical memory as the memory footprint of the job.
- Condor automatically adds the expression `(Disk >= DiskUsage)` if not already specified. This ensures that the job will run on a machine with enough disk space for the job's local I/O (if there is any).
- A pool administrator may define configuration variables that cause expressions to be added to `requirements`. These configuration variables are `APPEND_REQUIREMENTS`, `APPEND_REQ_VANILLA`, and `APPEND_REQ_STANDARD`. These configuration variables give pool administrators the flexibility to set policy for a local pool.
- Older versions of Condor needed to add confusing clauses about WINNT and the FileSystemDomain to vanilla universe jobs. This made sure that the jobs ran on a machine where files were accessible. The Windows version supported automatically transferring files with the vanilla job, while the Unix version relied on a shared file system. Since the Unix version of Condor now supports transferring files, these expressions are no longer added to the `requirements` for a job.

### When I use *condor_compile* to produce a job, I get an error that says, "Internal ld was not invoked!". What does this mean?

*condor_compile* enforces a specific behavior in the compilers and linkers that it supports (for example *gcc*, *g77*, *cc*, *CC*, *ld*) where a special linker script provided by Condor must be invoked during the final linking stages of the supplied compiler or linker.

In some rare cases, as with *gcc* compiled with the options **–with-as** or **–with-ld**, the enforcement mechanism we rely upon to have *gcc* choose our supplied linker script is not honored by the compiler. When this happens, an executable is produced, but the executable is devoid of the Condor libraries which both identify it as a Condor executable linked for the standard universe and implement the feature sets of remote I/O and transparent process checkpointing and migration.

Often, the only fix in order to use the compiler desired, is to reconfigure and recompile the compiler itself, such that it does not use the errant options mentioned.

With Condor's standard universe, we highly recommend that your source files are compiled with the supported compiler for your platform. See section 1.5 for the list of supported compilers. For a Linux platform, the supported compiler is the default compiler that came with the distribution. It is often found in the directory `/usr/bin`.

### Can I submit my standard universe SPARC Solaris 2.6 jobs and have them run on a SPARC Solaris 2.7 machine?

No. You may only use binary compatibility between SPARC Solaris 2.5.1 and SPARC Solaris 2.6 and between SPARC Solaris 2.7 and SPARC Solaris 2.8, but not between SPARC Solaris 2.6 and

SPARC Solaris 2.7. We may implement support for this feature in a future release of Condor.

### Can I submit my standard universe SPARC Solaris 2.8 jobs and have them run on a SPARC Solaris 2.9 machine?

No. Although normal executables are binary compatible, technical details of taking checkpoints currently prevents this particular combination. Note that this applies to standard universe jobs only.

### Why have standard universe jobs in Condor 6.6.x have begun unexpectedly segmentation faulting during a checkpoint after an upgrade of Redhat Enterprise Linux 3 to current update levels?

Redhat has apparently back-ported a 2.6 kernel feature called "exec_shield" to the current patch levels of the RHEL3 product line. This feature is designed to make buffer overflow attacks incredibly difficult to exploit. However, it has the unfortunate side effect of completely breaking all user land checkpointing algorithms including the one Condor utilizes. The solution is to turn off the kernel feature for each execution of a standard universe job in the Condor system. The method employed to do this is with `USER_JOB_WRAPPER` and a shell script that looks much like this one:

```
#! /bin/sh

sa="/usr/bin/setarch"

if [ -f $sa ]; then
    exec $sa i386 ${1+"$@"}
fi

exec ${1+"$@"}
```

Place this shell script into the `$(SBIN)` directory of your Condor installation with the name of *fix_std_univ* and make sure to *chmod 755 fix_std_univ* it. Then, set `USER_JOB_WRAPPER = $(SBIN)/fix_std_univ` in your global config file(or the config files which will affect your Linux install of Condor). Then do a *condor_reconfig* of your pool. When a standard universe job is run on a machine, if the *setarch* program is available (under Linux with the "exec_shield" feature), then it will run the executable in the i386 personality, which turns off the "exec_shield" kernel feature.

### Why do my vanilla jobs keep cycling between suspended and unsuspended?

Condor tries to provide a number, the "Condor Load Average" (reported in the machine ClassAd as `CondorLoadAvg`), which is intended to represent the total load average on the system caused

by any running Condor job(s). Unfortunately, it is impossible to get an accurate number for this without support from the operating system. This is not available. So, Condor does the best it can, and it mostly works in most cases. However, there are a number of ways this statistic can go wrong.

The old default Condor policy was to suspend if the non-Condor load average went over a certain threshold. However, because of the problems providing accurate numbers for this (described below), some jobs would go into a cycle of getting suspended and resumed. The default suspend policy now shipped with Condor uses the solution explained here.

While there are too many technical details of why `CondorLoadAvg` might be wrong for a short answer here, a brief explanation is presented. When a job has periodic behavior, and the load it places upon a machine is changing over time, the system load also changes over time. However, Condor thinks that the job's share of the system load (what it uses to compute the CondorLoad) is also changing. So, when the job was running, and then stops, both the system load and the Condor load start falling. If it all worked correctly, they'd fall at the exact same rate, and `NonCondorLoad` would be constant. Unfortunately, `CondorLoadAvg` falls faster, since Condor thinks the job's share of the total load is falling, too. Therefore, `CondorLoadAvg` falls faster than the system load, `NonCondorLoad` goes up, and the old default `SUSPEND` expression becomes true.

It appears that Condor should be able to avoid this problem, but for a host of reasons, it can not. There is no good way (without help from the operating systems Condor runs on; the help does not exist) to get this right. The only way to compute these numbers more accurately without support from the operating system is to sample everything at such a high rate that Condor itself would create a large load average, just to try to compute the load average. This is Heisenberg's uncertainty principle in action.

A similar sampling error can occur when Condor is starting a job within the vanilla universe with many processes and with a heavy initial load. Condor mistakenly decides that the load on the machine has gotten too high while the job is in the initialization phase and kicks the job off the machine.

To correct this problem, Condor needs to check to see if the load of the machine has been high over an interval of time. There is an attribute, `CpuBusyTime` that can be used for this purpose. This macro returns the time `$(CpuBusy)` (defined in the default configuration file) has been true, or 0 if `$(CpuBusy)` is false. `$(CpuBusy)` is usually defined in terms of non-Condor load. These are the default settings:

```
NonCondorLoadAvg    = (LoadAvg - CondorLoadAvg)
HighLoad            = 0.5
CPUBusy             = ($(NonCondorLoadAvg) >= $(HighLoad))
```

To take advantage of `CpuBusyTime`, you can use it in your `SUSPEND` expression.

Here is an example:

```
SUSPEND = (CpuBusyTime > 3 * $(MINUTE)) && ((CurrentTime - JobStart) > 90)
```

The above policy says to only suspend the job if the CPU has been busy with non-Condor load at least three minutes and it has been at least 90 seconds since the start of the job.

## Why might my job be preempted (evicted)?

There are four circumstances under which Condor may evict a job. They are controlled by different expressions.

Reason number 1 is the user priority: controlled by the PREEMPTION_REQUIREMENTS expression in the configuration file. If there is a job from a higher priority user sitting idle, the *condor_negotiator* daemon may evict a currently running job submitted from a lower priority user if PREEMPTION_REQUIREMENTS is True. For more on user priorities, see section 2.7 and section 3.4.

Reason number 2 is the owner (machine) policy: controlled by the PREEMPT expression in the configuration file. When a job is running and the PREEMPT expression evaluates to True, the *condor_startd* will evict the job. The PREEMPT expression should reflect the requirements under which the machine owner will not permit a job to continue to run. For example, a policy to evict a currently running job when a key is hit or when it is the 9:00am work arrival time, would be expressed in the PREEMPT expression and enforced by the *condor_startd*. For more on the PREEMPT expression, see section 3.5.

Reason number 3 is the owner (machine) preference: controlled by the RANK expression in the configuration file (sometimes called the startd rank or machine rank). The RANK expression is evaluated as a floating point number. When one job is running, a second idle job that evaluates to a higher RANK value tells the *condor_startd* to prefer the second job over the first. Therefore, the *condor_startd* will evict the first job so that it can start running the second (preferred) job. For more on RANK, see section 3.5.

Reason number 4 is if Condor is to be shutdown: on a machine that is currently running a job. Condor evicts the currently running job before proceeding with the shutdown.

## Condor does not stop the Condor jobs running on my Linux machine when I use my keyboard and mouse. Is there a bug?

There is no bug in Condor. Unfortunately, recent Linux 2.4.x and all Linux 2.6.x kernels through version 2.6.10 do not post proper state information, such that Condor can detect keyboard and mouse activity. Condor version 6.6.8 and later versions of Condor implement workarounds to piece together the needed state information for PS/2 devices. A better fix of the problem utilizes the kernel patch linked to from the directions posted at http://www.cs.wisc.edu/condor/kernel.patch.html. This patch works better for PS/2 devices, and may also work for USB devices. A future version of Condor will implement better recognition of USB devices, such that the kernel patch will also definitively work for USB devices.

### What signals get sent to my jobs when Condor needs to preempt or kill them, or when I remove them from the queue? Can I tell Condor which signals to send?

The answer is dependent on the universe of the jobs.

Under the scheduler universe, the signal jobs get upon *condor_rm* can be set by the user in the submit description file with the form of

```
remove_kill_sig = SIGWHATEVER
```

If this command is not defined, Condor further looks for a command in the submit description file with the form

```
kill_sig = SIGWHATEVER
```

And, if that command is also not given, Condor uses SIGTERM.

For all other universes, the jobs get the value of the submit description file command `kill_sig`, which is SIGTERM by default.

If a job is killed or evicted, the job is sent a `kill_sig`, unless it is on the receiving end of a hard kill, in which case it gets SIGKILL.

Under all universes, the signal is sent only to the parent PID of the job, namely, the first child of the *condor_starter*. If the child itself is forking, the child must catch and forward signals as appropriate. This in turn depends on the user's desired behavior. The exception to this is (again) where the job is receiving a hard kill. Condor sends the value SIGKILL to all the PIDs in the family.

### Why does my Linux job have an enormous ImageSize and refuse to run anymore?

Sometimes Linux jobs run, are preempted and can not start again because Condor thinks the image size of the job is too big. This is because Condor has a problem calculating the image size of a program on Linux that uses threads. It is particularly noticeable in the Java universe, but it also happens in the vanilla universe. It is not an issue in the standard universe, because threaded programs are not allowed.

On Linux, each thread appears to consume as much memory as the entire program consumes, so the image size appears to be (number-of-threads * image-size-of-program). If your program uses a lot of threads, your apparent image size balloons. You can see the image size that Condor believes your program has by using the -l option to condor_q, and looking at the ImageSize attribute.

When you submit your job, Condor creates or extends the requirements for your job. In particular, it adds a requirement that you job must run on a machine with sufficient memory:

```
Requirements = ... ((Memory * 1024) >= ImageSize) ...
```

(Note that memory is the execution machine's memory in megabytes while ImageSize is in kilobytes). When your application is threaded, the image size appears to be much larger than it really is, and you may not have a machine with sufficient memory to handle this requirement.

Unfortunately, calculating the correct ImageSize is rather hard to fix on Linux, and we do not yet have a good solution. Fortunately, there is a workaround while we work on a good solution for a future release.

In the Requirements expression above, Condor added `(Memory * 1024) >= ImageSize)` on your behalf. You can prevent Condor from doing this by giving it your own expression about memory in your submit file, just as:

```
Requirements = Memory > 1024
```

You will need to change 1024 to a reasonably good estimate of the actual image size of your program, in kilobytes. This expression says that your program requires 1 megabyte of memory. If you underestimate the memory your application needs, you may have bad performance if you job runs on machines that have insufficient memory.

In addition, if you have modified your machine policies to preempt jobs when they get big a ImageSize, you will need to change those policies.

## Why does the time output from *condor_status* appear as [?????] ?

Condor collects timing information for a large variety of uses. Collection of the data relies on accurate times. Being a distributed system, clock skew among machines causes errant timing calculations. Values can be reported too large or too small, with the possibility of calculating negative timing values.

This problem may be seen by the user when looking at the output of *condor_status*. If the **ActivityTime** field appears as [?????], then this calculated statistic was negative. *condor_status* recognizes that a negative amount of time will be nonsense to report, and instead displays this string.

The solution to the problem is to synchronize the clocks on these machines. An administrator can do this using a tool such as *ntp*.

## The user condor's home directory cannot be found. Why?

This problem may be observed after installation, when attempting to execute

```
~condor/condor/bin/condor_config_val  -tilde
```

and there is a user named condor. The command prints a message such as

```
Error: Specified -tilde but can't find condor's home directory
```

In this case, the difficulty stems from using NIS, because the Condor daemons fail to communicate properly with NIS to get account information. To fix the problem, a dynamically linked version of Condor must be installed.

### Condor commands (including *condor_q*) are really slow. What is going on?

Some Condor programs will react slowly if they expect to find a *condor_collector* daemon, yet cannot contact one. Notably, *condor_q* can be very slow. The *condor_schedd* daemon will also be slow, and it will log lots of harmless messages complaining. If you are not running a *condor_collector* daemon, it is important that the configuration variable `COLLECTOR_HOST` be set to nothing. This is typically done by setting `CONDOR_HOST` with

```
CONDOR_HOST=
COLLECTOR_HOST=$(CONDOR_HOST)
```

or

```
COLLECTOR_HOST=
```

### Where are my missing files? The command `when_to_transfer_output = ON_EXIT_OR_EVICT` is in the submit description file.

Although it may appear as if files are missing, they are not. The transfer does take place whenever a job is preempted by another job, vacates the machine, or is killed. Look for the files in the directory defined by the `SPOOL` configuration variable. See section 2.5.4, on page 37 for details on the naming of the intermediate files.

## 7.4 Condor on Windows

### Will Condor work on a network of mixed Unix and Windows machines?

You can have a Condor pool that consists of both Unix and Windows machines.

Your central manager can be either Windows or Unix. For example, even if you had a pool consisting strictly of Unix machines, you could use a Windows box for your central manager, and vice versa.

Submitted jobs can originate from either a Windows **or** a Unix machine, and be destined to run on Windows **or** a Unix machine. Note that there are still restrictions on the supported universes for jobs executed on Windows machines.

So, in summary:

1. A single Condor pool can consist of both Windows and Unix machines.
2. It does not matter at all if your Central Manager is Unix or Windows.
3. Unix machines can submit jobs to run on other Unix or Windows machines.
4. Windows NT machines can submit jobs to run on other Windows or Unix machines.

### What versions of Windows will Condor run on?

See Section 1.5, on page 5.

### My Windows program works fine when executed on its own, but it does not work when submitted to Condor.

First, make sure that the program really does work outside of Condor under Windows, that the disk is not full, and that the system is not out of user resources.

As the next consideration, know that some Windows programs do not run properly because they are dynamically linked, and they cannot find the `.dll` files that they depend on. Version 6.4.x of Condor sets the `PATH` to be empty when running a job. To avoid these difficulties, do one of the following

1. statically link the application
2. wrap the job in a script that sets up the environment
3. submit the job from a correctly-set environment with the command

   ```
   getenv = true
   ```

   in the submit description file. This will copy your environment into the job's environment.
4. send the required `.dll` files along with the job using the submit description file command **transfer_input_files**.

### Why is the *condor_master* daemon failing to start, giving an error about "In StartServiceCtrlDispatcher, Error number: 1063"?

In Condor for Windows, the *condor_master* daemon is started as a service. Therefore, starting the *condor_master* daemon as you would on Unix will not work. Start Condor on Windows machines using either

```
net start condor
```

or start the Condor service from the Service Control Manager located in the Windows Control Panel.

### Jobs submitted from Windows give an error referring to a credential.

Jobs submitted from a Windows machine require a stashed password in order for Condor to perform certain operations on the user's behalf. Refer to section 6.2.3 for information about password storage on Windows. The command which stashes a password for a user is *condor_store_cred*. See the manual page on on page 714 for usage details.

The error message that Condor gives if a user has not stashed a password is of the form:

```
ERROR: No credential stored for username@machinename

        Correct this by running:
        condor_store_cred add
```

### Jobs submitted from Unix to execute on Windows do not work properly.

A difficulty with defaults causes jobs submitted from Unix for execution on a Windows platform to remain in the queue, but make no progress. For jobs with this problem, log files will contain error messages pointing to shadow exceptions.

This difficulty stems from the defaults for whether file transfer takes place. The workaround for this problem is to place the line

```
TRANSFER_FILES  = ALWAYS
```

into the submit description file for jobs submitted from a Unix machine for execution on a Windows machine.

### When I run *condor_status* I get a communication error, or the Condor daemon log files report a failure to bind.

Condor uses the first network interface it sees on your machine. This problem usually means you have an extra, inactive network interface (such as a RAS dial up interface) defined before to your regular network interface.

To solve this problem, either change the order of your network interfaces in the Control Panel, or explicitly set which network interface Condor should use by adding the following parameter to your Condor configuration file:

```
NETWORK_INTERFACE = ip-address
```

Where `ip-address` is the IP address of the interface you wish Condor to use.

### My job starts but exits right away with status 128.

This can occur when the machine your job is running on is missing a DLL (Dynamically Linked Library) required by your program. The solution is to find the DLL file the program needs and put it in the TRANSFER_INPUT_FILES list in the job's submit file.

To find out what DLLs your program depends on, right-click the program in Explorer, choose Quickview, and look under "Import List".

### How can I access network files with Condor on Windows?

Five methods for making access of network files work with Condor are given in section 6.2.7.

### What is wrong when *condor_off* cannot find my host, and *condor_status* does not give me a complete host name?

Given the command

```
condor_off hostname2
```

an error message of the form

```
Can't find address for master hostname2.somewhere.edu
```

appears. Yet, when looking at the host names with

```
condor_status -master
```

the output is of the form

```
hostname1.somewhere.edu
hostname2
hostname3.somewhere.edu
```

To correct this incomplete host name, add an entry to the configuration file for `DEFAULT_DOMAIN_NAME` that specifies the domain name to be used. For the example given, the configuration entry will be

```
DEFAULT_DOMAIN_NAME = somewhere.edu
```

After adding this configuration file entry, use *condor_restart* to restart the Condor daemons and effect the change.

### Does USER_JOB_WRAPPER work on Windows machines?

The USER_JOB_WRAPPER configuration variable does work on Windows machines. The wrapper must be either a batch script with a file extension of `.bat` or `.cmd`, or an executable with a file extension of `.exe` or `.com`.

An example of a batch script sets environment variables:

```
REM set some environment variables
set LICENSE_SERVER=192.168.1.202:5012
set MY_PARAMS=2

REM Run the actual job now
%*
```

### *condor_store_cred* is failing, and I'm sure I'm typing my password correctly.

First, make sure the *condor_schedd* is running.

Next, check the SchedLog. It will contain more detailed information about the failure. Frequently, the error is a result of `PERMISSION DENIED` errors. You can read more about properly configuring security settings on page 282.

### My submit machine cannot have more than 120 jobs running concurrently. Why?

Windows is likely to be running out of desktop heap. Confirm this to be the case by looking in the log for the *condor_schedd* daemon to see if *condor_shadow* daemons are immediately exiting with status 128. If this is the case, increase the desktop heap size. Open the registry key:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\SubSystems\Window
```

The SharedSection value can have three values separated by commas. The third value controls the desktop heap size for non-interactive desktops, which the Condor service uses. The default is 512 (Kbytes). 60 *condor_shadow* daemons consume about 256 Kbytes, hence 120 shadows can run with the default value. To be able to run a maximum of 300 *condor_shadow* daemons, set this value at 1280.

Reboot the system for the changes to take effect. For more information, see Microsoft Article Q184802.

## 7.5 Grid Computing

### What must be installed to access grid resources?

A single machine with Condor installed such that jobs may be submitted is the minimum software necessary. If matchmaking or glidein is desired, then a single machine must not only be running Condor such that jobs may be submitted, but also fill the role of a central manager. A Personal Condor installation may satisfy both.

### I am the administrator at Physics, and I have a 64-node cluster running Condor. The administrator at Chemistry is also running Condor on her 64-node cluster. We would like to be able to share resources. How do we do this?

Condor's flocking feature allows multiple Condor pools to share resources. By setting configuration variables within each pool, jobs may be executed on either cluster. See the manual section on flocking, section 5.2, for details.

### What is glidein?

Glidein provides a way to temporarily add a resource to a local Condor pool. Glidein uses Globus resource-management software to run jobs on the resource. Those jobs are initially portions of Condor software, such that Condor is running on the resource, configured to be part of the local pool. Then, Condor may execute the user's jobs. There are several benefits to working in this way. Standard universe jobs may be submitted to run on the resource. Condor can also dynamically schedule jobs across the grid.

See the section on Glidein, section 5.4 of the manual for further information.

### Using my Globus gatekeeper to submit jobs to the Condor pool does not work. What is wrong?

The Condor configuration file is in a non-standard location, and the Globus software does not know how to locate it, when you see either of the following error messages.

first error message

```
% globus-job-run \
  globus-gate-keeper.example.com/jobmanager-condor /bin/date

Neither the environment variable CONDOR_CONFIG, /etc/condor/,
nor ~condor/ contain a condor_config file.  Either set
CONDOR_CONFIG to point to a valid config file, or put a
```

```
"condor_config" file in /etc/condor or ~condor/ Exiting.

GRAM Job failed because the job failed when the job manager
attempted to run it (error code 17)
```

second error message

```
% globus-job-run \
   globus-gate-keeper.example.com/jobmanager-condor /bin/date

ERROR: Can't find address of local schedd GRAM Job failed
because the job failed when the job manager attempted to run it
(error code 17)
```

As described in section 3.2.2, Condor searches for its configuration file using the following ordering.

1. File specified in the CONDOR_CONFIG environment variable
2. /etc/condor/condor_config
3. ~condor/condor_config
4. $(GLOBUS_LOCATION)/etc/condor_config

Presuming the configuration file is not in a standard location, you will need to set the CONDOR_CONFIG environment variable by hand, or set it in an initialization script. One of the following solutions for an initialization may be used.

1. Wherever *globus-gatekeeper* is launched, replace it with a minimal shell script that sets CONDOR_CONFIG and then starts *globus-gatekeeper*. Something like the following should work:

   ```
   #! /bin/sh
   CONDOR_CONFIG=/path/to/condor_config
   export CONDOR_CONFIG
   exec /path/to/globus/sbin/globus-gatekeeper "$@"
   ```

2. If you are starting *globus-gatekeeper* using *inetd*, *xinetd*, or a similar program, set the environment variable there. If you are using *inetd*, you can use the *env* program to set the environment. This example does this; the example is shown on multiple lines, but it will be all on one line in the *inetd* configuration.

   ```
   globus-gatekeeper stream tcp nowait root /usr/bin/env
   env CONDOR_CONFIG=/path/to/condor_config
   /path/to/globus/sbin/globus-gatekeeper
   -co /path/to/globus/etc/globus-gatekeeper.conf
   ```

   If you're using *xinetd*, add an env setting something like the following:

```
service gsigatekeeper
{
    env = CONDOR_CONFIG=/path/to/condor_config
    cps = 1000 1
    disable = no
    instances = UNLIMITED
    max_load = 300
    nice = 10
    protocol = tcp
    server = /path/to/globus/sbin/globus-gatekeeper
    server_args = -conf /path/to/globus/etc/globus-gatekeeper.conf
    socket_type = stream
    user = root
    wait = no
}
```

## 7.6 Troubleshooting

### If I see PERMISSION DENIED in my log files, what does that mean?

Most likely, the Condor installation has been misconfigured and Condor's access control security functionality is preventing daemons and tools from communicating with each other. Other symptoms of this problem include Condor tools (such as *condor_status* and *condor_q*) not producing any output, or commands that appear to have no effect (for example, *condor_off* or *condor_on*).

The solution is to properly configure the HOSTALLOW_* and HOSTDENY_* settings (for host/IP based authentication) or to configure strong authentication and set ALLOW_* and DENY_* as appropriate. Host-based authentiation is described in section 3.6.10 on page 282. Information about other forms of authentication is provided in section 3.6.1 on page 261.

### What happens if the central manager crashes?

If the central manager crashes, jobs that are already running will continue to run unaffected. Queued jobs will remain in the queue unharmed, but can not begin running until the central manager is restarted and begins matchmaking again. Nothing special needs to be done after the central manager is brought back on line.

### Why did the *condor_schedd* daemon die and restart?

The *condor_schedd* daemon receives signal 25, dies, and is restarted when the history file reaches a 2 Gbyte size limit. Until a larger history file size or the rotation of the history file is supported in Condor, try one of these work arounds:

1. When the history file becomes large, remove it. Note that this causes a loss of the information

in the history file, but the *condor_schedd* daemon will not die.

2. When the history file becomes large, move it.

3. Stop keeping the history. Only *condor_history* accesses the history file, so this particular functionality will be gone. To stop keeping the history, place

   ```
   HISTORY=
   ```

   in the configuration, followed by a *condor_reconfig* command to recognize the change in currently executing daemons.

### When I ssh/telnet to a machine to check particulars of how Condor is doing something, it is always vacating or unclaimed when I know a job had been running there!

Depending on how your policy is set up, Condor will track *any* tty on the machine for the purpose of determining if a job is to be vacated or suspended on the machine. It could be the case that after you ssh there, Condor notices activity on the tty allocated to your connection and then vacates the job.

### What is wrong? I get no output from *condor_status*, but the Condor daemons are running.

One likely error message within the collector log of the form

```
DaemonCore: PERMISSION DENIED to host <xxx.xxx.xxx.xxx> for command 0 (UPDATE_STARTD_AD)
```

indicates a permissions problem. The *condor_startd* daemons do not have write permission to the *condor_collector* daemon. This could be because you used domain names in your HOSTALLOW_WRITE and/or HOSTDENY_WRITE configuration macros, but the domain name server (DNS) is not properly configured at your site. Without the proper configuration, Condor cannot resolve the IP addresses of your machines into fully-qualified domain names (an inverse lookup). If this is the problem, then the solution takes one of two forms:

1. Fix the DNS so that inverse lookups (trying to get the domain name from an IP address) works for your machines. You can either fix the DNS itself, or use the DEFAULT_DOMAIN_NAME setting in your Condor configuration file.

2. Use numeric IP addresses in the HOSTALLOW_WRITE and/or HOSTDENY_WRITE configuration macros instead of domain names. As an example of this, assume your site has a machine such as foo.your.domain.com, and it has two subnets, with IP addresses 129.131.133.10, and 129.131.132.10. If the configuration macro is set as

```
HOSTALLOW_WRITE = *.your.domain.com
```

and this does not work, use

```
HOSTALLOW_WRITE = 192.131.133.*, 192.131.132.*
```

Alternatively, this permissions problem may be caused by being too restrictive in the setting of your HOSTALLOW_WRITE and/or HOSTDENY_WRITE configuration macros. If it is, then the solution is to change the macros, for example from

```
HOSTALLOW_WRITE = condor.your.domain.com
```

to

```
HOSTALLOW_WRITE = *.your.domain.com
```

or possibly

```
HOSTALLOW_WRITE = condor.your.domain.com, foo.your.domain.com, \
bar.your.domain.com
```

Another likely error message within the collector log of the form

```
DaemonCore: PERMISSION DENIED to host <xxx.xxx.xxx.xxx> for command 5 (QUERY_STARTD_ADS)
```

indicates a similar problem as above, but read permission is the problem (as opposed to write permission). Use the solutions given above.

## Why does Condor leave mail processes around?

Under FreeBSD and Mac OSX operating systems, misconfiguration of of a system's outgoing mail causes Condor to inadvertently leave paused and zombie mail processes around when Condor attempts to send notification e-mail. The solution to this problem is to correct the mailer configuration.

Execute the following command as the user under which Condor daemons run to determine whether outgoing e-mail works.

```
$ uname -a | mail -v your@emailaddress.com
```

If no e-mail arrives, then outgoing e-mail does not work correctly.

Note that this problem does not manifest itself on non-BSD Unix platforms, such as Linux.

## 7.7 Other questions

### Is there a Condor mailing-list?

Yes. There are two useful mailing lists. First, we run an extremely low traffic mailing list solely to announce new versions of Condor. Follow the instructions for Condor World at http://www.cs.wisc.edu/condor/mail-lists/. Second, our users can be extremely knowledgeable, and they help each other solve problems using the Condor Users mailing list. Again, follow the instructions for Condor Users at http://www.cs.wisc.edu/condor/mail-lists/.

### My question isn't in the FAQ!

If you have any questions that are not listed in this FAQ, try looking through the rest of the manual. Try joining the Condor Users mailing list, where our users support each other in finding answers to problems. Follow the instructions at http://www.cs.wisc.edu/condor/mail-lists/. If you still can't find an answer, feel free to contact us at condor-admin@cs.wisc.edu.

Note that Condor's free e-mail support is provided on a best-effort basis, and at times we may not be able to provide a timely response. If guaranteed support is important to you, please inquire about our paid support services.

CHAPTER

# EIGHT

# Version History and Release Notes

## 8.1 Introduction to Condor Versions

This chapter provides descriptions of what features have been added or bugs fixed for each version of Condor. The first section describes the Condor version numbering scheme, what the numbers mean, and what the different *release series* are. The rest of the sections each describe a specific release series, and all the Condor versions found in that series.

### 8.1.1 Condor Version Number Scheme

Starting with version 6.0.1, Condor adopted a new, hopefully easy to understand version numbering scheme. It reflects the fact that Condor is both a production system and a research project. The numbering scheme was primarily taken from the Linux kernel's version numbering, so if you are familiar with that, it should seem quite natural.

There will usually be two Condor versions available at any given time, the *stable* version, and the *development* version. Gone are the days of "patch level 3", "beta2", or any other random words in the version string. All versions of Condor now have exactly three numbers, seperated by "."

- The first number represents the major version number, and will change very infrequently.
- *The thing that determines whether a version of Condor is "stable" or "development" is the second digit. Even numbers represent stable versions, while odd numbers represent development versions.*
- The final digit represents the minor version number, which defines a particular version in a given release series.

### 8.1.2 The Stable Release Series

People expecting the stable, production Condor system should download the stable version, denoted with an even number in the second digit of the version string. Most people are encouraged to use this version. We will only offer our paid support for versions of Condor from the stable release series.

*On the stable series, new minor version releases will only be made for bug fixes and to support new platforms.* No new features will be added to the stable series. People are encouraged to install new stable versions of Condor when they appear, since they probably fix bugs you care about. Hopefully, there won't be many minor version releases for any given stable series.

### 8.1.3 The Development Release Series

Only people who are interested in the latest research, new features that haven't been fully tested, etc, should download the development version, denoted with an odd number in the second digit of the version string. We will make a best effort to ensure that the development series will work, but we make no guarantees.

On the development series, new minor version releases will probably happen frequently. People should not feel compelled to install new minor versions unless they know they want features or bug fixes from the newer development version.

*Most sites will probably never want to install a development version of Condor for any reason.* Only if you know what you are doing (and like pain), or were explicitly instructed to do so by someone on the Condor Team, should you install a development version at your site.

NOTE: Different releases within a development series cannot be installed side-by-side within the same pool. For example, the protocols used by version 6.1.6 are not compatible with the protocols used in version 6.1.5. When you upgrade to a new development release, make certain you upgrade all machines in your pool to the same version.

After the feature set of the development series is satisfactory to the Condor Team, we will put a code freeze in place, and from that point forward, only bug fixes will be made to that development series. When we have fully tested this version, we will release a new stable series, resetting the minor version number, and start work on a new development release from there.

## 8.2 Upgrade Surprises

Occasional changes to Condor can cause unexpected errors or results to users. Here is a list of changes to note and be aware of.

- When upgrading from 6.6.x 6.7.x, jobs that rely on the environment variable `CONDOR_SCRATCH_DIR` need to be changed to use `_CONDOR_SCRATCH_DIR`. An underscore was added to the beginning of this variable.

- A necessary change was introduced in version 6.7.15, such that standard universe jobs are better identified and have more correct resumption semantics on other machines with proper platform identification signatures (which contain things such as operating system, OS distribution, and kernel memory model). This change affects all standard universe jobs that remain in the job queue across an upgrade from any Condor release previous to 6.7.15 to any Condor release of 6.7.15 or more recent.

  The resulting policy changes are documented in section 3.5.4.

  Suggestions for dealing with this backwards compatibility issue are given in the Frequently Asked Questions in section 7.1.

## 8.3 Stable Release Series 6.8

This is a stable release series of Condor. It is based on the 6.7 development series. All new features added or bugs fixed in the 6.7 series are available in the 6.8 series. As usual, only bug fixes (and potentially, ports to new platforms) will be provided in future 6.8.x releases. New features will be added in the forthcoming 6.9.x development series.

The 6.8.x series supports a different set of platforms than 6.6.x. Please see the updated table of available platforms in section 1.5 on page 5.

The details of each version are described below.

### Version 6.8.6

Release Notes:

- Condor is now officially supported on Microsoft Vista.
- Condor is now officially supported on MacOS running natively on Intel CPUs. (and Condor bianries for Intel MacOS are now available for download).
- Condor now uses Globus 4.0.5 for GSI, pre-WS GRAM, and GridFTP support.

New Features:

- On all UNIX ports of Condor except MacOSX, AIX, and Tru64, separate debug symbol files are now supported. This allows meaningful debugging of core files in addition to attaching to stripped executables during runtime.
- *condor_dagman* now prints reports of pending nodes to the `dagman.out`, if it has been waiting more than `DAGMAN_PENDING_REPORT_INTERVAL` seconds without seeing any node job events. (This is to help diagnose the problem if *condor_dagman* gets ”stuck”.)

- Optimized the submission of grid-type gt4 grid universe jobs to the remote resource. Submission now takes one operation instead of three.
- The *condor_shadow* will obtain a session key to the *condor_schedd* at the start of the job instead of potentially waiting until the job completes. This reduces the chances of re-running already completed jobs in the event of authentication failures (for instance, if a Kerberos KDC is down or overloaded).

Bugs Fixed:

- On MacOS, Condor is more robust about how it monitors characteristics (such as image size) of a running job. Fixed several issues that would cause the *condor_startd* to exit on MacOS 10.4 running on Intel processors.
- Fixed bug in the local universe where the local universe execute directory was not removed when the job could not start. The most common case was an incorrectly named executable file.
- Fixed a bug that prevented dollar dollar expansions with a default argument that contained forward slashes from expanding properly. An example that now works correctly, but exhibited the incorrect behavior:

  ```
  $$(SomeVariable:/path/to/file)
  ```

- The Windows installer now works on MS Vista. Also, it does not pop up any command windows.
- The *condor_ckpt_server* was fixed to honor `HIGHPORT` and `LOWPORT`. While the well-known ports for the checkpoint data server have not changed, the helper processes that perform the store and restore (which communicate directly with the standard universe job) now bind to ports within specified ranges. Note that this will limit the number of simultaneous store/restore requests to the number of available ports.
- Fixed a bug in *condor_dagman* that caused recovery/bootstrap mode to be very slow on large DAGs (i.e., ones with hundreds of thousands of nodes).
- Fixed a bug that caused *condor_dagman* to incorrectly deal with VARS lines specifying more than one macro name (this bug was introduced in version 6.8.5).
- Fixed a bug in the configuration macro `RANDOM_INTEGER` when used as part of a larger expression. The entire configuration value containing the reference to `RANDOM_INTEGER` was being replaced by the chosen random integer, rather than just having `RANDOM_INTEGER`() itself be replaced.
- Fixed a bug in the GSI configuration parameters. If `GSI_DAEMON_DIRECTORY` was set and `GRIDMAP` was not set, then Condor would look in the wrong location for the GSI private key and mapfile.

- *condor_q* would produce garbage output in its error message when failing to contact the collector specified via **-pool**.
- In Unix only, fixed a file descriptor leak that could cause the *condor_schedd* daemon to crash.
- File transfer failures for spooled jobs no longer result in *condor_schedd* child processes hanging around for 8 hours before finally exiting. Too many such processes occasionally resulted in memory exhaustion.
- Fixed a bug in *condor_dagman*: `DIR` and `ABORT-DAG-ON` specifications were not propagated to rescue DAGs.
- Added a workaround for a bug in the Globus GRAM JobManager (http://bugzilla.mcs.anl.gov/globus/show_bug.cgi?id=5467) that can cause very short jobs' standard output and error to be lost.
- Disable GSI authorization callouts for the gridftp server that Condor starts to perform file transfers for grid-type gt4 grid universe jobs.

Known Bugs:

- Grid universe type GT4 (web services GRAM) does not work properly on Itanium-based machines, because it requies Java 1.5, which is not available on the Itanium (ia_64).

Additions and Changes to the Manual:

- Several updates to the DAGMan documentation (section 2.11).

## Version 6.8.5

Release Notes:

- The Globus libraries used by Condor now include the following advisory packages:
  - globus_gss_assist-3.23
  - globus_xio-0.35
  - globus_gram_protocol-6.5
  - globus_gass_transfer-2.12

  See http://www.globus.org/toolkit/advisories.html for details on the bugs fixed by these updated packages. The patch given in Globus Bugzilla 5091 (http://bugzilla.mcs.anl.gov/globus/show_bug.cgi?id=5091) is also included.

New Features:

- A clipped port to x86 Debian 4.0 has been added.
- The functionality embodied in *condor_q* **-better-analyze** is now available for X86_64 native ports of Condor.
- We now supply distinct, native ports for Mac OS X 10.3 and 10.4.
- There is a new configuration macro `COLLECTOR_REQUIREMENTS` that may be used to filter out unwanted ClassAd updates. For more information, see section 3.3.17.
- Added a **-f** option to *condor_store_cred*, which generates a pool password file that can be used for the PASSWORD authentication method on Unix Condor installations.

Bugs Fixed:

- The config file entry `HOSTALLOW_DAEMON` is now looked at in addition to `ALLOW_DAEMON` .
- Fixed a bug where under certain conditions Condor's file logging codes would perform a segmentation fault.
- Removed periodic re-indexing of the quill history_vertical table. This should not be needed with the current schema, and it should speed up database re-indexing operations.
- Fixed a bug that would cause the dedicated scheduler to crash, if the *condor_schedd* was suspended or blocked for more than approximately 10 minutes. The most likely cause of a suspension is a *condor_schedd* executable mounted from a remote NFS file system.
- Fixed a bug where if **-lc** was specified multiple times for the compiler when using *condor_compile* (some tools like *pgf90* do this), *condor_compile* would fail to link the application and emit a multiply defined symbol error for many symbols.
- Fixed a bug where Condor erroneously indicates that a scheduler universe's job executable is missing or not executable. This occurred if the scheduler universe job had been submitted with **CopyToSpool = false** in the submit description file, and the user had a umask which prevented the user named `condor` from following the search path to the user-owned executable.
- Fixed a bug that could cause the *condor_schedd* to crash if it received too many matches in one negotiation cycle (more than 1000 on a Linux platform).
- Fixed a bug in which *condor_history* did not honor the **-format** flag properly when Quill is in use.
- Fixed a bug in which a java property that includes surrounding double quote marks caused the detection of a java virtual machine to go awry. The fix, which may change in the future, changes any extra double quotes within a property value to single quotes.
- Fixed a bug in which the *condor_quill* daemon crashed occasionally when the Postgres database server was unavailable.

- The Solaris 9 Condor package can be used under Solaris 10 again. Changes in 6.7.20 broke this compatibility.
- *condor_dagman* now does a better job, especially in recovery mode, of detecting potentially incorrect submit events. Those have Condor IDs not matching what is expected.
- *condor_dagman* now truncates existing node job user log files to zero length, rather than deleting the log files. This prevents breaking the link if a user log file is set up as a link.
- When starting a GridFTP server to handle file transfers for gt4 grid jobs, the *condor_gridmanager* now properly sets the GLOBUS_TCP_PORT_RANGE and GLOBUS_TCP_SOURCE_RANGE environment variables if appropriate.
- Fixed a bug that could cause a security session to get deleted by the server (for example, the *condor_schedd*) before the client (for example, the *condor_shadow*) was done using it. This bug can be observed as communication failure the next time the client tried to connect to the server. In some cases, this caused jobs to be re-queued to be run again, because the final update of the job queue failed.
- If a grid job becomes held while it's still submitted to the remote resource and is then removed, the *condor_gridmanager* will now attempt to remove the job from the remote resource before letting it leave the local job queue.
- Fixed a bug in the *condor_c-gahp* that caused it to not use the user's credential for authentication with the remote schedd on some connections.
- The *condor_c-gahp* now properly lists all of the commands it supports in response to the COMMANDS command.
- Fix a bug in how the *condor_c-gahp* updates configuration parameter `GSI_DAEMON_NAME` to include the job's credential if it has one.
- Removed the 5096-character restriction on the length of DAG macro values (and names) in *condor_dagman*.
- Condor-G will now notice when jobs are missing from the status reports sent by the Grid Monitor. Jobs can disappear for short periods of time under normal circumstances, but a prolonged absence is often a sign of problems on the remote machine. The amount of time that a job can go missing from the Grid Monitor status reports before the *condor_gridmanager* reacts can be set by the new configuration parameter `GRID_MONITOR_NO_STATUS_TIMEOUT` . The default is 15 minutes.
- *condor_q* -analyze will now print a warning if a job being analyzed is already completed or if a grid universe job being analyzed has already been matched.
- In *condor_shadow*, when forwarding an updated X509 proxy to an executing job, the logic for whether to delegate or copy the proxy (determined by configuration parameter `DELEGATE_JOB_GSI_CREDENTIALS` ) was reversed. The authentication logic for this operation was also incorrect, causing the operation to fail in many instances.

- Made a small improvement to the reliability of Condor's process ancestry tracking under Linux. However, jobs that create children with more than 4096 bytes of environment are still problematic, due to a Linux kernel limitation that prevents reading more than 4k from /proc/¡pid¿/environ. The only truly reliable way to ensure that Condor is aware of all processes spawned by a Unix job is to use `VMx_USER`.
- *condor_glidein* option **-run_here** no longer fails when the current working directory is not in PATH.
- *condor_glidein* option **-runtime** would cause runtime errors at startup under some batch systems. The problematic parentheses characters are no longer generated as part of the environment value that is set by this option.
- On rare occasions, the *condor_startd* will compute a negative MIPS rating when performing benchmarks on the machine. This caused the `Mips` attribute to disappear from the machine ad. Now, the *condor_startd* ignores these bogus results. The cause of the negative MIPS ratings is still unknown.
- Fixed a bug that caused *condor_dagman* to hang if it processed, in recovery mode, a node for which all submit attempts failed and a POST script was run.
- Fixed a bug that would cause the *condor_negotiator*'s memory usage to grow over time when job or machine ClassAds made use of ClassAd functions that do regular expression matching operations.
- Fixed a bug that was preventing Condor daemons from caching DNS information for hosts authenticated via HOSTALLOW settings (i.e. no strong authentication). The collector, in particular, should spend much less time on IP to hostname lookups.
- When a job has an X509 proxy file (as indicated by the `X509UserProxy` attribute in the job ad), the *condor_starter* now always sets `X509_USER_PROXY` in the job's environment to point to a copy of that proxy file.
- Fixed several bugs that could cause the *condor_c-gahp* to time out when talking to the *condor_schedd* and falsely report that commands completed successfully. A common result is grid type condor grid universe jobs being placed on hold because the *condor_gridmanager* mistakenly thinks they disappeared from the remote *condor_schedd*'s queue.
- Fixed a bug in Stork which was causing it to write the output and error log files as the wrong user, and read the input file as the wrong user.
- Fixed a bug in Stork which was causing it to kill hung jobs as the wrong user.
- Fixed some possible static buffer overflows related to the transferring of a job's data files.
- Jobs with standard output and error going to the same file should not lose data in the common case.
- Heavily loaded condor daemons (e.g. *condor_schedd*) had a problem when they got behind processing the exit status of child process (e.g. *condor_shadow*). The problem was that the

daemon would continue to expect status updates from its child, even after the child had exited, and when the daemon decided that the lack of status updates meant that the child was hung, the daemon would try to kill any process that happened to have the same pid as the child which had already exited. In the case of the schedd, this would also result in the job run attempt being marked as a failure and the job would remain in the queue to run again. Condor no longer activates the "hung child" procedure for jobs which have exited but which have not yet had their exit status processed internally by the daemon.

- For grid-type condor jobs, made the *condor_gridmanager* more tolerant of unexpected responses from the remote condor_schedd.
- On HPUX and AIX, fixed a bug that could cause Condor's process family tracking logic to lose track of processes.
- Fixed a memory error that would cause *condor_q* to sometimes crash when using Quill.
- Fixed a problem where the Windows *condor_credd* would be inaccessible to other Condor components if CREDD_HOST were set to a DNS alias and not the canonical DNS name.
- Fixed a bug in the *condor_shadow* on Windows where it would fail to correctly perform the PASSWORD authentication method.
- The Windows *condor_credd* now uses the configuration parameter CREDD_HOST, if defined, to set its name when advertising itself to the *condor_collector*. Thus, if CREDD_HOST is set to something other than then *condor_credd*'s hostname, clients can still locate the daemon.
- Fixed a bug in the *condor_c-gahp* that could cause it to not perform hold, release, or remove commands on jobs in the remote *condor_schedd*.
- Fixed the default value of configuration parameter STARTD_AD_REEVAL_EXPR.

Known Bugs:

- None.

## Version 6.8.4

Release Notes:

- None.

New Features:

- Added new tool *condor_dump_history* which will enable schema migration to future Quill schema versions.

- Quill can now automatically rebuild the indexes on the *PostgreSQL* database tables. Some sites reported that even with auto vacuuming turned on, the indexes on the tables were growing without bounds. Rebuilding the indexes fixes that problem. Rebuilding is disabled by setting the parameter `QUILL_SHOULD_REINDEX` to `False`. Re-indexing happens immediately after the history file is purged of old data. So, if Quill is configured to never delete history data, the tables are never re-indexed. Also, *condor_quill* was changed so that the history deletion also happens at start time. This ensures that old history rows are deleted if Quill crashes before the scheduled deletion time.
- Added more information to StarterLog for an error message involved in file transfers:

  ```
  Download acknowledgment missing attribute: Result.
  ```

  The extra information is a full dump of the ClassAd that was received, in order to help determine why the expected attribute was not found.
- Added output to the `dagman.out` file documenting when *condor_dagman* shortcuts node retries because of *condor_submit* failures or a helper command failure.

Bugs Fixed:

- Fixed a bug in *condor_q* that only happened when running with a Quill database and using the long (-l) option. The bug was introduced in 6.8.3. The bug truncated the output of *condor_q*, and only displayed some of the job attributes.
- Fixed a bug in *condor_submit* that caused standard universe jobs to be unable to open their standard output or standard error, if **should_transfer_files** is **YES** or **IF_NEEDED** in the submit description file.
- Fixed a bug in *condor_glidein* that could cause it to request the queue unknown when submitting its setup job to GRAM, leading to failures.
- The `OnExitRemove` expression generated for DAGMan by *condor_submit_dag* evaluated to UNDEFINED for some values of `ExitCode`, causing *condor_dagman* to go on hold.
- Fixed a bug in which garbage values (random bits from memory) were sometimes written to the pool history file in the field representing the backfill state.
- *condor_submit_dag* now generates a submit file (`.condor.sub`) for *condor_dagman* that sends `stdout` and `stderr` to separate files. This has always been recommended, and recent versions of Condor cause `stdout` and `stderr` to overwrite each other if they are directed to the same file.
- Fixed several bugs for grid type **nordugrid** jobs. The *condor_gridmanager* would create an invalid RSL for these jobs and save their output to the wrong location in some cases.
- *condor_glidein* now properly escapes glidein tarball URLs that contain characters that have special meaning to GRAM RSL. It also turns on TCP updates to the *condor_collector*, if they are enabled on the submit machine.

- When using the submit file option **getenv=true**, environment variables containing a newline in their value are no longer inserted into the job's environment. The *condor_schedd* daemon does not allow newlines within ClassAd values, so the attempt to insert such values resulted in failure of job submission and caused the *condor_schedd* daemon to abort.
- Fixed a bug that caused *condor_dagman* to hang if a node with a POST script and retries initially runs but fails, and then has all *condor_submit* attempts fail on the retry.
- Fixed a problem in the Windows installer where the `DAEMON_LIST` parameter would be incorrectly set if the "Join an existing Condor pool" option was selected or the "Submit jobs to Condor pool" option was unchecked. In the first case, a *condor_collector* and *condor_negotiator* would incorrectly be run on the machine. In the second case, a *condor_schedd* would incorrectly be run. The problem exists in all previous 6.8 and 6.9 series releases.
- Fixed a bug in the handling of local universe jobs for a very busy *condor_schedd* daemon. When a local universe job completed, the *condor_starter* might not be able to connect to the *condor_schedd* daemon to update final information about the job, such as the exit status. Under this circumstance, the *condor_starter* would hang indefinitely. The bug is fixed by having the *condor_starter* attempt to retry a few times (with a delay in between each attempt) before exiting with a fatal error. The fatal error causes the job to restart.

Known Bugs:

- Setting `DAGMAN_DELETE_OLD_LOGS` to false can cause *condor_dagman* to have problems (including hanging), especially when running a rescue DAG. If you want to keep your old user log files, the best thing to do is to rename them before each *condor_dagman* run. If you do run with `DAGMAN_DELETE_OLD_LOGS` set to false, check your `dagman.out` file for error messages about submit event Condor IDs not matching the expected value. If you get such an error, you will probably have to *condor_rm* the *condor_dagman* job, remove or rename the old user log file(s) and run the rescue DAG. (Note: this bug also applies to earlier versions of *condor_dagman*.)

## Version 6.8.3

Release Notes:

- In this release, the command *condor_q -long* does not work when querying the Quill database. Instead, use the command *condor_q -direct quilld -long*, or use a previous version of *condor_q*.
- Performed a security audit of all places where Condor opens files, to make certain files are opened with a reasonable permission mode and with the O_EXCL flag whenever possible.

New Features:

- Added the JOB_INHERITS_STARTER_ENVIRONMENT configuration macro. When set to True, jobs inherit all environment variables from the *condor_starter*. This is useful for glidein jobs that need to access environment variables from the batch system running the glidein daemons. The default for this configuration macro is False, so existing behavior is unchanged. This feature does not apply to standard and pvm universe jobs.
- Changed the default UDP receive buffer for the *condor_collector* from 1M to 10M. This value can be configured with the (existing) COLLECTOR_SOCKET_BUFSIZE macro.

  NOTE: For some Linux distributions, it may be necessary to configure a larger value than the default; this parameter is /proc/sys/net/core/rmem_max . You can see the values that the *condor_collector* actually used by enabling D_FULLDEBUG for the *condor_collector* and looking at the log line that looks like this:

  Reset OS socket buffer size to 2048k (UDP), 255k (TCP).
- Added a new configuration macro to control the size of the TCP send buffers for the *condor_collector*. This macro used to be the same as COLLECTOR_SOCKET_BUFSIZE. The new macro is COLLECTOR_TCP_SOCKET_BUFSIZE , and it defaults to 128K.
- Added a clipped port for SuSE Linux Enterprise Server 9 running on the PowerPC architecture. Note the known bug below.
- The *condor_schedd* now maintains a birth date for the job queue. Nothing in Condor currently uses this feature, but future versions of *condor_quill* may require it.
- There is a new configuration file macro RANDOM_INTEGER(min,max[,step]). It produces a pseudo-random integer within the range min and max, inclusive at configuration time.

Bugs Fixed:

- Fixed a deadlock situation between the *condor_schedd* and the *condor_startd* that can significantly impact the *condor_schedd*'s performance. The likelihood of the deadlock increased based upon the number of VMs advertised by the *condor_startd*.
- Fixed a bug reading the user job log on Windows that caused occasional DAGMan confusion. Thanks to Fairview Software, Inc. for both finding the bug and writing a patch.
- Fixed a denial of service problem: Condor daemons no longer freeze for 20 seconds when a client connects to them and then sends no data. This behavior is common with port scanners.
- Fixed a race condition with *condor_quill* caused by *PostgreSQL*'s default transaction isolation level being "read committed". This bug would cause truncated *condor_q* reads when using Quill.
- Fixed a bug where the *condor_ckpt_server* would segfault when turned off with *condor_off* **-fast**.
- Fixed a bug in the *condor_startd* where it could die with SIGABRT when a *condor_starter* exited under certain rare circumstances. The bug seems to have been most likely to appear on x86_64 Linux machines, but could potentially affect all platforms.

- Fixed a problem with *condor_history* when running with Quill enabled, which caused it to allocate an unbounded amount of memory.
- Fixed a problem with *condor_q* when running with Quill, which caused it to silently truncate the printing of the job queue.
- Fixed a bug in the *condor_gridmanager* that caused the following configuration files parameters to be ignored for grid types condor and nordugrid jobs: `GRIDMANAGER_RESOURCE_PROBE_INTERVAL`, `GRIDMANAGER_MAX_PENDING_SUBMITS_PER_RESOURCE`, and `GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE`.
- Fixed a bug in *condor_run* that caused it to abort on non-fatal warnings from *condor_submit* and print incorrect error messages.
- Fixed a bug in the *condor_gridmanager* dealing with grid type gt4 grid universe jobs. If the job's standard output or error was not specified in the job ClassAd, the *condor_gridmanager* would create an improper GRAM RSL string, causing the job to fail.
- Fixed a bug in the *condor_gridmanager* that could cause it to delegate the wrong credential when refreshing the credentials for a grid type gt4 grid universe job.
- The *condor_gridmanager* could get into a state where it would no longer start up Globus jobmanagers for grid type gt2 grid universe jobs, if previous requests failed due to connection errors. This bug has been fixed.
- The *condor_c-gahp* now properly exits when the pipe to its parent goes away. Before, it would fill its log with large amounts of useless messages, before exiting several minutes later.
- Fixed a bug where a problem opening standard input, output, or error, the standard universe might generate an incorrect warning in the *condor_shadow*'s log.
- The *condor_gridmanager* now recovers properly when a proxy refresh fails for a gt2 grid universe job in the stage-out state. Before, the job would become held with a hold reason of "Globus error 3: an I/O operation failed".
- A number of fixes to minor typos and incorrect formatting in Condor's log files.
- When `REQUEST_CLAIM_TIMEOUT` was reached and the *condor_schedd* failed to contact the *condor_startd* to release the claim, the *condor_schedd* would periodically try releasing the claim indefinitely, possibly resulting in a lengthy communication delay each time.
- Under Windows, Condor daemons such as the *condor_schedd* were sometimes limiting their use of pending connect operations more than they should have. This would result in the message, "file descriptor safety level exceeded".
- *condor_fetchlog* no longer allows or documents the -dagman option. The option's appearance was an error. The option never worked.

- The *condor_schedd* ensures that the initial job queue log file contains a sequence number for use by Quill. This fixes a case in which no sequence number was inserted, because the initial rotation of this (empty) file failed. Quill also now reports exactly what the problem is if it reads a job queue log in this state, rather than simply crashing. This problem has so far only been observed under Windows.
- Fixed a problem on Windows where, when submitting a job with a sandbox (for example, using the **-s** or **-r** option to *condor_submit*), an erroneous file permissions check in the *condor_schedd* would result in a failed submission.
- The *condor_startd* would crash shortly after start up if the `RANK` expression contained any use of the unary minus operator. This patch should also fix any other cases where Condor daemons crashed due to the use of the unary minus operator in ClassAd expressions.
- Stork now writes a terminated event to the user log when it removes a transfer job from its queue because of failures to invoke a transfer module. Without this event, DAGMan would not notice that these jobs had left the queue.
- Fixed a problem where the *condor_schedd* on Windows would incorrectly reject a job if the client provided an `Owner` attribute that was correct but differed in case from the authenticated name. This bug was thought to have been fixed in Condor 6.8.0.
- Fixed problems with *condor_store_cred* behaving strangely when storing or removing a user name that is some initial substring of "condor_pool". Specifying such a user name would be incorrectly interpreted as equivalent to specifying the **-c** option.
- Fixed a problem with *condor_glidein* spewing lots of text to the screen when checking the status of a job it submitted.
- A new version of the GT4 GAHP is included, with the following changes:
  - A new `axis.jar` from Globus fixes a thread safety bug that can cause lockups in subscriptions for WS notifications. See Globus Bugzilla 4858 (http://bugzilla.globus.org/bugzilla/show_bug.cgi?id=4858).
  - Fixed bugs that caused memory related to destroyed jobs to not be reclaimed in both the client and the server.
  - Removed redundant usage of Secure Message, Secure Conversation, and Transport Security when talking to a WS GRAM service. Now, only Transport Security is used.
- Fixed memory leaks in *condor_quill*.
- Fixed a bug that might have caused *condor_startd* problems launching the *condor_starter* for the standard universe on 64-bit systems.
- Improved Condor's file transfer. If you request that Condor automatically transfer back your output, it now detects changes better. Previously, it would only transfer back files that had a more recent timestamp than the spool date. Now, it will transfer back any file that has changed in date (including being dated in the past) or changed in size.

Known Bugs:

- SuSE Linux Enterprise Server 9 on PowerPC only: The default Java interpreter on SuSE Linux Enterprise Server 9 running on the PowerPC architecture has compatibility problems with this release of Condor. The problem exhibits itself as the *condor_startd* hanging, never reporting itself to the *condor_collector*. The workaround is to either disable the Java universe (set JAVA to an empty string), or disable just-in-time compilation when running in the Java universe with the following configuration setting:

  ```
  JAVA_EXTRA_ARGUMENTS = -Djava.compiler=NONE
  ```

## Version 6.8.2

Release Notes:

- Condor now uses Globus 4.0.3 for GSI, GRAM, and GridFTP support. This includes a patch for the OpenSSL vulnerability detailed in CVE-2006-4339 and http://www.openssl.org/news/secadv_20060905.txt. It also includes fixes for Globus Bugzilla 4689 (http://bugzilla.globus.org/bugzilla/show_bug.cgi?id=4689) and a bug that can cause duplicate UUIDs to be generated for WS GRAM jobs.
- The *condor_schedd* daemon no longer forks separate processes to change ownership of job directories in the spool. Previously on Unix-like systems, this would create a new process before a job started running and after it finished running. Some sites with very busy *condor_schedd* daemons were encountering scaling problems.

New Features:

- Because, by default, the *condor_startd* daemon references the job ClassAd attribute NumCkpts, Condor's default configuration will now round up the value of NumCkpts, in order to improve matchmaking performance. See the entry on SCHEDD_ROUND_ATTR in section 3.3.11.
- Enhanced the RHEL3 x86_64 port of Condor to include the standard universe.
- *condor_submit_dag* **-f** no longer deletes the dagman.out file. *condor_submit_dag* without the **-f** option will now submit a DAGMan run even if the dagman.out file exists. In this case, the file will be appended to.
- Added a property to the Windows installer program to determine whether the Condor service will be started after installation. The property name is STARTSERVICE, and the default value is "Y".

Bugs Fixed:

- A bug caused the *condor_master* daemon to kill only immediate children within the process tree, upon an abnormal exit of the *condor_master* daemon. The *condor_master* daemon now kills all descendant processes.
- Fixed a bug where if the file system was full, the debugging log files (for example `SchedLog`) would silently lose messages. Now, if the disk is full, the Condor daemons will exit.
- Fixed a bug in the *condor_schedd* daemon that caused it to stop negotiating for grid universe jobs in the case that it decided it could not spawn any new *condor_shadow* processes.
- Added the ProcessId class (which more uniquely identifies a process than a PID does) to the *condor_dagman* abort duplicate runs feature. This makes it less likely that a given instance of *condor_dagman* will mistakenly conclude that another instance of *condor_dagman* is already running on the same DAG. Also fixed an unrelated bug in the abort duplicate runs feature that could cause a *condor_dagman* to not abort itself when it should.
- Condor daemons leaked memory (consuming more and more memory over time) when parsing ClassAds that use functions with arguments.
- Fixed a bug in the *condor_starter* daemon, which caused it to look in the wrong place for the job's executable, if `TransferExecutable` was set to `True` in the job ClassAd.
- *condor_history* no longer crashes if `HISTORY` is not defined in the Condor configuration file.
- Fixed an unintentional change to the value of **-Condorlog** in a *condor_dagman* submit description file: it is once again the log file of the first node job.
- Fixed a bug in *condor_q* that would cause *condor_q* **-hold** or *condor_q* **-run** to exit with an error on some platforms.
- Fixed a bug on Unix platforms, in which a misconfiguration of `MAIL` would cause the *condor_master* daemon to restart all of its child daemons whenever it tried (and failed) to send e-mail to the administrator.
- Network related error messages have been improved to make debugging easier. For example, when timing out on a read or write operation, the peer's address is now included in the error message.
- An invalid value for `UPDATE_INTERVAL` now causes the *condor_startd* daemon to abort. Previously, it would continue running, but some invalid values (for example, 0) could cause it to stop sending periodic ClassAd updates to the *condor_collector*, even after being reconfigured with a valid value. Only a complete restart of the *condor_startd* daemon was sufficient to get it out of this state.
- Fixed a bug that caused X.509 limited proxies to be delegated as impersonation (i.e. non-limited) proxies. Any authentication attempted with the resulting proxies would fail.
- Fixed a couple bugs that would cause Condor to lose track of some Condor-related processes and subsequently fail to clean up (kill) these processes.

- Fixed a bug that would cause *condor_history* to crash when dealing with rotated history files. Note that history file rotation is turned on by default. (See Section 3.3.3 for descriptions of ENABLE_HISTORY_ROTATION and MAX_HISTORY_ROTATIONS .)

Known Bugs:

- None.

## Version 6.8.1

Release Notes:

- Version 6.8.1 fixes important bugs, some of which have security implications. All users are encouraged to upgrade, and full disclosure of the vulnerabilities will be given at the end of October 2006.
- Condor is now linked against GSI from Globus 4.0.2. This includes a patch for Globus Security Advisories 2006-01 (http://www.globus.org/mail_archive/security-announce/2006/08/msg00000.html) and 2006-02 (http://www.globus.org/mail_archive/security-announce/2006/08/msg00001.html). It also includes a patch for the OpenSSL vulnerability detailed in CVE-2006-4339 and http://www.openssl.org/news/secadv_20060905.txt.
- The PCRE (Perl Compatible Regular Expressions) library used by Condor is now dynamically linked and shipped as a DLL with Condor for Windows, rather than being statically linked.

New Features:

- Added an optional argument to the *condor_dagman* ABORT-DAG-ON command that allows the DAGMan exit code to be specified separately from the node value that causes the abort; also, a DAG can now be aborted on a zero exit code from a node.
- Added the ALLOW_FORCE_RM configuration variable. If this expression evaluates to True, then an *condor_rm* -f attempt is allowed. If it evaluated to False, the attempt is disallowed. The expression is evaluated in the context of the job ClassAd. If not defined, the value defaults to True, matching the behavior of previous Condor releases.
- *condor_dagman* will now reject DAGs for which any of the nodes' user job log files are on NFS (because of the unreliability of NFS file locking, this can cause DAGs to fail). This feature can be turned off by setting the DAGMAN_LOG_ON_NFS_IS_ERROR configuration macro to False (the default is True).
- *condor_submit* can now be configured to reject jobs for which the log file is on NFS. To do this, set the LOG_ON_NFS_IS_ERROR configuration macro to True. The default is that condor_submit will issue a warning for a log file on NFS.

- Added the DAGMAN_ABORT_DUPLICATES configuration macro, which causes *condor_dagman* to attempt to detect at startup whether another *condor_dagman* is already running on the same DAG; if so, the second *condor_dagman* will abort itself.
- The new configuration variable NETWORK_MAX_PENDING_CONNECTS may be used to limit the maximum number of simultaneous network connection attempts. This is primarily relevant to the *condor_schedd* daemon, which may try to connect to large numbers of *condor_startd* daemons when claiming them. The *condor_negotiator* may also connect to large numbers of *condor_startd* daemons when initiating security sessions used for sending MATCH messages. On Unix, the default is to allow up to eighty percent of the process file descriptor limit. On Windows, the default is 1600.
- Added some more debug output to *condor_dagman* to clarify fatal errors.
- The -format argument to *condor_q* and *condor_status* can now take an expression in addition to a simple attribute name.
- DRMAA is now available on most Linux platforms, Windows and PPC MacOS.

Bugs Fixed:

- When a large number of jobs (roughly 200 or more) are running from a single *condor_schedd* daemon, and those jobs are using job leases (the default in 6.8), it is possible for the *condor_schedd* daemon to enter a state where it crashes on startup until all of the job leases expire.
- Condor jobs submitted with the NiceUser priority were not being matched if the NEGOTIATOR_MATCHLIST_CACHING setting was TRUE (which is enabled by default).
- Fixed a Quill bug that prevented it from running on Windows. The symptom showed with errors in the QuillLog such as

  ```
  POLLING RESULT: ERROR
  ```

- Fixed a bug in Quill where it would cause errors such as

  ```
  duplicate key violates unique constraint "history_vertical_pkey"
  ```

  in the QuillLog and the *PostgreSQL* log file. These errors triggered a significant slowdown in the performance of Quill and the database. This would only happen when a job attribute changed type from a string type to a numeric type, or vice versa.
- In those unusual cases where Condor is unable to create a new process, it shuts down cleanly, eliminating a small possibility of data corruption.
- Fixed a bug with the gt4 and nordugrid grid universe jobs that caused the stdout and stderr of a job to not be transferred correctly, if the given file names had absolute paths.

- *condor_dagman* now echos warnings from *condor_submit* and *stork_submit* to the `dagman.out` file.
- Fixed a bug introduced in 6.7.20, causing the *condor_ckpt_server* to exit immediately after starting up, unless Condor's security negotiation was disabled.
- `MAX_<SUBSYS>_LOG` defaults to one Megabyte, even if the setting is missing from the configuration. Previously it was 64 Kilobytes.
- Fixed a bug related to non-blocking connect that could occasionally cause Condor daemons to crash.
- Fixed a rare bug where an exceptionally large query to the *condor_collector* could cause it to crash. The most common cause was a single *condor_schedd* daemon restarting, and trying to recover a large number of job leases at once. More than approximately 250 running jobs on a single *condor_schedd* daemon would be necessary to trigger this bug.
- When using the `JOB_PROXY_OVERRIDE_FILE` configuration parameter, the X.509 proxy will now be properly forwarded for Condor-C jobs.
- Greatly reduced the chance that a Condor-C job in the REMOVED state will be HELD due to an expired proxy or failure to talk to the remote *condor_schedd*.
- Fixed error and debug messages added in Condor version 6.7.20 that incorrectly reported IP and port numbers. These messages were intended to report the peer's address, but they were instead reporting the local address of the network socket.
- Fixed a bug introduced in Condor version 6.7.20 which could cause Condor daemons to die with the message

  ```
  PANIC -- OUT OF FILE DESCRIPTORS
  ```

  The conditions causing this related to failed attempts to send updated status to the *condor_collector* daemon, with both non-blocking updates and security negotiation enabled (the defaults).
- Also fixed a bug in the negotiator with the same effect as above, except it only happened with the configuration setting `NEGOTIATOR_USE_NONBLOCKING_STARTD_CONTACT`=False.
- Fixed a bug in *condor_schedd* under Solaris that could also cause file descriptors to become exhausted over time when many machines were claimed in a short spans of time (e.g. over 100) and the *condor_schedd* process file descriptor limit was near 256.
- Fixed a bug in *condor_schedd* under Windows that could cause network sockets to be allocated and never released back to the system. The circumstances that could cause this were very rare. The error message in the logs indicating that this problem was happening is

  ```
  ERROR: DuplicateHandle() failed in Sock::set_inheritable
  ```

  In cases where this error message is displayed, the network socket is closed.

- Under some conditions, when making TCP connections, Condor was still trying to connect for the full duration of the operation timeout (often 10 or 20 seconds), even if the connection attempt was refused (for example, because the port being accessed is not accepting connections). Now, the connect operation finishes immediately after the first such failure, allowing the Condor process to continue with other tasks.
- Fixed the problems relating to credential cache problems in the Kerberos authentication mechanism. The current version of Kerberos is 1.4.3.
- Fixed bugs in the SSL authentication mechanism that caused the *condor_schedd* to crash when submitting a job (on Unix) and caused all tools and daemons to crash on Windows when using SSL.
- Some of the binaries required to use Condor-C on Windows were mistakenly not included in previous releases of Condor. This has been fixed.
- Fixed a problem on Windows where the *condor_startd* could fail to include some attributes in its ClassAd. This would result in some jobs incorrectly not being matched to that machine. This only happened if `CREDD_HOST` was defined and Condor daemons on the execute machine were unable to authenticate with the *condor_credd*.
- Fixed a *condor_dagman* bug which had prevented the `$(DAGManJobId)` attribute from being expanded in job submit files (for example, when used as the value to define the **Priority** command).
- Fixed a bug in *condor_submit* that caused parallel universe jobs submitted via Condor-C to become mpi universe jobs.
- Fixed a bug which could cause Condor daemons to hang if they try to write to the standard error stream (`stderr`) on some platforms. In general, this should never happen, but can, due to third party libraries (beyond our control) trying to write error or other messages.
- Fixed *condor_status* to report error messages.
- Fixed a bug in which setting the configuration variable

  ```
  NEGOTIATOR_CONSIDER_PREEMPTION = False
  ```

  caused an incorrect calculation. The fraction of the pool already being claimed by a user was calculated using the wrong total number of *condor_startd* daemons. This could cause some *condor_startd* daemons to remain unclaimed, even when there were jobs available to run on them.
- Fixed a security vulnerability in Condor's FS and FS_REMOTE authentication methods. The vulnerability allowed an attacker to impersonate another user on the system, potentially allowing submission of jobs as a different user. This may allow escalation to root privilege if the Condor binaries and configuration files have improper permissions. The fix is not backwards compatible, which means all daemons and tools using FS authentication must be running Condor 6.8.1 or greater. The same applies to FS_REMOTE; All daemons and tools using FS_REMOTE must be using Condor 6.8.1 or greater. In practice, this means that for FS, all

Condor binaries on one host must be version 6.8.1 or greater, but versions can be different from host to host. For FS_REMOTE it means all binaries across all hosts must be 6.8.1 or greater.

- Fixed a couple race conditions in stork and the credd where credential files were possibly created with improper permissions before being set to owner permissions.
- Fixed a bug in the *condor_gridmanager* that caused it to delegate 12-hour proxies for grid-type gt4 jobs and then not refresh them.
- Fixed a bug in the *condor_gridmanager* that caused a directory needed for staging-in of grid-type gt4 job files to be removed when the *condor_Gridmanager* exited, causing the stage-in to fail.
- Fixed a bug that caused the *checkpoint server* to restart because of (ostensibly) getting an unexpected errno from select().
- Fixed a bug on Windows where setting **output** or **error** to a relative or absolute path (as opposed to a simple file name without path information) would not work properly.
- History file rotation did not previously work on Windows because the name of a rotated files would contain an ISO 8601 extended format timestamp, which contains colon characters. The naming convention for rotated files has been modified to use ISO 8601 basic format, avoiding this problem.
- The CLAIMTOBE authentication method (which is inherently insecure and should only be used for testing or other special circumstances) previously would authenticate without providing the "domain" portion of the user name. As an example, a user would be authenticated as simply "user" rather than "user@cs.wisc.edu". This problem has been fixed, but the new protocol is not backwards compatible so the fix is turned off by default. Correct behavior can be enabled by setting the `SEC_CLAIMTOBE_INCLUDE_DOMAIN` parameter to `True`.
- Fixed a bug with the `NEGOTIATOR_MATCHLIST_CACHING` that would cause very low-priority jobs (like jobs submitted with `nice_user=True`) to not match even if resources were available.
- Fixed a buffer overflow that could crash the *condor_negotiator*.
- `SCHEDD_ROUND_ATTR_<XXXX>` preserves the value being rounded up when it is a multiple of the power of 10 specified for rounding. Previously, the value would be incremented; now it remains the same. For example, if SCHEDD_ROUND_ATTR_¡xxxx¿=2 and the value being rounded up is 100, it now remains 100, rather than being incremented to 200.
- Fixed *condor_updates_stats* to report it's version number correctly.

Known Bugs:

- The **-completedsince** option to *condor_history* works when Quill is enabled. The behavior of *condor_history* **-completedsince** is undefined when Quill is *not* enabled.

### Version 6.8.0

Release Notes:

- The default configuration for Condor now requires that HOSTALLOW_WRITE be explicitly set. Condor will refuse to start if the default configuration is used unmodified. Existing installations should not need to change anything. For those who desire the earlier default, you can set it to "*", but note that this is potentially a security hole allowing anyone to submit jobs or machines to your pool.
- Most Linux distributions are now supported using dynamically linked binaries built on a RedHat Enterprise Linux 3 machine. Recent security patches to a number of Linux distributions have rendered the binaries built on RedHat 9 machines ineffective. The download pages have been changed to reflect this, but Linux users should be aware of this change. The recommended download for most x86 Linux users is now: condor-6.8.0-linux-x86-rhel3-dynamic.tar.gz.
- Some log messages have been clarified or moved to different debugging levels. For example, certain messages that looked like errors were printed to D_ALWAYS, even though nothing was wrong and the system was behaving as expected.
- The new features and bugs fixed in the rest of this section only refer to changes made since the 6.7.20 release, not the last stable release (6.6.11). For a complete list of changes since 6.6.11, read the 6.7 version history in section 8.4 on page 513.

New Features:

- Version 1.4 of the Condor DRMAA libraries are now included with the Condor release. For more information about DRMAA, see section 4.4.2 on page 402.
- Version 1.0.15 of the Condor GAHP is now used for Condor-G and Condor-C.
- Added the **-outfile_dir** command-line argument to *condor_submit_dag*. This allows you to change the directory in which *condor_dagman* writes the dagman.out file.
- Added a new **–summary** (also **-s**) option to the *condor_update_stats* tool. If enabled, this prevents it from displaying the entire history for each machine and only displays the summary info.

Bugs Fixed:

- Fixed a number of potential static buffer overflows in various Condor daemons and libraries.
- Fixed some small memory leaks in the *condor_startd*, *condor_schedd*, and a potential leak that effected all Condor daemons.

- Fixed a bug in Quill which caused it to crash when certain long attributes appeared in a job ad.
- The startd would crash after a reconfig if the address of a collector had not been resolved since the previous reconfig (e.g. because DNS was down during that time).
- Once a Condor daemon failed to lookup the IP address of the collector (e.g. because DNS was down), it would fail to contact the collector from that time until the next reconfig. Now, each time Condor tries to contact the collector, it generates a fresh DNS query if the previous attempt failed.
- When using Condor-C or the -s or -r command-line options to condor_submit, the job's standard output and error would be placed in the job's initial working directory, even if the job ad said to place them in a different directory.
- Greatly sped up the parsing of large DAGs (by a factor of 50 or so) by using a hash table instead of linear search to find DAG nodes.
- Fixed a bug in *condor_dagman* that caused an EXECUTABLE_ERROR event from a node job to abort the DAG instead of just marking the relevant node as failed.
- Fixed a bug in *condor_collector* that caused it to discard machine ads that don't have an IP address field (either StartdIpAddr or STARTD_IP_ADDR). The *condor_startd* will always produce a StartdIpAddr field, but machine ads published through *condor_advertise* may not.
- When using `BIND_ALL_INTERFACES` on a dual-homed machine, a bug introduced in 6.7.18 was causing Condor daemons to sometimes incorrectly report their IP addresses, which could cause jobs to fail to start running.
- Made the event checking in *condor_dagman* less strict: added the new "allow duplicate events" value to the `DAGMAN_ALLOW_EVENTS` macro (this value is part of the default); 16 value now also allows terminate event before submit; changed "allow all events" to "allow almost all events" (all except "run after terminal event"), so it is more useful.
- *condor_dagman* and *condor_submit_dag* now report **-NoEventChecks** as ignored rather than deprecated.
- Fixed a bug in the *condor_dagman* **-maxidle** feature: a shadow exception event now puts the corresponding job into the idle state in *condor_dagman*'s internal count.
- Fixed a problem on Windows where daemons would sometimes crash when dealing with UNC path names.
- Fixed a problem where the *condor_schedd* on Windows would incorrectly reject a job if the client provided an `Owner` attribute that was correct but differed in case from the authenticated name.
- Fixed a *condor_startd* crash introduced in version 6.7.20. This crash would appear if an execute machine was matched for preemption but then not claimed in time by the appropriate *condor_schedd*.

- Resolved an issue where the *condor_startd* was unable to clean up jobs' execute directories on Windows when the *condor_master* was started from the command line rather than as a service.
- Added more patches to Condor's DRMAA interface to make it more compatible with Sun Grid Engine's DRMAA interface.
- Removed the unused `D_UPDOWN` debug level and added the `D_CONFIG` debug level.
- Fixed a bug that caused *condor_q* with the **-l** or **-xml** arguments to print out duplicate attributes when using Quill.
- Fixed a bug that prevented Condor-C jobs (universe grid jobs of type condor) from submitting correctly if `QUEUE_ALL_USERS_TRUSTED` is set to True.
- Fixed a bug that could cause the *condor_negotiator* to crash if the pool contains several different versions of the *condor_schedd* and in the config file `NEGOTIATOR_MATCHLIST_CACHING` is set to True.
- Changed the default value for config file entry `NEGOTIATOR_MATCHLIST_CACHING` from False to True. When set to True, this will instruct the negotiator to safely cache data in order to improve matchmaking performance.
- The Condormaster now recognizes *condor_quill* as a valid Condor daemon without any manual configuration on the part of site administrators. This simplifies the configuration changes required to enable Quill.
- Fixed a rare bug in the *condor_starter* where if there was a failure transferring job output files back to the submitting host, it could hang indefinitely, and the job appeared as if it was continuing to run.

Known Bugs:

- The **-completedsince** option to *condor_history* works when Quill is enabled. The behavior of *condor_history* **-completedsince** is undefined when Quill is *not* enabled.

## 8.4 Development Release Series 6.7

This is the development release series of Condor. The details of each version are described below.

### Version 6.7.20

Release Notes:

- Condor no longer supports SGI IRIX platforms. No futher releases for this platform will be built or distributed.
- *condor_submit* on Windows no longer checks that the schedd has access to the submitter's credential if invoked with the **-n** or **-r** option. It is therefore necessary to make sure ahead of time that the credential is correctly stored with *condor_store_cred* before doing a remote submit.
- Version 1.3.2 of the Generic Connection Broker (GCB) library is now used for building Condor, and it is the 1.3.2 versions of the *gcb_broker* and *gcb_relay_server* programs that are included in this release. For more information about GCB, see section 3.7.3 on page 301.

New Features:

- Added a variety of built-in functions to ClassAds. Examples of new functionality include the ability to express conditionals, string operations, and regular expression matching.
- Condor can now map authenticated names (e.g. an X509 subject name or Kerberos principle) to canonical Condor user names via a unified mapfile.
- condor_stats and the view server are now aware of the new backfill state for machines, and record and report statistics on it.
- Condor now supports running backfill jobs on Windows machines. See section 3.12.9 on page 359 for more information about running backfill jobs with Condor.
- Condor-C is now supported on Windows. When using Condor-C to direct a job to a Windows remote schedd, one must be careful to ensure that their credential is accessible to the remote schedd and that the `NTDomain` attribute in the remote job ClassAd is set correctly. In particular, if the local schedd resides in a different Windows domain from that of the remote schedd, it is necessary to include a line like the following in the submit file:

  ```
  +remote_NTDomain = "OTHERDOMAIN"
  ```

- Added a `SUBMIT_MAX_PROCS_IN_CLUSTER` configuration parameter to allow administrators to limit the number of jobs that can be submitted in a single cluster when using *condor_submit*. This parameter defaults to 0, which implies no limit.
- Added config file parameter `QUEUE_ALL_USERS_TRUSTED` which can be used to disable authorization checks to the job queue. See section 3.3.11 on page 184.
- *condor_dagman* now re-checks immediately before job submission that every node job submit file defines a log file.
- *condor_dagman* now requires that all Stork submit files used in a DAG define a log file.
- New macro functionality in job ClassAds: $$([ClassAd Expression]). The contained ClassAd expression is evaluated when the job is matched. "My" refers to attributes in the job's ClassAd, "Target" refers to attributes in the machine classad.

- Condor can now run a program to obtain it's configuration parameters. If a configuration filename (such the environment variable CONDOR_CONFIG or the configuration parameter LOCAL_CONFIG_FILE) ends with a vertical bar ("| "), it is executed and its standard output is parsed for configuration parameters. If LOCAL_CONFIG_FILE is used in this way, then it can only contain a single item, and spaces in the value will be interpreted as part of the command to be executed.

- Added the *condor_dagman* configuration parameter `DAGMAN_PROHIBIT_MULTI_JOBS` , which prohibits *condor_dagman* from running a DAG that references node job submit files that queue multiple jobs (other than parallel universe).

- A number of types of failures to run a job now result in the job going on hold, rather than immediately being returned to the idle state to be tried again. This currently does not apply to standard universe jobs. The types of errors that now result in the job going on hold are failure to execute the specified program, failure to transfer files, failure to open input or output files, and failure to access the job's initial working directory. In all such cases, a specific hold reason is specified in the job ClassAd, along with a numeric hold code and subcode. If you wish to automatically retry in such cases (the old behavior), then you can specify a `PeriodicRelease` expression that checks for specific hold states.

- Strong authentication using SSL is now available for web-service clients using the SOAP (BirdBath) interface commands. The Condor daemons can communicate via HTTPS on a specified port, and clients must present a client-side SSL certificate.

- Previously, the *condor_schedd* only would communicate with one web-service client at a time. This restriction has now been removed; multiple simultaneous transactions to the schedd via the SOAP (BirdBath) interface is now supported.

- *condor_submit* will now issue a warning if the user / job log is on an NFS mounted file system.

- When a job terminates or is removed and its working directory is on an NFS mounted file system, the condor_schedd creates and removes a file in the working directory to force the NFS client to sync with the NFS server and see any files written by the job.

- Non-blocking connect operations are now used in two cases: sending ClassAd updates from Condor Daemons to the collector and sending match information from the negotiator to the startd. Both of these operations are UDP-based (unless you enable TCP updates to the collector), so non-blocking connects would not be an issue, except that TCP connections are required whenever it is necessary to establish a new security session. An example where the new non-blocking behavior is helpful is when a machine is down and TCP connections to it timeout. Daemons that try to connect to it using non-blocking connections will no longer stop everything they are doing for the full duration of the timeout.

  This feature may result in a greater number of sockets being open at one time than previously (especially in the negotiator). There is not yet support for placing a limit on the number of simultaneous connection attempts, therefore, if you need to turn off the use of non-blocking connects, you may do so with the following configuration settings:

  `NONBLOCKING_COLLECTOR_UPDATE` = False
  `NEGOTIATOR_USE_NONBLOCKING_STARTD_CONTACT` = False

- *condor_quill* now includes an additional “schema version” table. If the database was created prior to 6.7.20, the new table is automatically added by the 6.7.20 Quill daemon.
- *condor_submit* will now issue a warning if the user / job log is on an NFS mounted file system.

Bugs Fixed:

- Fixed a bug introduced in 6.7.19 which prevented MPI jobs from running in some situations.
- Fixed a bug in the Dedicated scheduler where parallel jobs with job leases would get restarted from scratch if the collector had crashed when the schedd restarted.
- Fixed a rare bug where after a job has been marked terminated in the user’s job log, it could be run again and another execute event could be written to the user’s job log after the terminate event. This caused problems with DAGMan, which really depends on the this log being correct. This fix is applicable to the vanilla, standard, parallel, mpi, and java universes.
- Fixed several bugs in quill that would cause it to crash when very long values for job classad attributes where in the queue. This typically happened with jobs submitted with ”getenv = true” in the submit file, and very large environments.
- Fixed a bug in ClassAds where an attribute name which was constructed from a keyword with one or more digits appended would cause parse errors. Discovered with T1 and F1(short form of true and false).
- Fixed a bug introduced in 6.7.19, which could cause the startd to get into a state where it would reject all new claims (requiring a restart of the startd to resume normal operation). The symptom of this bug was a recurring message in the startd such as this:

```
5/19 23:51:47 vm1: ClaimId from schedd (<xxx.xxx.xxx.xxx:45877>#1147893880#1208)
5/19 23:51:47 vm1: State change: claiming protocol failed
```

- Fixed a bug introduced in 6.7.19 which could cause the startd to abort (signal 6 under unix) when a match timed out for a claim preempting the existing preempting claim.
- Fixed a bug which could cause *condor_quill* to perform an illegal memory access and potential segmentation fault.
- When a job is matched to a remote machine through flocking, the remote machine is given WRITE and DAEMON permissions on the submit machine. This functionality previously worked, but broken some time in the 6.7 series.
- Fixed a bug which could cause *condor_status* (and probably other tools and possibly even daemons) to crash on Solaris machines if address resolution fails because of an ill-configured DNS server.
- Fixed a bug on unix which could cause the Condor daemons to mistakenly think a child process was successfully created (when the fork() system call returned -1).

- Fixed a bug which could cause network write operations to block Condor daemons for the full networking timeout time when the connection was closed while the daemon was still writing to it.
- Streaming standard input, output, and error has been fixed for Windows jobs.
- Grid-type gt4 jobs will now go on hold if Condor can't delegate their X509 proxies to the remote Delegation Service.
- The sample configuration file `condor_config.local.credd` contained a typo which has now been fixed.
- Fixed a bug which caused the checkpoint server (*condor_ckpt_server*) to publish an incorrect IP address in the `Machine` attribute of it's ClassAd.
- Fixed some rare bugs when the Condor claiming protocol fails while a machine is running a backfill job. The *condor_startd* now correctly recovers from these failures.
- Fixed a bug in Condor's file-transfer mode that could cause file transfer errors to go unnoticed by one side of the connection. One possible result of this would be a job leaving the queue "successfully" when there was actually an error copying back one of the output files. The possibility of this bug happening was much less for large files (greater than 65536 bytes).
- DAG variable names beginning with "queue" can goof up DAG node job submits; *condor_dagman* now checks for such variable names and fails with an explicit error message if there are any. This bug has probably existed for a long time, but was just recently discovered.
- *condor_off* -peaceful was still resulting in a shutdown timeout after `GRACEFUL_SHUTDOWN_TIMEOUT`, which would then cause jobs to be preempted.
- If `PeriodicHold` or `PeriodicRemove` triggered for a held or idle job, the hold or abort event would not be written to the user log if XML logging was enabled.
- Vanilla universe jobs were failing to run if the executable was specified as a relative path and `transfer_executable` was set to false.
- Many user log events for grid universe jobs were not written in XML format when `log_xml` was set to true.
- Gridftp server jobs automatically submitted to handle file transfers for grid-type gt4 jobs now properly leave the queue when they enter the REMOVED state.
- Fixed a bug in the event checking code used by *condor_dagman* and *condor_check_userlogs* which caused errors to be reported when they should not have been for parallel universe jobs. Also fixed a bug in *condor_check_userlogs* that caused it to sometimes not report an error when it should have.
- PBS and LSF grid universe jobs now fully handle the new job attributes `Arguments` and `Environment`. Previously, jobs would be put on hold if the values couldn't be converted to the old representation used in attributes `Args` and `Env`.

- A bug that was introduced in version 6.7.18 in which Condor would fail to send e-mail when trying to send to multiple recipients has been fixed.
- Fixed a bug in *condor_dagman* that caused it to abort without generating a rescue DAG file if a node job user log somehow contained a bad event type number.
- When the `EXECUTE_LOGIN_IS_DEDICATED` config file option is set to True, the *condor_starter* would occasionally crash. This bug has been fixed.
- When the `LOCAL_CONFIG_DIR` config file option was set to an invalid path, the daemons and command line tools would segfault. This has been fixed.
- Fixed the **-dag** option to *condor_q*. Ever since version 6.7.7, this option did not print DAG node names correctly if there were multiple DAGMan jobs submitted.
- Eliminated a bug where a malformed ClassAd attribute could make the schedd's on-disk job queue unreadable.
- Fixed a number of cases where a large ClassAd value could crash Condor.

Known Bugs:

- None.

## Version 6.7.19

Release Notes:

- A major security hole has been fixed in the checkpoint server. The hole allows arbitrary files owned by the condor UID, or the UID of a Personal Condor running a checkpoint server, to be read and written. Users who can not upgrade to this release of Condor are urged to replace the *condor_ckpt_server* binary with the version in this release. This applies to all versions of Condor, including the 6.6 series.

  To replace only the *condor_ckpt_server* binary,

  1. Download the 6.7.19 *condor_ckpt_server* binary, from the contrib page at http://www.cs.wisc.edu/condor/downloads/contrib.license.html.
  2. Rename the binary, and place it in the `$(SBIN)` directory.
  3. Turn off the checkpoint server that is currently running, by turning off Condor on the machine where it is running.

     ```
     condor_off
     ```

  4. Change the configuration variable that specifies the path and name of the checkpoint server to the renamed 6.7.19 *condor_ckpt_server* binary. For example

```
CKPT_SERVER = $(SBIN)/ckpt_server.v6719
```

5. Reconfigure, and turn Condor back on.

```
condor_reconfig -master
condor_on
```

6. Check the checkpoint server log to verify that the correct 6.7.19 *condor_ckpt_server* binary is running.

- Condor is no longer available for HPUX 10.20 on Hewlett Packard PA-RISC or Red Hat Linux on ALPHA.
- The **globus** universe and the follow commands that have been used in submit description files are retired. Please remove from submit description files: **globusscheduler**, **grid_type**, **jobmanager_type**, **nordugrid_resource**, **remote_pool**, **remote_schedd**, **unicore_u_site**, and **unicore_v_site**. For the newer syntax of **grid** universe jobs, please see section 5.3 on the **grid** universe, as well as the *condor_submit* manual page on page 716.
- The utilities *condor_store_cred*, *condor_get_cred*, *condor_list_cred*, and *condor_rm_cred* for dealing with credentials for Stork on UNIX have been renamed to *stork_store_cred*, *stork_get_cred*, *stork_list_cred*, and *stork_rm_cred*, respectively. This is because the Windows *condor_store_cred* tool, which can be used to set the shared secret for the password authentication method, is now present on all platforms.

New Features:

- A *condor_schedd* can now submit jobs directly to a local PBS or LSF installation. To do this, submit the job with a **universe** of **grid** and a **grid_resource** of **pbs** or **lsf**.
- Implemented all of the functions from new classads into the ”old” classads now in Condor.
- Condor’s format for storing the history file has been improved so that some queries will now go much faster. In particular, *condor_history* now accepts the **-backwards** option, which will take advantage of this change. Queries that only reference the job’s cluster id and proc id will be able to take advantage of this speed increase, and in the near future, more fast queries will be supported. You need to make no changes in order to deal with this new history file format, unless you want to be able to search your entire history file backwards, in which case you should run the new *condor_convert_history* program.
- Condor can now delegate a job’s GSI X509 credentials when transferring them over the wire, instead of copying them. This is much more secure when communications are not encrypted. As this can be a major performance hit when submitting large numbers of jobs remotely, the old behavior can be forced by setting `DELEGATE_JOB_GSI_CREDENTIALS` to False in the configuration file.

- Added configuration parameter `NO_DNS` , which allows Condor to work on machines with no DNS. When this option is set to True, Condor will use pseudo-hostnames constructed from a machine's IP address and `DEFAULT_DOMAIN_NAME` , rather than attempting to resolve hostnames into IP addresses and vice-versa.
- The `JobLeaseDuration` now defaults to 20 minutes for all jobs that support this feature (everything except standard and PVM universe, and jobs that request streaming I/O). This way, by default, if the submit host crashes or there is a short network outage, the *condor_schedd* will be able to reconnect to jobs that were executing at the time of problem.
- Condor daemons now touch their daemon log file periodically. When a daemon starts up, it prints to the log the last time the log file was modified. This lets an admin estimate when a daemon stopped running. The configuration parameter `TOUCH_LOG_INTERVAL` sets the time between touches (in seconds) and defaults to 60 seconds.
- Added the ability to pass a specific *condor_config_val* program to the cron/Hawkeye "modules". If "HAWKEYE_CONFIG_VAL" is specified in the configuration, an environment variable with the same name and the same value will be added to all cron job environments. This change has no effect if the above macro is not specified in the configuration. The above name "HAWKEYE_CONFIG_VAL" is derived from the cron name (i.e. STARTD_CRON_NAME or SCHEDD_CRON_NAME).
- *condor_submit_dag* now generates a submit file with `copy_to_spool` set to `false`. This reduces the load and saves file space on the submit machine, especially if you are running multiple instances of *condor_dagman*.
- The authorization levels in Condor's security system now form a hierarchy. A client with DAEMON or ADMINISTRATOR access also have WRITE access. A client with WRITE, NEGOTIATOR, or CONFIG access also have READ access.
- Added configuration parameter `GRIDMANAGER_EMPTY_RESOURCE_DELAY` , which sets how long the *condor_gridmanager* retains information about a grid resource after it has no active jobs to that resource.
- Added configuration parameter `JOB_PROXY_OVERRIDE_FILE` , which lets an admin force a particular X509 proxy to be used for all grid universe jobs, overriding whatever proxy may be specified in the job ad.
- *condor_dagman* no longer uses the `popen()` system call when running commands; this provides better security and allows it to run on Windows without being a service.
- Added a new version of DRMAA which includes fixes and updates per DRMAA spec finalization.
- For grid-type gt4 jobs, the resource lifetime on the remote server will be based on `job_lease_duration`, if it's set.
- Improved the error message in *condor_dagman* for `pclose()` failures after submitting a node job.

- Added new job attribute `GridResourceUnavailableTime`, which is equivalent to `GlobusResourceUnavailableTime`, but is used for all grid universe jobs. One benefit of this new attribute is that grid resource up/down user log events are logged correctly when the gridmanager crashes and restarts.
- Added the ability to set `GROUP_AUTOREGROUP` on a per-group basis, using the syntax GROUP_AUTOREGROUP_¡groupname¿ = True/False.
- Added configuration variable `SYSAPI_GET_LOADAVG` to control if Condor should attempt to fetch the system load average. See section 3.3.3.
- Added configuration variable `SCHEDD_ROUND_ATTR_<XXXX>`. See description in section 3.3.11 on page 186.
- The password authentication method now works on all platforms. It was previously only available on Windows. UNIX platforms will store the pool password in the file defined by the configuration parameter `SEC_PASSWORD_FILE` . This file will be owned by the real UID that Condor runs as and will only be accessible by that user.
- A new tool, *condor_userlog_job_counter*, has been added. Given a userlog file as an argument, it determines the number of queued (e.g., submitted but not yet terminated or aborted) jobs recorded in that userlog, and returns that value as an exit code. It returns 255 if there are more than 254 queued jobs or to indicate an error (e.g., a userlog reading/parsing error, no events found, a job count ¡0 or ¿254, improper usage, etc.).
- The *condor_chirp* tool has been added to the Windows distribution.
- The environment variable `X509_USER_PROXY` is now set to the full path of the proxy for scheduler universe jobs if a proxy is associated with the job.

Bugs Fixed:

- Fixed a bug where *condor_q* would exit with a non-zero exit status even though it found and displayed the requested information or job queue.
- Fixed a bug in the dedicated scheduler where parallel and mpi jobs with more than one proc in a cluster would only have the Scheduler attribute set in the first proc.
- Fixed a bug in the *condor_collector* that could cause it to crash if it's configured as a view collector (i.e. `KEEP_POOL_HISTORY` is TRUE). In particular, machine ads with a State value of "Backfill" could trigger this crash.
- Fixed a related bug in *condor_stats* that could cause a crash when encountering a machine state of "Backfill".
- Disconnected starter-shadow connections (job leases) now work for flocked jobs.
- Fixed numeric value wrap-around bug for the totals in *condor_status*.

- Grid universe jobs sent to Globus Toolkit 2 resources now generate an evict user log event when the job transitions from Running to Idle, along with another execute even when the job restarts. Previously no events were logged in these cases, leading to the potentially confusing situation where a job would be Idle in the queue, but the last job log entry would indicate that the job was Running.
- All grid universe jobs now properly handle the new job attributes `Arguments`, `Environment`, and `TransferOutputRemaps`.
- In some cases, a restart of Condor was required to properly handle a change in the undocumented configuration parameter `SIGNIFICANT_ATTRIBUTES`. Now, a *condor_reconfig* is sufficient.
- Fixed a permissions problem that would cause automatic X509 proxy renewal for vanilla universe jobs to fail.
- Fixed a bug introduced in 6.7.17 that caused the configuration parameter `ENABLE_GRID_MONITOR` to be ignored. The value would always be considered true.
- Improved fault recovery of gt2 grid jobs. This includes a work-around for Globus bugzilla ticket 871.
- When the *condor_gridmanager* cancels a job after `GlobusResubmit` evaluates to true, it will no longer put the job on hold if the cancel fails.
- Fixed the default COLLECTOR_QUERY_WORKERS entry in the example central manager config_config; due to a cut and paste error it was COLLECTOR_CLASS_HISTORY_SIZE.
- In some cases, `CondorLoadAvg` was reporting a different result, depending on the setting of `NUM_CPUS`, even with everything else, such as the actual number of cpus, being the same. The specific case in which this effect was noticeable was when the machine load was greater than `NUM_CPUS`. `CondorLoadAvg` is now independent of the setting of `NUM_CPUS`.
- When the Grid Monitor encounters problems, Condor will now try to restart the Globus JobManagers for the affect grid universe jobs, limited by `GRIDMANAGER_MAX_JOBMANAGERS_PER_RESOURCE` . The previous behavior caused problems with sites that don't have a fork JobManager, and Condor wouldn't react when a job's proxy expired.
- Fixed a bug that could cause extra Grid Monitor file to accumulate under /tmp until the *condor_gridmanager* exited.
- Fixed a problem in which preempting claims waiting on retiring jobs (i.e. waiting on MaxJobRetirementTime) could get preempted without sufficient rank or priority (because the new preemption only had to beat the retiring job, not the preempting claim). Furthermore, both the new preempting claim and the original preempting claim had the same claim id, so they collided in a way that ultimately caused both to be removed, and the respective jobs would go back into unmatched state. The result was unnecessary negotiation churn and slower convergence of resource usage to the desired distribution. Now, preemption of preempting claims during long job retirement is correctly handled.

- Fixed a bug that caused the shadow to transfer a job's files twice to the starter if the files were stored in Condor's spool directory.
- On some systems, when Condor starts a gridftp server for gt4 grid jobs, all transfers to or from the server will fail if it's not told where its executable is located (using '-exec' on the command line). Condor now gives this option to the gridftp server.
- Fixed a bug in *condor_dagman* that could cause DAGMan to crash if all submits fail for several nodes that have POST scripts. This bug existed in versions 6.7.17 and 6.7.18.
- Fixed a bug in how Condor determines the version of a Condor executable. This was preventing the grid universe from working on Tru64 5.1 on Alpha.
- Fixed a bug in the *condor_gridmanager* that could cause gt2 grid jobs with an invalid proxy to become stuck (the *condor_gridmanager* would do nothing with the jobs and not acknowledge a hold or removal).
- Fixed a bug on Win32 that caused a failure when sending a WM_CLOSE message to a job when the Condor daemons are running as a normal user (i.e. not running as LocalSystem). Also, fixed a thread handle leak when sending a WM_CLOSE message.
- On Win32, fixed a bug that would cause the *condor_master* to exit upon a *condor_restart* command when started as a service with a service name other than "condor" (the default name used by the installer).
- Fixed a bug that could cause the *condor_master* to crash when sending a shutdown fast to a child process after the `SHUTDOWN_GRACEFUL_TIMEOUT` timeout expired.
- Fixed a bug with automatically setting the undocumented `SIGNIFICANT_ATTRIBUTES` configuration parameter in order to speed up negotiation— previously, the job's Requirements expression was not correctly considered. With certain scheduling policy expressions, this bug could have resulted in jobs staying idle in the queue when they should have been launched.
- It used to be impossible to use the `SUBMIT_EXPRS` configuration setting to provide default values for job submit file keywords that were recognized by *condor_submit*. For example, administrators could define a default value for a custom job attribute, but not something like `Notification` or `WantRemoteIO`. Now, administrators can use `SUBMIT_EXPRS` for any settings, whether they are regular *condor_submit* keywords or custom job attributes.
- Fixed a bug in the *condor_startd* that could cause resources to get stuck in the Backill/Killing state if both the `START_BACKFILL` and `EVICT_BACKFILL` expressions evaluated to TRUE at the same time.

Known Bugs:

- None.

### Version 6.7.18

Release Notes:

- A security team at UW-Madison is conducting an ongoing security audit of the Condor system and has identified a few important vulnerabilities. Condor versions 6.6.11 and 6.7.18 fix these security problems and other bugs. There have been no reported exploits, but all sites are urged to upgrade immediately.

  The Condor Team will publish detailed reports of these vulnerabilities on 2006-04-24, 4 weeks from the date when the fixes were first released (2006-03-27). This will allow all sites time to upgrade before enough information to exploit these bugs is widely available.

- The **-flock** option in *condor_cold_start* and *condor_cold_stop* has been replaced by **-filelock** to avoid any confusion between file locking and Condor job flocking.

- As of 6.7.17, Quill's database schema has been slightly altered. For more information, please see the corresponding 6.7.17 version history entry in section 8.4 on page 529.

Security Bugs Fixed:

- Bugs in previous versions of Condor could allow any user who can submit jobs on a machine to gain access to the "condor" account (or whatever non-privileged user the Condor daemons are running as). This bug can not be exploited remotely, only by users already logged onto a submit machine in the Condor pool.

- The security of the "condor_config_val -set" feature was found to be insufficient, so this feature is now disabled by default. There are new configuration settings to enable this feature in a secure manner. Please read the descriptions of `ENABLE_RUNTIME_CONFIG` , `ENABLE_PERSISTENT_CONFIG` and `PERSISTENT_CONFIG_DIR` in the example configuration file shipped with the latest Condor releases, or in section 3.3.5 on page 155.

New Features:

- Added a new `LOCAL_CONFIG_DIR` configuration setting. This now allows entire directories of files to be included as though they were configuration files. See 3.3.3 for more info.

- You can now put extra information into the notification email. The information is a list of attributes, which you provide. For example, if your submit file has "+EmailAttributes = "RemoteHost, Requirements"", then RemoteHost and Requirements will be listed in the notification email.

- Added a new clipped port of Condor to HP-UX 11 running on the HP-PA architecture.

- Condor is now much better at recognizing when a grid-type gt2 grid universe job failure is unrecoverable and at cleaning up failed or canceled job submissions. This should reduce the number of jobs that perpetually return to held state when released.

- When job attribute `GlobusResubmit` evaluates to true for grid-type gt2 jobs, the *condor_gridmanager* will try to cancel the existing job before starting the new submission. If the cancel attempt fails, the *condor_gridmanager* will proceed with the new submission anyway.
- When `BIND_ALL_INTERFACES` is enabled, Condor daemons now advertise their IP address as that of the network interface used to contact the collector. This makes it possible, for example, to have a schedd on a multi-homed machine flock jobs to Condor pools in two separate networks, because the schedd can advertise a different IP address to the two collectors. *condor_cod* also benefits in the case where the startd is reachable through a network interface other than the default one that would normally be advertised. This change also produces improved default behavior in cases such as *condor_glidein* where the startd lands on a dual-homed machine with both public and private IP addresses.
- In *condor_dagman*, the informational messages about hitting the **-maxidle**, **-maxjobs**, **-maxpre**, and **-maxpost** limits are no longer printed to the `dagman.out` file by default. To see these messages, add **-debug 4** to the *condor_submit_dag* command line. A summary of the total number of job and script deferrals is now printed by default each time the node status is printed and at the end of the `dagman.out` file. This can be turned off by setting the debug level to 2 or lower on the *condor_submit_dag* command line.
- Added support for a new configuration setting, `STARTD_RESOURCE_PREFIX` . For more information, see section 3.3.10 on page 174.
- The number of CPUs Condor detects may now have an upper bound. The `MAX_NUM_CPUS` configuration setting controls this.
- When preempting a claim, the *condor_negotiator* now prints the startd rank of the job that is being preempted and the startd rank of the job it is causing the preemption.
- Improved the error messages from *condor_check_userlogs*, especially if it fails because it doesn't have write permission on the log files (unfortunately, the log reading code requires write locks to avoid collisions between multiple readers and writers).
- Improved error messages in *condor_dagman* when `getcwd()` fails (this is only relevant if the **-UseDagDir** flag is used).
- Added QUILL_MANAGE_VACUUM to determine whether Quill needs to perform vacuuming tasks or not. In the latter case, vacuuming tasks can be automatically managed by *PostgreSQL* version 8.1 onwards. Please see Quill's section in the Administrator's Manual for more details.
- The Grid Monitor now works at sites where /etc/grid-security/certificates is out of date, but $(GLOBUS_LOCATION)/share/certificates is not.
- A new authentication method, PASSWORD, has been added; it provides mutual authentication between a client and server using a shared secret. Password authentication currently only works on Windows, and only for daemon-to-daemon communication.

- Condor on Windows now supports running jobs as the submitting user. This feature requires the use of a central daemon for storing users' passwords (the *condor_credd*). See the example configuration file `condor_config.local.credd` included with the Condor distribution for more information.
- Added a **-n** option to *condor_store_cred* to allow for storing a password to a remote host.
- Support for DRMAA on Windows has been added.
- Kerberos support has been upgraded to use version 1.4.3 of the Kerberos library. This adds support for Kerberos as an authentication method on Windows.
- Added the new *condor_replication* daemon which works with *condor_had* to enable replication of data for daemons configured for high availability. In particular, *condor_replication* can be configured to replicate the accountant log so the a fail-over *condor_negotiator* can share the user priority state from the primary *condor_negotiator*.
- The *condor_collector* now has the ability to receive ClassAds via it's SOAP interface.

Bugs Fixed:

- Fixed a memory corruption bug in *condor_quill* where it could miscalculate the hostname of the db server to which it connects.
- Fixed a security hole in *condor_quill* where the daemon would emit the quillwriter user's password in cleartext into the *condor_quill* logfile.
- Fixed a bug in 6.7.17 that could cause the schedd state to be wiped out, clearing the contents of the job queue. The most likely case in which this problem could have happened is when the disk containing the spool directory became full and the schedd restarted several times due to failures writing to job_queue.log. The problem no longer exists in 6.7.18, but for users who cannot upgrade immediately, the workaround to prevent the bug from ever happening is to add the following line to the configuration file:

  ```
  MAX_JOB_QUEUE_LOG_ROTATIONS = 0
  ```

- Fixed a bug that could have caused corruption of the job queue log file in very rare circumstances involving a full disk. This potential problem existed in all previous versions of Condor.
- Fixed a problem with parallel universe jobs with multiple procs (i.e. multiple queue statements in one submit file). Before, in such a case, the user log would have multiple submit events per cluster but only one terminate event. This caused confusion for dagman. Now there is one submit event and one terminate event for such parallel universe jobs.
- A ClassAd bug that has existed since the Condor 6.3 series has been fixed, and it might affect your pool. In a ClassAd, MY and TARGET are supposed to narrow the scope for looking up a ClassAd variable. For instance, in a job's requirements, MY refers to the job's attributes, and TARGET refers to the machine's attributes. Unfortunately, since 6.3 MY and TARGET

actually made a search order, not a scope restriction. That is, if a job's requirements had TARGET.foo and foo was undefined in the machine ad, it would look in the job ad for the value instead of deciding that foo was undefined. This is now fixed. However, there is a chance that users have made Classad expressions that confused MY and TARGET but worked. With this bug fix, they might not work anymore. We expect this bug fix to affect few users, but it may be tricky to understand for those of you that it affects. We needed to make this bug fix because the bug caused problems for some users that could not be worked around.

- Fixed a vague error message for the standard universe as to now emit reason for failure when reading a checkpoint image.
- Fixed a bug which was causing erroneous load average numbers for the AIX port of Condor.
- Fixed a bug which caused the update proxy command to the *condor_schedd* to fail if the job was running and Condor was started as root.
- Fixed a bug which was causing jobs to never leave the "run" state if the *condor_schedd*'s cron/hawkeye feature is enabled. This bug was introduced with the addition of the cron logic to the *condor_schedd* in 6.7.8.
- Improved how the *condor_gridmanager* reacts to proxy delegation commands failing for grid-type gt4 jobs. Before, it could end up retrying the commands every couple seconds. Now, it retries them every 5 minutes.
- Fixed Condor's code for automatically starting a gridftp server for grid-type gt4 jobs to work when Condor is started as root.
- Scheduler universe jobs no longer inherit the environment of the *condor_schedd*.
- Fixed a bug causing jobs to fail to run when submitted from a 6.7.15+ *condor_schedd* to an older *condor_starter*. This problem only affected jobs with no argument specification in the submit file or jobs with arguments specified in the new syntax (surrounded by double quotes).
- The *condor_c-gahp* now ensures that arguments and environment in the job ClassAd are converted to a syntax understood by the target schedd. Previously (starting with 6.7.15), jobs with empty arguments/environment, or jobs using the new syntax for these would fail to run when submitted as Condor-C jobs targeting a pre 6.7.15 schedd.
- When running grid-type gt4 jobs with an automatically-started gridftp server, a restart of Condor could cause all of the gt4 jobs to be canceled and resubmitted due to the gridftp server's port changing. Now, the old port will be reused when possible.
- Fixed a bug that could cause the *condor_gridmanager* to crash when running grid-type gt4 jobs with an automatically-started gridftp server.
- Fixed a bug that caused the *condor_c-gahp* to exit if a file transfer failed.
- When a grid-type condor job is removed, any active file transfer for the job is aborted. Previously, the transfer would be allowed to complete before the job was canceled.
- Fixed bug where FileLock.pm was not included in release.

- Clusters of jobs using transfer-file mode and output or error files containing path information and references to `$(Process)` or `$(Cluster)` were incorrectly storing the output files in the initial working directory rather than the specified path. This happened for all jobs in the cluster except for the first job (process 0). This bug was introduced in 6.7.13.
- Previously, setting the default job environment within `SUBMIT_EXPRS` did not work, because *condor_submit* would always override this default with an empty environment.
- Since 6.7.15, Condor-C has incorrectly handled the use of both **remote_env** and **remote_args**. The normal **environment** and **arguments** commands were honored, but the 'remote' versions were ignored unless the corresponding 'normal' command happened to be set to a double-quoted value (i.e. the new syntax for these settings). Now that this problem is fixed, when **remote_env** or **remote_args** is specified, it correctly sets the respective property of the job ClassAd in the remote schedd. It is still preferable to use the **environment** and **arguments** commands instead of the setting remote attributed directly, because then you can use either the new double-quoted environment/argument syntax or the old one, and *condor_submit* will automatically set the correct ClassAd attributes.
- Fixed a bug in the **-l** option to *condor_q* which, when querying Quill, used to display the attributes of the last cluster in every job ad even though they were submitted as part of different clusters.
- Fixed a bug in the Quill daemon which used to incorrectly parse classad attribute values of the form "number and some stuff" (e.g. attribute=Rank and value=1000 * memory).
- Added HAD to the default DC_DAEMON_LIST, both in the default condor_config and in the default list hard coded into *condor_master*.
- Java universe jobs submitted with the old-style arguments syntax (argument string not surrounded by double quotes in the submit file) would fail to run (and therefore stay in the schedd job queue) if the path to Condor's execute directory contained a space (e.g. "Program Files"). This bug was introduced in 6.7.15.
- Fixed a problem introduced in 6.7.17 where repeated "ProcAPI sanity failure" entries would appear in daemon logs on Windows.
- The **-submitter** option to *condor_q* has been fixed to handle submitters in accounting groups.
- The *condor_shadow* now correctly handles the case where RESERVED_SWAP is set to 0.

Known Bugs:

- None.

## Version 6.7.17

Release Notes:

- The default output for *condor_status* was changed between 6.7.16 and 6.7.17 to support the new *Backfill* state which Condor resources can now enter (described below in more detail).
- Added two new columns to the Quill database schema to support historical job queue logs (see `MAX_JOB_QUEUE_LOG_ROTATIONS` in the New Features section below). These are log_seq_num and log_creation_time. For a description of those two columns, check out the schema of the JobQueuePollingInfo table in section 3.11.3 on page 335.

  Databases created by versions of Quill prior to 6.7.17 must be updated to reflect these two new columns. This can be achieved by either dropping the database and letting Quill recreate it on the next polling cycle, or by manually adding the two columns and initializing their values via the following sql commands:

  ```
  alter table jobqueuepollinginfo add column log_seq_num bigint;
  alter table jobqueuepollinginfo add column log_creation_time bigint;
  update jobqueuepollinginfo set log_seq_num = 0, log_creation_time=0;
  ```

  If the schema is being manually changed, it must be done so **before** the *condor_quill* daemon is started.

New Features:

- Added support for Condor resources to perform backfill computations when there are no Condor jobs to run. Condor can be configured such that whenever a machine is in the Unclaimed/Idle state and otherwise has nothing else to do, the *condor_startd* will automatically spawn backfill jobs to continue to perform useful work. Currently, Condor only supports using the Berkeley Open Infrastructure for Network Computing (BOINC) to provide the backfill jobs (see http://boinc.berkeley.edu for more information about BOINC). See section 3.12.9 on page 359 for more information about running backfill jobs with Condor. At this time, backfill jobs are not supported on windows machines.
- The history file, which is a flat file for each submitting computer that stores information about all jobs completed on that computer is now rotated automatically. By default, the file will be rotated when it is more than 20MB and two backup files will be allowed (for a total of three history files with 60MB of data). This means that older history will be lost once it is rotated out. You can disable the history file rotation if you like, and you can change the number and size of the backup files. *condor_history* has been updated to understand these backup history files.
- Added parallel universe support to *condor_dagman* (*condor_dagman* can now handle submit files that submit more than one Condor job proc).
- Added a **-format** option to the *condor_history* command which behaves just like the -format option to *condor_status* and *condor_q* commands.
- Added remove and get_job_attr options to the *condor_chirp* command line tool. Changed parallel universe script to use them.

- When the Grid Monitor encounters problems, Condor no longer tries to restart the Globus JobManagers for all of the affected grid universe jobs. Restarting the JobManagers can easily bring down a remote headnode. Condor will attempt to restart the Grid Monitor, but there will be no update of job status in the mean time.
- When started as root on a Linux 32-bit x86 machine, Condor daemons will leave core files in the log directory when they crash. Recent changes to the Linux kernel default to blocking these core files. This change means Condor behaves more consistently across different Unix-like operating systems.
- Made several changes to make Condor-G much less likely to overload a pre-WS GRAM server for grid-type gt2 jobs. Added configuration parameter GRIDMANAGER_MAX_JOBMANAGERS_PER_RESOURCE, which limits the number of globus-job-manager processes Condor will let run on the server at a time. Streaming of output for gt2 jobs is disabled if GRIDMANAGER_MAX_JOBMANAGERS_PER_RESOURCE isn't set to unlimited. If the Grid Monitor encounters problems, the *condor_gridmanager* doesn't restart the globus-job-managers of the affected jobs. Fixed a couple bugs in the Grid Monitor that could cause it to spawn extra polling processes on the server.
- Added support for Parallel scheduling groups for the parallel universe. This is useful if you have machines connected by InfiniBand switches, and want to constrain your parallel jobs to never run across two different switches.
- Added a new suite of tools to dynamically deploy Condor. The most important of these tools are *condor_cold_start* and *condor_cold_stop*. Another significant subset of this suite are tools to determine whether a process is alive or dead. The most advanced of which are the *uniq_pid_midwife* and *uniq_pid_undertaker*. Currently these programs are only supported on Linux.
- Added `MAX_JOB_QUEUE_LOG_ROTATIONS` to control how many historical job queue logs are kept when the job queue log is rotated. These historical logs are used by Quill to avoid missing information in Quill's job history information when the schedd rotates to a new log. The default value for this configuration setting is 1, so one old copy of the job queue log file will be kept.
- Added support for DRMAA on the Mac OSX platform.
- Enabled `COLLECTOR_QUERY_WORKERS` in the default *condor_collector* configuration, and set this value to 16. This replaces the previous implicit default of 0 and will result in a more responsive *condor_collector* in the common case. Note this `COLLECTOR_QUERY_WORKERS` has no effect on non-UNIX systems (Windows).
- `HIGHPORT` and `LOWPORT` can now specify ports below 1024 when Condor is started as root on Unix systems. This always worked on Windows.
- It is now possible to specify separate port ranges for binding incoming (listen) sockets and outgoing (connect) sockets by using `IN_LOWPORT` /`IN_HIGHPORT` and `OUT_LOWPORT` /`OUT_HIGHPORT` . if not present, we still fall back to the regular `LOWPORT` /`HIGHPORT` settings.

- Port ranges from `LOWPORT` , `HIGHPORT` , `IN_LOWPORT` , `IN_HIGHPORT` , `OUT_LOWPORT` , and `OUT_HIGHPORT` are now passed to Globus through the correct environment variables.

Bugs Fixed:

- Previously, the *condor_startd* would not recompute the `CurrentRank` attribute each time a new job was spawned, but only computed it whenever a new claim was made. Now, the *condor_startd* correctly recomputes `CurrentRank` each time a new job starts running.
- When running a gridftp server for grid-type gt4 jobs, Condor will now start the server so as to ignore /etc/grid-security/gridftp.conf and $GLOBUS_LOCATION/etc/gridftp.conf. These files may contain options that would cause the gridftp server to fail when not run as root. Also, Condor's gridftp server is started to ensure that it does not erroneously try to load libraries from an existing Globus installation, causing the gridftp server to crash.
- Fixed a bug where jobs using the grid (or globus) universe that specified an AccountingGroup would never run because the *condor_gridmanager* would fail to start.
- Fixed a bug introduced in 6.7.14 where the job attributes RemoteUserCpu and RemoteSysCpu were incorrectly reported as 0 in the history file and the job queue for non-standard universe jobs.
- Fixed a physical memory reporting bug for the Mac OSX port of Condor.
- Since the addition of the "new" cron syntax (introduced in version 6.7.11), the *condor_startd* has (silently) ignored any jobs defined with the "old" syntax if any jobs are defined with the "new" syntax. Now, the *condor_startd* will honor both definitions, but will log a warning to it's log file if any jobs with the "old" syntax are found (whether or not any new jobs are found). The *condor_schedd* (which also has the "cron" logic) will behave in the same way.
- The bug which was causing the "Cron" job command lines to have the name added each invocation has been fixed.
- Fixed some messages about keyboard and mouse idle time had been logged too often in the *condor_startd* logs under certain conditions to be logged less often.
- Fixed the **-dag** option to *condor_q*. Previously, this did not print DAG node names as it should have. (This bug has existed since approximately v6.7.11.)
- Fixed a bug that could cause the *condor_gridmanager* to crash if the GridJobId attribute for a gt2 job became mangled. The cause of mangling seen by some users is still unknown.
- Submission from 6.7.15 or 6.7.16 *condor_submit* to a 6.7.14 or earlier *condor_schedd* was not working unless the submit file explicitly set both arguments and environment using the old syntax. Now *condor_submit* automatically converts the environment and argument syntax when necessary. If the conversion is not possible, due to limitations in the old syntax, *condor_submit* will generate an error message and refuse to complete the submission.
- *condor_submit* now returns an error if the executable file specified in the submit file exists but is zero length.

Known Bugs:

- RPM packages of Condor may refuse to install because of a failed dependency on perl(FileLock). The module in question is missing from the bundles. As a workaround, use rpm's –nodeps option to ignore the requirement. (Bug introduced in version 6.7.17)
- The new dynamic deployment tools (*condor_cold_start* and others) may fail because FileLock.pm is missing. If you would like to use the new dynamic deployments tools, contact condor-admin@cs.wisc.edu to receive a copy of FileLock.pm. (Bug introduced in version 6.7.17)
- Jobs with no arguments submitting using Condor versions 6.7.15 up to and including 6.7.17 that try to run on a pre-6.7.15 *condor_starter* will fail to start. The *condor_starter* will fail and exit. The job will not run until matched with a *condor_starter* from 6.7.15 or later. The workaround is to always specify arguments in the submit file using the old syntax. You must specify the arguments, even if they are empty. For example: "argument=". Existing jobs in the queue can be modified with *condor_qedit*. For example: "*condor_qedit* ¡jobid¿ Args '""""'. Jobs submitted prior to upgrading to 6.7.15 or later are not affected. (Bug introduced in version 6.7.15)
- Enabling the cron/Hawkeye feature of the *condor_schedd* causes jobs to never leave the "run" state. This bug was introduced with the addition of the cron logic to the *condor_schedd* in version 6.7.8. This functionality is not enabled by default, so most users will not encounter it. This does not affect the cron/Hawkeye feature in *condor_startd*. (Bug introduced in version 6.7.8)
- Multi-cluster condor submits will cause *condor_dagman* to hang. This bug was introduced by the implementation of parallel universe support. Prior to version 6.7.17, any Condor submit file creating more than one Condor process would be treated as an error by *condor_dagman*. Now this is not the case, because a single cluster with multiple processes will work; but *condor_dagman* does not deal properly with multi-cluster submits (e.g., a submit file queuing jobs with different executables). As a workaround, take care that submit files submitted by *condor_dagman* only submit multiple processes, not multiple clusters. (Bug introduced in version 6.7.17.)
- The **-maxjobs** and **-maxidle** settings for *condor_dagman* are inconsistent: maxjobs applies to job clusters, but maxidle applies to individual processes. Note that this only makes any difference in the case of node submit files that queue more than one process, which has only been supported since version 6.7.17. (Bug introduced in version 6.7.17.)
- *condor_dagman* does not properly handle failures when removing jobs for a failed node. Note that this only makes any difference in the case of node submit files that queue more than one process, which has only been supported since version 6.7.17. If one process for a node fails, the entire cluster is considered failed, and any other processes in that cluster are removed. If removing the processes fails, *condor_dagman* may hang, waiting for those processes to abort. (Bug introduced in version 6.7.17.)

- The FileLock.pm perl module (written in-house) was not included in this release. As a direct result the **-flock** option of condor_cold_start will not work. This can be remedied by downloading FileLock.pm from: ftp://ftp.cs.wisc.edu/condor/temporary/filelock/FileLock.pm and installing it in the `lib` directory of your Condor installation. (Bug introduced in version 6.7.17)
- In some circumstances, the schedd state may be wiped out, clearing the contents of the job queue. The most likely case in which this problem can happen is when the disk containing the spool directory becomes full and the schedd restarts several times due to failures writing to job_queue.log. The problem no longer exists in 6.7.18, but for users who cannot upgrade immediately, the workaround to prevent the bug from ever happening is to add the following line to your config file:

  ```
  MAX_JOB_QUEUE_LOG_ROTATIONS = 0
  ```

## Version 6.7.16

Release Notes:

- None.

New Features:

- Support for running a personal Condor on Windows using *condor_master* **-f**

Bugs Fixed:

- Support for NorduGrid jobs was accidentally left out of the *condor_gridmanager* in previous releases. This has been corrected.
- The *condor_starter* was refusing to run jobs if it could not perform a reverse-DNS lookup of the submit-host. Now that this is fixed, when the reverse-DNS lookup fails, the job can still run, but Condor will not be able to verify the authenticity of the submit-host's uid domain. In this case, if you enable `TRUST_UID_DOMAIN` , everything will function as normal, minus the verification of the domain; if you do not enable `TRUST_UID_DOMAIN`, the starter will treat the job as being from a different uid domain, regardless of what uid domain the job advertises.
- Fixed a few bugs with `transfer_output_remaps` that caused files to be remapped while in a temporary sandbox. Now, the remapping occurs only when the files are returned to the job submitter.
- Fixed some minor memory leaks in the *condor_gridmanager*.

- Fixed a bug in 6.7.15 that was causing startd cron jobs to fail to run if the old-style configuration setting `STARTD_CRON_JOBS` was used instead of the new-style configuration setting `STARTD_CRON_JOBLIST` .

Known Bugs:

- The command line string used in starting *cron* jobs is correct the first time the job is run, but incorrect each subsequent time the job is run. The error is that the job's name is incorrectly appended to the previous run's command line. As an example, the first time the job is run with the correct command line

  ```
  /path/to/job jobname
  ```

  The second time, this job is run with the incorrect command line

  ```
  /path/to/job jobname jobname
  ```

  And, the third time, this job is run with the incorrect command line

  ```
  /path/to/job jobname jobname jobname
  ```

- On Windows only (as far as we know) a *condor_rm* of a scheduler universe job after a *condor_qedit* may cause the schedd to crash. (This bug has existed at least since 6.7.12.)
- On Windows only (as far as we know) a *condor_hold* followed by a *condor_release* on a job sometimes results in the job being removed instead of going into the idle state. (This bug has existed at least since 6.7.12.)
- On Windows only, *condor_dagman* may fail with a "DLL not initialized" error (exit code -1073741502). (This bug has existed at least since 6.7.12.)

## Version 6.7.15

Release Notes:

- If you have not used the undocumented configuration setting `SIGNIFICANT_ATTRIBUTES` , there is no need to read the rest of this paragraph. For sites that have been using `SIGNIFICANT_ATTRIBUTES` in the config file, we suggest removing that setting, because Condor now automatically selects the list of attributes that are used to cluster job ClassAds into distinct ads for negotiation. In 6.7.15, any setting of `SIGNIFICANT_ATTRIBUTES` will be combined with the automated list of attributes that Condor produces. In the future, this behavior may change (e.g. it might override the automated behavior rather than combining with it). If you know in advance that your use of Condor heavily depends on `SIGNIFICANT_ATTRIBUTES` *not* including some attributes that *are* used in

requirements expressions (e.g. ImageSize), then you should be aware that 6.7.15 provides *no* way for you to suppress such attributes. In that case, we recommend that you wait for this issue to be addressed before upgrading. This should not concern most users–especially anyone who is not even using SIGNIFICANT_ATTRIBUTES , or who has defined SIGNIFICANT_ATTRIBUTES to include all attributes that are used in requirements expressions (which is the normal usage case).

- Added a clipped port of Condor to YellowDog Linux 3.0 on the PowerPC architecture.
- "Cron" jobs defined with the "old" configuration syntax (usually through "STARTD_CRON_JOBS" or "HAWKEYE_CRON_JOBS" – see the *condor_startd* manual section for more details) are broken. Using the "new" syntax ("STARTD_CRON_JOBLIST") will work around this problem.

New Features:

- For those platforms which support it, libcondorapi.so is now produced and available in the lib/ directory after installing Condor.
- The negotiation protocol between the *condor_schedd* and the *condor_negotiator* daemons has been improved for both scalability and correctness. In general, most sites will see faster negotiation cycles when many jobs are submitted after upgrading both the negotiator and all schedd daemons to version 6.7.15. This means the scheduling overhead per job is reduced. If you have used the undocumented macro SIGNIFICANT_ATTRIBUTES , please read the note above in the release notes, because this new automated behavior affects the use of that configuration setting–in most cases making it unnecessary.
- Due to kernel bugs between the Linux 2.4.x and 2.6.x kernels, Condor now implements "checkpointing signatures" which allow more fine grained and automatic control over whether or not a particular machine is willing to resume a job using a previously created checkpoint. This functionality is homogenized across all platforms which provide the standard universe feature set.
- Grid matchmaking ads are now aged and replaced by the negotiator based on a configurable classad expression from the condor config file. This configuration parameter is called STARTD_AD_REEVAL_EXPR . In previous versions, this was done strictly based on the UpdateSequenceNumber field in the ad. The default value for the new parameter behaves the same as the older, hard-coded algorithm.
- Condor can now dynamically start its own gridftp server to handle file transfers for grid-type gt4 jobs. The gridftp server appears as a job in the queue and disappears when it's no longer needed.
- Automatic renewal of job proxies from a MyProxy server now works for all grid universe jobs. Before, it only worked for grid-type gt2 jobs.
- *condor_dagman* now reports to its POST scripts uniquely distinguishable return codes for non-exe job failures (e.g., *condor_dagman*, batch-system, or other external errors such as failed

batch job submission, or batch job removal). In the past these errors were reported as various signals (e.g., SIGABRT for job removal or SIGUSR1 for failed job submission), making it impossible to distinguish them from the real signals as which they were masquerading. We now represent these errors using the previously-unused return-code space below -64 (we start below -1000, in fact). As before, 0-255 reflect normal exe return codes, and -1 to -64 represent signals 1 to 64 – but now -1000 and below represent DAGMan, batch-system, or other external errors.

- Added the `DAGMAN_RETRY_NODE_FIRST` configuration macro to *condor_dagman* to control whether failed nodes are retried before or after other ready nodes. The default is FALSE (*condor_dagman*'s previous behavior), which means that failed nodes will be retried after other ready nodes.
- Added a new (backward compatible) syntax for job arguments and environment, allowing special characters to be escaped in a uniform way. The old limit of 4096 characters in the job arguments has also been removed. See *condor_submit* manual for details of the new syntax.
- Added more configuration parameters to the *condor_master*'s restart / backoff mechanism. You can now configure the initial value of the backoff time (via `MASTER_BACKOFF_CONSTANT` ). Additionally, you can now set daemon specific values for all of these parameters. See the *condor_master* entry in the manual for more details.
- *condor_userprio* now supports **-setaccum -setbegin -setlast** options to set the Accumulated Usage, Begin Usage Time, and Last Usage time of a submitter. This is in addition to the existing **-setprio** and **-setfactor** options. These options can be used to safely reconstruct priority information if the only backup data available is the output from *condor_userprio* **-l**
- An updated DRMAA version is available on supported platforms. The previous DRMAA implementation has been removed.
- Added new per-job Stork user logs. Stork user logs are now optional, and specified in the job submit file. Stork now uses Condor user log output format, including optional XML format. Previous, per-server Stork user log in `LOG/Stork.user_log` is now deprecated, and will be removed in a future release.
- *condor_dagman* now supports the new, per-job Stork user logs. "Old-style" Stork logs (specified with **-Storklog** on the *condor_submit_dag* command line) are supported for now, but this support will probably be eliminated in the 6.7.16 release.
- Added new per-job Stork input, output and error output file specifications. Stork job output is now optional, and specified in the job submit file. Previous, per-server Stork user log in `LOG/Stork-module.stderr` and `LOG/Stork-module.stdout` has been removed.
- The Condor installer for Windows is now MSI compliant.

Bugs Fixed:

- Fixed a bug introduced in Condor 6.7.14 that caused the GT2 GAHP server to ignore configuration parameters `LOWPORT` and `HIGHPORT` and the GT4 GAHP to fail at startup.

- *condor_status* **-any** now reports quill ads when quill is enabled.
- *condor_restart* **-peaceful** was causing *condor_master* to only do a graceful shutdown, rather than a peaceful one. This means that `GRACEFUL_SHUTDOWN_TIMEOUT` would come into effect if jobs running under the startd took too long to finish. However, **-peaceful** restart did work in the case where a specific subsystem (e.g. **-startd**) was specified.
- When run from a privileged (root) Stork server, modules lose `LD_LIBRARY_PATH` and other key environments, for security reasons. This is not actually a Stork bug, but a feature of glibc. When run with a dynamically linked `globus-url-copy`, the contributed modules for the HTTP, FTP and GSIFTP transfer protocols will fail. To compensate, these modules can now restore their environment via the pre-existing `STORK_ENVIRONMENT` configuration macro. Unprivileged (user level) Storks are not affected by this behavior.
- Jobs that are are placed on held because of `on_exit_hold` evaluated to TRUE or jobs that stay in the queue after finishing because `on_exit_remove` evaluated to FALSE again correctly report the expression as being a ”job attribute”, not ”UNKNOWN (never set)”.
- *condor_glidein* was creating a default configuration with `UPDATE_INTERVAL` =20, which causes unnecessary scaling problems in large glidein pools. It now simply leaves this value undefined so that the default behavior may be assumed.
- Fixed a bug that could cause the *condor_gridmanager* to crash when a grid-type condor grid universe job left the queue.
- When using job leases with the condor grid-type, a completed job will now leave the remote *condor_schedd*’s queue when the lease expires.
- Fixed a bug in the `fullpath()` function that tests whether a file path is a full path – paths of the form `"c:/"` were not recognized as full paths, which could lead to something being prepended to what was already a full path, thereby creating an invalid path.
- Fixed a problem with `WhenToTransferOutput`=ALWAYS. The bug affected jobs that were evicted after producing one or more intermediate files that were removed by the job before finally running to completion in a subsequent run. Condor was treating the missing intermediate files as an error and the job would typically keep running and failing until the user intervened. In addition to fixing this bug, file transfer error messages are now propagated back to the shadow log and the user log, making it easier to debug problems related to file-transfers.
- *condor_submit* was not paying attention to `transfer_output_remaps` when doing permissions checks on output files.

Known Bugs:

- The command line string used in starting *cron* jobs is correct the first time the job is run, but incorrect each subsequent time the job is run. The error is that the job’s name is incorrectly appended to the previous run’s command line. As an example, the first time the job is run with the correct command line

```
/path/to/job jobname
```

The second time, this job is run with the incorrect command line

```
/path/to/job jobname jobname
```

And, the third time, this job is run with the incorrect command line

```
/path/to/job jobname jobname jobname
```

### Version 6.7.14

Release Notes:

- None.

New Features:

- The Condor grid universe can now be used to submit jobs to Nordugrid and Unicore resources.
- The Condor daemons now automatically restart when the system clock jumps more than 20 minutes in either direction. This may happen if the machine running Condor entered a ”sleep” state. This resolves a variety of minor problems.
- Added a *-direct* debugging option to *condor_q* which, when using or querying a quill installation, allows talking directly to the rdbms, the quill daemon, or the schedd without performing the queue location discovery algorithm.
- *condor_schedd* provides more flexibility in how local and scheduler universe jobs are started. The new configuration macros `START_LOCAL_UNIVERSE` and `START_SCHEDULER_UNIVERSE` allow administrators to control whether *condor_schedd* will start an idle local or scheduler universe job. If a job's respective universe macro evaluates to true, *condor_schedd* will then evaluate the `Requirements` expression for the job. Only if both conditions are met will a job be allowed to begin execution.
- *condor_schedd* advertises how many local and scheduler universe jobs are currently running or idle in its ClassAd. The total number of running jobs is denoted by the `TotalLocalJobsRunning` and `TotalSchedulerJobsRunning` attributes. The total number of idle jobs is denoted by the `TotalLocalJobsIdle` and `TotalSchedulerJobsIdle`.
- A job submission can now specify the exact time that it should be executed at using the `DeferralTime` attribute. The time is specified as the number seconds since the Unix epoch (00:00:00 UTC, Jan 1, 1970). An additional attribute `DeferralWindow` can be specified along with the deferral time that will allow a job to run even if it misses the execution time. The window is the number of seconds in the past that Condor will allow for a missed job to execute. This feature is not supported for scheduler universe jobs.

- Added the concept of a "controlling" daemon to the *condor_master*. This feature is currently used only for "High Availability" (HA) configurations involving the *condor_had* daemon. To properly use these Condor HA features you must set this macro.

  To configure the *condor_negotiator* daemon to be controlled by the *condor_had*, you should add an entry to your condor_config:

  ```
  MASTER_NEGOTIATOR_CONTROLLER = HAD
  ```

  This will cause the *condor_master* to treat the *condor_had* as the "controller" of the *condor_negotiator*.

- Grid-type condor grid universe jobs now respect configuration parameters `GRIDMANAGER_MAX_PENDING_SUBMIT_PER_RESOURCE` and `GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE` .

- Grid universe jobs can now determine their **grid_type** via matchmaking, in addition to which resource they will be submitted to. A **grid** universe job may become any **grid_type** job, depending on what resource ad it is matched with.

- Added support for a new configuration value, `STARTD_CRON_AUTOPUBLISH` . This setting can be used to tell the *condor_startd* to automatically publish a new update to the *condor_collector* whenever any of the *cron* modules it is configured to run have produced output. For more information, see the description of `STARTD_CRON_AUTOPUBLISH` in section 3.3.10 on page 176.

- Reduced delay in negotiation when a job is released. A reschedule request is sent to the negotiator when a job is released from hold. This reduces the delay in several cases, most notably when using Condor-C or "condor_submit -s". Previously the negotiator would not be notified and would normally wait until the next scheduled negotiation cycle.

- Added three new user log events: GridResourceUp, GridResourceDown, and GridSubmit. They are equivalent to the existing Globus-specific log events, but are used for all grid universe jobs.

- When known, CPU-usage information will be reflected in the Terminated user log event for grid universe jobs.

- Changed ClassAd expression evaluation so that logical and and logical or are short-circuited. This means that an expression like `TARGET.foo && TARGET.bar` will not evaluate `TARGET.bar` if `TARGET.foo` evaluates to false. This will speed up some expressions, particularly those involving user-defined functions. Although this was thoroughly tested, this is the sort of change that could have subtle, unexpected behavior, so please be on the lookout for problems that might be caused by it.

- Added the *condor_check_userlogs* command, which checks user log files for "illegal" events.

- New settings `SYSTEM_PERIODIC_HOLD` , `SYSTEM_PERIODIC_RELEASE` , and `SYSTEM_PERIODIC_REMOVE` . These expressions behave identically to the job expressions `periodic_hold`, `periodic_release`, and `periodic_remove`, but are evaluated for all jobs in the queue. If not present, they default to FALSE.

- An improved version of the DRMAA C library is available for download from http://prdownloads.sourceforge.net/condor-ext/condor_drmaa_6_7_14_src.tgz
- Added `CLAIM_WORKLIFE` configuration option. The startd will not allow claims older than the specified number of seconds to run more jobs. Any existing job that is running when the worklife expires, however, is allowed to continue to run as normal.

Bugs Fixed:

- Fixed the following problems with the Condor SOAP interface: a) placing a job on hold now stops the job as expected, b) fixed potential schedd segfaults when sending NULL buffers via SOAP, c) fixed compatibility problems with .NET clients
- Fixed a potential security problem where any machine in the pool could advertise an additional *condor_negotiator* in the pool. Now, the *condor_collector* will only accept negotiator classads from machines listed in the `HOSTALLOW_NEGOTIATOR` variable. This bug has been in Condor since version 6.7.4.
- Fixed bug in the dedicated scheduler where on busy pools running mixed parallel and sequential jobs, it would incorrectly try to preempt dedicated jobs.
- Fixed some problems when Microsoft .NET clients communicate with Condor via SOAP. The issues were resolved by upgrading the version of gsoap included inside of Condor to gsoap ver 2.7.6c.
- Fixed the bug in the *condor_ckpt_server* from version 6.7.13 where it would give clients the wrong IP address and no checkpointing was possible. This would result in the following sorts of errors in the log file generated by the *condor_shadow* (by default, `ShadowLog`):

  ```
  Read: connect() failed - errno = 111
  Read: open_tcp_stream() failed
  Read: ERROR:open_ckpt_file failed, aborting ckpt
  ```

  Version 6.7.14 of the *condor_ckpt_server* is working properly once again.
- Fixed bugs in Condor's Generic Connection Broker (GCB) support. Condor version 6.7.14 is linked with a new version of the GCB library (1.3.1) that fixes a major bug in how GCB handles UDP messages. Previous versions of GCB had a UDP receive buffer that was far too small, resulting in many dropped UDP packets. Now, GCB will dynamically allocate more buffer space as needed. The new version of GCB also adds support for comments (any line beginning with #) in the GCB routing table. For more information about GCB, see section 3.7.3 on page 301.
- Update job information such as ImageSize, RemoteUserCpu and RemoteSysCpu at job completion. Previously this was only done periodically.
- Fixed a bug that could cause the *condor_gridmanager* to crash when a job using job leases left the queue.

- Fixed a bug that could cause the *condor_schedd* to repeatedly start the *condor_gridmanager* to manage jobs that were complete. This would happen when LeaveJobInQueue evaluated to True.
- When GSI_DAEMON_TRUSTED_CA_DIR is set, pass the setting down to the gt4 gahp server.
- Fixed a bug in *condor_dagman* that caused the UNLESS-EXIT feature to not work with POST scripts (the return value from a POST script was not tested against the UNLESS-EXIT value).
- Fixed a bug in *condor_dagman* that caused POST scripts to work incorrectly with node retries: if the node job failed for a node with retries, the POST script was only run on the last retry.
- Fixed a bug in *condor_dagman* that caused rescue DAGs to fail if the original DAG was run with the **-UseDagDir** command-line flag. (This bug was introduced at some point after version 6.7.10 and before version 6.7.13.)
- Improved usage of uid caching introduced in 6.6.0. This will further reduce load on NIS servers. See the discussion of PASSWD_CACHE_REFRESH in section 3.3.3 on page 151 for more details.
- Fixed a bug in 6.7.13 for Windows causing incorrect handling of absolute paths in the job's output/error if the path began with a forward slash rather than a backslash.
- Fixed a bug in the *condor_master* that caused "condor_off -subsystem" (and similar commands) to fail if the daemon name wasn't hard-coded into the *condor_master*. The *condor_master* now handle any daemon listed in the DAEMON_LIST for these commands.
- Fixed a bug in the condor_gridmanager that caused it to undercount already-submitted jobs at start-up for purposes of job throttling to Globus grid resources.
- Improved the handling of job leases for grid-type condor jobs. There was a race condition between the lease expiring and the *condor_gridmanager* attempting to extend the lease. Also, a lease could be set and not extended well before the job was actually submitted. In certain cases, the forwarded lease could exceed JobLeaseDuration.
- The startd no longer advertises itself as available to run jobs when it is in shutdown mode (e.g. waiting for jobs to finish). This was a noticeable problem when using large values for MAXJOBRETIREMENTTIME on multi-VM startds; while waiting for one of its VMs to finish running a job, the startd would be available for matching to jobs, but it would reject them when the schedd tried to start them, possibly causing an endless cycle of matching, attempting to run, and failing.
- Fixed some minor typos and formatting bugs in some of the log messages generated by the *condor_ckpt_server*.
- For grid-type condor jobs, the *condor_gridmanager* now notices when a job disappears from the remote *condor_schedd* unexpectedly.
- When getting "connection refused", Condor command-line tools and daemons no longer continuously retry the connection attempt until timing out. These retries were causing 10 second or longer delays when trying to connect to Condor services which, for one reason or another, were no longer listening on the expected TCP port.

- Fixed a bug in the *condor_c-gahp* that could cause it to crash if it fails to connect to a remote *condor_schedd* when submitting a job.
- Minor memory leaks have been fixed.
- Fixed a bug in Quill that could result in an infinite loop in *condor_q* when querying Quill.
- *condor_preen* now looks at the `HISTORY` setting in the configuration file when considering files to erase. Previously it assumed the history file was always called "history."

Changes:

- The *condor_dagman* log file path is converted to an absolute path inside *condor_dagman* itself, so that the logging works for multi-directory rescue DAGs (which it didn't before), but the `.condor.sub` files are still portable.
- Added the Stork log file (if any) to the list of log files that *condor_dagman* lists in the `dagman.out` file.
- *condor_dagman* now reports the node return value for all failed nodes.
- Attributes names forced into the job ad via '+' are no longer converted to lower-case. This conversion was a side-effect of a bug-fix in 6.7.11 and caused problems with code that assumed that Condor would preserve the case of attribute names.
- Job policy expressions are now evaluated on COMPLETED and REMOVED jobs in the schedd.

Known Bugs:

- The `NEGOTIATOR_MATCHLIST_CACHING` setting is broken. It should not be used. This setting is `FALSE` by default, but if set to `TRUE`, the *condor_negotiator* will crash.
- Jobs that are are placed on held because of `on_exit_hold` evaluated to TRUE or jobs that stay in the queue after finishing because `on_exit_remove` evaluated to FALSE will erroneously report the reason as "UNKNOWN (never set)".

## Version 6.7.13

Release Notes:

- Added a new natively compiled clipped port for the Red Hat Enterprise Linux 3 IA64 distribution.

New Features:

- Added support complete support for Quill on Windows, so job queues can now be accessed via a relation database. Quill is now available on all Condor supported platforms. See page 330 for more information.
- Added support in Condor for the Generic Connection Broker (GCB). This is a system for managing network connections across public and private networks. More information about GCB can be found in section 3.7.3 on page 301.
- Added a new configuration option, `BIND_ALL_INTERFACES` This is a boolean value that controls if Condor should bind and listen to all the network interfaces on a multi-homed machine. If set to TRUE, the value of `NETWORK_INTERFACE` will only control what IP address is published by Condor daemons, even though they will still be listening on all interfaces. The default is FALSE.
- Added a **-pool** option to *condor_submit*. It lets you submit jobs to a *condor_schedd* in a different pool. The other options to *condor_submit* now have long names, but the single-character versions still work.
- "grid_resource" can now be used to directly set the new grid universe job attribute "GridResource." The old attributes still work, but they will be ignored if "grid_resource" is present. As a side-effect, "stream_output" and "stream_error" will default to "False" for all jobs.
- X509 user proxies are now updated for vanilla universe jobs. If a job specifically sets x509userproxy and is using file transfer, when the proxy file is updated, it will be transfered to the running job.
- If a cycle is detected in the DAG while running, *condor_dagman* now prints (in the `dagman.out` file) the status of all DAG nodes.
- BeginTransaction call in *condor_schedd*'s SOAP interface now notifies the caller if too many transactions are currently running via an error code of FAIL. Previous behavior was to abort a running transaction in order to allow the BeginTransaction call to succeed.
- `MAX_SOAP_TRANSACTION_DURATION` config option added so that a single transaction cannot take up too many *condor_schedd* resourced. This option specifies an optional maximum duration between SOAP calls in a single transaction.
- If a machine is acting as both a submit and an execute node, and it cannot communicate with the central manager, it will attempt to run jobs locally. If Condor specific terms, if the *condor_schedd* fails to hear from the central manager, it will attempt to run jobs on a locally running *condor_startd*. The `SCHEDD_ASSUME_NEGOTIATOR_GONE` config macro was added to support this feature; see page 186 for details.
- You can now specify per-subsystem entries in your condor_config file by prepending the subsystem name and a period to the normal name. The per-subsystem settings take precedence over the regular settings.

- *condor_dagman* now recovers automatically after being abruptly killed by something other than Condor itself (e.g., by Unix initd during a "fast" system shutdown). This is accomplished through the use of a default `OnExitRemove` expression inserted by *condor_submit_dag* which instructs the *condor_schedd* not to treat death by SIGKILL as a valid exit condition for *condor_dagman*.
- Added submit attribute **globus_xml**, for use with grid-type gt4 jobs. The given XML text will be inserted at the end of the XML job description written by Condor for submission to the WS-GRAM server.
- For grid-type gt4 jobs, if a URL scheme is missing from the resource name, "https://" will be inserted automatically.
- Added submit attribute **transfer_output_remaps**. This specifies the name (and optionally path) to use when downloading output files from the completed job. Normally output files are transferred back to the initial working directory with the same name they had in the execution directory. This gives you the option to save them with a different path or name.

Bugs Fixed:

- Fixed a bug concerning backslash escaping in classad attribute values when *condor_q* was using quill.
- Fixed a bug where *condor_q* could not accept multiple jobids on the command line.
- Fixed parallel universe ssh script to now clean up all temporary files it creates.
- Fixed a bug in the dedicated scheduler that caused it to request resources it could not use, resulting in longer job startup times.
- Fixed a bug in the *condor_schedd* that caused grid-type gt2 jobs submitted by an older *condor_submit* or in the queue during an upgrade (version 6.7.10 or earlier) to go on hold if the grid_type was "globus".
- Fixed a bug in *condor_submit* that caused it to not set `JobGridType` in the job ad for grid universe jobs when submitting to a *condor_schedd* older than version 6.7.11.
- When using file transfer, transferring the results back to the submit machine could silently fail for Condor releases 6.7.0 though 6.7.12. This was relatively rare through 6.7.10. For 6.7.11 and 6.7.12, the bug would be easily triggered if a vanilla job had an X509 user proxy associated with it. This is now fixed.
- Fixed a logic bug in the *condor_schedd*. Previously, if there was an error expanding any `$$(attribute)` references in a job classad when trying to spawn a *condor_shadow*, the *condor_schedd* would die with the fatal exception "Impossible: GetJobAd() returned NULL for X.Y but that job is already known to exist". Now, the *condor_schedd* correctly distinguishes between a non-fatal error expanding `$$(attribute)` and the fatal error of the job already being gone (which is, in fact, impossible). This bug was first introduced in Condor version 6.7.1.

- The reason strings generated when a user job policy expression fires are now consistent for grid universe jobs.
- The *condor_gridmanager* now evaluates the periodic job policy expressions at the interval set by `PERIODIC_EXPR_INTERVAL`.
- Fixed a bug which prevented standard universe from working on a linux kernel post 2.6.12.2.
- The *condor_schedd* used to crash in certain cases if a given job was vacated using *condor_vacate_job*, then put on hold and released. The bug only appeared if a specific job id was given to *condor_vacate_job*, as opposed to specifying a username or another constraint. Now, the use of *condor_vacate_job* for individual job identifiers is safe and the *condor_schedd* will not crash. This bug has been in Condor since support for *condor_vacate_job* was first added in version 6.7.0.
- Fixed a bug that caused the *condor_gridmanager* to crash if a grid-type condor job ad contained the attribute `remote_`.
- Fixed a bug with the FS_REMOTE authentication mechanism that caused it to fail occasionally when using NFS.
- Fixed a bug in which a double terminated event in a DAG node with a POST script could cause *condor_dagman* to abort the DAG and claim that a cycle exists in the DAG.
- In the DAG status messages in `dagman.out` files, *condor_dagman* now shows nodes with queued PRE or POST scripts in the Pre or Post columns. Previously, these nodes were shown in the Un-Ready column.
- Fixed the GetFile SOAP call on the *condor_schedd* so that it behaves more like POSIX read() and does not report errors when trying to read more data than is available.
- Fixed a hash function bug that could cause *condor_dagman* to crash.
- `JobCurrentStartDate` and `JobLastStartDate` are no longer changed in the job ad when the *condor_schedd* and *condor_shadow* reconnect to a running job after a crash.
- *condor_dagman* now allows POST scripts to be used with DATA nodes in a DAG (previously this caused the DAG to hang).
- Using the new **Remote_** simplified syntax no longer generates unnecessary debug messages.
- Fixed a bug in estimating the size of attribute value buffers that caused quill to crash. This arose when job ads had variables with very large values (more than 3KB).
- Fixed a bug in the *condor_gridmanager* that could cause it to crash when the `Rematch` attribute evaluates to True.
- The default base scratch directory for WS-GRAM doesn't exist on most server machines. Added a work-around to create the directory as part of the job submission.

- Starting in version 6.7.11, the execute host reported for grid jobs in the user log execute event can contain spaces. The C++ user log reading code now properly reads the entire string for these events.
- Fixed a bug that caused the *condor_gridmanager* to die when it tried to renew the job lease of a grid-type condor job.
- Fixed a bug that was causing the *condor_schedd* to crash on Solaris if the cron macros aren't defined.
- Fixed a bug where output may be lost when spooling (with the -s option to *condor_submit* or implicitly with Condor-C). This bug could only happen if the job terminated within one second of starting.
- Fixed a bug affecting transferral of output and error files where the file specified in the submit file contains path information. The file was being staged back into the initial working directory and then it was copied to the final path specified. The bug is that if there was an error copying the file to the final location, the intermediate copy would not be deleted and the job would still exit successfully, as though it had succeeded. Now, no intermediate copy of the file is made, and errors in transferring the file will be treated as a failure to run the job, which will typically cause the job to return to idle state and run again.

Changes:

- Added a couple missing parameters to the example configuration file `condor_config.generic`.
- Slightly cleaned up event checking error messages in *condor_dagman*.
- Fixed a bug in the *condor_c-gahp* that caused it to crash when handling grid-type condor jobs with job leases.
- Starting in 6.7.11, the "JM-Contact" field of the "Job submitted to Globus" user log event was mis-printed. This has been corrected.
- Fixed bug that prevented Stork detection of hung jobs.
- Fixed an obscure bug that incorrectly quoted the status of completed jobs, visible via *stork_status*.

Known Bugs:

- The *condor_ckpt_server* is broken in version 6.7.13. Please do not attempt to use it. It is safe to use the 6.7.12 *condor_ckpt_server* in a pool running 6.7.13 until the 6.7.14 release is out. Of course, the 6.7.12 *condor_ckpt_server* will not work with GCB, so sites wishing to use both GCB and a *condor_ckpt_server* will have to wait for 6.7.14.
- Rescue DAGs generated from DAGs run with the **-UseDagDir** command-line flag no longer work. (The original run with **-UseDagDir** should work, but if it fails and generates a rescue DAG, the rescue DAG will *always* fail.)

### Version 6.7.12

Release Notes:

- 6.7.12 addresses several critical bugs in 6.7.11. 6.7.11 should not be used.

Bugs Fixed:

- Fixed a serious bug introduced in 6.7.11 which prevented *condor_dagman* from successfully removing its own jobs from the Condor queue after receiving a *condor_rm* request from the *condor_schedd*.
- Fixed a serious bug introduced in 6.7.11 where the *condor_master* on Windows would not properly shut down.

### Version 6.7.11

Release Notes:

- Condor is now linked against GSI from Globus 4.0.1.
- GSI security and the grid universe should now work in the Alpha Linux port.
- All Condor release packages are now compressed with GNU's *gzip*. We no longer ship releases compressed with the vendor's *compress* utility.

New Features:

- Added a new feature called **Quill** to Condor which allows an SQL server to mirror the job queue in order to speed up queries about the job queue via *condor_q* and *condor_history*. Please see page 330 for the description of this feature.
- *condor_dagman* has a new **-maxidle** command-line argument that can be used to throttle DAG job submissions according to the number of idle jobs in the DAG.
- *stork_submit* is now able to search for X.509 credentials in the standard locations.
- The *condor_negotiator* can now limit how long it negotiates with a single submitter before moving on to the next one.
- On platforms and filesystems that support files larger than 2 GB, the history file can now be larger than 2 GB.
- Added two options to *condor_q*: **-jobads** and **-machineads**. They will take ads from files instead of the schedd and collector, respectively. These options are mostly useful for debugging.

- Added a new, hopefully less confusing, Cron (Hawkeye) configuration syntax. The old syntax is still supported, but should be considered deprecated, and will eventually go away. The new syntax splits the old colon separated "name:prefix:executable:period" string into separate macros.
- Improved support for job leases. "job_lease_duration" now works for grid-type condor jobs. New job ad attribute "TimerRemove" specifies a specific time at which a job should be removed. These attributes will be passed through multiple layers of grid-type condor jobs.
- Grid universe jobs now use a unified pair of attributes ("GridResource" and "GridJobId") to identify the remote resource. This will make it possible to match jobs to multiple types of resources. The submit file syntax remains the same for now, except that "remote_pool" is now required for grid-type condor jobs.
- Significantly improved response time for *condor_q* when job classads are larger than 4 kbytes (by disabling TCP Nagle algorithm as appropriate).

Bugs Fixed:

- Fixed bug in the dedicated scheduler where if the *condor_startd* rejected a match, the *condor_schedd* would never retry new matches for that machine. This would result in MPI and parallel jobs sticking in the Idle state, and the message "DedicatedScheduler::negotiate sent match for machine, but we've already got it".
- Fixed problem with the parallel universe to allow for LAM jobs to get SIGTERM on exit so they can exit cleanly.
- Fixed a bug that was visible to the end user as file transfer failures on a busy system. The root problem was that if the *condor_negotiator* gave out the same match twice (due to having stale info in the *condor_collector* when trying to negotiate), the *condor_schedd* would be confused, attempt to re-use the match, fail to do so, and then kill the previous (legitimate) use of the match. This bug was introduced in version 6.7.4.
- Fixed bug in the parallel universe that caused the schedd to crash when reconnecting to jobs that couldn't be reconnected to.
- Fixed bug in parallel shadow which caused Shadow Exceptions in parallel jobs when the components exited in the wrong order.
- Fixed a bug in *condor_dagman* that caused it to fail on Windows for DAGs with nodes having absolute paths to their log files. (This bug was introduced in version 6.7.10.)
- Fixed a bug whereby *condor_dagman* could crash after executing the POST script of a node whose Condor job had never been successfully submitted due to repeated *condor_submit* failures. (This bug was introduced in 6.7.7 or earlier.)
- Fixed a bug in a debug message. If an error occurred during file transfer, Condor would print the wrong expected filesize in the error message on some platforms.

- Fixed bug where *stork_submit* was corrupting log notes passed from the command line. This bug also had the effect of disabling Stork jobs running from DAGMan versions v6.7.10, and later.
- If you have DATA nodes in your DAG but no Stork log specified (with the **-Storklog** argument), *condor_dagman* now fails with an explanatory message when parsing the DAG file(s). (Previously, it would just wait forever for the Stork jobs to finish, because it wouldn't see the relevant events.)
- In *condor_dagman*, argument quoting for *stork_submit* now matches argument quoting for *condor_submit*.
- Corrected how *condor_submit* handles attributes forced into the job ad with '+'. Now, the attribute names are case-insensitive, they are not treated as normal submit attributes, and they always over-ride normal submit attributes.
- Fixed bugs that would cause a segfault when reading a classad from a file. Triggered by consecutive blank lines and lines containing only white-space.
- Fixed a bug that could cause duplicated output when a gt4 grid job is executed more than once.
- Fixed a bug that could cause the *condor_gridmanager* to assert if it tried to delegate credentials for gt4 grid jobs before the gahp server was started.
- Fixed a race condition that could cause condor grid-type jobs to be held with hold reason "Spooling input data files".
- *condor_glidein* now correctly handles extracting necessary information from modern Condor configurations where `NEGOTIATOR_HOST` is not defined.
- Refinements in how grid universe components track jobs. Grid universe jobs are less likely to generate multiple terminate events in the job's user log. There will also be slight performance improvements are redundant work is no longer done.
- Fixed a bug in *condor_dagman* that caused it to core dump on a 'job reconnected' event from a node job.
- *condor_submit* will now exit zero as long as the submission succeeds. Debugging output will still be printed if the internal reschedule fails.
- On Windows, exited child processes of the Condor services will be handled in order of termination. This fixes the problem where jobs submitted from a Windows machine appear to run much longer than normal because the *condor_schedd* fails to notice that a *condor_shadow* exits when the system is very busy.
- Fixed a bug that caused scheduler universe jobs to often wait five minutes (or whatever `SCHEDD_INTERVAL` is set to) before running.
- Fixed a bug that prevented the *condor_starter* from running on a Win32 machine with a FAT32 filesystem.

- A reschedule command will now be sent to the *condor_schedd* whenever a job is released from held state. This should make grid-type condor jobs start much faster.
- Config parameters `GAHP` and `GAHP_ARGS` have been deprecated. `GT2_GAHP` should be used instead.

Changes:

- *condor_configure* no longer creates a

  ```
  $(LOCAL_DIR)/ViewHist
  ```

  directory, which was begun in version 6.7.10. This directory was of limited value for most users.

Known Bugs:

- None.

## Version 6.7.10

Release Notes:

- This release contains all of the bug fixes and improvements from the 6.6 stable series up to and including version 6.6.10.
- The Mac OS X binaries shipped with this release were built on OS 10.3. Previous versions of Condor for OS X were built with version 10.2. Condor is officially dropping support for Mac OS 10.2 with this release (though it is possible the 10.3 binaries still work, we have not verified it either way). These binaries are known to work with Mac OS 10.4 ("Tiger"), as well.
- There is a minor bug in version 6.7.10's *condor_configure* script. It will create a directory called `ViewHist` in the local directory (next to `log`, `spool`, etc). This directory is not used by Condor at all, except in the case of a *condor_view* collector (which is optional, and not enabled by default). This behavior will be removed in version 6.7.11, and *condor_configure* will go back to not creating the `ViewHist` directory.

New Features:

- *condor_dagman* can now run multiple DAGs in separate directories.

- Added DAGMAN_CONDOR_SUBMIT_EXE , DAGMAN_STORK_SUBMIT_EXE , DAGMAN_CONDOR_RM_EXE , and DAGMAN_STORK_RM_EXE configuration settings to specify the *condor_submit*, *stork_submit*, *condor_rm*, and *stork_rm* executables used by *condor_dagman*. If unset (which they are by default), *condor_dagman* looks for each in the PATH.
- For Condor-C jobs, the *condor_gridmanager* will retry and delay failed connections to a remote *condor_schedd* like it does for Condor-G jobs. The same configuration settings apply (GRIDMANAGER_CONNECT_FAILURE_RETRY_COUNT and GRIDMANAGER_RESOURCE_PROBE_INTERVAL ).
- **remote_initialdir** is now supported in all universes except for standard universe. Previously, it was only supported in the grid universe.
- **+Remote_** syntax for Condor-C jobs has been simplified for the specific commands of **universe**, **remote_schedd**, **remote_pool**, **globus_rsl**, and **globus_scheduler**.
- Added default user priority factors for accounting groups. More on accounting groups will be available in future versions of the manual.
- The *condor_startd* can now be configured to write out the ClaimId of the next available claim for each virtual machine to separate files. This functionality will enable enhanced fault tolerance in future versions of Condor. For more information, see section 3.3.10 for details on STARTD_SHOULD_WRITE_CLAIM_ID_FILE and STARTD_CLAIM_ID_FILE , the two configuration settings that control this behavior.

Bugs Fixed:

- Fixed bugs on the Win32 platform in the *condor_schedd* that could cause jobs to never complete when the *condor_schedd* is busy with many jobs running at once.
- Fixed a bug on Windows where if lots of jobs submitted were from the same *condor_schedd*, some of the *condor_shadow* processes would block for an extremely long time trying to get a lock for writing to the ShadowLog file. Now, log writing happens more fairly, and no *condor_shadow* processes can be delayed indefinitely.
- *condor_submit* **-name** formerly had no effect on Windows and did not work properly. This is now fixed.
- Significantly sped up the removal of large groups of jobs by changing the default value of JOB_IS_FINISHED_INTERVAL from 1 to 0 (see section 3.3.11 for details on this setting).
- Improved performance of the *condor_schedd* when not running as root. In version 6.7.7, the new code to support the scheduler universe with Condor-C involved adding some additional overhead to the *condor_schedd*. However, this overhead is not needed unless the *condor_schedd* is running as root. In version 6.7.10, the *condor_schedd* notices if it is not root and does an optimization to avoid the overhead.

- Fixed a bug that caused the gridmanager to crash if a gt2, gt3, or gt4 grid job had a proxy that couldn't be read properly. Now the job gets put on hold.
- The Condor-C GAHP now performs file staging in a separate process, allowing remote grid jobs to be started earlier.
- When contacting the embedded web server on Condor daemons, authentication is no longer requested. The previous authentication requirement didn't provide any additional security, and could confuse users.
- Fixed rare bug that could cause *condor_submit* to crash when both getenv=true and environment=... were in a submit file and when very large variable names were in the environment.
- Fixed a rare bug where the *condor_schedd* would die with a fatal exception under extremely heavy load on the machine. The error message was:

```
ERROR ``Impossible: Create_Thread child_errno (xxx) is not
ERRNO_PID_COLLISION!'' at line 6181 in file daemon_core.C
```

- Fixed a rare bug where certain attributes in a job description file could cause the *condor_schedd* to crash when restarting and parsing the `job_queue.log` file.
- Improved performance of standard universe jobs when `WantRemoteIO` is set to false in the job ClassAd. In this case, Condor's checkpointing libraries now avoid some additional communication with the *condor_shadow* which are not required if there's no remote IO.
- Fixed some messages in the Condor log files that were improperly formatted, or contained incomplete information.
- Improved some user-log-reading error messages in *condor_dagman*.
- Removed support for deprecated **-NoPostFail** option from *condor_dagman*. (The same functionality can be achieved through the use of a simple POST script.)
- Fixed bug in dedicated scheduler, where under heavy load, the schedd would occasionally try to start the same job twice, and subsequently exit with the message:

```
ERROR ``Trying to run job x.x, but already marked RUNNING!''
```

- Fixed bug in dedicated scheduler, so that it now creates a spool directory for each condor proc of a parallel or MPI job with multiple requirements.

Known Bugs:

- On Windows only, *condor_dagman* fails for DAGs with nodes having absolute log file paths in their submit files.
- *condor_dagman* does not correctly handle the case where all submit attempts for a node job fail, and the node has a POST script. If this happens for a single node in a DAG, it is usually okay, but if it happens for a second node, *condor_dagman* will crash.

- The Condor-C GAHP now performs file staging in a separate process, allowing remote grid jobs to be started earlier.
- Using the new **Remote_** syntax simplification causes *condor_submit* to display debug messages to standard output, possibly confusing programs that parse *condor_submit*'s output. Fixed in 6.7.13.

### Version 6.7.9

Release Notes:

- This release contains all of the bug fixes and improvements from the 6.6 stable series up to and including version 6.6.10.

New Features:

- The Parallel Universe has been added. For more information, see section 2.10 on page 70.
- The environment variable `X509_USER_PROXY` is set to the full path of the proxy if a proxy is associated with the job. This is usually done using **x509userproxy** in the submit file. This currently works in the local, java, and vanilla universes.
- *condor_submit* generates more precise error messages in some failure cases.
- *condor_hold*, *condor_release* and *condor_rm* now allow the user to change the HoldReason, ReleaseReason or RemoveReason with the -reason flag.
- *condor_dagman* no longer does a one-second sleep before each submit if all node jobs have the same log file. (The sleep is still needed if there are multiple log files, for unambiguous ordering of events during bootstrapping.) Note that if `DAGMAN_SUBMIT_DELAY` is specified, the specified delay takes effect whether or not all jobs have the same log file.

Bugs Fixed:

- Many crashes related to running the Dedicated Scheduler have been fixed.
- Setting COLLECTOR_HOST or NEGOTIATOR_HOST with a port but without a hostname no longer causes the *condor_master* to crash.
- The Condor-G Grid Monitor now works with Globus 4.0 pre-Web Services GRAM.
- Several deadlocks in the Condor-C GAHP server have been fixed.

### Version 6.7.8

Release Notes:

- This release contains all of the bug fixes and improvements from the 6.6 stable series up to and including version 6.6.9.

New Features:

- Controlling whether or not a standard universe job asks the *condor_shadow* about how/where to open every single file can be better controlled with the `want_remote_io` attribute in the submit description file. This attribute can be set to true or false and it is true be default. If set to false, then this attribute forces a standard universe job in Condor to always look to the local file system when opening files and not to contact the shadow. This increases performance of user jobs where the jobs open a very large amount of files in a small space of time. However, the user jobs must be matched to machines that have the same UID_DOMAIN and FILESYSTEM_DOMAIN, as per vanilla universe jobs with a homogeneous file system.
- *condor_dagman* now has the capability to run more than one independent DAG in a single *condor_dagman* process.
- User policy expressions (on_exit_remove and on_exit_hold) now work for scheduler universe jobs.
- TotalCpus and TotalMemory are now set in machine ads.
- *condor_dagman* now tolerates the "two terminated events for a single job" bug by default. There is a new bit in `DAGMAN_ALLOW_EVENTS` to control whether this bug is considered a fatal error in a *condor_dagman* run.
- Added a new debug formatting flag, `D_PID`, that prints out the process id (PID) of the process writing a given entry to a log file. This is useful in Condor daemons (such as the *condor_schedd*) where the daemon can fork() multiple processes to perform various tasks and it is helpful to see what log messages are coming from forked process versus the main thread of execution. The default `SCHEDD_DEBUG` in the sample configuration files shipped with Condor now includes this flag.
- When *condor_dagman* writes rescue files, each node is now specified with the same number of retries as was specified in the original DAG, rather than with only the "remaining" number of retries based on the failed run. The latter behavior can be restored by setting `DAGMAN_RESET_RETRIES_UPON_RESCUE` to false.
- Added "Hawkeye" capabilities to *condor_schedd*. It's configured identically to that of *condor_startd*, but using "SCHEDD" in place of "STARTD", in particular for the "SCHEDD_CRON_NAME" macro.

Bugs Fixed:

- Fixed a bug in *condor_dagman* that prevented POST scripts from being used with jobs that write XML-format logs.
- The event-checking code used by *condor_dagman* now defaults to allowing an execute event before the submit event for the same job; if this happens, there will be a warning, but the DAG will continue. See section 3.3.23 for more info.
- *condor_userprio* option **-pool** was failing with "Can't find address for negotiator" since version 6.7.5.
- Fixed a bug the prevented SOAP clients from being able to access a job's spooled data files if the *condor_schedd* restarted.
- Fixed a bug that caused the *condor_gridmanager* to panic when trying to retire a job from the queue that was already gone. This could cause multiple terminate events to be logged for some jobs.
- Fixed a bug that caused match-making to not work for Condor-C jobs.
- Added workaround for a Globus bug that can cause re-execution of a completed GT2 job in the correct failure case (Globus bugzilla ticket 3411).
- Properly extend the lifetime of GT4 jobs and credentials on the remote server.

## Version 6.7.7

Release Notes:

- This release contains all of the bug fixes and improvements from the 6.6 stable series up to and including version 6.6.9.

New Features:

- The `STARTD_EXPRS` list can now be on a per-VM basis, and entries on the list can also be specific to a VM. See 3.12.7 for more details.
- The `LOCAL_CONFIG_FILE` can now be overridden. This now allows files to include other local config files. See 3.3.3 for more info.
- Resources that are claimed but suspended can now optionally not be charged for at the accountant. When the resource is unsuspended, the accountant will resume charging for usage. This is controlled by the `NEGOTIATOR_DISCOUNT_SUSPENDED_RESOURCES` config file entry, and it defaults to false.
- The `DAGManJobID` attribute which condor_dagman inserts into the classad of every job it submits now contains only its cluster ID (instead of a cluster.proc ID pair), so that it may be referenced as an integer in DAG job submit files. This allows, for example, a user to

automatically set the relative local queue priority of jobs based on the condor_dagman job that submitted them, so that jobs submitted by "older" DAGs will start before jobs submitted by "newer" DAGs (assuming they are otherwise identical).

- GSI authentication can now be used when Condor-C jobs are submitted from one condor_schedd to another.
- File permissions are now preserved when a job's data files are transferred between unix machines. File transfers that involve a windows machine or older version of Condor remain as before.
- Condor-C now supports the scheduler remote universe.
- condor_advertise now publishes a "MyAddress" if none is provided in the source ClassAd. This will prevent the collector from throwing out ads with no address (see Bugs Fixed).
- Added a new *condor_dagman* parameter `DAGMAN_ALLOW_EVENTS` controlling which "bad" events are not considered fatal errors; the **-NoEventChecks** command-line argument is deprecated and has no effect.
- *condor_fetchlog* now takes an optional log file extension in order to select logs such as "StarterLog.vm2".

Bugs Fixed:

- Fixed a throughput performance bottle neck when standard universe jobs vacate when the user has specified `WantCheckpoint` equal to False in the submit file.
- Added initial support for the `getdents()`, `getdents64()`, `glob()`, and the family of functions `opendir()`, `readdir()`, `closedir()` for the standard universe.

  It is recommended that you do not directly invoke `getdents()` or `getdents64()`, but instead use the other POSIX functions specified above.

  There are two caveats: these calls will not work in heterogeneous contexts, and you may not call `getdents()` directly when *condor_compile*ing a 32-bit program while specifying the 64-bit interfaces for the Unix API.
- In versions 6.7.4 through 6.7.6, Computing On Demand (COD) support was broken due to a bug in how Condor daemons parsed their command line arguments. The bug was introduced with the changes to provide a web services (SOAP) interface to Condor. This bug has been fixed and COD support is now working again.
- In version 6.7.6, the `DAGParentNodeNames` attribute which *condor_dagman* adds to all DAG job classads could grow too long and cause job submission to fail. Now, if the `DAGParentNodeNames` value would be too long to add to the job classad, the attribute is instead left undefined and a warning is emitted in the DAGMan debugging log. This behavior means that such a node can be reliably distinguished from a node with no parents, as the latter will have a `DAGParentNodeNames` attribute defined but empty.

- In version 6.7.3, the value of the X509UserProxySubject job attribute was changed in such a way that Condor-G jobs submitted by a newer condor_submit to an older condor_schedd could fail to run. Now, condor_submit reverts to the old behavior when talking to an old condor_schedd.
- Bug-fixes and improvements to grid_type gt4:
  - Condor will now delegate a single proxy to the GT4 server for multiple. If the local proxy is refreshed, Condor will forward the refreshed copy to the server.
  - Exit codes are now recorded properly.
  - `JAVA_EXTRA_ARGUMENTS` now used when invoking the GT4 GAHP server (which is written in java).
  - If `LOWPORT` and `HIGHPORT` are set in the config file, the GT4 GAHP server will now obey the port restriction.
  - Fixed a bug that caused Condor not to notice when some GT4 jobs completed.
  - Fixed a bug in handling the job's environment for GT4 jobs. Condor incorrectly used `<name>=<value>` for each variable's name.
  - Improved hold reason in certain cases when a GT4 job goes on hold.
  - condor_q -globus now works properly for GT4 jobs. Also, the resource name in the user log execute event is printed properly for GT4 jobs.
  - Fixed a bug that could cause Condor to not detect when a GT4 job completes. This was triggered by Condor not properly recognizing the StageOut Globus job state.
- Fixed a bug that can cause the condor_gridmanager to abort if `PeriodicRelease` evaluates to true while it's putting a job on hold.
- Fixed a bug in *condor_dagman* that caused the DAG to be aborted if a job generated an executable error event.
- Fixed a bug in *condor_dagman* on Windows that would cause it to hang or crash on exit.
- MPI universe jobs now honor the `JOB_START_DELAY` configuration setting.
- The *condor_collector* now throws out startd, schedd, and License ClassAds that don't have a valid IP address (used in it's hashing). The collector now correctly will fall back to "MyAddress" if it's provided.
- Fixed a bug in *condor_dagman* that could cause *condor_dagman* to fail an assertion if PRE or POST scripts are throttled with the **-maxpre** or **-maxpost** *condor_submit_dag* command line flags.

## Version 6.7.6

Release Notes:

- Version 6.7.6 contains all the bug fixes and improvements from the 6.6 stable series up to and including version 6.6.9.

New Features:

- Added support for libc's ( )system) function for standard universe executables. This call is not checkpoint-safe in that the standard universe job could call it twice or more times in the event of a resumption from an earlier checkpoint. The invocation of this call by the shadow on behalf of the user job is controlled by a configuration file parameter called SHADOW_ALLOW_UNSAFE_REMOTE_EXEC and is off by default. The full environment of the user job is preserved during the invocation of ( )system) and this might cause problems in heterogeneous submission contexts of the user is not careful.

- Added support for a web services (SOAP) interface to Condor. For more information, see and section 4.4.1 on page 396.

  NOTE: Due to a bug in gSOAP, the SOAP support in Condor 6.7.6 does not work with all SOAP toolkits. Some of the responses that gSOAP generates contain unqualified tags. Therefore, SOAP toolkits that are strict (such as gSOAP or .Net) will not accept these poorly formed responses. SOAP toolkits that are more lax in the responses they accept (such as Axis, SOAP::Lite, or ZSI) will work with version 6.7.6. This problem has already been fixed and the solution will be released in Condor version 6.7.7.

- Added support for the GT4 grid_type in Condor's grid universe. This new grid type supports jobs submitted to grid resources controlled by Globus Toolkit version 4 (GT4).

  New configuration settings are required to support jobs submitted for the GT4 grid type. These settings have been added to the default configuration files shipped with Condor, but sites that are upgrading an existing installation and choosing to keep their old configuration files must add these settings to allow GT4 jobs to work:

```
## The location of the wrapper for invoking GT4 GAHP server
GT4_GAHP = $(SBIN)/gt4_gahp

## The location of GT4 files. This should normally be lib/gt4
GT4_LOCATION = $(LIB)/gt4

## gt4-gahp requires gridftp server. This should be the address of gridftp
## server to use
GRIDFTP_URL_BASE = gsiftp://$(FULL_HOSTNAME)
```

- Condor version 6.7.6 includes the Stork data movement system, the Condor Credential Daemon (*condor_credd*), and support for using MyProxy for credential management. However, currently these are only supported in our release for Linux using the 2.4 kernel with glibc version 2.3 (RedHat 9, etc). All of these features require changes to the Condor configuration files to function properly. The default configuration files shipped with Condor already include all the new settings, but sites upgrading an existing installation must add these new

settings to their Condor configuration. For a list of settings and more information, see section 3.3.29 on page 214 for Stork, section 3.3.19 on page 199 for *condor_credd*, and section 3.3.27 on page 213 for MyProxy. For more information about MyProxy, you can also see http://grid.ncsa.uiuc.edu/myproxy

- Added preliminary support for the High Availability Daemon (HAD).
- Added a new SCHED_UNIV_RENICE_INCREMENT configuration variable used by the *condor_schedd* for scheduler universe jobs, analogous to the existing JOB_RENICE_INCREMENT variable used by the *condor_startd* for other job universes. The SCHED_UNIV_RENICE_INCREMENT variable is undefined by default, and when undefined, defaults to 0 internally.
- The relative priority of a user's own jobs in the local condor_schedd queue is no longer limited to the range -20 to +20, but can be any integer value.
- DAGMan Improvements:
  - *condor_dagman* now inserts a DAGParentNodeNames attribute into classad of all Condor jobs it submits, containing the names of the job's parents in the DAG. The list is in the form of a comma-delimited string.
  - Added the *condor_dagman* arguments **-noeventchecks** and **-allowlogerror** to *condor_submit_dag*.
- *condor_glidein* Improvements:
  - Added *condor_glidein* options for setting up GSI authentication.
  - Added *condor_glidein* option -run_here for direct execution of Glidein, instead of submitting it for remote execution. You may also save a script for doing this and then run the script through whatever mechanism you want (like some batch system interface not supported by Condor-G).
- Added support for the NEGOTIATOR_CYCLE_DELAY configuration setting, which is only intended for expert administrators. For more information, see section 3.3.18 on page 196.

Bugs Fixed:

- Previous versions of the *condor_master* had a bug where if the administrator attempted to use <SUBSYS>_ARGS to pass **-p** to any Condor daemon to have it listen on a specific, fixed port, the underlying daemon would not honor the flag. Now, the *condor_master* correctly supports using <SUBSYS>_ARGS to define a port using **-p**. For more information about <SUBSYS>_ARGS, see section 3.3.9 on page 164.
- Removed case-sensitivity of command-line argument names in *condor_submit_dag*.
- Fixed the -r (remote schedd) option in *condor_submit_dag*.

- Condor versions 6.7.1 through 6.7.5 exhibit a bug in which the commands *condor_off*, *condor_restart*, and *condor_vacate* did not handle the **-pool** command-line option correctly. The bug caused these commands to correctly query the central manager of the remote pool, and to incorrectly send the command to the central manager machine. This bug has now been fixed, and these tools no longer send the command to the central manager machine.

Known Bugs:

- None.

## Version 6.7.5

Release Notes:

- None.

New Features:

- Added DAG aborting feature – a DAG can be configured to abort immediately if a node exits with a given exit value.
- The dedicated scheduler can now preempt running MPI jobs from appropriately configured machines. See 3.12.8 for details.
- The MPI universe now supports submit files with multiple procs (queue commands), each with distinct requirements. This is useful for placing the head node of an MPI job on a specific machine, and the rest of the nodes elsewhere. See 2.10.5 for details.
- The *condor_negotiator* now publishes its own ClassAd to the *condor_collector* which includes the IP address and port where it is listening. This negotiator ClassAd can be viewed using the new **-negotiator** option with *condor_status*. In addition to removing an unnecessary fixed port for the *condor_negotiator*, this change corrects some problems with commands that attempted to communicate directly with the *condor_negotiator*. These bugs were first listed in the Known Bugs section of the 6.6.0 version history.

  To enable this feature and have the *condor_negotiator* listen on a dynamic port, you must comment out the `NEGOTIATOR_HOST` setting in your configuration file. The new example configuration files shipped with version 6.7.4 and later will already have this setting undefined. However, if you upgrade your binaries and retain an older copy of your configuration files, you should consider commenting out `NEGOTIATOR_HOST`.

  To disable this feature and have the *condor_negotiator* still listen on a well-known port, you can uncomment the `NEGOTIATOR_HOST` setting in the default configuration. For example:

  ```
  NEGOTIATOR_HOST = $(CONDOR_HOST)
  ```

Pools that are comprised of older versions of Condor and a 6.7.4 or later central manager machine should either continue to use their old `condor_config` file (which will still have `NEGOTIATOR_HOST` defined) or they should re-define the `NEGOTIATOR_HOST` setting in the new example configuration files which are used during the installation process.

- Added optional `DAGMAN_RETRY_SUBMIT_FIRST` configuration parameter that tells *condor_dagman* whether to immediately retry the submit if a node submit fails, or to put that job at the end of the ready jobs queue. The default is TRUE, which retries the failed submit before trying to submit any other jobs.
- The schedd now uses non-blocking connection attempts when contacting startds. This prevents the long (typically 40 second) hang of all schedd operations when the connection attempt does not complete, due to network problems.

Bugs Fixed:

- Fixed a performance problem with the standard universe when `gettimeofday()` is called in a very tight loop by the application.
- Fixed the default value of `OPSYS` in the MacOSX version of Condor. Once again, Condor reports `OSX` for all versions of MacOSX. This bug was introduced in version 6.7.3 of Condor.
- Fixed a bug in *condor_dagman* that caused it to be killed if the `DAGMAN_MAX_SUBMIT_ATTEMPTS` parameter was set to too high a value.
- Fixed a bug in *condor_gridmanager* that caused it to crash if the grid_monitor was activated.
- Fixed support for the getdents64() system call inside the standard universe on Linux and Solaris.
- Fixed a bug in *condor_dagman* that dealt incorrectly with the problem of Condor sometimes writing both a terminated and an aborted event for the same job. The spurious aborted event is now ignored.

Known Bugs:

- None.

## Version 6.7.3

Release Notes:

- This release contains all the bug fixes from the 6.6 stable series up to and including version 6.6.7, and some of the fixes that will be included in version 6.6.8. The bug fixes in version 6.6.8 that were not included in version 6.7.3 are listed in a separate section of the 6.6.8 version history.

New Features:

- Added Full Ports of Condor to Redhat Fedora Core 1, 2 and 3 on the 32-bit x86 architecture. Please read the Linux platform specific section 6.1.6 in this manual for more information on caveats with this port.
- Added a feature to *condor_dagman* that will allow VARS names to include numerics and underscores.
- Added optional `COLLECTOR_HOST_FOR_NEGOTIATOR` configuration parameter to indicate which *condor_collector* the *condor_negotiator* on this (local) host should query first. This is designed to improve negotiation performance.
- Added a new *condor_dagman* capability to allow the DAG to continue if it encounters a double run of the same node job (set the `DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION` parameter to true to do this).
- Added Condor-C: the "condor" grid_type. Condor-C allows jobs to be handed from one *condor_schedd* to another *condor_schedd*.
- Added **setup_here** option to *condor_glidein* for cases where direct installation is desired instead of submitting a setup job to the remote gatekeeper. (For example, this is useful when doing an installation onto AFS.)
- If `RemoteOwner` is exported via `STARTER_VM_EXPRS` into the ad of other virtual machines, the *condor_negotiator* automatically inserts `RemoteUserPrio` into the ad as well, so policy expressions can now take into account the priority of jobs running on other virtual machines on the same host.
- Linux 2.6 kernels do not update the access time for console devices, so Condor was unable to detect if there has been activity at the keyboard or mouse. As a work-around, Condor now polls /proc/interrupts to detect if the keyboard has requested attention. This does not work for USB keyboards or pseudo TTYs, so `ConsoleIdle` on 2.6 kernels will be wrong for some devices. Future versions of Condor or Linux may correct this.
- *condor_dagman* no longer removes the X509_USER_PROXY environment variable. This should allow users to set the environment variable before invoking *condor_submit_dag* and have the jobs submitted by *condor_dagman* correctly find the proxy file.

Bugs Fixed:

- Fixed a *condor_dagman* bug that could cause it to leave jobs running when aborting a DAG.
- Fixed a *condor_dagman* bug which, if its debug level was set to zero (silent), could cause it to to improperly recognize persistent *condor_submit* failures.

- Fixed a bug in Condor's file transfer mechanism that showed up when users tried to use streaming output for either STDOUT or STDERR. There were situations where Condor would attempt to transfer back the STDOUT or STDERR file from the execution host, even though these files didn't exist and all the data was already streamed back to the submit host. Now, if either `stream_output` or `stream_error` are set to true in the job submit description file, Condor will transfer any other output but will not attempt to transfer back STDOUT or STDERR.
- The Condor user log library (libcondorapi) now correctly handles execute events that lack a hostname.

Known Bugs:

- Unfortunately, the default `OPSYS` value for the MacOSX version of Condor was incorrectly changed in version 6.7.3. Condor used to always report `OSX`, but in version 6.7.3 it will report either `OSX10_2`, `OSX10_3`, or `OSX_UNK`. This is wrong, since Condor jobs submitted to any version of OSX should be able to run on any other version of OSX, and the above change needlessly partitions resources and complicates things for end-users. Therefore, anyone running version 6.7.3 on MacOSX is encouraged to add the following line to their global `condor_config` file:

  ```
  OPSYS = OSX
  ```

  If your pool is already running the new release, you can cause the above change to take effect by running the following command on your pool's central manager machine (or any machine listed in the `HOSTALLOW_ADMINISTRATOR` list) after you have changed the `OPSYS` value in your configuration:

  ```
  condor_reconfig -all
  ```

  However, if you have already submitted jobs to your pool with the old `OPSYS` value, the `Requirements` expression in those jobs will still refer to the incorrect value. In this case, you should either a) wait for the jobs to complete before making the above change, b) remove the jobs and resubmit them after you've made the change, or c) manually run *condor_qedit* on the jobs to change their `Requirements` expressions.
- When running in recovery mode on a DAG that has PRE scripts, *condor_dagman* may attempt more than the specified number of retries of a node (counting retries attempted during the first run of the DAG). This is because if a node fails because of the PRE script failing, that fact is not recorded in the log, so that retry is missed in recovery mode.

## Version 6.7.2

Release Notes:

- Condor Version 6.7.2 includes some bug fixes from Version 6.6.7, but none from Version 6.6.8.
- MPI users who are upgrading from previous versions of Condor to version 6.7.2 will need to modify the `MPI_CONDOR_RSH_PATH` configuration macro of their dedicated resource to be `$(LIBEXEC)` instead of `$(SBIN)`. Users who are installing Condor version 6.7.2 for the first time will not need to make any changes.

New Features:

- Added an `INCLUDE` configuration file variable to define the location of header files shipped with Condor that are currently needed to be included when compiling Condor APIs. When `INCLUDE` is defined, *condor_config_val* can be used to list header files.
- A Condor pool can now support multiple Collectors. This should improve stability due to automatic failover. All daemons will now send updates to ALL of the specified collectors. All daemons/tools will query the Collectors in sequence, until an appropriate response is received. Thus if one (or more) of the Collectors are down, the pool will continue to function normally, as long as there is at least one functioning Collector. You can specify multiple (comma-separated) collector host (and port) addresses in the `COLLECTOR_HOST` entry in the configuration file. A given *condor_master* can only run one Collector.
- When the *condor_master* is started with the **-r** option to indicate that it should quite after a period of time, the *condor_startd* will now indicate how much time is remaining before it exits. It does this by advertising TimeToLive in the machine ClassAd.
- Added new macro `JOB_START_COUNT` that works in conjunction with existing macro `JOB_START_DELAY` to throttle job starts. Together, this macro pair provides greater flexibility tuning job start rate given available *condor_schedd* performance.
- Added a `LIBEXEC` directory to the install process. Support commands that the Condor system needs will be added to this directory in future releases. This directory should not be added to a user or system-wide path.
- Added the ability to decide for each file that condor transfers whether it should be encrypted or not, using encrypt_input_files, dont_encrypt_input_files, encrypt output files, and dont_encrypt_output_files in the job's submit file.
- Added DISABLE_AUTHENTICATION_IP_CHECK which will work around problems on dual-homed machines where the IP address is reported incorrectly to condor. This is particularly a problem when using Kerberos on multi-homed machines.

Bugs Fixed:

- Fixed a bug on Linux systems caused by both Condor and the Linux distribution having a library file called `libc.a`. The problem caused the link step to fail on Condor API programs. The evaluation order to determine the location of library files caused use of the wrong file, given the duplicate naming. The bug is fixed by renaming the Condor library files.

- When the *condor_startd* is evaluating the state of each virtual machine (VM), it now refreshes any ClassAd attributes which are shared from other virtual machines (using STARTD_VM_EXPRS) before it tries to evaluate. This way, if a given VM changes its state, all other VMs will immediately see this state change.
- Fixed a bug where you couldn't transfer input files larger than 2 gigabytes.
- Condor can now detect the size of memory on a Linux machine with the 2.6 kernel.
- JAR files specified in the submit file were not being transfered along with the job unless they were also explicitly placed in the list of input files to transfer. Now, the JAR files are implicitly added to the list of input files to transfer.

Known Bugs:

- None.

## Version 6.7.1

Release Notes:

- Version 6.7.1 contains all of the features, ports, and bug fixes from the previous stable series, up to and including version 6.6.6. There are a few additional bugs that have been fixed in the 6.6.x stable series which have not yet been released, but which will appear in version 6.6.7. These bug fixes have been included in version 6.7.1, and appear in the "Bugs fixes included from version 6.6.7" list below. In addition, a number of new features and some bug fixes have been made, which are described below in more detail.
- None.

New Features:

- Added an option to DAGMan's retry ability. If a DAG specifies something like "RETRY job 10 unless-exit 9", then the retries will only happen if the node doesn't exit with a value of 9.
- Condor-G can now submit jobs to Globus 3.2 (WS) (for jobs with `universe = grid`, `grid_type = gt3`). Submitting to Globus 3.0 (as in Condor 6.7.0) is no longer supported. Submitting to pre-WS Globus (2.x) is still supported (`grid_type = gt2`).
- Added new startd policy expression MaxJobRetirementTime. This specifies the maximum amount of time (in seconds) that the startd is willing to wait for a job to finish on its own when the startd needs to preempt the job (for owner preemption, negotiator preemption, or graceful startd shutdown).

- Added -peaceful shutdown/restart mode. This will shut down the startd without killing any jobs, effectively treating both `MaxJobRetirementTime` and `GRACEFUL_SHUTDOWN_TIMEOUT` as infinite. The default shutdown/restart mode is still -graceful, which behaves according to whatever `MaxJobRetirementTime` and `GRACEFUL_SHUTDOWN_TIMEOUT` are. The behavior of -fast mode is unchanged; it kills jobs immediately, regardless of the other timeout settings.
- Jobs can now be submitted as "noop" jobs. Jobs submitted with `noop_job = true` will not be executed by Condor, and instead will immediately have a terminate event written to the job log file and removed from the queue. This is useful for DAGs where the pre-script determines the job should not run.
- Added preliminary support for the Tool Daemon Protocol (TDP) into Condor. This protocol is still under development, but the goal is to provide a generic way for scheduling systems (daemons) to interact with monitoring tools. Assuming this protocol is adopted by other scheduling systems and by various monitoring tools, it would allow arbitrary combinations of tools and schedulers to co-exist, function properly, and provide monitoring services for jobs running under the schedulers. This initial support allows users to specify a "tool" that should be spawned along-side their regular Condor job. On Linux, the ability to have the batch Condor job suspend immediately upon start-up is also implemented, which allows a monitoring tool to attach with ptrace() before the job's main() function is called.

Bugs Fixed:

- Fixed a significant memory leak in the *condor_schedd* that was introduced in version 6.7.0. In 6.7.0, the *condor_schedd* would leak a copy of ClassAd for every job it tried to spawn (on average, around 2000 bytes per job).
- Fixed the bugs in Condor's MPI support that were introduced in version 6.7.0. Condor now supports MPI jobs linked with MPICH 1.2.4 and older. Improved Condor's log messages and email notifications when MPI jobs run on multiple virtual machines (the messages now include the appropriate "vmX" identifier, not just the hostname). Unfortunately, due to changes in MPICH between version 1.2.4 and 1.2.5, Condor's MPI support is not compatible with MPICH 1.2.5. We will be addressing this problem in a future release.

Bugs fixes included from version 6.6.7:

- Fixed an important bug in the low-level code that Condor uses to transfer files across a network. There were certain temporary failure cases that were being treated as permanent, fatal errors. This resulted in file transfers that aborted prematurely, causing jobs to needlessly re-run. The code now gracefully recovers from these temporary errors. This should significantly help throughput for some sites, particularly ones that transfer very large files as output from their jobs.
- Fixed a number of bugs in the **-format** option to *condor_q* and *condor_status*. Now, these tools will properly handle printing boolean expressions in all cases. Previously, depending

on how the boolean evaluated, either the expression was printed, or the tool could crash. Furthermore, the tools do a better job of handling the different types of format conversion strings and printing out the appropriate value. For example, if a user tries to print out a boolean attribute with `condor_status -format "%d\n" HasFileTransfer`, the *condor_status* tool will evaluate `HasFiletransfer` and print either a 0 or a 1 (FALSE or TRUE). If, on the other hand, a user tries to print out a boolean attribute with `condor_status -format "%s\n" HasFileTransfer`, the *condor_status* tool will print out the string "FALSE" or "TRUE" as appropriate.

- The ClassAd attribute scope resolution prefixes, `MY.` and `TARGET.`, are no longer case sensitive.
- *condor_dagman* now does better checking for inconsistent events (such as getting multiple terminate events for a single job). This checking can be disabled with the **-NoEventChecks** command-line option.

Known Bugs:

- None.

## Version 6.7.0

Release Notes:

- Version 6.7.0 contains all of the features, ports, and bug fixes from the previous stable series, up to and including version 6.6.4. In addition, a number of new features and some bug fixes have been made, which are described below in more detail.

New Features:

- Added support for vanilla and Java jobs to reconnect when the connection between the submitting and execution nodes is lost for any reason. Possible reasons for this disconnect include: network outages, rebooting the submit machine, restarting the Condor daemons on the submit machine, etc. If the execution machine is rebooted or the Condor daemons are restarted, reconnection is not possible. To take advantage of this reconnect feature, jobs must be submitted with a `JobLeaseDuration`. There are new events in the UserLog related to disconnect and reconnect.
- Added a new Condor tool, *condor_vacate_job*. This command is similar to *condor_vacate*, except the kinds of arguments it takes define jobs in a job queue, not machines to vacate. For example, a user can vacate a specific job id, all the jobs in a given cluster, all the jobs matching a job queue constraint, or even all jobs owned by that user. The owner of a job can always vacate their own jobs, regardless of the pool security policy controlling *condor_vacate* (which is an administrative command which acts directly on machines). See the new command reference, section 9 on page 762 for details.

- Added a new "High Availability" service to the *condor_master*. You can now specify a daemon which can have "fail over" capabilities (i.e. the master on another machine can start a matching daemon if the first one fails). Currently, this is only available over a shared file system (i.e. NFS), and has only been tested for the *condor_schedd*.
- Scheduler universe jobs on UNIX can now specify a `HoldKillSig`, the signal that should be sent when the job is put on hold. If not specified, the default is to use the `KillSig`, and if that is not defined, the job will be sent a SIGTERM. The submit file keyword to use for defining this signal is `hold_kill_sig`, for example, `hold_kill_sig = SIGUSR1`.
- The *condor_startd* can now support policies on SMP machines where each virtual machine (VM) has knowledge of the other VMs on the same host. For example, if a job starts running on one of the VMs, a job running on another VM could immediately be suspended. This is accomplished by using the new configuration variable `STARTD_VM_EXPRS` , which is a list of ClassAd attribute names that should be shared across all VMs on the machine. For each VM on the machine, every attribute in this list is looked up in the VM-specific machine ClassAd, the attribute name is given a prefix indicating what VM it came from, and then inserted into the machine ClassAds of all the other VMs.
- The *condor_startd* publishes four new attributes into the machine ClassAds it generates when it is in the Claimed state: `TotalJobRunTime`, `TotalJobSuspendTime`, `TotalClaimRunTime`, `TotalClaimSuspendTime`. These attributes keep track of the total time the resource was either running a job (in the Busy activity) or had a job suspended, regardless of how many suspend/resume cycles the job went through. The first two attributes (with "Job" in the name) keep track for a single job (i.e. since the last time the resource was Claimed/Idle). The last two attributes (with "Claim" in the name) keep track of these totals across all jobs that ran under the same claim (i.e. since the last state change into the Claimed state).
- Added a **-num** option to the *condor_wait* tool to wait for a specified number of jobs to finish.
- Added a configuration option `STARTER_JOB_ENVIRONMENT` so the admin can configure the default environment inherited by user jobs.
- Added a (configurable, defaults to off) feature to the *condor_schedd* to allow backup the spool file before doing anything else.
- The "Continuous" option of the *condor_startd* "cron" jobs is being deprecated. It's being replaced by two new options which control separate aspects of it's behavior:
  - "WaitForExit" specifies the "exit timing" mode
  - "ReConfig" specifies that the job can handle SIGHUPs, and it should be sent a SIGHUP when the *condor_startd* is reconfigured.
- A lot of the items logged by the *condor_startd* "cron" logic, changed to D_FULLDEBUG (from D_ALWAYS), etc.
- Added `NEGOTIATOR_PRE_JOB_RANK` and `NEGOTIATOR_POST_JOB_RANK` . These expressions are applied respectively before and after the user-supplied job rank when deciding

which of the possible matches to choose. (The existing expression `PREEMPTION_RANK` is applied after `NEGOTIATOR_POST_JOB_RANK` .) The pool administrator may use these expressions to steer jobs in ways that improve the overall performance of the pool. For example, using the pre job rank, preemption may be avoided as long as there are idle machines, even when the user-supplied rank expression prefers a machine that happens to be busy. Using the post job rank, one could steer jobs towards machines that are known to be dedicated to batch jobs, or one could enforce breadth-first instead of depth-first filling of a cluster of multiprocessor machines.

- Added the ability for Condor to transfer files larger than 2G on platforms that support large files. This works automatically for transferred executables, input files and output files.
- Added the ability for jobs to stream back standard input, output, and error files while running. This is activated by the **stream_input**, **stream_output**, and **stream_error** options to *condor_submit*. Note that this feature is incompatible with the new feature described above where the shadow and starter can reconnect in certain circumstances.
- Added support for vanilla jobs to be mirrored on a second *condor_schedd*. The jobs are submitted to the second *condor_schedd* on hold and will be released if the second *condor_schedd* hasn't heard from the first *condor_schedd* (actually, a *condor_gridmanager* running under the first *condor_schedd*) for a configurable amount of time. Once the second *condor_schedd* releases the jobs, the first *condor_schedd* acts as a mirror, reflecting the state of the jobs on the second *condor_schedd*. To use this mirroring feature, jobs must be submitted with a `mirror_schedd` parameter in the submit file and require no file transfer.

Bugs Fixed:

- Fixed a bug in the *condor_startd* "cron" logic which caused the *condor_startd* to except when trying to delete a job that could never be run (i.e. invalid executable, etc).
- Fixed a bug in *condor_startd* "cron" logic which caused it to not detect when the starting of a "job" failed.
- Fixed several bugs in the reconfiguration handling of the *condor_startd* "cron" logic. In particular, even if the job has the "reconfig" option set (or "continuous"), the job(s) won't be sent a SIGHUP when the startd first starts, or when the job itself is first run (until it outputs its first output block, defined by the "-" separator).

Known Bugs:

- Condor's MPI support (for MPICH 1.2.4) was broken by other changes in version 6.7.0. Support for MPI jobs will return in Condor version 6.7.1.

| *Architecture* | *Operating System* |
|---|---|
| Hewlett Packard PA-RISC (both PA7000 and PA8000 series) | HPUX 10.20 |
| Sun SPARC Sun4m,Sun4c, Sun UltraSPARC | Solaris 2.6, 2.7, 8, 9 |
| Silicon Graphics MIPS (R5000, R8000, R10000) | IRIX 6.5 (clipped) |
| Intel x86 | Red Hat Linux 7.1, 7.2, 7.3, 8.0<br>Red Hat Linux 9<br>Windows 2000 Professional and Server, 2003 Server (clipped<br>Windows XP Professional (clipped) |
| ALPHA | Digital Unix 4.0<br>Red Hat Linux 7.1, 7.2, 7.3 (clipped)<br>Tru64 5.1 (clipped) |
| PowerPC | Macintosh OS X (clipped)<br>AIX 5.2L (clipped) |
| Itanium | Red Hat Linux 7.1, 7.2, 7.3 (clipped)<br>SuSE Linux Enterprise 8.1 (clipped) |

Table 8.1: Condor 6.7.0 supported platforms

## 8.5 Stable Release Series 6.6

This is a stable release series of Condor. It is based on the 6.5 development series. All new features added or bugs fixed in the 6.5 series are available in the 6.6 series. The details of each version are described below.

### 8.5.1 Version 6.6.12

Release Notes:

- Contains only a couple bug fixes.

Bugs fixed that are included in version 6.7.19:

- None.

Bugs fixes irrelevant to the 6.7 series:

- Fixed a bug which caused the *condor_collector* incorrectly handle Collector ads in which the `Machine` attribute is missing, or Storage ads in which the `Name` is missing. In these cases, a *condor_collector* running on some platforms (notably, Solaris) could crash.

Known Bugs:

- None.

## Version 6.6.11

Release Notes:

- A security team at UW-Madison is conducting an onging security audit of the Condor system and has identified a few important vulnerabilities. Condor versions 6.6.11 and 6.7.18 fix these security problems and other bugs. There have been no reported exploits, but all sites are urged to upgrade immediately.

  The Condor Team will publish detailed reports of these vulnerabilities on 2006-04-24, 4 weeks from the date when the fixes were first released (2006-03-27). This will allow all sites time to upgrade before enough information to exploit these bugs is widely available.

Security Bugs Fixed:

- Bugs in previous versions of Condor could allow any user who can submit jobs on a machine to gain access to the "condor" account (or whatever non-privileged user the Condor daemons are running as). This bug can not be exploited remotely, only by users already logged onto a submit machine in the Condor pool.
- The security of the "condor_config_val -set" feature was found to be insufficient, so this feature is now disabled by default. There are new configuration settings to enable this feature in a secure manner. Please read the descriptions of `ENABLE_RUNTIME_CONFIG`, `ENABLE_PERSISTENT_CONFIG` and `PERSISTENT_CONFIG_DIR` in the example configuration file shipped with the latest Condor releases, or in section 3.3.5 on page 155.

Other bugs fixed that are included in version 6.7.18:

- Fixed a bug which could cause the *condor_collector* to crash when it receives certain types of malformed ads.
- Fixed a bug which caused the *condor_collector* incorrectly handle ads in which the `UpdateInterval` attribute is set. In particular, the previous versions of the *condor_collector* will use the `UpdateInterval` value as the maximum *lifetime* of the ad when aging the ads, which could cause it to remove the ad prematurely. The *condor_collector* now looks at the `ClassAdLifetime` attribute, and uses its value (if set). NOTE: No current Condor daemons are publishing either of these attributes, but may do so in the future.

Bugs fixed that are included in version 6.7.14:

- Fixed a rare problem in the *condor_negotiator* where a poorly formed classad from a single *condor_schedd* could halt negotiation for the entire pool. This poorly formed ad could only

happen in extrememly rare circumstances, but it was possible. Now, the *condor_negotiator* will simply ignore poorly formed classads and continue to negotiate with any other *condor_schedd* in the system that has idle jobs.

- Fixed a bug which caused log messages which should contain "PRIV_USER_FINAL" to be "PRIV_USER_FINALPRIV_FILE_OWNER". It's also possible that this same bug could cause crashes if any daemon attempts to log a message which would refer to "PRIV_FILE_OWNER".
- Fixed a bug which caused the *condor_starter* to exit with an error when the sum total of the file transfer size exceeded 2G. This, in turn, caused a "shadow exeception", and the job would fail.

Bugs fixed that are included in version 6.7.11:

- In very rare cases, the *condor_startd* could get into an infinite loop if a job it was managing was suspended and then there were fatal errors trying to send commands to evict the corresponding *condor_starter*. This bug has been fixed, and the *condor_startd* will now correctly recover (and cleanup all processes) if it fails to send commands to a starter managing a suspended job.
- Condor on Solaris has been patched to work around a Solaris stdio limitation of 255 maximum file descriptors. Before this patch, heavily loaded Condor daemons running on Solaris, particularly the *condor_schedd*, could exit complaining about lack of file descriptors for dprintf.
- Fixed a bug where the *condor_starter* would follow symbolic links to directories, when calculating job disk usage. This could cause an incorrect job disk usage calculation, or hang the starter upon encountering an infinite directory loop. This bug only affected Unix platforms.
- For Globus jobs, the Rematch expression is now evaluated when a submit fails (in addition to when a submit commit times out).
- Fixed a bug that caused the *condor_gridmanager* to go into an infinite loop if an entry in the job's environment string was missing an equals sign.

Bugs fixed that are included in version 6.7.9:

- Fixed a bug where the *condor_startd* would erroneously compute the console idle time utilizing a file called /proc/interrupts on unix machines that were not linux.
- Fixed a bug where the *condor_negotiator* might dump core if it was reconfiged in the middle of a negotiation cycle.
- Fixed a bug where the *condor_negotiator* might dump core if a startd had a name longer than 63 bytes.
- Fixed a bug that could cause *condor_userprio* to crash if the data it gets back from the *condor_negotiator* is invalid.

- Fixed a bug where DEFAULT_PRIO_FACTOR was ignored if ACCOUNTANT_LOCAL_DOMAIN was not defined.

Bugs fixes irrelevant to the 6.7 series:

- Added the **-NoEventChecks** and the **-AllowLogError** command-line flags to *condor_submit_dag* and the *condor_submit_dag* man page (they were already in *condor_dagman*). Added **-r** and **-debug** to the *condor_submit_dag* man page (they were already in *condor_submit_dag*, just not documented).
- Made command-line arguments case insensitive in the Windows version of *condor_submit_dag*; also fixed log file checks in that version.

Known Bugs:

- A bug has been found which can cause a *condor_collector* to crash on some platforms (notably, Solaris). This can happen if the *condor_collector* receives a Collector ad in which the Machine attribute is missing, or a Storage ad in which the Name is missing. There is no security threat involved in either case.

## Version 6.6.10

Release Notes:

- Most of the fixes included in this release were also included in version 6.7.7 (see below).
- The QUEUE_CLEAN_INTERVAL timer is reset during a *condor_schedd* reconfig only if this timer value has been changed. Previously, the timer was reset during all *condor_schedd* reconfigs, which could prevent the job_queue.log file from being cleaned. Note that this timer is always reset upon a *condor_schedd* startup. See the related change for truncating the job_queue.log below, for this same release.
- Previously, the *condor_schedd* would over-react and exit if it tried to send a user email and SMTP_SERVER was undefined; now it simply prints an error in the SchedLog and moves on.

Bugs fixed that are included in version 6.7.7:

- Fixed a bug that could cause the file job_queue.log in the Condor SPOOL directory to grow unnecessarily large, thereby slowing down the startup and/or shutdown times for the *condor_schedd* daemon.
- Fixed a critical bug where the console idle time for PS/2 keyboards and mice was not being updated correctly.

- Fixed a bug in the *condor_collector* that could cause it to crash when parsing certain types of invalid ClassAds. In particular, if a Machine, Schedd or License ClassAd sent to the *condor_collector* has an IP address field which is empty (which should never happen), the *condor_collector* will crash.
- Fixed some bugs in how the *condor_schedd* handles a graceful shutdown (either because of a *condor_off*) or a SIGTERM on UNIX):
  - There was a minor bug if JOB_START_DELAY was set to 0 that would prevent the *condor_schedd* from correctly cleaning up during graceful shutdown. Now, the *condor_schedd* will properly shutdown, even if JOB_START_DELAY is set to 0.
  - Fixed a bug when there are scheduler universe jobs that were recently submitted to the queue. Previously, the shutdown code would not evict scheduler universe jobs that had been submitted since the last SCHEDD_INTERVAL (which defaults to 5 minutes). So, if a user submitted a scheduler universe job and then someone shutdown Condor on that machine, the *condor_schedd* would wait until the next SCHEDD_INTERVAL had elapsed before evicting the job. Now, the schedd will always attempt to evict scheduler universe jobs during a shutdown, without waiting for this interval to pass.
- A number of Windows-specific bugs were fixed:
  - It was possible under certain circumstances for execute directories to not be cleaned up properly. This has been fixed.
  - Certain Asian locales would cause the *condor_starter* to crash due to character translation problems. This has been fixed.
  - Condor will now properly report memory sizes that exceed 2 GB.
  - The *condor_starter* would be unable to run jobs if the LOG path had a period (.) in it. This has been fixed.
  - The *condor_startd* would leak memory, especially on SMP machines. This has been fixed.
  - The *condor_master* would crash immediately on Windows 2003 Server if the firewall was enabled. This has been fixed.
- Fixed a bug in *condor_dagman* that could cause *condor_dagman* to fail an assertion if PRE or POST scripts are throttled with the **-maxpre** or **-maxpost** *condor_submit_dag* command line flags.

Bugs fixed that are NOT included in version 6.7.7:

- Fixed a bug where enabling the grid_monitor for any globus job handled by something other than a hard-coded list of jobmanager names would cause the job to stay idle forever. The hard-coded list of jobmanager names was: condor, fork, lsf, pbs, and remote. A jobmanager by any other name (e.g. condor_rh9, or lcgpbs) would cause the problem. This bug was originally fixed in internal releases of 6.7.0, but it was reintroduced by mistake in all public releases.

- Fix the way *condor_version* handles command line arguments (there were a number of problems and inconsistencies) and added a **-help** option and usage message.
- Fixed some memory leaks in the *condor_startd* that would be induced by calling *condor_reconfig* or *condor_status* **-d**.
- By design, Condor daemons will exit if their parent process exits. On Windows, a bug introduced in v6.5.x series broke this behavior. This is now fixed.
- On Windows, users would often observe the *condor_master* failing to add exceptions for the Condor daemons to the Windows Firewall on Windows XP SP2 or Windows 2003 Server SP1. The *condor_master* will now retry for a longer period of time to add these exceptions, and the number of retries has now been made configurable. See section 3.3.9 on page 168 for more information.

Known Bugs:

- None.

## Version 6.6.9

Release Notes:

- Most of the fixes included in this release were also included in version 6.7.5. However, at the end of this section, a few fixes that were added to 6.6.9 after 6.7.5 was released are mentioned separately.

Bugs fixed that are included in version 6.7.5:

- Fixed a security bug in the *condor_schedd* that could enable a maliciously modified *condor_submit* tool to overwrite files in the Condor `SPOOL` subdirectory, including the job queue.
- Fixed a bug where under very pathological file permission failure conditions with a standard universe job, there would be a cycle of an execute event followed by a termination event in the user log when the job had not actually ran.

Bugs fixed that are NOT included in version 6.7.5:

- Fixed a memory management bug introduced in version 6.6.8 that could result in deallocated memory being referenced after a child process forked from a Condor daemon exits.
- Fixed bugs in some Condor tools that failed to locate *condor_startd* daemons that contained multiple `@` signs in their `Name` attribute. For example, a virtual machine from a multiple-CPU *condor_startd* spawned using glidein would have the name: `vm1@[pid]@[hostname]`. All Condor tools that need to communicate with a *condor_startd* like this will now succeed.

- Removed a fixed-length buffer in the code that handled the SUBSYS_EXPRS config file setting. Previously, if any attributes referred to were larger than approximate 1000 bytes, Condor daemons would crash. Now, there is no limit to the size of the attributes listed in SUBSYS_EXPRS. For more information about this setting, see section 3.3.5 on page 156.
- Fixed a bug which would cause Condor to fail to cache user GID information and potentially overwhelm NIS servers.
- Fixed another bug which could cause UDP machine updates to be dropped by the *condor_collector*.

Known Bugs:

- If a DAG node has both retries and a POST script, and the actual Condor job for the node fails, the POST script is not run except after the last retry of the job (or if the job succeeds). (The POST script should be run each time the node job is run, whether the job succeeds or not.)
- Occasionally, Condor generates both a terminated event and an aborted event for a job that is aborted. If this happens for a DAG node job, *condor_dagman* considers this an error and aborts the DAG. If you run into this problem, you can avoid the abort by adding the **-NoEventChecks** flag to argument list in the *condor_dagman* submit file generated by *condor_submit_dag* (you have to do *condor_submit_dag -no_submit* and hand-edit the resulting submit file). However, if you get the double events on a node that has retries, *condor_dagman* will assert. The only fix for this is to upgrade to a 6.7.5 or newer *condor_dagman*. You can do this by simply installing a newer *condor_dagman* executable, without any other changes to your Condor installation. It is fine to run a 6.7 *condor_dagman* on a 6.6 Condor installation.
- 
- In a DAG, if a node job generates an executable error event, the DAG is aborted. This can be worked around by adding the **-NoEventChecks** flag to argument list in the *condor_dagman* submit file generated by *condor_submit_dag* (you have to do *condor_submit_dag -no_submit* and hand-edit the resulting submit file).

## Version 6.6.8

Release Notes:

- Most of the fixes included in this release were also included in version 6.7.3. However, at the end of this section, a few fixes that were added to 6.6.8 after 6.7.3 was released are mentioned separately.

New Features:

- None.

Bugs Fixed:

- In version 6.6.7, we fixed bugs related to the **-format** option to various Condor tools. However, some sites were using **-format** in ways we did not expect, by not specifying any 'string at all. This used to work, given the old buggy code that handled **-format**, but the changes in version 6.6.7 broke this, and format strings without a 'Now, if the format string does not contain a 'the attribute name which follows it is once again ignored, and the format string is printed directly without any modification. For example, to print out the machine's `Name` (always defined) and the `RemoteUser` (only defined if the machine is claimed), and always print a newline (to keep the formatting legible), this command will now work:

```
% condor_status -f "%s " Name -f "%s " RemoteUser -f "\n" bogus
bird.cs.wisc.edu biguser@raven.cs.wisc.edu
condor.cs.wisc.edu
dodo.cs.wisc.edu biguser@raven.cs.wisc.edu
lark.cs.wisc.edu biguser@raven.cs.wisc.edu
raven.cs.wisc.edu
...
```

- Windows bug fixes:
  - Fixed a bug in that would cause Condor to fail to gracefully shutdown user jobs that are console applications (including batch scripts).
  - Fixed an issue that would cause *condor_store_cred* to fail if the user did not have `NETWORK` logon rights.
  - *condor_store_cred* **query** command would appear to succeed, even if the stored credential was invalid (e.g. the password was changed but the password stash was not updated). This has been fixed.
  - Fixed a bug that would cause the *condor_startd* to crash under certain conditions during job eviction. This bug was introduced in Condor version 6.6.6.
  - Fixed a bug that would cause *condor_dagman* to crash if it was submitted as a non-Administrator user.
  - Fixed a bug that would cause Condor to occasionally kill processes that didn't belong to it during job eviction or daemon restarts.
  - On startup, the *condor_master* would occasionally fail to add the daemons to the Windows XP firewall exception list because of a race with the Windows SharedAccess service. This bug has been fixed.
  - If a user submitted a job with an invalid executable, the starter would often wedge until the job was preempted. Now, the starter attempts to detect invalid executables and prevent wedging.

  - Fixed issues that would cause *condor_startd* to "disappear" from the pool because of dropped machine ad updates. This fix applies to all platforms, but the symptoms were exhibited predominantly on Windows machines.
  - Fixed a bug that could cause `HIGHPORT` and `LOWPORT` parameters to be ignored if a Windows machine ran for several weeks without being rebooted.

- Starting with RedHat 9, newer versions of Linux began to produce core files named `core.<pid>`. This broke functionality in Condor that managed and transferred back any core file created by the job, since the *condor_starter* was unable to locate the proper file. Now, Condor will correctly transfer back core files, even if they are created as `core.<pid>`. This functionality works in all universes, and is independent of Condor's file transfer mechanism.
- Fixed a bug that was causing *condor_startd* to consume large amounts of memory over long periods of time.
- Fixed a bug that was causing *condor_startd* to fail to start up with the message, "caInsert: Can't insert CpuBusy into target ClassAd."
- Fixed a long-standing bug in Condor regarding the configuration settings `LOWPORT` and `HIGHPORT` . When these were enabled (to restrict Condor's port usage to a specified range), Condor would fail to set the `SO_KEEPALIVE` option on sockets it created. This meant that in the case of a hard machine failure (such as a sudden power outage, etc) on one machine, Condor daemons communicating with that machine would never notice it had died. Now, the `SO_KEEPALIVE` option is properly set on all sockets, even with `LOWPORT` and `HIGHPORT` defined.
- Fixed a bug that caused *condor_rm* **-forcex** to not remove jobs that make use of `leave_in_queue`. If invoked using a cluster id, username, or constraint expression, *condor_rm* would report success but the jobs would remain in the queue. Now, the jobs will leave the queue.
- When a held job is released, job ad attributes HoldReasonCode and HoldReasonSubCode are now properly moved to LastHoldReasonCode and LastHoldReasonSubCode.
- Fixed a bug that would cause the `RemoveReason` attribute for a job to be set incorrectly in some circumstances. Specifically, this was when a job was not running and a `periodic_remove` expression caused the job to be cancelled.
- Fixed *condor_submit* such that submit description file commands written with syntax both of `ThisStyle` and `this_style` will work.
- Fixed a very rare but serious bug in Condor that was originally introduced in version 6.3.0. Under exceptional circumstances (a very heavily loaded machine where a huge number of processes are being spawned all the time, and where the *condor_schedd* is managing many thousands of jobs in the queue), it was possible for the *condor_schedd* to run a job twice. We have fixed the underlying problem that lead to the *condor_schedd* making this mistake, rendering this error impossible.

- Fixed a bug that occurred when submitted Condor-G jobs while using the grid monitor. If the grid job monitor returned a FAILED status for a job while the jobmanager is asleep, the *condor_gridmanager* could sometimes end up in a loop, continuously restarting the remote Globus jobmanager then putting it back to sleep.

Known Bugs:

- None

Bugs fixed that are not included in version 6.7.3:

- Fixed a discrepancy in the `SUBSYS_ADDRESS_FILE` setting. Previously, this setting did not work for `SUBSYS` values of `COLLECTOR` or `NEGOTIATOR` (for example, defining `COLLECTOR_ADDRESS_FILE` had no effect). Now, if either of these is defined in the configuration file, the corresponding Condor daemon will write out the address and port it is using to the specified file. Normally, the *condor_collector* and *condor_negotiator* listen on a well-known, fixed port. However, on single-machine, Personal Condor installations, these address files allow all of the Condor daemons and tools to locate the *condor_collector* and *condor_negotiator*, even if they are using a dynamically assigned port. For more information about the `SUBSYS_ADDRESS_FILE` setting, please see the description in section 3.3.5 on page 156. For more information about using non-standard ports for the *condor_collector* and *condor_negotiator*, please see the description of "Non Standard Ports for Central Managers" in section 3.7.1 on page 295.

## Version 6.6.7

Release Notes:

- None.

New Features:

- Added a feature to the *condor_master* which automatically adds the Condor daemons to the Windows Firewall exception list. This only applies to machines running Windows XP SP2.

Bugs Fixed:

- Fixed a bug specific to Windows that could cause, in rare occurrences due to a race condition, Condor to fail to properly signal the job to suspend, continue, or preempt.
- When Condor transfers the job executable using the file transfer mechanism, it used to leave the binary sitting as a world-writable file inside the execute directory on UNIX. Now, executable files transferred by Condor have the proper permissions (mode 0755).

- Fixed an important bug in the low-level code that Condor uses to transfer files across a network. There were certain temporary failure cases that were being treated as permanent, fatal errors. This resulted in file transfers that aborted prematurely, causing jobs to needlessly re-run. The code now gracefully recovers from these temporary errors. This should significantly help throughput for some sites, particularly ones that transfer very large files as output from their jobs.
- Fixed a bug in the file transfer mechanism which caused segmentation faults when very long input/output/intermediate file lists were used.
- Fixed a number of bugs in the **-format** option to *condor_q* and *condor_status*. Now, these tools will properly handle printing boolean expressions in all cases. Previously, depending on how the boolean evaluated, either the expression was printed, or the tool could crash. Furthermore, the tools do a better job of handling the different types of format conversion strings and printing out the appropriate value. For example, if a user tries to print out a boolean attribute with `condor_status -format "%d\n" HasFileTransfer`, the *condor_status* tool will evaluate `HasFiletransfer` and print either a 0 or a 1 (FALSE or TRUE). If, on the other hand, a user tries to print out a boolean attribute with `condor_status -format "%s\n" HasFileTransfer`, the *condor_status* tool will print out the string "FALSE" or "TRUE" as appropriate.
- The ClassAd attribute scope resolution prefixes, `MY.` and `TARGET.`, are no longer case sensitive.
- *condor_dagman* now generates a fatal error if any node submit files are missing the log file attribute. This behavior can be overridden with the **-AllowLogError** command-line option.
- *condor_dagman* now does better checking for inconsistent events (such as getting multiple terminate events for a single job). This checking can be disabled with the **-NoEventChecks** command-line option.
- Under Tru64, Condor would sometimes fail to start a job while setting the resource limits on behalf of the job. This error appears to be the result of a kernel issue. A workaround has been implemented which will leave the limits of the job unmodified and run the job when this specific error situation arises.
- On Windows, occasionally Condor would exhibit erratic behavior when a machine resumes from sleeping. This has been fixed.
- On Windows, occasionally Condor would fail to bind to any available interfaces due to a mishandling of a function return value. This has been fixed.

Known Bugs:

- None.

### Version 6.6.6

Release Notes:

- A *condor_dagman* job will fail and report a cycle in the DAG when XML logs are used in a single or multiple log format. The Post Script completion event does not get converted to XML and Dagman never sees them complete or fail because of the format of the event.

New Features:

- The checkpoint server has moved from contrib module status to being a normal part of Condor.
- When the first start running, all Condor daemons will now try to print to their log file the full path to the binary they are executing. Unfortunately, we can only reliably get this information on Linux, Solaris, MacOSX, and Windows platforms. On other platforms, this information will only be printed to the log file in certain cases that depend on how the daemon was invoked. This new feature was added to aid in debugging problems where sites were not running the version of the Condor daemons they thought they were due to problems in custom-built startup scripts.
- *condor_wait* is now available in the Windows port.
- Added a fix to the accountant that allows users to specify user priorities with *condor_userprio* before any jobs have been submitted.
- Added support for running batch files under Windows when using the `STARTD_CRON` or `USER_JOB_WRAPPER` attributes.
- Moved from Globus 2.2.2 to Globus 2.2.4 for Condor-G, except for the DUX 4.0f platform.

Bugs Fixed:

- Windows bug fixes:
  - Fixed a bug which could cause Condor to kill processes that aren't related to Condor or the job it was running at the time.
  - Fixed a problem that could cause daemons or tools to crash when they looked up information about processes running on the system.
  - Fixed a problem with the collector dropping TCP updates with pools larger than roughly 20 machines. This issue only occurs with `UPDATE_COLLECTOR_WITH_TCP` enabled.
  - Fixed an issue with *condor_store_cred* reporting success when in fact under certain circumstances the store command actually failed.
  - Removed *condor_kbdd_dll*. It is no longer used.
  - Fixed an issue with *condor_birdwatcher* that caused it to leak resource handles.

  - Fixed an issue with the Windows port of *condor_dagman* that would cause it to crash when POST scripts were used.

- Fixed a bug where the environment of jobs in any universe could be corrupted.
- The *condor_startd* now properly cleans up execute directories on root-squashed NFS mounts.
- Fixed a problem where the *condor_starter* could crash if the job it was running used Condor's file transfer mechanism and the full path names to the job's files became longer than a few hundred characters.
- The `image_size` attribute of a job on Mac OS X is much closer to the values that *ps* returns. Previously it would be highly inflated.
- Fixed a memory leak in the *condor_gridmanager*.
- Added the **-Storklog** argument to *condor_submit_dag* to make it compatible with the older perl script of the same name.
- Removed support for the **-libc** option for *condor_version*.
- Added a fix to *condor_compile* where if our internal *ld* managed to not be invoked during linking of a standard universe executable, a warning is emitted.
- Fixed a minor bug in the file transfer mechanism. Specifically, if a VANILLA job had `when_to_transfer_output` set to ON_EXIT_OR_EVICT, wrote more than one output file, and was actually evicted, the condor *condor_shadow* would have a fatal run-time error (shadow exception) and your job would be rerun.
- DAGMan bug fixes:
  - If submit files for individual nodes referred to the same log file with different paths, *condor_dagman* would read log events incorrectly and the DAG would fail. *condor_dagman* is now able to recognize that the different paths actually refer to the same log file.
  - Fixed a bug where DAGMan failed to monitor Stork job logs.
  - If a node submit file doesn't specify a log file, the warning message now gets printed out in the the DAGMan log file.
  - Fixed a bug that caused *condor_dagman* to fail if first node submit file has continuation in log file line.
- Bugs related to configuration
  - Fixed a bug where Condor daemons could crash if `COLLECTOR_HOST` or `NEGOTIATOR_HOST` was defined to be something bogus.
  - Fixed potential crash in the *condor_collector* when `COLLECTOR_NAME` was too long.
  - The default setting for `POOL_HISTORY_DIR` is no longer `SPOOL` . Using the spool directory would result in history files being obliterated by *condor_preen*.
- Fixed a bug which could result in a daemon crashing while it was writing to its logfile.

- Fixed a signal handling bug in the checkpoint server which could cause the daemon to hang sometimes.
- The Kerberos map file now tolerates spaces on either side of the equals sign instead of generating a parse error.
- The **-analyze** option to *condor_q* is only meaningful for certain universes. *condor_q* now warns if the output might not be meaningful.
- Java universe: when jar files are transferred to the execute machine (with `should_transfer_files` or `transfer_input_files`) the *condor_starter* will use the local path (in the execute directory) for the jarfiles, instead of the original path specified in the submit file.
- Previously, if a scheduler universe job died with a signal, the *condor_schedd* would write multiple (conflicting) events into the UserLog file: a terminate event and an abort event. Now, only the terminate event is written, not the abort event.
- Fixed a minor bug where if the *condor_schedd* crashed or was killed at just the wrong moment while a job was being removed because the `periodic_remove` expression had evaluated to TRUE, the job might have been successfully removed but the `RemoveReason` attribute could have been lost. Now, both actions are taken together atomically. If a job is successfully removed, it will always have a `RemoveReason` attribute.
- Fixed a memory leak in the *condor_collector*.

Known Bugs:

- None.

## Version 6.6.5

Release Notes:

- None.

New Features:

- None.

Bugs Fixed:

- Fixed a bug introduced in Condor version 6.6.2 that could cause *condor_dagman* to segfault while parsing some DAG files, or fail to recognize already-completed nodes in a rescue DAG.

- Fixed a bug in *condor_dagman*, whereby it could fail to automatically discover a Condor job's userlog file if the job's submit file did not have whitespace surrounding the equal sign on the log file line.
- Fixed a bug in *condor_submit* that appears to only have effected OSX machines. Previously, submit files that only defined a single job and used `queue` without any numerical modifiers would result in an error like this:

  ```
  ERROR: "test.sub" doesn't contain any "queue" commands -- no jobs queued
  ```

  Now, *condor_submit* will properly process and submit the job from job description files that contain a single `queue` statement with no modifiers.
- Fixed a bug in the AIX *condor_starter* that was causing the starter to sometimes kill itself when the job completed. Because this happened before the *condor_starter* reported the job completion back to the *condor_shadow*, such a job would be restarted.
- Fixed a few memory and registry handle leaks in the *condor_schedd* and *condor_startd*. These leaks particularly affected Windows systems.
- On Windows, Condor was known to have trouble accessing config files with UNC paths (with appropriate permissions set). This has been fixed.
- On Windows, *condor_store_cred* would fail if the account did not have `Log on Locally` privileges, even if the account was allowed to log in interactively. This has been fixed.
- Fixed a bug on Windows that would cause the *condor_schedd* to crash if `D_FULLDEBUG` was turned on, and the submitting user account did not have Administrator access rights.

Known Bugs:

- *condor_dagman* can fail to detect a job's progress if another job in the DAG specifies the same underlying userlog file using a different path or filename (e.g., log=foo and log=./foo) in its submit file.

## Version 6.6.4

Release Notes:

- This version only contains platform-specific bug fixes. Therefore, it was only released for the two effected platforms.

Bugs Fixed:

- Fixed a major bug in the Windows NT/2000 port that caused the Condor daemons to crash when attempting to authenticate.

- Fixed the bug in Condor's file transfer mechanism for Mac OSX that was introduced in version 6.6.3.

Known Bugs:

- None.

## Version 6.6.3

Release Notes:

- The Globus universe support for versions of Globus prior to 2.2 (specifically, those using GRAM 1.5 or earlier) has been removed.

New Features:

- The Globus universe now supports submitting jobs to Globus Toolkit 3.2 installations.

Bugs Fixed:

- The negotiator no longer crashes when a grid site ClassAd sets WantAdRevaluate but does not contain an UpdateSequenceNumber.
- Globus universe jobs were failing to go on hold when a $$() expression could not be expanded.
- On Windows, the system-wide TEMP variable is included in the execute environment if it is not specified in the submit file.
- Fixed a rarely-occurring bug when the child process forked by the schedd gets stuck in an infinite loop when the user does "condor_submit -s". This should also fix problems when the child process forked by the collector would sometimes get stuck in an infinite loop when `COLLECTOR_QUERY_WORKERS > 0` in the config file.

Known Bugs:

- The Condor file transfer mechanism is broken on Mac OSX in Condor version 6.6.3. OSX users should either upgrade to version 6.6.4, or install a patched *condor_starter* binary available from http://www.cs.wisc.edu/condor/binaries/condor-6.6.3-patch1-MacOSX-PPC.tar.Z.

### Version 6.6.2

Release Notes:

- There will be another release, 6.6.3, within a few weeks. We decided to release this version now because it adds the AIX platform and has some bug fixes which we thought important enough for a release. However, if you are not affected by the bugs fixed (see below) you may wish to wait for 6.6.3.

New Features:

- Clipped support for AIX 5.2. This means VANILLA universe only - no checkpointing or STANDARD universe.
- The setting `GRIDMANAGER_GLOBUS_COMMIT_TIMEOUT` allows configuring the two phase commit timeout in Globus. This maps to the two_phase setting in the Globus RSL.
- Added a new configuration variable, `DAGMAN_MAX_SUBMIT_ATTEMPTS` , that controls how many times in a row *condor_dagman* will attempt to execute *condor_submit* for a given job before giving up. It cannot be set to less than 1 attempt, or more than 10; if left undefined, it defaults to 6.
- Added a new tool *condor_updates_stats* to dump out the update statistics information from ClassAds in a human readable format. Condor 6.6.1, by default, publishes "update statistics" into the ClassAds as published by the *condor_collector*. This program parses this output and displays it to the user in a readable format.
- Changed the default *condor_dagman* behavior so that it doesn't check for cycles at startup, only at runtime, since the former could be expensive for large DAGs. Added a boolean `DAGMAN_STARTUP_CYCLE_DETECT` config attribute to re-enable cycle-detection at startup.
- *condor_dagman* now offers a configuration variable, `DAGMAN_MAX_SUBMITS_PER_INTERVAL` , which controls how many individual jobs *condor_dagman* will submit in a row before servicing other requests (such as a *condor_rm*).
- The grid_monitor now automatically detects jobmanager scripts on the remote gatekeeper. Previously it was limited to supporting the condor, fork, lsf, pbs, and remote jobmanager scripts.
- A new parameter, `SEC_DEBUG_PRINT_KEYS` , controls whether or not the keys used for encryption get printed into the log. The default is false.

Bugs Fixed:

- Jobs that make use of Condor's file transfer mechanism were not automatically authorized to read/write input/output files when flocking to machines that did not happen to be in the `HOSTALLOW_WRITE` list. This bug has existed since 6.3.

- Eliminated a small chance that a grid_monitor log file or state file might be reused. The unique identifying numbers are now unique across the entire gridmanager, not each Globus resource.
- Eliminated a race condition which might cause the grid monitor to erroneously decide that the status file was broken when in fact it was being uploaded and was empty.
- The grid monitor now attempts to restart transfers in the event of globus-url-copy hanging.
- Removed some settings from the default configuration files shipped with Condor that are no longer used in the code.
- Fixed bugs in *condor_dagman* parsing of submit files (to determine node log files). Previously, a submit file line beginning with "log" (e.g., "LogLock = True") would be interpreted as a log file line. Also, if "log" was defined twice in the submit file, *condor_dagman* would incorrectly use the first definition, rather than the last.
- Re-added PVM support for IRIX 6.5.
- Fixed an indirect bug whereby *condor_dagman* could fail with an assertion error if it encounters both a terminate and a abort event in the userlog for the same job; this can happen due to a bug in the *condor_schedd*, which is not yet fixed.
- *condor_dagman* now works right with nodes that have an initialdir specified in the node submit file. (Previously, specifying an initialdir only worked if the log file path was absolute.)
- *condor_dagman* now responds more quickly to a request to be removed from the queue (via *condor_rm*), even if it is in the midst of submitting jobs. Previously, *condor_dagman* would finish submitting all ready jobs before responding to a removal request, which could take a long time, and forced it to immediately remove all the jobs it had just submitted unnecessarily.
- Fixed keyboard idle reporting on Mac OS X. Previously, the code would often return -1 on newer hardware.

Known Bugs:

- If a scheduler universe job terminates via a signal, the *condor_schedd* logs both a terminate event and an abort event to the userlog.
- Keyboard activity is not reported for pseudo-ttys on Mac OS X, only the physically connected keyboard

## Version 6.6.1

Release Notes:

- *condor_analyze* is not included in the downloads of Version 6.6.1. The existing binary from Version 6.6.0 is likely to work on all platforms for which it was released.

New Features:

- Added full support (including standard universe jobs with checkpointing and remote system calls) for Linux i386 RedHat 9 (using gcc/g++ version 3.2.2 and glibc version 2.3.2).
- Added full support (including standard universe jobs with checkpointing and remote system calls) for Linux i386 RedHat 8 (using gcc/g++ version 3.2 and glibc version 2.2.93).
- The time it takes *condor_dagman* to submit jobs has been reduced slightly to improve up the startup time of large DAGs.
- In order to help reduce load on the *condor_schedd* when *condor_dagman* is submitting jobs, there is a new config variable, DAGMAN_SUBMIT_DELAY , to specify the number of seconds *condor_dagman* will sleep before submitting each job.
- Enabled the "update statistics" in the *condor_collector* by default in both the executable and in the default configuration.
- Command-line arguments to *condor_dagman* are now handled case-insensitively.
- Added support for Condor-G and strong authentication to Condor for IRIX 6.5, but removed support for checkpointing and remote system calls. We plan to add support in Condor for IRIX's kernel-level checkpointing in a future release.
- Added a **-p** option to *condor_store_cred* so that users can now specify the the password on the command line instead of getting prompted for it.
- The gahp_server helper process for Condor-G includes patches from the LHC Computing Grid Project to increase data transfer performance of the Condor-G client. Previous versions of Condor-G could bog down in accepting new transfer requests, producing a variety of errors.
- Added a new configuration setting, SUBMIT_SEND_RESCHEDULE which controls whether or not *condor_submit* should automatically send a *condor_reschedule* command when it is done. Previously, *condor_submit* would always send this reschedule so that the *condor_schedd* knew to start trying to find matches for the new jobs. However, for submit machines that are managing a huge number of jobs (thousands or tens of thousands), this step would hurt performance in such a way that it became an obstacle to scalability. In this case, an administrator can set SUBMIT_SEND_RESCHEDULE to FALSE, this extra step is not performed, and the *condor_schedd* will try to find matches whenever the periodic timer in the *condor_negotiator* (NEGOTIATOR_INTERVAL) goes off.
- Pool administrators can now specify the length of time before the *condor_starter* sends its initial update to the *condor_shadow* by defining STARTER_INITIAL_UPDATE_INTERVAL . The default is 8 seconds. This setting would not normally need changing except to fine-tune a heavily loaded system.
- Administrators can now specify the default session duration for each Condor subsystem. This allows for fine tuning the image size of running Condor daemons if the memory footprint is a concern. The default for tools is 1 minute, the default for *condor_submit* is one hour, and the default for daemons is 100 days. This does not mean that tools cannot run more than one minute or submit cannot run for more than an hour; it only affects memory usage.

- Added new configuration setting GRID_MONITOR_HEARTBEAT_TIMEOUT . If this many seconds pass without hearing from the grid_monitor, it is assumed to be dead. Defaults to 300 (5 minutes). Increasing this number will improve the ability of the grid_monitor to survive in the face of transient problems but will also increase the time before Condor notices a problem. Prior to this change the gridmanager always waited 5 minutes, the user could not change the setting.
- Added new configuration setting GRID_MONITOR_RETRY_DURATION . If something goes wrong with the grid_monitor at a particular site (like GRID_MONITOR_HEARTBEAT_TIMEOUT expiring), it will be retried for this many seconds. Defaults to 900 (15 minutes). If we can't successfully get it going again the grid monitor will be disabled for that site until 60 minutes have passed. Prior to this change the condor_gridmanager wait 60 minutes after any failure.

Bugs Fixed:

- Fixed bugs related to network communication and timeouts that impact scalability in Condor:
  - Fixed a bug inside Condor's network communication layer that could result in Condor daemons blocking trying to read more data after a socket had already been closed.
  - Fixed a *condor_negotiator* bug that could, in certain rare circumstances, cause a *condor_schedd* to hang for five minutes while trying to communicate with it.
  - Fixed a bug in which TCP connections would re-authenticate needlessly when Condor's strong authentication was enabled. This was not harmful but incurred a bit of overhead, especially when using Kerberos authentication.
- Fixed bugs related to network security sessions which were getting cleared out. If the timing was unfortunate, this could cause some jobs to fail immediately after completion. So, Condor no longer clears out security sessions periodically (it used to happen every 8 hours) nor does it do so when a daemon receives a *condor_reconfig* command.
- Fixed a bug in the standard universe where C++ code that threw an exception would result in abortion of the executable instead of the delivery of the exception. This bug affects Condor version 6.6.0 for Redhat 7.x.
- Fixed a *condor_shadow* bug that could result in a fatal error if the following 3 conditions were met: (1) the job enables Condor's file transfer mechanism, (2) the job wants Condor to automatically figure out what files to transfer back (the default), and (3) the job does not specify a userlog.
- Fixed bug whereby *condor_dagman*, if removed from the queue via *condor_rm*, could fail to remove all of its submitted jobs if any of their submit events had not yet appeared in the userlog.
- Fixed a few bugs in *condor_preen*:
  - It will no longer potentially remove files related to a valid Computing on Demand (COD) claim on an otherwise idle machine.

  - *condor_preen* will no longer keep reporting that it had successfully removed a directory which was in fact failing to be removed.

- Fixed the faulty argument parsing in *condor_rm*, *condor_release*, and *condor_hold*. Before you could accidentally type `condor_rm -analyze`, and it would remove all of your jobs. Now it gives an error.
- On Windows, when you type a command like `condor_reconfig.exe` instead of `condor_reconfig`, you no longer get an error.
- Fixed a bug on Windows that would cause "GetCursorPos() failed" to appear repeatedly in the StartLog. The startd now uses a different function to track mouse activity that does not have a tendency to fail.
- Fixed a bug on Windows that would prevent some *condor_shadow* daemons from obtaining a lock to their log file under heavy load, and thus causing them to EXCEPT().
- Fixed a bug on Windows where file transfers would incorrectly fail because of bad permissions when using domain accounts with nested groups, or when UNC paths were used.
- Fixed the bug where the *condor_starter* would fail to transfer back core files created by Vanilla, Java and MPI universe jobs. This bug was introduced in Condor version 6.5.2. Now, Condor correctly transfers back any core files created by faulty user jobs in any job universe.
- In some circumstances, *condor_history* would fail to read information about some jobs, and would report errors. In particular, when jobs had large environments, it would fail. This has been corrected.
- Fixed a rare bug affecting *condor_dagman* when job-throttling was enabled: if *condor_dagman* was removed from the queue together with some of its own jobs (e.g., via `condor_rm -a`), it would quickly submit new jobs to replace them before recognizing that it needs to exit. It now shuts down immediately without submitting and then removing these unnecessary jobs.
- Fixed a potential security problem that was introduced in Condor version 6.5.5 when the `REQUIRE_LOCAL_CONFIG_FILE` configuration setting was added. This setting used to default to FALSE if it was not defined in the configuration files. It now defaults to TRUE. If administrators define local configuration files for the machines in their pool, it should be a fatal error if those files don't exist unless the administrators actively disable this check by defining `REQUIRE_LOCAL_CONFIG_FILE` to be FALSE.
- Fixed a bug on Windows that would cause the *condor_startd* to EXCEPT() if the *condor_starter* exited and left orphaned processes to be cleaned up. This bug first appeared in 6.5.0.
- Fixed a bug on Windows that would cause graceful shutdowns on Windows (such as when `condor_vacate` is called) to fail to complete.
- The gahp_server helper program, which provides Globus services to Condor-G, was always dynamically linked, even in statically-linked releases. The statically linked distributions of Condor now include a static gahp_server.

- Fixed minor bug in parsing XML user log files that contain empty strings.
- Fixed the messages written to the Condor daemon log files in various error conditions to be more informative and clear:
  - The error message in the SchedLog that indicates that swap space has been depleted has been rephrased so it appears to be significant.
  - Certain serious error messages are now being written to the `D_ALWAYS` debug level that used to only appear if other debug levels were enabled.
  - Clarified log messages related to errors looking up user information in the passwd database on UNIX and for creating dynamic users on Windows.
  - Log messages related to keep-alives sent between the *condor_schedd* and *condor_startd* (written to `D_PROTOCOL`) now include the `ClaimId` on both sides, so that it is easier to find potential problems and figure out which keep-alive messages correspond to what resources.
  - Added more useful information to certain errors relating to security sessions and strong authentication.
  - Fixed the formatting of some messages to correctly include a newline at the end of the message.
- Fixed a bug in the *condor_configure* installation tool. Previously, it would set `MAIL_PATH`, which doesn't exist in Condor and had no effect. Now, *condor_configure* correctly sets `MAIL` , instead.
- Fixed bug in userlog code in the CondorAPI library to prevent segmentation faults.
- Clarified log messages for Condor-G's GridmanagerLog, especially those relating to the grid monitor.
- Fixed potential race condition when using the grid monitor. Condor-G now identifies partial grid monitor status updates and waits for the update to complete.
- The grid_monitor is slightly more robust in the face of unexpected behavior by the Globus jobmanager. This is only a partial fix, for complete success you really need the Globus patch at http://bugzilla.globus.org/bugzilla/show_bug.cgi?id=1425
- Internal timeouts in the grid_monitor have been increased, increasing robustness during transient errors.

Known Bugs:

- Submission of MPI jobs from a Unix machine to run on Windows machines (or vice versa) fails for machine_count ¿ 1. This is not a new bug. Cross-platform submission of MPI jobs between Unix and Windows has always had this problem.
- A multiple install of Condor's standard universe support libraries onto an NFS server for the purposes of having a heterogeneous mix of Linux distribution revisions all being able to utilize the same *condor_compile* does not function correctly if Redhat 9 is one of the distributions.

### Version 6.6.0

New Features:

- The *condor_dagman* debugging log now reports the total number of "Un-Ready" Nodes (i.e. those waiting for unfinished dependencies) in its periodic summaries. In the past, the omission of this state led to confusion because the total of all reported job states didn't always match the total number of jobs in the DAG.
- Most Condor commands (*condor_on*, *condor_off*, *condor_restart*, *condor_reconfig*, *condor_vacate*, *condor_checkpoint*, *condor_reschedule*) now support a **-all** command-line option to specify which daemons to act on. This is more efficient and much easier to use than previous methods for accomplishing the same effect. Using **-all** with *condor_off* correctly leaves the existing *condor_master* processes running on each host, so that a subsequent *condor_on* would work. See section **??** on page **??** for more details on proper use of **-all** with *condor_off* and *condor_on*

Bugs Fixed:

- Fixed a bug under Solaris 8 with Update 6+, and Solaris 9 where Condor would incorrectly report the console and mouse idle times as zero.
- The standard-universe fetch_files feature was not cleaning up temporary files on the execution machine.
- In rare circumstances, a Linux kernel bug results in conflicting information about system boot time (`/proc/stat` and `/proc/uptime`). Specifically, the "btime" field in `/proc/stat` suddenly jumps to the present moment and then stays at that value. This was resulting in incorrect estimation of process ages, which caused Condor's estimation of CondorLoadAvg to be completely wrong. A more robust heuristic is now being used.
- A long configuration line with with continuation lines can cause the config file parser to not properly skip the leading whitespace from the continued lines. This has been corrected.
- The Grid Monitor now will automatically probe for and work with "unknown" batch systems.
- Fixed a bug where under certain circumstances *condor_dagman* would fail to detect an unsuccessful invocation of *condor_submit*, and would instead report the job as successfully submitted with job id 0.0.
- Fixed a bug which was causing problems when a periodic_remove expression for a scheduler universe job evaluates to true. Under these conditions, the schedd did not log the job termination to the job log. Additionally, the schedd would exit with an error status.
- Fixed a recently-introduced *condor_dagman* bug where the number of node retries (specified with the RETRY keyword) wasn't being updated after some failures; instead, the node would be allowed to retry indefinitely if it kept failing.

- Fixed a recently-introduced bug where shutting down the *condor_schedd* caused *condor_dagman* to remove all its jobs from the queue and write a rescue file, rather than simply exiting so that it could recover automatically upon restart.
- Changed the default "Periodic Expression Interval" parameter (PERIODIC_EXPR_INTERVAL) from 60 seconds to 300 seconds.
- Whenever *condor_reconfig* was used to re-configure multiple daemons which included the *condor_collector* for a pool, the command would start to fail after the *condor_collector* was reconfigured due to problems with security sessions in Condor's strong authentication code. This situation no longer causes problems for the *condor_reconfig* tool, and it can properly re-configure multiple daemons at once, even if one of them is the *condor_collector* for a pool.
- Most Condor commands (*condor_on*, *condor_off*, *condor_restart*, *condor_reconfig*, *condor_vacate*, *condor_checkpoint*, *condor_reschedule*) now check to make sure they are not sending a duplicate command if the user specifies the same target machine or daemon twice. For example:

  ```
  condor_reconfig hostname1 hostname2 hostname1
  ```

  will only send a single reconfig command to `hostname1`.
- Fixed a bug in the HPUX version of Condor which was causing the startd to occasionally abort operation. This has been in Condor since version 6.1.1.
- The Condor daemons will no longer overwhelm NIS servers when large numbers of daemons are running. Condor now caches uid and group information internally, and refreshes the cache entries on a specified interval (which defaults to 5 minutes). See section 3.3.3 on page 151 for more details.

Known Bugs:

- The *condor_preen* program does not know about Computing on Demand (COD) claims. If there are no regular Condor jobs on a given machine, but there are COD claims, and *condor_preen* is spawned, it will remove files related to the COD claims. In version 6.6.0, sites using COD are encouraged to disable *condor_preen* by commenting out the PREEN setting in the config files. This bug has been fixed in Condor version 6.6.1.
- Normally, if a user's job crashes and creates a core file on a remote execution machine, the *condor_starter* will automatically transfer the core file back to the submit machine. However, beginning in Condor version 6.5.2, if a vanilla, Java, or MPI universe job creates a core file, the *condor_starter* will fail to transfer it back. This bug will be fixed in version 6.6.1.
- There are a few bugs related to Condor tools failing to correctly locate the *condor_negotiator* daemon. These bugs usually show up if a site is using non-standard ports for the central manager daemon. However, some of the bugs show up regardless of if the negotiator is listening on the standard port or not.

- `condor_config_val -negotiator` queries the *condor_collector*, instead of querying the *condor_negotiator* like it should.
- Using the **-pool** option to `condor_q -analyze` will not work. The tool will fail to find and query the *condor_negotiator* for user priorities which it needs to determine why jobs may not be running.
- The Condor tools that support either the **-negotiator** or **-collector** options do not work when a user also specifies the **-pool** to define a remote pool to communicate with. The tools print a somewhat confusing message in this case.
- Most Condor tools that support `-pool hostname` will also recognize `-pool hostname:port` if the remote *condor_collector* is listening on a non-standard port. However, the *condor_findhost* tool does not work if given a **-pool** option that includes a port.

| *Architecture* | *Operating System* |
|---|---|
| Hewlett Packard PA-RISC (both PA7000 and PA8000 series) | HPUX 10.20 |
| Sun SPARC Sun4m,Sun4c, Sun UltraSPARC | Solaris 2.6, 2.7, 8, 9 |
| Silicon Graphics MIPS (R5000, R8000, R10000) | IRIX 6.5 |
| Intel x86 | Red Hat Linux 7.1, 7.2, 7.3<br>Red Hat Linux 8 (clipped) |
| | Red Hat Linux 9 (clipped) |
| | Windows NT 4.0 Workstation and Server (clipped) |
| | Windows 2000 Professional and Server, 2003 Server (clipped |
| | Windows XP Professional (clipped) |
| ALPHA | Digital Unix 4.0<br>Red Hat Linux 7.1, 7.2, 7.3 (clipped) |
| | Tru64 5.1 (clipped) |
| PowerPC | Macintosh OS X (clipped) |
| Itanium | Red Hat Linux 7.1, 7.2, 7.3 (clipped) |

Table 8.2: Condor version 6.6.0 supported platforms

CHAPTER

# NINE

# Command Reference Manual (man pages)

# *cleanup_release*

uninstall a previously installed software release installed by *install_release*

## Synopsis

**cleanup_release** [**-help**]

**cleanup_release** *install-log-name*

## Description

*cleanup_release* uninstalls a previously installed software release installed by *install_release*. The program works through the install log in reverse order, removing files as it goes. Each delete is logged in the install log to allow recovery from a crash. The install log name is provided as the *install-log-name* argument to this program.

## Options

**-help** Display brief usage information and exit.

## Exit Status

*cleanup_release* will exit with a status of 0 (zero) upon success, and non-zero otherwise.

## See Also

*install_release* (on page 774).

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_advertise*

Send a ClassAd to the *condor_collector* daemon

## Synopsis

***condor_advertise*** [**-help** | **-version**] [**-pool** *centralmanagerhostname[:portname]*] [**-debug**] [**-tcp**] *update-command classad-filename*

## Description

*condor_advertise* sends a ClassAd to the *condor_collector* daemon on the central manager machine. The ClassAd is contained in a file, which is specified by the second required argument. Which daemon's ClassAd to update is specified by the first required argument. The *update-command* may be one of the following strings:

**UPDATE_STARTD_AD**

**UPDATE_SCHEDD_AD**

**UPDATE_MASTER_AD**

**UPDATE_GATEWAY_AD**

**UPDATE_CKPT_SRVR_AD**

**UPDATE_NEGOTIATOR_AD**

**UPDATE_HAD_AD**

**UPDATE_AD_GENERIC**

**UPDATE_SUBMITTOR_AD**

**UPDATE_COLLECTOR_AD**

**UPDATE_LICENSE_AD**

**UPDATE_STORAGE_AD**

*condor_advertise* can also be used to invalidate and delete ClassAds currently held by the *condor_collector* daemon. In this case the *update-command* will be one of the following strings:

**INVALIDATE_STARTD_ADS**

**INVALIDATE_SCHEDD_ADS**

**INVALIDATE_MASTER_ADS**

**INVALIDATE_GATEWAY_ADS**

**INVALIDATE_CKPT_SRVR_ADS**

**INVALIDATE_NEGOTIATOR_ADS**

**INVALIDATE_HAD_ADS**

**INVALIDATE_ADS_GENERIC**

**INVALIDATE_SUBMITTOR_ADS**

**INVALIDATE_COLLECTOR_ADS**

**INVALIDATE_LICENSE_ADS**

**INVALIDATE_STORAGE_ADS**

For any of these INVALIDATE commands, the ClassAd in the required file consists of three entries. The file contents will be similar to:

```
MyType = "Query"
TargetType = "Machine"
Requirements = Name == "condor.example.com"
```

The definition for `MyType` is always `Query`. `TargetType` is set to the `MyType` of the ad to be deleted. This `MyType` is `DaemonMaster` for the *condor_master* ClassAd, `Machine` for the *condor_startd* ClassAd, `Scheduler` for the *condor_schedd* ClassAd, and `Negotiator` for the *condor_negotiator* ClassAd. `Requirements` is an expression evaluated within the context of ads of `TargetType`. When `Requirements` evaluates to `True`, the matching ad is invalidated. A full example is given below.

## Options

**-help** Display usage information

**-version** Display version information

**-pool** ***centralmanagerhostname[:portname]*** Specify a pool by giving the central manager's hostname and an optional port number. The default is the `COLLECTOR_HOST` specified in the configuration file.

**-tcp** Use TCP for communication. Without this option, UDP is used.

**-debug** Print debugging information as the command executes.

## General Remarks

The job and machine ClassAds are regularly updated. Therefore, the result of *condor_advertise* is likely to be overwritten in a very short time. It is unlikely that either Condor users (those who submit jobs) or administrators will ever have a use for this command. If it is desired to update or set a ClassAd attribute, the *condor_config_val* command is the proper command to use.

For those administrators who do need *condor_advertise*, you can optionally include these attributes:

DaemonStartTime - The time the service you are advertising started running. Measured in seconds since the Unix epoch.

UpdateSequenceNumber - An integer that begins at 0 and increments by one each time you re-advertise the same ad.

If both of the above are included, the *condor_collector* will automatically include the following attributes:

UpdatesTotal - The actual number of advertisements for this daemon that the *condor_collector* has seen.

UpdatesLost - The number of advertisements that for this daemon that the *condor_collector* expected to see, but did not.

UpdatesSequenced - The total of UpdatesTotal and UpdatesLost.

UpdatesHistory - See `COLLECTOR_DAEMON_HISTORY_SIZE` in section 3.3.17.

## Examples

Assume that a machine called condor.example.com is turned off, yet its *condor_startd* ClassAd does not expire for another 20 minutes. To avoid this machine being matched, an administrator chooses to delete the machine's *condor_startd* ClassAd. Create a file (called `remove_file` in this example) with the three required attributes:

```
MyType = "Query"
TargetType = "Machine"
Requirements = Name == "condor.example.com"
```

This file is used with the command:

```
% condor_advertise INVALIDATE_STARTD_ADS remove_file
```

## Exit Status

*condor_advertise* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_check_userlogs*

Check user log files for errors

## Synopsis

***condor_check_userlogs*** *UserLogFile1* [*UserLogFile2 . . .UserLogFileN* ]

## Description

*condor_check_userlogs* is a program for checking a user log or set of users logs for errors. Output includes an indication that no errors were found within a log file, or a list of errors such as an execute or terminate event without a corresponding submit event, or multiple terminated events for the same job.

*condor_check_userlogs* is especially useful for debugging *condor_dagman* problems. If *condor_dagman* reports an error it is often useful to run *condor_check_userlogs* on the relevant log files.

## Exit Status

*condor_check_userlogs* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial

Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_checkpoint*

send a checkpoint command to jobs running on specified hosts

## Synopsis

***condor_checkpoint*** [**-help** | **-version**]

***condor_checkpoint*** [**-debug**] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>" . . .* ]| [**-all**]

***condor_checkpoint*** [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *hostname* ]| [**-addr** *"<a.b.c.d:port>"*] . . . [ | **-all**]

## Description

*condor_checkpoint* sends a checkpoint command to a set of machines within a single pool. This causes the startd daemon on each of the specified machines to take a checkpoint of any running job that is executing under the standard universe. The job is temporarily stopped, a checkpoint is taken, and then the job continues. If no machine is specified, then the command is sent to the machine that issued the *condor_checkpoint* command.

The command sent is a periodic checkpoint. The job will take a checkpoint, but then the job will immediately continue running after the checkpoint is completed. *condor_vacate*, on the other hand, will result in the job exiting (vacating) after it produces a checkpoint.

If the job being checkpointed is running under the standard universe, the job produces a checkpoint and then continues running on the same machine. If the job is running under another universe, or if there is currently no Condor job running on that host, then *condor_checkpoint* has no effect.

There is generally no need for the user or administrator to explicitly run *condor_checkpoint*. Taking checkpoints of running Condor jobs is handled automatically following the policies stated in the configuration files.

## Options

**-help** Display usage information

**-version** Display version information

**-debug** Causes debugging information to be sent to `stderr` based on the value of the configuration variable `TOOL_DEBUG`

**-pool** ***centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's hostname and an optional port number

**-name** ***hostname*** Send the command to a machine identified by *hostname*

***hostname*** Send the command to a machine identified by *hostname*

**-addr** ***"<a.b.c.d:port>"*** Send the command to a machine's master located at *"<a.b.c.d:port>"*

***"<a.b.c.d:port>"*** Send the command to a machine located at *"<a.b.c.d:port>"*

**-all** Send the command to all machines in the pool

## Exit Status

*condor_checkpoint* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Examples

To send a *condor_checkpoint* command to two named machines:

```
% condor_checkpoint  robin cardinal
```

To send the *condor_checkpoint* command to a machine within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command sends the command to a the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_checkpoint -pool condor.cae.wisc.edu -name cae17
```

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_chirp*

Access files or job ClassAd from an executing job

## Synopsis

***condor_chirp*** [**-help**]

***condor_chirp*** **fetch** *RemoteFileName LocalFileName*

***condor_chirp*** **put** [**-mode** *mode*] [**-perm** *UnixPerm*] *LocalFileName RemoteFileName*

***condor_chirp*** **remove** *RemoteFileName*

***condor_chirp*** **get_job_attr** *JobAttributeName*

***condor_chirp*** **set_job_attr** *JobAttributeName AttributeValue*

## Description

*condor_chirp* is run from a user job while executing. It accesses files or job ClassAd attributes on the submit machine. Files can be read, written or removed. Job attributes can be read, and most attributes can be updated.

Descriptions using the terms *local* and *remote* are given from the point of view of the executing program.

If the input file name for **put** is a dash, *condor_chirp* uses standard input as the source. If the output file name for **fetch** is a dash, *condor_chirp* writes to standard output instead of a local file.

Jobs that use *condor_chirp* must have the attribute `WantIOProxy` set to `True` in the job ad. To do this, place

```
+WantIOProxy = true
```

in the submit description file for the job.

*condor_chirp* only works for jobs run in the vanilla, mpi, parallel and java universes.

The optional **-mode** *mode* argument is one or more of the following characters describing the *RemoteFileName* file.

- w: open for writing
- a: force all writes to append

- t: truncate before use
- c: create the file, if it does not exist
- x: fail if 'c' is given, and the file already exists

The optional **-perm** *UnixPerm* argument describes the file access permissions in a Unix format (for example, 660).

## Options

**-help** Display usage information and exit.

**fetch** Copy the *RemoteFileName* from the submit machine to the execute machine.

**remove** Remove the *RemoteFileName* file from the submit machine.

**put** Copy the *LocalFileName* from the execute machine to the submit machine. Perm is the unix permission to open the file with.

**get_job_attr** Prints the named job ClassAd attribute to standard output.

**set_job_attr** Sets the named job ClassAd attribute with the given attribute value.

## Examples

To copy a file from the submit machine to the execute machine while the user job is running, run

```
% condor_chirp fetch remotefile localfile
```

To print to standard output the value of the `Requirements` expression from within a running job, run

```
% condor_chirp get_job_attr Requirements
```

Note that the remote (submit-side) directory path is relative to the submit directory, and the local (execute-side) directory is relative to the current directory of the running program.

To append the word "foo" to a file on the submit machine, run

```
% echo foo | condor_chirp put -mode wat - RemoteFile
```

## Exit Status

*condor_chirp* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_cod*

manage COD machines and jobs

## Synopsis

***condor_cod*** [**-help** | **-version**]

***condor_cod*** *request* [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname* ]| [**-addr** *"<a.b.c.d:port>"*] [[**-help — -version**] | [**-debug — -timeout N — -classad file**] ][**-requirements expr**]

***condor_cod*** *release* **-id** *ClaimID* [[**-help — -version**] | [**-debug — -timeout N — -classad file**] ][**-fast**]

***condor_cod*** *activate* **-id** *ClaimID* [[**-help — -version**] | [**-debug — -timeout N — -classad file**] ][**-keyword string — -jobad filename — -cluster N — -proc N — -requirements expr**]

***condor_cod*** *deactivate* **-id** *ClaimID* [[**-help — -version**] | [**-debug — -timeout N — -classad file**] ][**-fast**]

***condor_cod*** *suspend* **-id** *ClaimID* [[**-help — -version**] | [**-debug — -timeout N — -classad file**] ]

***condor_cod*** *resume* **-id** *ClaimID* [[**-help — -version**] | [**-debug — -timeout N — -classad file**] ]

## Description

*condor_cod* issues commands that manage and use COD claims on machines, given proper authorization.

Instead of specifying an argument of *request*, *release*, *activate*, *deactivate*, *suspend*, or *resume*, the user may invoke the *condor_cod* tool by appending an underscore followed by one of these arguments. As an example, the following two commands are equivalent:

```
condor_cod release -id "<128.105.121.21:49973>#1073352104#4"

condor_cod_release -id "<128.105.121.21:49973>#1073352104#4"
```

To make these extended-name commands work, hard link the extended name to the *condor_cod* executable. For example on a Unix machine:

```
ln condor_cod_request condor_cod
```

The *request* argument gives a claim ID, and the other commands (*release, activate, deactivate, suspend*, and *resume*) use the claim ID. The claim ID is given as the last line of output for a *request*, and the output appears of the form:

```
ID of new claim is: "<a.b.c.d:portnumber>#x#y"
```

An actual example of this line of output is

```
ID of new claim is: "<128.105.121.21:49973>#1073352104#4"
```

Also see section 4.3 for more a complete description of COD.

## Options

**-help** Display usage information

**-version** Display version information

**-pool** ***centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's hostname and an optional port number

**-name** ***scheddname*** Send the command to a machine identified by *scheddname*

**-addr** ***"<a.b.c.d:port>"*** Send the command to a machine located at *"<a.b.c.d:port>"*

**request** Create a new COD claim

**release** Relinquish a claim and kill any running job

**activate** Start a job on a given claim

**deactivate** Kill the current job, but keep the claim

**suspend** Suspend the job on a given claim

**resume** Resume the job on a given claim

## General Remarks

## Examples

## Exit Status

*condor_cod* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_cold_start*

install and start Condor on this machine

## Synopsis

***condor_cold_start* -help**

***condor_cold_start*** [**-basedir** *directory*] [**-force**] [**-setuponly** | **-runonly**] [**-arch** *architecture*] [**-site** *repository*] [**-localdir** *directory*] [**-runlocalconfig** *file*] [**-logarchive** *archive*] [**-spoolarchive** *archive*] [**-execarchive** *archive*] [**-filelock**] [**-pid**] [**-artifact** *filename*] [**-wget**] [**-globuslocation** *directory*] **-configfile** *file*

## Description

*condor_cold_start* installs and starts Condor on this machine, setting up or using a predefined configuration. In addition, it has the functionality to determine the local architecture if one is not specified. Additionally, this program can install pre-made `log`, `execute`, and/or `spool` directories by specifying the archived versions.

## Options

**-arch *architecturestr*** Use the given *architecturestr* to fetch the installation package. The string is in the format:

<condor_version>-<machine_arch>-<os_name>-<os_version>

(for example 6.6.7-i686-Linux-2.4). The portion of this string <condor_version> may be replaced with the string ”latest” (for example, latest-i686-Linux-2.4) to substitute the most recent version of Condor.

**-artifact *filename*** Use *filename* for name of the artifact file used to determine whether the *condor_master* daemon is still alive.

**-basedir *directory*** The directory to install or find the Condor executables and libraries. When not specified, the current working directory is assumed.

**-execarchive *archive*** Create the Condor `execute` directory from the given *archive* file.

**-filelock** Specifies that this program should use a POSIX file lock midwife program to create an artifact of the birth of a *condor_master* daemon. A file lock undertaker can later be used to

determine whether the *condor_master* daemon has exited. This is the preferred option when the user wants to check the status of the *condor_master* daemon from another machine that shares a distributed file system that supports POSIX file locking, for example, AFS.

**-force** Overwrite previously installed files, if necessary.

**-globuslocation *directory*** The location of the globus installation on this machine. When not specified `/opt/globus` is the directory used. This option is only necessary when other options of the form **-*archive** are specified.

**-help** Display brief usage information and exit.

**-localdir *directory*** The directory where the Condor `log`, `spool`, and `execute` directories will be installed. Each running instance of Condor must have its own local directory.

**-logarchive *archive*** Create the Condor log directory from the given *archive* file.

**-pid** This program is to use a unique process id midwife program to create an artifact of the birth of a *condor_master* daemon. A unique pid undertaker can later be used to determine whether the *condor_master* daemon has exited. This is the default option and the preferred method to check the status of the *condor_master* daemon from the same machine it was started on.

**-runlocalconfig *file*** A special local configuration file bound into the Condor configuration at runtime. This file only affects the instance of Condor started by this command. No other Condor instance sharing the same global configuration file will be affected.

**-runonly** Run Condor from the specified installation directory without installing it. It is possible to run several instantiations of Condor from a single installation.

**-setuponly** Install Condor without running it.

**-site *repository*** The ftp, http, gsiftp, or mounted file system directory where the installation packages can be found (for example, `www.cs.example.edu/packages/coldstart`).

**-spoolarchive *archive*** Create the Condor spool directory from the given *archive* file.

**-wget** Use *wget* to fetch the `log`, `spool`, and `execute` directories, if other options of the form **-*archive** are specified. *wget* must be installed on the machine and in the user's path.

**-configfile *file*** A required option to specify the Condor configuration file to use for this installation. This file can be located on an http, ftp, or gsiftp site, or alternatively on a mounted file system.

## Exit Status

*condor_cold_start* will exit with a status value of 0 (zero) upon success, and non-zero otherwise.

## Examples

To start a Condor installation on the current machine, using `http://www.example.com/Condor/deployment` as the installation site:

```
% condor_cold_start \
  -configfile http://www.example.com/Condor/deployment/condor_config.mobile \
  -site http://www.example.com/Condor/deployment
```

Optionally if this instance of Condor requires a local configuration file `condor_config.local`:

```
% condor_cold_start \
  -configfile http://www.example.com/Condor/deployment/condor_config.mobile \
  -site http://www.example.com/Condor/deployment \
  -runlocalconfig condor_config.local
```

## See Also

*condor_cold_stop* (on page 617), *filelock_midwife* (on page 770), *uniq_pid_midwife* (on page 792).

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and

Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_cold_stop*

reliably shut down and uninstall a running Condor instance

## Synopsis

***condor_cold_stop* -help**

***condor_cold_stop*** [**-force**] [**-basedir** *directory*] [**-localdir** *directory*] [**-runlocalconfig** *file*] [**-cleaninstall**] [**-cleanlocal**] [**-stop**] [**-logarchive** *archive*] [**-spoolarchive** *archive*] [**-execarchive** *archive*] [**-filelock**] [**-pid**] [**-artifact** *file*] [**-nogurl**] [**-globuslocation** *directory*] **-configfile** *file*

## Description

*condor_cold_stop* reliably shuts down and uninstall a running Condor instance. This program first uses *condor_local_stop* to reliably shut down the running Condor instance. It then uses *condor_cleanup_local* to create and store archives of the `log`, `spool`, and `execute` directories. Its last task is to uninstall the Condor binaries and libraries using *cleanup_release*.

## Options

**-artifact *file*** Uses *file* as the artifact file to determine whether the *condor_master* daemon is still alive.

**-basedir *directory*** Directory where the Condor installation can be found. When not specified, the current working directory is assumed.

**-cleaninstall** Remove the Condor installation. If none of the options **-cleaninstall**, **-cleanlocal**, or **-stop** are specified, the program behaves as though all of them have been provided.

**-cleanlocal** The program will remove the `log`, `spool`, `exec` directories for this Condor instance. If none of the options **-cleaninstall**, **-cleanlocal**, or **-stop** are specified, the program behaves as though all of them have been provided.

**-configfile *file*** The same configuration file path given to *condor_cold_start*. This program assumes the file is in the installation directory or the current working directory.

**-execarchive *archive*** The program will create a tar'ed and gzip'ed archive of the `execute` directory and stores it as *archive.* The *archive* can be a file path or a grid-ftp url.

**-filelock** Determine whether the *condor_master* daemon has exited using a file lock undertaker. This option must match the corresponding option given to *condor_cold_start*.

**-force** Ignore the status of the *condor_schedd* daemon (whether it has jobs in the queue or not) when shutting down Condor.

**-globuslocation *directory*** The directory containing the Globus installation. This option is required if any of the options of the form **-*archive** are used, and Globus is not installed in `/opt/globus`.

**-localdir *directory*** Directory where the `log`, `spool`, and `execute` directories are stored for this running instance of Condor. Required if the **-cleanlocal** option is specified.

**-logarchive *archive*** The program will create a tar'ed and gzip'ed archive of the `log` directory and stores it as *archive.* The *archive* can be a file path or a grid-ftp url.

**-nogurl** Do not use *globus-url-copy* to store the archives. This implies that the archives can only be stored on mounted file systems.

**-pid** Determine whether the *condor_master* daemon has exited using a unique process id undertaker. This option must match the corresponding option given to *condor_cold_start*.

**-runlocalconfig *file*** Bind *file* into the configuration used by this instance of Condor. This option should the one provided to *condor_cold_start*.

**-spoolarchive *archive*** The program will create a tar'ed and gzip'ed archive of the `spool` directory and stores it as *archive*. The *archive* can be a file path or a grid-ftp url.

**-stop** The program will shut down this running instance of Condor. If none of the options **-cleaninstall**, **-cleanlocal**, or **-stop** are specified, the program behaves as though all of them have been provided.

## Exit Status

*condor_cold_stop* will exit with a status value of 0 (zero) upon success, and non-zero otherwise.

## Examples

To shut down a Condor instance on the target machine:

```
% condor_cold_stop -configfile condor_config.mobile
```

To shutdown a Condor instance and archive the log directory:

```
% condor_cold_stop -configfile condor_config.mobile \
  -logarchive /tmp/log.tar.gz
```

## See Also

*condor_cold_start* (on page 613), *filelock_undertaker* (on page 772), *uniq_pid_undertaker* (on page 794).

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_compile*

create a relinked executable for submission to the Standard Universe

## Synopsis

***condor_compile*** *cc* | *CC* | *gcc* | *f77* | *g++* | *ld* | *make* | ...

## Description

Use *condor_compile* to relink a program with the Condor libraries for submission into Condor's Standard Universe. The Condor libraries provide the program with additional support, such as the capability to checkpoint, which is required in Condor's Standard Universe mode of operation. *condor_compile* requires access to the source or object code of the program to be submitted; if source or object code for the program is not available (i.e. only an executable binary, or if it is a shell script), then the program must submitted into Condor's Vanilla Universe. See the reference page for *condor_submit* and/or consult the "Condor Users and Administrators Manual" for further information.

To use *condor_compile*, simply enter "condor_compile" followed by whatever you would normally enter to compile or link your application. Any resulting executables will have the Condor libraries linked in. For example:

```
condor_compile cc -O -o myprogram.condor file1.c file2.c ...
```

will produce a binary "myprogram.condor" which is relinked for Condor, capable of checkpoint/migration/remote-system-calls, and ready to submit to the Standard Universe.

If the Condor administrator has opted to fully install *condor_compile*, then *condor_compile* can be followed by practically any command or program, including make or shell-script programs. For example, the following would all work:

```
condor_compile make

condor_compile make install

condor_compile f77 -O mysolver.f

condor_compile /bin/csh compile-me-shellscript
```

If the Condor administrator has opted to only do a partial install of *condor_compile*, the you are restricted to following *condor_compile* with one of these programs:

```
        cc (the system C compiler)

        acc (ANSI C compiler, on Sun systems)

        c89 (POSIX compliant C compiler, on some systems)

        CC (the system C++ compiler)

        f77 (the system FORTRAN compiler)

        gcc (the GNU C compiler)

        g++ (the GNU C++ compiler)

        g77 (the GNU FORTRAN compiler)

        ld (the system linker)
f90 (the system FORTRAN 90 compiler)
```

NOTE: If you use explicitly call "ld" when you normally create your binary, simply use:

```
        condor_compile ld <ld arguments and options>
```

instead.

NOTE: f90 (FORTRAN 90) is only supported on Solaris and Digital Unix.

## Exit Status

*condor_compile* is a script that executes specified compilers and/or linkers. If an error is encountered before calling these other programs, *condor_compile* will exit with a status value of 1 (one). Otherwise, the exit status will be that given by the executed program.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized

without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_config_bind*

bind together a set of configuration files

## Synopsis

***condor_config_bind* -help**

***condor_config_bind* -o** *outputfile configfile1 configfile2* [*configfile3...*]

## Description

*condor_config_bind* dynamically binds two or more Condor configuration files through the use of a new configuration file. The purpose of this tool is to allow the user to dynamically bind a local configuration file into an already created, and possible immutable, configuration file. This is particularly useful when the user wants to modify a configuration but cannot actually make any changes to the global configuration file (even to change the list of local configuration files). This program does not modify the given configuration files. Rather, it creates a new configuration file that specifies the given configuration files as local configuration files.

Condor evaluates each of the configuration files in the given command-line order (left to right). A value defined in two or more of the configuration files results in the last one evaluated defining the value. It overrides any others. To bind a new local configuration into a global configuration, specify the local configuration second within the command-line ordering.

## Options

***configfile1*** First configuration file to bind.

***configfile2*** Second configuration file to bind.

***configfile3...*** An optional list of other configuration files to bind.

**-help** Display brief usage information and exit

**-o *output_file*** Specifies the file name where this program should output the binding configuration.

## Exit Status

*condor_config_bind* will exit with a status value of 0 (zero) upon success, and non-zero on error.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_config_val*

Query or set a given Condor configuration variable

## Synopsis

***condor_config_val*** [*options*] *variable...*

***condor_config_val*** [*options*] **-set** *string...*

***condor_config_val*** [*options*] **-rset** *string...*

***condor_config_val*** [*options*] **-unset** *variable...*

***condor_config_val*** [*options*] **-runset** *variable...*

***condor_config_val*** [*options*] **-tilde**

***condor_config_val*** [*options*] **-owner**

***condor_config_val*** [*options*] **-config**

***condor_config_val*** [*options*] **-verbose** *variable...*

## Description

*condor_config_val* can be used to quickly see what the current Condor configuration is on any given machine. Given a list of variables, *condor_config_val* will report what each of these variables is currently set to. If a given variable is not defined, *condor_config_val* will halt on that variable, and report that it is not defined. By default, *condor_config_val* looks in the local machine's configuration files in order to evaluate the variables.

*condor_config_val* can also be used to quickly set configuration variables for a specific daemon on a given machine. Each daemon remembers settings made by *condor_config_val*. The configuration file is not modified by this command. Persistent settings remain when the daemon is restarted. Runtime settings are lost when the daemon is restarted. In general, modifying a host's configuration with *condor_config_val* requires the `CONFIG` access level, which is disabled on all hosts by default. See section 3.6.1 on page 261 for more details. Beginning with Condor version 6.3.2, administrators have more fine-grained control over which access levels can modify which settings. See section 3.6.1 on page 261 for more details.

NOTE: The changes will not take effect until you perform a *condor_reconfig*.

NOTE: It is generally wise to test a new configuration on a single machine to ensure you have no syntax or other errors in the configuration before you reconfigure many machines. Having bad syntax or invalid configuration settings is a fatal error for Condor daemons, and they will exit. Far

better to discover such a problem on a single machine than to cause all the Condor daemons in your pool to exit.

## Options

**-name** ***machine_name*** Query the specified machine's *condor_master* daemon for its configuration.

**-pool** ***centralmanagerhostname[:portnumber]*** Use the given central manager and an optional port number to find daemons.

**-address** <***ip:port***> Connect to the given ip/port.

**-master** | **-schedd** | **-startd** | **-collector** | **-negotiator** The daemon to query (if not specified, master is default).

**-set** ***string***... Set a persistent config file entry. The string must be a single argument, so you should enclose it in double quotes. The string must be of the form "variable = value".

**-rset** ***string***... Set a runtime config file entry. See the description for **-set** for details about the string to use.

**-unset** ***variable***... Unset a persistent config file variable.

**-runset** ***variable***... Unset a runtime config file variable.

**-tilde** Return the path to the Condor home directory.

**-owner** Return the owner of the *condor_config_val* process.

**-config** Print the current configuration files in use.

**-verbose** ***variable***... Returns the configuration file name and line number where a configuration variable is defined.

**variable**... The variables to query.

## Exit Status

*condor_config_val* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Examples

To request the schedd daemon on host perdita to give the value of the MAX_JOBS_RUNNING configuration variable:

```
% condor_config_val -name perdita -schedd MAX_JOBS_RUNNING
500
```

To request the schedd daemon on host perdita to set the value of the MAX_JOBS_RUNNING configuration variable to the value 10.

```
% condor_config_val -name perdita -schedd -set "MAX_JOBS_RUNNING = 10"
Successfully set configuration "MAX_JOBS_RUNNING = 10" on
schedd perdita.cs.wisc.edu <128.105.73.32:52067>.
```

A command that will implement the change just set in the previous example.

```
% condor_reconfig -schedd perdita
Sent "Reconfig" command to schedd perdita.cs.wisc.edu
```

A re-check of the configuration variable reflects the change implemented:

```
% condor_config_val -name perdita -schedd MAX_JOBS_RUNNING
10
```

To set the configuration variable MAX_JOBS_RUNNING back to what it was before the command to set it to 10:

```
% condor_config_val -name perdita -schedd -unset MAX_JOBS_RUNNING
Successfully unset configuration "MAX_JOBS_RUNNING" on
schedd perdita.cs.wisc.edu <128.105.73.32:52067>.
```

A command that will implement the change just set in the previous example.

```
% condor_reconfig -schedd perdita
Sent "Reconfig" command to schedd perdita.cs.wisc.edu
```

A re-check of the configuration variable reflects that variable has gone back to is value before initial set of the variable:

```
% condor_config_val -name perdita -schedd MAX_JOBS_RUNNING
500
```

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_configure*

Configure or install Condor

## Synopsis

***condor_configure*** [--**help**]

***condor_configure*** [--**install[=**<**path/to/release.tar**>**]** ] [--**install-dir=**<**path**>] [--**local-dir=**<**path**>] [--**make-personal-condor**] [--**type** = < **submit, execute, manager** >] [--**central-manager** = < **hostname**>] [--**owner** = < **ownername** >] [--**make-personal-stork**] [--**stork**] [--**credd**] [--**verbose**]

## Description

*condor_configure* is a Perl script that installs and/or configures Condor on Unix machines. It will run with Perl 5.003 or more recent versions.

*condor_configure* is designed to be run more than one time where required. It can install Condor (with a correct configuration), or it can change the configuration files. Note that changes in the configuration files do not result in changes while Condor is running. To effect changes while Condor is running, it is necessary to further do *condor_reconfig* or *condor_restart*. *condor_reconfig* is required where the currently executing daemons need to be informed of configuration changes. *condor_restart* is required where the options --**make-personal-condor** or --**type** are used, since these affect which daemons are running.

Running *condor_configure* with no options results in the help screen being printed.

## Options

**—help** Print help screen and exit

**—install** Perform installation, assuming that the current working directory contains the `release.tar` file. Where a directory path is supplied, it specifies that directory containing the `release.tar` file. Without further options, the configuration is that of a Personal Condor, a complete one-machine pool. If used as an upgrade within an existing installation directory, existing configuration files and local directory are preserved.

**—install-dir=**<**path**> Specifies the path where Condor should be installed or the path where it already is installed. The default is the current working directory.

**—local-dir=**<**path**> Specifies the location of the local directory, which is the directory that generally contains the local (machine-specific) configuration file as well as the directories where Condor daemons write their run-time information (`spool`, `log`, `execute`). This location is indicated by the `LOCAL_DIR` variable in the configuration file. When installing (that is, if —install is specified), *condor_configure* will properly create the local directory in the location specified. If none is specified, the default value is given by the evaluation of `$(RELEASE_DIR)/local.$(HOSTNAME)`.

During subsequent invocations of *condor_configure* (that is, without the —install option), if the —local-dir option is specified, the new directory will be created and the `log`, `spool` and `execute` directories will be moved there from their current location.

**—make-personal-condor** Installs and configures for Personal Condor, a fully-functional, one-machine pool.

**—type=** < **submit, execute, manager** > One or more of the types may be listed. This determines the roles that a machine may play in a pool. In general, any machine can be a submit and/or execute machine, and there is one central manager per pool. In the case of a Personal Condor, the machine fulfills all three of these roles.

**—central-manager=**<**hostname**> Instructs the current Condor installation to use the specified machine as the central manager. This modifies the configuration variables `COLLECTOR_HOST` and `NEGOTIATOR_HOST` to point to the given host name). The central manager machine's Condor configuration needs to be independently configured to act as a manager using the option —type=manager.

**—owner=**<**ownername**> Set configuration such that Condor daemons will be executed as the given owner. This modifies the ownership on the `log`, `spool` and `execute` directories and sets the `CONDOR_IDS` value in the configuration file, to ensure that Condor daemons start up as the specified effective user. See section 3.6.12 on UIDs in Condor on page 289 for details. This is only applicable when *condor_configure* is run by root. If not run as root, the owner is the user running the *condor_configure* command.

**—make-personal-stork** Creates a Personal Stork, using the *condor_credd* daemon.

**—stork** Configures the Stork data placement server. Use this option with the **—credd** option.

**—credd** Configure the the *condor_credd* daemon (credential manager daemon).

**—verbose** Print information about changes to configuration variables as they occur.

## Exit Status

*condor_configure* will exit with a status value of 0 (zero) upon success, and it will exit with a nonzero value upon failure.

## Examples

Install Condor on the machine (machine1@cs.wisc.edu) to be the pool's central manager. On machine1, within the directory that contains the unzipped Condor distribution `.tar.gz` file:

```
% condor_configure --install --type=submit,execute,manager
```

This will allow the machine to submit and execute Condor jobs, in addition to being the central manager of the pool.

To change the configuration such that machine2@cs.wisc.edu is an execute-only machine (that is, a dedicated computing node) within a pool with central manager on machine1@cs.wisc.edu, issue the command on that machine2@cs.wisc.edu from within the directory where Condor is installed:

```
% condor_configure --central-manager=machine1@cs.wisc.edu --type=execute
```

To change the location of the `LOCAL_DIR` directory in the configuration file, do (from the directory where Condor is installed):

```
% condor_configure --local-dir=/path/to/new/local/directory
```

This will move the `log`,`spool`,`execute` directories to `/path/to/new/local/directory` from the current local directory.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and

Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_convert_history*

Convert the history file to the new format

## Synopsis

***condor_convert_history*** [**-help**]

***condor_convert_history*** *history-file1* [*history-file2...*]

## Description

As of Condor version 6.7.19, the Condor history file has a new format to allow fast searches backwards through the file. Not all queries can take advantage of the speed increase, but the ones that can are significantly faster.

Entries placed in the history file after upgrade to Condor 6.7.19 will automatically be saved in the new format. The new format adds information to the string which distinguishes and separates job entries. In order to search within this new format, no changes are necessary. However, to be able to search the entire history, the history file must be converted to the updated format. *condor_convert_history* does this.

Turn the *condor_schedd* daemon off while converting history files. Turn it back on after conversion is completed.

Arguments to *condor_convert_history* are the history files to convert. The history file is normally in the Condor spool directory; it is named `history`. Since the history file is rotated, there may be multiple history files, and all of them should be converted. On Unix platform variants, the easiest way to do this is:

```
cd `condor_config_val SPOOL`
condor_convert_history history*
```

*condor_convert_history* makes a back up of each original history files in case of a problem. The names of these back up files are listed; names are formed by appending the suffix `.oldver` to the original file name. Move these back up files to a directory other than the spool directory. If kept in the spool directory, *condor_history* will find the back ups, and will appear to have duplicate jobs.

## Exit Status

*condor_convert_history* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_dagman*

meta scheduler of the jobs submitted as the nodes of a DAG or DAGs

## Synopsis

***condor_dagman*** [**-debug** *level*] [**-rescue** *filename*] [**-maxidle** *numberOfJobs*] [**-maxjobs** *numberOfJobs*] [**-maxpre** *NumberOfPREscripts*] [**-maxpost** *NumberOfPOSTscripts*] [**-noeventchecks**] [**-allowlogerror**] [**-usedagdir**] (**-condorlog** *filename* | **-storklog** *filename*) **-lockfile** *filename* [**-waitfordebug**] **-dag** *dag_file* [***-dag*** *dag_file_2 . . .* ***-dag*** *dag_file_n* ]

## Description

*condor_dagman* is a meta scheduler for the Condor jobs within a DAG (directed acyclic graph) (or multiple DAGs). In typical usage, a submitter of jobs that are organized into a DAG submits the DAG using *condor_submit_dag*. *condor_submit_dag* does error checking on aspects of the DAG and then submits *condor_dagman* as a Condor job. *condor_dagman* uses log files to coordinate the further submission of the jobs within the DAG.

As part of *daemoncore*, the set of command-line arguments given in section 3.9.2 work for *condor_dagman*.

Arguments to *condor_dagman* are either automatically set by *condor_submit_dag* or they are specified as command-line arguments to *condor_submit_dag* and passed on to *condor_dagman*. The method by which the arguments are set is given in their description below.

*condor_dagman* can run multiple, independent DAGs. This is done by specifying multiple **-dag** *a*rguments. Pass multiple DAG input files as command-line arguments to *condor_submit_dag*.

Debugging output may be obtained by using the **-debug** *level* option. Level values and what they produce is described as

- level = 0; never produce output, except for usage info
- level = 1; very quiet, output severe errors
- level = 2; normal output, errors and warnings
- level = 3; output errors, as well as all warnings
- level = 4; internal debugging output
- level = 5; internal debugging output; outer loop debugging
- level = 6; internal debugging output; inner loop debugging
- level = 7; internal debugging output; rarely used

## Options

**-debug *level*** An integer level of debugging output. *level* is an integer, with values of 0-7 inclusive, where 7 is the most verbose output. This command-line option to *condor_submit_dag* is passed to *condor_dagman* or defaults to the value 3, as set by *condor_submit_dag*.

**-rescue *filename*** Sets the file name of the rescue DAG to write in the case of a failure. As passed by *condor_submit_dag*, the name of the file will be the name of the DAG input file concatenated with the string `.rescue`.

**-maxidle *NumberOfJobs*** Sets the maximum number of idle jobs allowed before *condor_dagman* stops submitting more jobs. Once idle jobs start to run, *condor_dagman* will resume submitting jobs. *NumberOfJobs* is a positive integer. This command-line option to *condor_submit_dag* is passed to *condor_dagman*. If not specified, the number of idle jobs is unlimited.

**-maxjobs *numberOfJobs*** Sets the maximum number of jobs within the DAG that will be submitted to Condor at one time. *numberOfJobs* is a positive integer. This command-line option to *condor_submit_dag* is passed to *condor_dagman*. If not specified, the default number of jobs is unlimited.

**-maxpre *NumberOfPREscripts*** Sets the maximum number of PRE scripts within the DAG that may be running at one time. *NumberOfPREScripts* is a positive integer. This command-line option to *condor_submit_dag* is passed to *condor_dagman*. If not specified, the default number of PRE scripts is unlimited.

**-maxpost *NumberOfPOSTscripts*** Sets the maximum number of POST scripts within the DAG that may be running at one time. *NumberOfPOSTScripts* is a positive integer. This command-line option to *condor_submit_dag* is passed to *condor_dagman*. If not specified, the default number of POST scripts is unlimited.

**-noeventchecks** This argument is no longer used; it is now ignored. Its functionality is now implemented by the `DAGMAN_ALLOW_EVENTS` configuration macro (see section 3.3.23).

**-allowlogerror** This optional argument has *condor_dagman* try to run the specified DAG, even in the case of detected errors in the user log specification.

**-usedagdir** This optional argument has causes *condor_dagman* to run each specified DAG as if the directory containing that DAG file was the current working directory. This option is most useful when running multiple DAGs in a single *condor_dagman*.

**-storklog *filename*** Sets the file name of the Stork log for data placement jobs.

**-condorlog *filename*** Sets the file name of the file used in conjunction with the **-lockfile** *filename* in determining whether to run in recovery mode.

**-lockfile *filename*** Names the file created and used as a lock file. The lock file prevents execution of two of the same DAG, as defined by a DAG input file. A default lock file ending with the suffix `.dag.lock` is passed to *condor_dagman* by *condor_submit_dag*.

**-waitfordebug** This optional argument causes *condor_dagman* to wait at startup until someone attaches to the process with a debugger and sets the wait_for_debug variable in main_init() to false.

**-dag *filename*** *filename* is the name of the DAG input file that is set as an argument to *condor_submit_dag*, and passed to *condor_dagman*.

## Exit Status

*condor_dagman* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Examples

*condor_dagman* is normally not run directly, but submitted as a Condor job by running condor_submit_dag. See the condor_submit_dag manual page 742 for examples.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_fetchlog*

Retrieve a daemon's log file that is located on another computer

## Synopsis

***condor_fetchlog*** [**-help** | **-version**]

***condor_fetchlog*** [**-pool** *centralmanagerhostname[:portnumber]*] [**-master** | **-startd** | **-schedd** | **-collector** | **-negotiator** | **-kbdd**] *machine-name subsystem[.extension]*

## Description

*condor_fetchlog* contacts Condor running on the machine specified by *machine-name*, and asks it to return a log file from that machine. Which log file is determined from the *subsystem[.extension]* argument. The log file is printed to standard output. This command eliminates the need to remotely log in to a machine in order to retrieve a daemon's log file.

For security purposes of authentication and authorization, this command requires an administrator's level of access. See section 3.6.1 on page 261 for more details about Condor's security mechanisms.

The *subsystem[.extension]* argument is utilized to construct the log file's name. Without an optional *.extension*, the value of the configuration variable named *subsystem*_LOG defines the log file's name. When specified, the *.extension* is appended to this value.

Acceptable strings for the argument *subsystem* are as given as possible values of the predefined configuration variable `$(SUBSYSTEM)`. See the definition in section 3.3.1.

A value for the optional *.extension* argument may be one of the three strings:

1. `.old`
2. `.vm<X>`
3. `.vm<X>.old`

Within these strings, `<X>` is substituted with the number of the virtual machine.

## Options

**-help** Display usage information

**-version** Display version information

**-pool *centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's host name and an optional port number

**-master** Send the command to the *condor_master* daemon (default)

**-startd** Send the command to the *condor_startd* daemon

**-schedd** Send the command to the *condor_schedd* daemon

**-collector** Send the command to the *condor_collector* daemon

**-kbdd** Send the command to the *condor_kbdd* daemon

## Examples

To get the *condor_negotiator* daemon's log from a host named `head.example.com` from within the current pool:

```
condor_fetchlog head.example.com NEGOTIATOR
```

To get the *condor_startd* daemon's log from a host named `execute.example.com` from within the current pool:

```
condor_fetchlog execute.example.com STARTD
```

This command requested the *condor_startd* daemon's log from the *condor_master*. If the *condor_master* has crashed or is unresponsive, ask another daemon running on that computer to return the log. For example, ask the *condor_startd* daemon to return the *condor_master*'s log:

```
condor_fetchlog -startd execute.example.com MASTER
```

## Exit Status

*condor_fetchlog* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_findhost*

find machine(s) in the pool that can be used with minimal impact on currently running Condor jobs and best meet any specified constraints

## Synopsis

***condor_findhost*** [**-help**] [**-m**] [**-n** *num*] [**-c** *c_expr*] [**-r** *r_expr*] [**-p** *centralmanagerhostname*]

## Description

*condor_findhost* searches a Condor pool of machines for the best machine or machines that will have the minimum impact on running Condor jobs if the machine or machines are taken out of the pool. The search may be limited to the machine or machines that match a set of constraints and rank expression.

*condor_findhost* returns a fully-qualified domain name for each machine. The search is limited (constrained) to a specific set of machines using the *-c* option. The search can use the *-r* option for rank, the criterion used for selecting a machine or machines from the constrained list.

## Options

**-help** Display usage information and exit

**-m** Only search for entire machines. Virtual machines within an entire machine are not considered.

**-n *num*** Find and list up to *num* machines that fulfill the specification. *num* is an integer greater than zero.

**-c *c_expr*** Constrain the search to only consider machines that result from the evaluation of *c_expr*. *c_expr* is a ClassAd expression.

**-r *r_expr*** *r_expr* is the rank expression evaluated to use as a basis for machine selection. *r_expr* is a ClassAd expression.

**-p *centralmanagerhostname*** Specify the pool to be searched by giving the central manager's host name. Without this option, the current pool is searched.

## General Remarks

*condor_findhost* is used to locate a machine within a pool that can be taken out of the pool with the least disturbance of the pool.

An administrator should set preemption requirements for the Condor pool. The expression

```
(Interactive =?= TRUE )
```

will let *condor_findhost* know that it can claim a machine even if Condor would not normally preempt a job running on that machine.

## Exit Status

The exit status of *condor_findhost* is zero on success. If not able to identify as many machines as requested, it returns one more than the number of machines identified. For example, if 8 machines are requested, and *condor_findhost* only locates 6, the exit status will be 7. If not able to locate any machines, or an error is encountered, *condor_findhost* will return the value 1.

## Examples

To find and list four machines, preferring those with the highest mips (on Drystone benchmark) rating:

```
condor_findhost -n 4 -r "mips"
```

To find and list 24 machines, considering only those where the `kflops` attribute is not defined:

```
condor_findhost -n 24 -c "kflops=?=undefined"
```

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team,

Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_glidein*

add a remote grid resource to a local Condor pool

## Synopsis

***condor_glidein*** [**-help**]

***condor_glidein*** [**-admin** *address*] [**-anybody**] [**-archdir** *dir*] [**-basedir** *basedir*] [**-count** *CPUcount*] [**¡Execute Task Options¿**] [**¡Generate File Options¿**] [**-gsi_daemon_name** *cert_name*] [**-idletime** *minutes*] [**-install_gsi_trusted_ca_dir** *path*] [**-install_gsi_gridmap** *file*] [**-localdir** *dir*] [**-memory** *MBytes*] [**-project** *name*] [**-queue** *name*] [**-runtime** *minutes*] [**-runonly**] [**¡Set Up Task Options¿**] [**-suffix** *suffix*] [**-vms** *VMcount*] *¡contact argument¿*

## Description

*condor_glidein* allows the temporary addition of a grid resource to a local Condor pool. The addition is accomplished by installing and executing some of the Condor daemons on the remote grid resource, such that it reports in as part of the local Condor pool. *condor_glidein* accomplishes two separate tasks: set up and execution. These separated tasks allow flexibility, in that the user may use *condor_glidein* to do only one of the tasks or both, in addition to customizing the tasks.

The set up task generates a script that may be used to start the Condor daemons during the execution task, places this script on the remote grid resource, composes and installs a configuration file, and it installs the *condor_master*, *condor_startd* and *condor_starter* daemons on the grid resource.

The execution task runs the script generated by the set up task. The goal of the script is to invoke the *condor_master* daemon. The Condor job *glidein_startup* appears in the queue of the local Condor pool for each invocation of *condor_glidein*. To remove the grid resource from the local Condor pool, use *condor_rm* to remove the *glidein_startup* job.

The Condor jobs to do both the set up and execute tasks utilize Condor-G and Globus protocols (gt2, gt3, or gt4) to communicate with the remote resource. Therefore, an X.509 certificate (proxy) is required for the user running *condor_glidein*.

Specify the remote grid machine with the command line argument *¡contact argument¿*. *¡contact argument¿* takes one of 4 forms:

1. *hostname*
2. *Globus contact string*
3. *hostname/jobmanager-¡schedulername¿*

4. **-contactfile** *filename*

The argument **-contactfile** *filename* specifies the full path and file name of a file that contains Globus contact strings. Each of the resources given by a Globus contact string is added to the local Condor pool.

The set up task of *condor_glidein* copies the binaries for the correct platform from a central server. To obtain access to the server, or to set up your own server, follow instructions on the Glidein Server Setup page, at http://www.cs.wisc.edu/condor/glidein. Set up need only be done once per site, as the installation is never removed.

By default, all files installed on the remote grid resource are placed in the directory `$(HOME)/Condor_glidein`. `$(HOME)` is evaluated and defined on the remote machine using a grid map. This directory must be in a shared file system accessible by all machines that will run the Condor daemons. By default, the daemon's log files will also be written in this directory. Change this directory with the **-localdir** option to make Condor daemons write to local scratch space on the execution machine. For debugging initial problems, it may be convenient to have the log files in the more accessible default directory. If using the default directory, occasionally clean up old log and execute directories to avoid running out of space.

## Examples

To have 10 grid resources running PBS at a grid site with a gatekeeper named gatekeeper.site.edu join the local Condor pool:

```
% condor_glidein -count 10 gatekeeper.site.edu/jobmanager-pbs
```

If you try something like the above and *condor_glidein* is not able to automatically determine everything it needs to know about the remote site, it will ask you to provide more information. A typical result of this process is something like the following command:

```
% condor_glidein \
    -count 10 \
    -arch 6.6.7-i686-pc-Linux-2.4 \
    -setup_jobmanager jobmanager-fork \
    gatekeeper.site.edu/jobmanager-pbs
```

The Condor jobs that do the set up and execute tasks will appear in the queue for the local Condor pool. As a result of a successful glidein, use *condor_status* to see that the remote grid resources are part of the local Condor pool.

A list of common problems and solutions is presented in this manual page.

## Generate File Options

**-genconfig** Create a local copy of the configuration file that may be used on the remote resource. The file is named `glidein_condor_config.<suffix>`. The string defined by <suffix> defaults to the process id (PID) of the *condor_glidein* process or is defined with the **-suffix** command line option. The configuration file may be edited for later use with the **-useconfig** option.

**-genstartup** Create a local copy of the script used on the remote resource to invoke the *condor_master*. The file is named `glidein_startup.<suffix>`. The string defined by <suffix> defaults to the process id (PID) of the *condor_glidein* process or is defined with the **-suffix** command line option. The file may be edited for later use with the **-usestartup** option.

**-gensubmit** Generate submit description files, but do not submit. The submit description file for the set up task is named `glidein_setup.submit.<suffix>`. The submit description file for the execute task is named `glidein_run.submit.<suffix>`. The string defined by <suffix> defaults to the process id (PID) of the *condor_glidein* process or is defined with the **-suffix** command line option.

## Set Up Task Options

**-setuponly** Do only the set up task of *condor_glidein*. This option cannot be run simultaneously with **-runonly**.

**-setup_here** Do the set up task on the local machine, instead of at a remote grid resource. This may be used, for example, to do the set up task of *condor_glidein* in an AFS area that is read-only from the remote grid resource.

**-forcesetup** During the set up task, force the copying of files, even if this overwrites existing files. Use this to push out changes to the configuration.

**-useconfig *config_file*** The set up task copies the specified configuration file, rather than generating one.

**-usestartup *startup_file*** The set up task copies the specified startup script, rather than generating one.

**-setup_jobmanager *jobmanagername*** Identifies the jobmanager on the remote grid resource to receive the files during the set up task. If a reasonable default can be discovered through

MDS, this is optional. *jobmanagername* is a string representing any gt2 name for the job manager. The correct string in most cases will be *jobmanager-fork*. Other common strings may be *jobmanager*, *jobmanager-condor*, *jobmanager-pbs*, and *jobmanager-lsf*.

## Execute Task Options

**-runonly** Starts execution of the Condor daemons on the grid resource. If any of the necessary files or executables are missing, *condor_glidein* exits with an error code. This option cannot be run simultaneously with **-setuponly**.

**-run_here** Runs *condor_master* directly rather than submitting a Condor job that causes the remote execution. To instead generate a script that does this, use **-run_here** in combination with **-gensubmit**. This may be useful for running Condor daemons on resources that are not directly accessible by Condor.

## Options

**-help** Display brief usage information and exit.

**-basedir** ***basedir*** Specifies the base directory on the remote grid resource used for placing files. The default directory is `$(HOME)/Condor_glidein` on the grid resource.

**-archdir** ***dir*** Specifies the directory on the remote grid resource for placement of the Condor executables. The default value for **-archdir** *i*s based upon version information on the grid resource. It is of the form <`basedir`>/<`condor-version`>-<`Globus canonicalsystemname`>. An example of the directory (without the base directory) for Condor version 6.1.13 running on a Sun Sparc machine with Solaris 2.6 is `6.1.13-sparc-sun-solaris-2.6`.

**-localdir** ***dir*** Specifies the directory on the remote grid resource in which to create log and execution subdirectories needed by Condor. If limited disk quota in the home or base directory on the grid resource is a problem, set **-localdir** to a large temporary space, such as `/tmp` or `/scratch`. If the batch system requires invocation of Condor daemons in a temporary scratch directory, '.' may be used for the definition of the **-localdir** option.

**-arch** ***architecture*** Identifies the platform of the required tarball containing the correct Condor daemon executables to download and install. If a reasonable default can be discovered through MDS, this is optional. A list of possible values may be found at http://www.cs.wisc.edu/condor/glidein/binaries. The architecture name is the same as the

tarball name without the suffix `tar.gz`. An example is 6.6.5-i686-pc-Linux-2.4.

**-queue *name*** The argument *name* is a string used at the grid resource to identify a job queue.

**-project *name*** The argument *name* is a string used at the grid resource to identify a project name.

**-memory *MBytes*** The maximum memory size in Megabytes to request from the grid resource.

**-count *CPUcount*** The number of CPUs requested to join the local pool. The default is 1.

**-vms *VMcount*** For machines with multiple CPUs, the CPUs maybe divided up into virtual machines. *VMcount* is the number of virtual machines that results. By default, Condor divides multiple-CPU resources such that each CPU is a virtual machine, each with an equal share of RAM, disk, and swap space. This option configures the number of virtual machines, so that multi-threaded jobs can run in a virtual machine with multiple CPUs. For example, if 4 CPUs are requested and **-vms** is not specified, Condor will divide the request up into 4 virtual machines with 1 CPU each. However, if **-vms** *2* is specified, Condor will divide the request up into 2 virtual machines with 2 CPUs each, and if **-vms** *1* is specified, Condor will put all 4 CPUs into one virtual machine.

**-idletime *minutes*** The amount of time that a remote grid resource will remain idle state, before the daemons shut down. A value of 0 (zero) means that the daemons never shut down due to remaining in the idle state. In this case, the **-runtime** option defines when the daemons shut down. The default value is 20 minutes.

**-runtime *minutes*** The maximum amount of time the Condor daemons on the remote grid resource will run before shutting themselves down. This option is useful for resources with enforced maximum run times. Setting **-runtime** to be a few minutes shorter than the enforced limit gives the daemons time to perform a graceful shut down.

***-anybody*** Sets the Condor `START` expression for the added remote grid resource to `True`. This permits any user's job which can run on the added remote grid resource to run. Without this option, only jobs owned by the user executing *condor_glidein* can execute on the remote grid resource. WARNING: Using this option may violate the usage policies of many institutions.

**-admin *address*** Where to send e-mail with problems. The default is the login of the user running *condor_glidein* at UID domain of the local Condor pool.

**-suffix *X*** Suffix to use when generating files. Default is process id.

**-gsi_daemon_name *cert_name*** Includes and enables GSI authentication in the configuration for the remote grid resource. The argument is the GSI certificate name that the daemons will use to authenticate themselves.

**-install_gsi_trusted_ca_dir *path*** The argument identifies the directory containing the trusted CA certificates that the daemons are to use (for example, `/etc/grid-security/certificates`). The contents of this directory will be installed at the remote site in the directory <`basedir`>`/grid-security`.

**-install_gsi_gridmap *file*** The argument is the file name of the GSI-specific X.509 map file that the daemons will use. The file will be installed at the remote site in <`basedir`>`/grid-security`. The file contains entries mapping certificates to user names. At the very least, it must contain an entry for the certificate given by the command-line option **-gsi_daemon_name** . If other Condor daemons use different certificates, then this file will also list any certificates that the daemons will encounter for the *condor_schedd*, *condor_collector*, and *condor_negotiator*. See section 3.6.3 for more information.

## Exit Status

*condor_glidein* will exit with a status value of 0 (zero) upon complete success, or with non-zero values upon failure. The status value will be 1 (one) if *condor_glidein* encountered an error making a directory, was unable to copy a tar file, encountered an error in parsing the command line, or was not able to gather required information. The status value will be 2 (two) if there was an error in the remote set up. The status value will be 3 (three) if there was an error in remote submission. The status value will be -1 (negative one) if no resource was specified in the command line.

Common problems are listed below. Many of these are best discovered by looking in the `StartLog` log file on the remote grid resource.

**WARNING: The file xxx is not writable by condor** This error occurs when *condor_glidein* is run in a directory that does not have the proper permissions for Condor to access files. An AFS directory does not give Condor the user's AFS ACLs.

**Glideins fail to run due to GLIBC errors** Check the list of available glidein binaries (http://www.cs.wisc.edu/condor/glidein/binaries), and try specifying the architecture name that includes the correct glibc version for the remote grid site.

**Glideins join pool but no jobs run on them** One common cause of this problem is that the remote grid resources are in a different file system domain, and the submitted Condor jobs have an implicit requirement that they must run in the same file system domain. See section 2.5.4 for details on using Condor's file transfer capabilities to solve this problem. Another cause of this problem is a communication failure. For example, a firewall may be preventing the

*condor_negotiator* or the *condor_schedd* daemons from connecting to the *condor_startd* on the remote grid resource. Although work is being done to remove this requirement in the future, it is currently necessary to have full bidirectional connectivity, at least over a restricted range of ports. See page 157 for more information on configuring a port range.

**Glideins run but fail to join the pool** This may be caused by the local pool's security settings or by a communication failure. Check that the security settings in the local pool's configuration file allow write access to the remote grid resource. To not modify the security settings for the pool, run a separate pool specifically for the remote grid resources, and use flocking to balance jobs across the two pools of resources. If the log files indicate a communication failure, then see the next item.

**The startd cannot connect to the collector** This may be caused by several things. One is a firewall. Another is when the compute nodes do not have even outgoing network access. Configuration to work without full network access to and from the compute nodes is still in the experimental stages, so for now, the short answer is that you must at least have a range of open (bidirectional) ports and set up the configuration file as described on page 157. Use the option **-genconfig**, edit the generated configuration file, and then do the glidein execute task with the option **-useconfig**.)

Another possible cause of connectivity problems may be the use of UDP by the *condor_startd* to register itself with the *condor_collector*. Force it to use TCP as described on page 158.

Yet another possible cause of connectivity problems is when the remote grid resources have more than one network interface, and the default one chosen by Condor is not the correct one. One way to fix this is to modify the glidein startup script using the **-genstartup** and **-usestartup** options. The script needs to determine the IP address associated with the correct network interface, and assign this to the environment variable `_condor_NETWORK_INTERFACE`.

**NFS file locking problems** If the **-localdir** option uses files on NFS (not recommended, but sometimes convenient for testing), the Condor daemons may have trouble manipulating file locks. Try inserting the following into the configuration file:

```
IGNORE_NFS_LOCK_ERRORS = True
```

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team,

Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_history*

View log of Condor jobs completed to date

## Synopsis

***condor_history*** [**-help**]

***condor_history*** [**-l**] [**-f** *filename*] [**-format** *formatString AttributeName*] [**-backwards**] [*-constraint expr* | *cluster* | *cluster.process* | *owner* | *-completedsince postgrestimestamp*]

## Description

*condor_history* displays a summary of all Condor jobs listed in the specified history files. If no history files are specified (with the **-f** option), the local history file as specified in Condor's configuration file (`$(SPOOL)/history` by default) is read. The default listing summarizes (in chronological order) each job on a single line, and contains the following items:

**ID** The cluster/process id of the job.

**OWNER** The owner of the job.

**SUBMITTED** The month, day, hour, and minute the job was submitted to the queue.

**CPU_USAGE** Remote user CPU time accumulated by the job to date in days, hours, minutes, and seconds.

**ST** Completion status of the job (C = completed and X = removed).

**COMPLETED** The time the job was completed.

**PRI** User specified priority of the job, ranges from -20 to +20, with higher numbers corresponding to greater priority.

**SIZE** The virtual image size of the executable in Megabytes.

**CMD** The name of the executable.

If a job ID (in the form of *cluster_id* or *cluster_id.proc_id*) or an *owner* is provided, output will be restricted to jobs with the specified IDs and/or submitted by the specified owner. The *-constraint* option can be used to display jobs that satisfy a specified boolean expression.

The history file is kept in chronological order, implying that new entries are appended at the end of the file. As of Condor version 6.7.19, the format of the history file is altered to enable faster reading of the history file backwards (most recent job first). History files written with earlier versions of

Condor, as well as those that have entries of both the older and newer format need to be converted to the new format. See the *condor_convert_history* manual page on page 633 for details on converting history files to the new format.

## Options

**-help** Display usage information and exit.

**-f *filename*** Use the specified file instead of the default history file.

**-backwards** List jobs in reverse chronological order. The job most recently added to the history file is first.

**-match *number*** Limit the number of jobs displayed to *number*.

**-name *quill-name*** Utilize the given Quill database for history information, instead of the history file.

**-constraint *expr*** Display jobs that satisfy the expression.

**-format *formatSpec*AttributeName** Display jobs with a custom format. See *condor_q* man page for details.

**-l** Display job ads in long format.

**-completedsince *postgrestimestamp*** When Quill is enabled, display only job ads that were in the Completed job state on or after the date and time given by the *postgrestimestamp*. The *postgrestimestamp* follows the syntax as given for *PostgreSQL* version 8.0. The behavior of this option is undefined when Quill is *not* enabled.

## Examples

To see all historical jobs since April 1, 2005 at 1pm,

```
%condor_history -completedsince '04/01/2005 13:00'
```

## Exit Status

*condor_history* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_hold*

put jobs in the queue into the hold state

## Synopsis

***condor_hold*** [**-help** | **-version**]

***condor_hold*** [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname* ]| [**-addr** *"<a.b.c.d:port>"*] *cluster. . .*| *cluster.process. . .*| *user. . .* | **-constraint** *expression . . .*

***condor_hold*** [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname* ]| [**-addr** *"<a.b.c.d:port>"*] **-all**

## Description

*condor_hold* places jobs from the Condor job queue in the hold state. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be held are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the QUEUE_SUPER_USERS macro) can place the job on hold.

A job in the hold state remains in the job queue, but the job will not run until released with *condor_release*.

A currently running job that is placed in the hold state by *condor_hold* is sent a hard kill signal. For a standard universe job, this means that the job is removed from the machine without allowing a checkpoint to be produced first.

## Options

**-help** Display usage information

**-version** Display version information

**-pool *centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's hostname and an optional port number

**-name *scheddname*** Send the command to a machine identified by *scheddname*

**-addr** ***"<a.b.c.d:port>"*** Send the command to a machine located at *"<a.b.c.d:port>"*

**-debug** Causes debugging information to be sent to `stderr` based on the value of the configuration variable `TOOL_DEBUG`

***cluster*** Hold all jobs in the specified cluster

***cluster.process*** Hold the specific job in the cluster

***user*** Hold all jobs belonging to specified user

**-constraint** ***expression*** Hold all jobs which match the job ClassAd expression constraint (within quotation marks). Note that quotation marks must be escaped with the backslash characters for most shells.

***-all*** Hold all the jobs in the queue

## See Also

*condor_release* (on page 688)

## General Remarks

To put a PVM universe job on hold, you must put each "process" in the PVM job cluster on hold. (In the PVM universe, each PVM job is assigned its own cluster number, and each machine class is assigned a "process" number in the job's cluster.) Putting a subset of the machine classes for a PVM job on hold is not supported.

## Examples

To place on hold all jobs (of the user that issued the *condor_hold* command) that are not currently running:

```
% condor_hold -constraint "JobStatus!=2"
```

Multiple options within the same command cause the union of all jobs that meet either (or both) of the options to be placed in the hold state. Therefore, the command

```
% condor_hold Mary -constraint "JobStatus!=2"
```

places all of Mary's queued jobs into the hold state, and the constraint holds all queued jobs not currently running. It also sends a hard kill signal to any of Mary's jobs that are currently running. Note that the jobs specified by the constraint will also be Mary's jobs, if it is Mary that issues this example *condor_hold* command.

## Exit Status

*condor_hold* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_master*

The master Condor Daemon

## Synopsis

***condor_master***

## Description

This daemon is responsible for keeping all the rest of the Condor daemons running on each machine in your pool. It spawns the other daemons, and periodically checks to see if there are new binaries installed for any of them. If there are, the *condor_master* will restart the affected daemons. In addition, if any daemon crashes, the *condor_master* will send e-mail to the Condor Administrator of your pool and restart the daemon. The *condor_master* also supports various administrative commands that let you start, stop or reconfigure daemons remotely. The *condor_master* will run on every machine in your Condor pool, regardless of what functions each machine are performing.

Section 3.1.2 in the Administrator's Manual has more information about the *condor_master* and other Condor daemons. See Section 3.9.2 for documentation on command line arguments for *condor_master*.

The `DAEMON_LIST` configuration macro is used by the *condor_master* to provide a per-machine list of daemons that should be started and kept running. For daemons that are specified in the `DC_DAEMON_LIST` configuration macro, the *condor_master* daemon will spawn them automatically appending a *-f* argument. For those listed in `DAEMON_LIST`, but not in `DC_DAEMON_LIST`, there will be no *-f* argument.

## Options

**-n *name*** Provides an alternate name for the *condor_master* to override that given by the `MASTER_NAME` configuration variable.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_master_off*

Shutdown Condor and the *condor_master*

## Synopsis

***condor_master_off*** [**-help**] [**-version**] [**hostname ...**]

## Description

*condor_master_off* no longer exists.

## General Remarks

*condor_master_off* no longer exists as a Condor command. Instead, use

```
condor_off -master
```

to accomplish this task.

## See Also

See the *condor_off* manual page.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and

Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_off*

Shutdown Condor daemons

## Synopsis

***condor_off*** [**-help** | **-version**]

***condor_off*** [**-graceful** | **-fast**] [**-debug**] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>" . . .* ]| [**-all**]
[**-subsystem** *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill*]

***condor_off*** [**-graceful** | **-fast**] [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *hostname* ]| [**-addr** *"<a.b.c.d:port>"*] . . . [ | **-all**]
[**-subsystem** *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill*]

## Description

*condor_off* shuts down a set of the Condor daemons running on a set of one or more machines. It does this cleanly so that checkpointable jobs may gracefully exit with minimal loss of work.

The command *condor_off* without any arguments will shut down all daemons except *condor_master*. The *condor_master* can then handle both local and remote requests to restart the other Condor daemons if need be. To restart Condor running on a machine, see the *condor_on* command.

With the **-subsystem** *master* option, *condor_off* will shut down all daemons including the *condor_master*. Specification using the **-subsystem** option will shut down only the specified daemon.

For security purposes (authentication and authorization), this command requires an administrator's level of access. See section 3.6.1 on page 261 for further explanation.

## Options

**-help** Display usage information

**-version** Display version information

**-graceful** Gracefully shutdown daemons (the default)

**-fast** Quickly shutdown daemons

**-peaceful** Wait indefinitely for jobs to finish

**-debug** Causes debugging information to be sent to `stderr` based on the value of the configuration variable `TOOL_DEBUG`

**-pool** ***centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's hostname and an optional port number

**-name** ***hostname*** Send the command to a machine identified by *hostname*

***hostname*** Send the command to a machine identified by *hostname*

**-addr** ***"<a.b.c.d:port>"*** Send the command to a machine's master located at *"<a.b.c.d:port>"*

***"<a.b.c.d:port>"*** Send the command to a machine located at *"<a.b.c.d:port>"*

**-all** Send the command to all machines in the pool

**-subsystem** ***master*** | ***startd*** | ***schedd*** | ***collector*** | ***negotiator*** | ***kbdd*** | ***quill*** Send the command to the named daemon. Without this option, the command is sent to the *condor_master* daemon.

## Exit Status

*condor_off* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Examples

To shut down all daemons (other than *condor_master*) on the local host:

```
% condor_off
```

To shut down only the *condor_collector* on three named machines:

```
% condor_off  cinnamon cloves vanilla -subsystem collector
```

To shut down daemons within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command shuts down all daemons except the *condor_master* on the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_off  -pool condor.cae.wisc.edu -name cae17
```

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_on*

Start up Condor daemons

## Synopsis

***condor_on*** [**-help** | **-version**]

***condor_on*** [**-debug**] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>"* . . . ]| [**-all**]
[**-subsystem** *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill*]

***condor_on*** [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *hostname* ]| [**-addr** *"<a.b.c.d:port>"*] . . . [ | **-all**]
[**-subsystem** *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill*]

## Description

*condor_on* starts up a set of the Condor daemons on a set of machines. This command assumes that the *condor_master* is already running on the machine. If this is not the case, *condor_on* will fail complaining that it cannot find the address of the master. The command *condor_on* with no arguments or with the **-subsystem** *master* option will tell the *condor_master* to start up the Condor daemons specified in the configuration variable `DAEMON_LIST`. If a daemon other than the *condor_master* is specified with the **-subsystem** option, *condor_on* starts up only that daemon.

This command cannot be used to start up the *condor_master* daemon.

For security purposes (authentication and authorization), this command requires an administrator's level of access. See section 3.6.1 on page 261 for further explanation.

## Options

**-help** Display usage information

**-version** Display version information

**-pool *centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's hostname and an optional port number

**-name *hostname*** Send the command to a machine identified by *hostname*

***hostname*** Send the command to a machine identified by *hostname*

**-addr *"<a.b.c.d:port>"*** Send the command to a machine's master located at *"<a.b.c.d:port>"*

***"<a.b.c.d:port>"*** Send the command to a machine located at *"<a.b.c.d:port>"*

**-all** Send the command to all machines in the pool

**-subsystem *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill*** Send the command to the named daemon. Without this option, the command is sent to the *condor_master* daemon.

**-debug** Causes debugging information to be sent to `stderr` based on the value of the configuration variable `TOOL_DEBUG`

## Exit Status

*condor_on* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Examples

To begin running all daemons (other than *condor_master*) given in the configuration variable `DAEMON_LIST` on the local host:

```
% condor_on
```

To start up only the *condor_negotiator* on two named machines:

```
% condor_on  robin cardinal -subsystem negotiator
```

To start up only a daemon within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command starts up only the *condor_schedd* daemon on the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_on -pool condor.cae.wisc.edu -name cae17 -subsystem schedd
```

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_preen*

remove extraneous files from Condor directories

## Synopsis

***condor_preen*** [**-mail**] [**-remove**] [**-verbose**]

## Description

*condor_preen* examines the directories belonging to Condor, and removes extraneous files and directories which may be left over from Condor processes which terminated abnormally either due to internal errors or a system crash. The directories checked are the LOG, EXECUTE, and SPOOL directories as defined in the Condor configuration files. *condor_preen* is intended to be run as user root (or user condor) periodically as a backup method to ensure reasonable file system cleanliness in the face of errors. This is done automatically by default by the *condor_master*. It may also be explicitly invoked on an as needed basis.

When *condor_preen* cleans the SPOOL directory, it always leaves behind the files specified in the `VALID_SPOOL_FILES` list in your config file. For the log directory, the only files removed or reported are those listed in the `INVALID_LOG_FILES` list. The reason for this difference is that, in general, you want to leave all files in the LOG directory alone, with a few exceptions (namely, core files). *condor_preen* still works if you supply a `VALID_LOG_FILES` list instead, but this usage is depricated. There are new log files for different things introduced all the time, and you wouldn't want to have to keep updating the list of files to leave alone in the LOG directory. For example, the SMP startd can spawn an arbitrary number of *condor_starter* processes, each with its own log file. On the other hand, there are only a small, fixed number of files in the SPOOL directory that the *condor_schedd* needs to keep around, so it is easier to specify the files you want to keep instead of the ones you want to get rid of.

## Options

**-mail** Send mail to the `PREEN_ADMIN` as defined in the Condor configuration files instead of writing to the standard output

**-remove** Remove the offending files and directories rather than just reporting on them

**-verbose** List all files found in the Condor directories, even those which are not considered extraneous

## Exit Status

*condor_preen* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_prio*

change priority of jobs in the condor queue

## Synopsis

***condor_prio*** [**-p** *priority*] [**+** | **-** *value*] [**-n** *schedd_name*] [**-pool** *pool_name*] *cluster* | *cluster.process* | *username* | *-a*

## Description

*condor_prio* changes the priority of one or more jobs in the condor queue. If a cluster_id and a process_id are both specified, *condor_prio* attempts to change the priority of the specified process. If a cluster_id is specified without a process_id, *condor_prio* attempts to change priority for all processes belonging to the specified cluster. If a username is specified, *condor_prio* attempts to change priority of all jobs belonging to that user. If the -a flag is set, *condor_prio* attempts to change priority of all jobs in the condor queue. The user must specify a priority adjustment or new priority. If the -p option is specified, the priority of the job(s) are set to the next argument. The user can also adjust the priority by supplying a + or - immediately followed by a digit. The priority of a job can be any integer, with higher numbers corresponding to greater priority. Only the owner of a job or the super user can change the priority for it.

The priority changed by *condor_prio* is only compared to the priority of other jobs owned by the same user and submitted from the same machine. See the "Condor Users and Administrators Manual" for further details on Condor's priority scheme.

## Options

**-p *priority*** Set priority to the specified value

**+** | **-** ***value*** Change priority by the specified value

**-n *schedd_name*** Change priority of jobs queued at the specified schedd in the local pool

**-pool *pool_name*** Change priority of jobs queued at the specified schedd in the specified pool

**-a** Change priority of all the jobs in the queue

## Exit Status

*condor_prio* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_q*

Display information about jobs in queue

## Synopsis

***condor_q*** [**-help**]

***condor_q*** [**-debug**] [**-global**] [**-submitter** *submitter*] [**-name** *name*] [**-pool** *centralmanagerhostname[:portnumber]*] [**-analyze**] [**-better-analyze**] [**-run**] [**-hold**] [**-globus**] [**-goodput**] [**-io**] [**-dag**] [**-long**] [**-xml**] [**-format** *fmt attr*] [**-cputime**] [**-currentrun**] [**-avgqueuetime**] [**-jobads** *file*] [**-machineads** *file*] [**-direct** *rdbms* | *quilld* | *schedd*] [{*cluster* | *cluster.process* | *owner* | ***-constraint*** *expression* . . .} ]

## Description

*condor_q* displays information about jobs in the Condor job queue. By default, *condor_q* queries the local job queue but this behavior may be modified by specifying:

- the **-global** option, which queries all job queues in the pool
- a schedd name with the **-name** option, which causes the queue of the named schedd to be queried
- a submitter with the **-submitter** option, which causes all queues of the named submitter to be queried

To restrict the display to jobs of interest, a list of zero or more restrictions may be supplied. Each restriction may be one of:

- a *cluster* and a *process* matches jobs which belong to the specified cluster and have the specified process number
- a *cluster* without a *process* matches all jobs belonging to the specified cluster
- a *owner* matches all jobs owned by the specified owner
- a **-constraint** *expression* which matches all jobs that satisfy the specified ClassAd expression. (See section 4.1 for a discussion of ClassAd expressions.)

If no *owner* restrictions are present in the list, the job matches the restriction list if it matches at least one restriction in the list. If *owner* restrictions are present, the job matches the list if it matches one of the *owner* restrictions *and* at least one non-owner restriction.

If the **-long** option is specified, *condor_q* displays a long description of the queried jobs by printing the entire job classad. The attributes of the job classad may be displayed by means of the **-format** option, which displays attributes with a `printf(3)` format. Multiple **-format** options may be specified in the option list to display several attributes of the job. If neither **-long** or **-format** are specified, *condor_q* displays a a one line summary of information as follows:

**ID** The cluster/process id of the condor job.

**OWNER** The owner of the job.

**SUBMITTED** The month, day, hour, and minute the job was submitted to the queue.

**RUN_TIME** Wall-clock time accumulated by the job to date in days, hours, minutes, and seconds.

**ST** Current status of the job, which varies somewhat according to the job universe and the timing of updates. U = unexpanded (never been run), H = on hold, R = running, I = idle (waiting for a machine to execute on), C = completed, and X = removed.

**PRI** User specified priority of the job, ranges from -20 to +20, with higher numbers corresponding to greater priority.

**SIZE** The virtual image size of the executable in megabytes.

**CMD** The name of the executable.

If the **-dag** option is specified, the OWNER column is replaced with NODENAME for jobs started by Condor DAGMan.

NOTE: The **-dag** option has no effect on a pre-v6.3.0 Condor queue, because older *condor_schedd* daemons don't pass the necessary DAG information to their jobs.

If the **-run** option is specified, the ST, PRI, SIZE, and CMD columns are replaced with:

**HOST(S)** The host where the job is running. For PVM jobs, a host count is displayed instead.

If the **-globus** option is specified, the ST, PRI, SIZE, and CMD columns are replaced with:

**STATUS** The state that Condor believes the job is in. Possible values are

**PENDING** The job is waiting for resources to become available in order to run.

**ACTIVE** The job has received resources, and the application is executing.

**FAILED** The job terminated before completion because of an error, user-triggered cancel, or system-triggered cancel.

**DONE** The job completed successfully.

**SUSPENDED** The job has been suspended. Resources which were allocated for this job may have been released due to a scheduler-specific reason.

**UNSUBMITTED** The job has not been submitted to the scheduler yet, pending the reception of the GLOBUS_GRAM_PROTOCOL_JOB_SIGNAL_COMMIT_REQUEST signal from a client.

**STAGE_IN** The job manager is staging in files, in order to run the job.

**STAGE_OUT** The job manager is staging out files generated by the job.

**UNKNOWN**

**MANAGER** A guess at what remote batch system is running the job. It is a guess, because Condor looks at the Globus jobmanager contact string to attempt identification. If the value is fork, the job is running on the remote host without a jobmanager. Values may also be condor, lsf, or pbs.

**HOST** The host to which the job was submitted.

**EXECUTABLE** The job as specified as the executable in the submit description file.

If the **-goodput** option is specified, the ST, PRI, SIZE, and CMD columns are replaced with:

**GOODPUT** The percentage of RUN_TIME for this job which has been saved in a checkpoint. A low GOODPUT value indicates that the job is failing to checkpoint. If a job has not yet attempted a checkpoint, this column contains `[?????]`.

**CPU_UTIL** The ratio of CPU_TIME to RUN_TIME for checkpointed work. A low CPU_UTIL indicates that the job is not running efficiently, perhaps because it is I/O bound or because the job requires more memory than available on the remote workstations. If the job has not (yet) checkpointed, this column contains `[??????]`.

**Mb/s** The network usage of this job, in Megabits per second of run-time.

If the **-io** option is specified, the ST, PRI, SIZE, and CMD columns are replaced with:

READ The total number of bytes the application has read from files and sockets.

WRITE The total number of bytes the application has written to files and sockets.

SEEK The total number of seek operations the application has performed on files.

XPUT The effective throughput (average bytes read and written per second) from the application's point of view.

BUFSIZE The maximum number of bytes to be buffered per file.

BLOCKSIZE The desired block size for large data transfers.

These fields are updated when a job produces a checkpoint or completes. If a job has not yet produced a checkpoint, this information is not available.

If the **-cputime** option is specified, the RUN_TIME column is replaced with:

**CPU_TIME** The remote CPU time accumulated by the job to date (which has been stored in a checkpoint) in days, hours, minutes, and seconds. (If the job is currently running, time accumulated during the current run is *not* shown. If the job has not produced a checkpoint, this column contains 0+00:00:00.)

The *-analyze* option may be used to determine why certain jobs are not running by performing an analysis on a per machine basis for each machine in the pool. The reasons may vary among failed constraints, insufficient priority, resource owner preferences and prevention of preemption by the `PREEMPTION_REQUIREMENTS` expression. If the *-long* option is specified along with the *-analyze* option, the reason for failure is displayed on a per machine basis.

The *-better-analyze* option does a more thorough job of determining why jobs are not running than *-analyze*. There are scalability issues present when run on a pool with a large number of machines, as well as when run to analyze a large number of queued jobs. The *-better-analyze* option make take an excessively long time to complete in these cases. Therefore, it is recommended to constrain *-better-analyze* to only analyze one job at a time.

## Options

**-help** Get a brief description of the supported options

**-global** Get queues of all the submitters in the system

**-debug** Causes debugging information to be sent to `stderr` based on the value of the configuration variable `TOOL_DEBUG`

**-submitter *submitter*** List jobs of specific submitter from all the queues in the pool

**-pool *centralmanagerhostname[:portnumber]*** Use the *centralmanagerhostname* as the central manager to locate schedds. (The default is the `COLLECTOR_HOST` specified in the configuration file.

**-analyze** Perform an approximate analysis to determine how many resources are available to run the requested jobs. These results are only meaningful for jobs using Condor's matchmaker. This option is never meaningful for Scheduler universe jobs and only meaningful for grid universe jobs doing matchmaking.

**-better-analyze** Perform a more time-consuming, but potentially more extensive analysis to determine how many resources are available to run the requested jobs.

**-run** Get information about running jobs.

**-hold** Get information about jobs in the hold state. Also displays the time the job was placed into the hold state and the reason why the job was placed in the hold state.

**-globus** Get information only about jobs submitted to grid resources described as **gt2**, **gt3**, or **gt4**.

**-goodput** Display job goodput statistics.

**-io** Display job input/output summaries.

**-dag** Display DAG jobs under their DAGMan.

**-name *name*** Show only the job queue of the named schedd

**-long** Display job ads in long format

**-xml** Display job ads in xml format. The xml format is fully defined at http://www.cs.wisc.edu/condor/classad/refman/.

**-format *fmt attr*** Display attribute or expression *attr* in format *fmt*. To display the attribute or expression the format must contain a single `printf(3)` style conversion specifier. Attributes must be from the job ClassAd. Expressions are ClassAd expressions and may refer to attributes in the job ClassAd. If the attribute is not present in a given ClassAd and cannot be parsed as an expression, then the format option will be silently skipped. The conversion specifier must match the type of the attribute or expression. %s is suitable for strings such as `Owner`, %d for integers such as `ClusterId`, and %f for floating point numbers such as `RemoteWallClockTime`. An incorrect format will result in undefined behavior. Do not use more than one conversion specifier in a given format. More than one conversion specifier will result in undefined behavior. To output multiple attributes repeat the **-format** option once for each desired attribute. Like `printf(3)` style formats, you can include other text that will be reproduced directly. You can specify a format without any conversion specifiers but you must still give attribute. You can include \n to specify a line break.

**-cputime** Instead of wall-clock allocation time (RUN_TIME), display remote CPU time accumulated by the job to date in days, hours, minutes, and seconds. (If the job is currently running, time accumulated during the current run is *not* shown.)

**-currentrun** Normally, RUN_TIME contains all the time accumulated during the current run plus all previous runs. If this option is specified, RUN_TIME only displays the time accumulated so far on this current run.

**-avgqueuetime** Display the average of time spent in the queue, considering all jobs not completed (those that do not have `JobStatus == 4` or `JobStatus == 3`.

**-jobads *file*** Display jobs from a list of ClassAds from a file, instead of the real ClassAds from the *condor_schedd* daemon. This is most useful for debugging purposes. The ClassAds appear as if *condor_q* **-l** is used with the header stripped out.

**-machineads *file*** When doing analysis, use the machine ads from the file instead of the ones from the *condor_collector* daemon. This is most useful for debugging purposes. The ClassAds appear as if *condor_status* **-l** is used.

**-direct *rdbms* | *quilld* | *schedd*** When the use of Quill is enabled, this option allows a direct query to either the rdbms, the *condor_quill* daemon, or the *condor_schedd* daemon for the requested queue information. It also prevents the queue location discovery algorithm from failing over to alternate sources of information for the queue in case of error. It is useful for debugging an installation of Quill. One of the strings *rdbms*, *quilld*, or *schedd* is required with this option.

**Restriction list** The restriction list may have zero or more items, each of which may be:

***cluster*** match all jobs belonging to cluster

***cluster.proc*** match all jobs belonging to cluster with a process number of *proc*

**-constraint *expression*** match all jobs which match the ClassAd expression constraint

A job matches the restriction list if it matches any restriction in the list Additionally, if *owner* restrictions are supplied, the job matches the list only if it also matches an *owner* restriction.

## General Remarks

The default output from *condor_q* is formatted to be human readable, not script readable. In an effort to make the output fit within 80 characters, values in some fields might be truncated. Furthermore, the Condor Project can (and does) change the formatting of this default output as we see fit. Therefore, any script that is attempting to parse data from *condor_q* is strongly encouraged to use the **-format** option (described above, examples given below).

Although *-analyze* provides a very good first approximation, the analyzer cannot diagnose all possible situations because the analysis is based on instantaneous and local information. Therefore, there

are some situations (such as when several submitters are contending for resources, or if the pool is rapidly changing state) which cannot be accurately diagnosed.

*-goodput*, *-cputime*, and *-io* are most useful for STANDARD universe jobs, since they rely on values computed when a job checkpoints.

## Examples

The **-format** option provides a way to specify both the job attributes and formatting of those attributes. There must be only one conversion specification per **-format** option. As an example, to list only Jane Doe's jobs in the queue, choosing to print and format only the owner of the job, the command line arguments for the job, and the process ID of the job:

```
%condor_q -submitter jdoe -format "%s" Owner -format " %s " Args -format "ProcId = %d\n" ProcId
jdoe 16386 2800 ProcId = 0
jdoe 16386 3000 ProcId = 1
jdoe 16386 3200 ProcId = 2
jdoe 16386 3400 ProcId = 3
jdoe 16386 3600 ProcId = 4
jdoe 16386 4200 ProcId = 7
```

To display only the JobID's of Jane Doe's jobs you can use the following.

```
%condor_q -submitter jdoe -format "%d." ClusterId -format "%d\n" ProcId
27.0
27.1
27.2
27.3
27.4
27.7
```

An example that shows the difference (first set of output) between not using an option to *condor_q* and (second set of output) using the **-globus** option:

```
 ID      OWNER            SUBMITTED     RUN_TIME ST PRI SIZE CMD
 100.0   smith          12/11 13:20   0+00:00:02 R  0   0.0  sleep 10

1 jobs; 0 idle, 1 running, 0 held
```

```
 ID      OWNER          STATUS  MANAGER  HOST                EXECUTABLE
 100.0   smith         ACTIVE fork     grid.example.com       /bin/sleep
```

## Exit Status

*condor_q* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_qedit*

modify job attributes

## Synopsis

***condor_qedit*** [**-n** *schedd-name*] [**-pool** *pool-name*] {*cluster* | *cluster.proc* | *owner* | *-constraint constraint*} *attribute-name attribute-value . . .*

## Description

*condor_qedit* modifies job attributes in the Condor job queue. The jobs are specified either by cluster number, cluster.proc job ID, owner, or by a ClassAd constraint expression. The attribute-value may be any ClassAd expression (integer, floating point number, string, expression).

## Options

**-n *schedd-name*** Modify job attributes in the queue of the specified schedd

**-pool *pool-name*** Modify job attributes in the queue of the schedd specified in the specified pool

## Examples

```
% condor_qedit -name north.cs.wisc.edu -pool condor.cs.wisc.edu 249.0 answer 42
Set attribute "answer".
% condor_qedit -name perdita 1849.0 In '"myinput"'
Set attribute "In".
% condor_qedit jbasney NiceUser TRUE
Set attribute "NiceUser".
% condor_qedit -constraint 'JobUniverse == 1' Requirements '(Arch == "INTEL") && (OpS
Set attribute "Requirements".
```

## General Remarks

You can view the list of attributes with their current values for a job with *condor_q* **-long**.

Strings must be specified with quotes (for example, ’”String”’).

If a job is currently running, modified attributes for that job will not take effect until the job restarts. For example, attempting to modify `PeriodicRemove` to affect when a running job will be removed from the queue will *not* affect the job, unless the job happens to be evicted from a machine and returns to the queue to be run again later. This is also true for other expressions, such as `PeriodicHold`, `PeriodicRelease`, and so forth.

*condor_qedit* will not allow modification of the following attributes to ensure security and correctness: `Owner`, `ClusterId`, `ProcId`, `MyType`, `TargetType`, and `JobStatus`.

Please use *condor_hold* to place a job in the hold state, and use *condor_release* to release a held job, instead of attempting to modify `JobStatus` directly.

## Exit Status

*condor_qedit* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_reconfig*

Reconfigure Condor daemons

## Synopsis

***condor_reconfig*** [**-help** | **-version**]

***condor_reconfig*** [**-debug**] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>" . . .* ]| [**-all**]
[**-subsystem** *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill*] [**-full**]

***condor_reconfig*** [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *hostname* ]|
[**-addr** *"<a.b.c.d:port>"*] . . . [ | **-all**]
[**-subsystem** *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill*] [**-full**]

## Description

*condor_reconfig* reconfigures all of the Condor daemons in accordance with the current status of the Condor configuration file(s). Once reconfiguration is complete, the daemons will behave according to the policies stated in the configuration file(s). The main exception is with the `DAEMON_LIST` variable, which will only be updated if the *condor_restart* command is used. There are a few other configuration settings that can only be changed if the Condor daemons are restarted. Whenever this is the case, it will be mentioned in section 3.3 on page 138 which lists all of the settings used to configure Condor. In general, *condor_reconfig* should be used when making changes to the configuration files, since it is faster and more efficient than restarting the daemons.

The command *condor_reconfig* with no arguments or with the **-subsystem** *master* option will cause the reconfiguration of the *condor_master* daemon and all the child processes of the *condor_master*.

For security purposes (authentication and authorization), this command requires an administrator's level of access. Note that changes to the `ALLOW_*` and `DENY_*` configuration variables require the **-full** option. See section 3.6.1 on page 261 for further explanation.

## Options

**-help** Display usage information

**-version** Display version information

**-full** Perform a full reconfiguration. In addition to re-reading the configuration files, a full reconfiguration will clear cached DNS information in the daemons. Use this option only

when the DNS information needs to be reinitialized.

**-debug** Causes debugging information to be sent to `stderr` based on the value of the configuration variable `TOOL_DEBUG`

**-pool** ***centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's hostname and an optional port number

**-name** ***hostname*** Send the command to a machine identified by *hostname*

***hostname*** Send the command to a machine identified by *hostname*

**-addr** ***"<a.b.c.d:port>"*** Send the command to a machine's master located at *"<a.b.c.d:port>"*

***"<a.b.c.d:port>"*** Send the command to a machine located at *"<a.b.c.d:port>"*

**-all** Send the command to all machines in the pool

**-subsystem** ***master*** | ***startd*** | ***schedd*** | ***collector*** | ***negotiator*** | ***kbdd*** | ***quill*** Send the command to the named daemon. Without this option, the command is sent to the *condor_master* daemon.

## Exit Status

*condor_reconfig* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Examples

To reconfigure the *condor_master* and all its children on the local host:

```
% condor_reconfig
```

To reconfigure only the *condor_startd* on a named machine:

```
% condor_reconfig -name bluejay -subsystem startd
```

To reconfigure a machine within a pool other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command reconfigures the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_reconfig -pool condor.cae.wisc.edu -name cae17
```

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_reconfig_schedd*

Reconfigure condor schedd

## Synopsis

***condor_reconfig_schedd*** [**-help**] [**-version**] [**hostname ...**]

## Description

*condor_reconfig_schedd* no longer exists.

## General Remarks

*condor_reconfig_schedd* no longer exists as a Condor command. Instead, use

```
condor_reconfig -schedd
```

to accomplish this task.

## See Also

See the *condor_reconfig* manual page.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and

Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_release*

release held jobs in the Condor queue

## Synopsis

***condor_release*** [**-help** | **-version**]

***condor_release*** [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname* ]| [**-addr** *"<a.b.c.d:port>"*] *cluster...*| *cluster.process...*| *user...* | **-constraint** *expression ...*

***condor_release*** [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname* ]| [**-addr** *"<a.b.c.d:port>"*] **-all**

## Description

*condor_release* releases jobs from the Condor job queue that were previously placed in hold state. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be released are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the `QUEUE_SUPER_USERS` macro) can release the job.

## Options

**-help** Display usage information

**-version** Display version information

**-pool *centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's hostname and an optional port number

**-name *scheddname*** Send the command to a machine identified by *scheddname*

**-addr *"<a.b.c.d:port>"*** Send the command to a machine located at *"<a.b.c.d:port>"*

**-debug** Causes debugging information to be sent to `stderr` based on the value of the configuration variable `TOOL_DEBUG`

***cluster*** Release all jobs in the specified cluster

***cluster.process*** Release the specific job in the cluster

***user*** Release jobs belonging to specified user

**-constraint *expression*** Release all jobs which match the job ClassAd expression constraint

***-all*** Release all the jobs in the queue

## See Also

*condor_hold* (on page 656)

## General Remarks

When releasing a held PVM universe job, you must release the entire job cluster. (In the PVM universe, each PVM job is assigned its own cluster number, and each machine class is assigned a "process" number in the job's cluster.) Releasing a subset of the machine classes for a PVM job is not supported.

## Examples

To release all of the jobs of a user named Mary:

```
% condor_release Mary
```

## Exit Status

*condor_release* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_reschedule*

Update scheduling information to the central manager

## Synopsis

***condor_reschedule*** [**-help** | **-version**]

***condor_reschedule*** [**-debug**] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>" . . .* ]| [**-all**]

***condor_reschedule*** [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *hostname* ]| [**-addr** *"<a.b.c.d:port>"*] . . . [ | **-all**]

## Description

*condor_reschedule* updates the information about a set of machines' resources and jobs to the central manager. This command is used to force an update before viewing the current status of a machine. Viewing the status of a machine is done with the *condor_status* command. *condor_reschedule* also starts a new negotiation cycle between resource owners and resource providers on the central managers, so that jobs can be matched with machines right away. This can be useful in situations where the time between negotiation cycles is somewhat long, and an administrator wants to see if a job in the queue will get matched without waiting for the next negotiation cycle.

A new negotiation cycle cannot occur more frequently than every 20 seconds. Requests for new negotiation cycle within that 20 second window will be deferred until 20 seconds have passed since that last cycle.

## Options

**-help** Display usage information

**-version** Display version information

**-pool** ***centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's hostname and an optional port number

**-name** ***hostname*** Send the command to a machine identified by *hostname*

***hostname*** Send the command to a machine identified by *hostname*

**-addr** ***"<a.b.c.d:port>"*** Send the command to a machine's master located at *"<a.b.c.d:port>"*

***"<a.b.c.d:port>"*** Send the command to a machine located at *"<a.b.c.d:port>"*

**-all** Send the command to all machines in the pool

**-debug** Causes debugging information to be sent to `stderr` based on the value of the configuration variable `TOOL_DEBUG`

## Exit Status

*condor_reschedule* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Examples

To update the information on three named machines:

```
% condor_reschedule robin cardinal bluejay
```

To reschedule on a machine within a pool other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command reschedules the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_reschedule -pool condor.cae.wisc.edu -name cae17
```

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized

without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_restart*

Restart the a set of Condor daemons

## Synopsis

***condor_restart*** [**-help** | **-version**]

***condor_restart*** [**-debug**] [**-name** *hostname* | *hostname* | **-addr** *"<a.b.c.d:port>"* | *"<a.b.c.d:port>" . . .* ]| [**-all**]
[**-subsystem** *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill*]

***condor_restart*** [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *hostname* ]|
[**-addr** *"<a.b.c.d:port>"*] . . . [ | **-all**]
[**-subsystem** *master* | *startd* | *schedd* | *collector* | *negotiator* | *kbdd* | *quill*]

## Description

*condor_restart* restarts a set of Condor daemon(s) on a set of machines. The daemon(s) will be put into a consistent state, killed, and then started anew.

If, for example, the *condor_master* needs to be restarted again with a fresh state, this is the command that should be used to do so. If the `DAEMON_LIST` variable in the configuration file has been changed, this command is used to restart the *condor_master* in order to see this change. The *condor_reconfigure* command cannot be used in the case where the `DAEMON_LIST` expression changes.

The command *condor_restart* with no arguments or with the **-subsystem** *master* option will safely shut down all running jobs and all submitted jobs from the machine(s) being restarted, then shut down all the child daemons of the *condor_master*, and then restart the *condor_master*. This, in turn, will allow the *condor_master* to start up other daemons as specified in the `DAEMON_LIST` configuration file entry.

For security purposes (authentication and authorization), this command requires an administrator's level of access. See section 3.6.1 on page 261 for further explanation.

## Options

**-help** Display usage information

**-version** Display version information

**-pool** ***centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's hostname and an optional port number

**-name** ***hostname*** Send the command to a machine identified by *hostname*

***hostname*** Send the command to a machine identified by *hostname*

**-addr** ***"<a.b.c.d:port>"*** Send the command to a machine's master located at *"<a.b.c.d:port>"*

***"<a.b.c.d:port>"*** Send the command to a machine located at *"<a.b.c.d:port>"*

**-all** Send the command to all machines in the pool

**-subsystem** ***master*** | ***startd*** | ***schedd*** | ***collector*** | ***negotiator*** | ***kbdd*** | ***quill*** Send the command to the named daemon. Without this option, the command is sent to the *condor_master* daemon.

**-debug** Causes debugging information to be sent to `stderr` based on the value of the configuration variable `TOOL_DEBUG`

## Exit Status

*condor_restart* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Examples

To restart the *condor_master* and all its children on the local host:

```
% condor_restart
```

To restart only the *condor_startd* on a named machine:

```
% condor_restart -name bluejay -subsystem startd
```

To restart a machine within a pool other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command restarts the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_restart -pool condor.cae.wisc.edu -name cae17
```

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_rm*

remove jobs from the Condor queue

## Synopsis

***condor_rm*** [**-help** | **-version**]

***condor_rm*** [**-debug**] [**-forcex**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname* ]| [**-addr** *"<a.b.c.d:port>"*] *cluster...*| *cluster.process...*| *user...* | **-constraint** *expression ...*

***condor_rm*** [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname* ]| [**-addr** *"<a.b.c.d:port>"*] **-all**

## Description

*condor_rm* removes one or more jobs from the Condor job queue. If the **-name** option is specified, the named *condor_schedd* is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be removed are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the QUEUE_SUPER_USERS macro) can remove the job.

When removing a grid job, the job may remain in the "X" state for a very long time. This is normal, as Condor is attempting to communicate with the remote scheduling system, ensuring that the job has been properly cleaned up. If it takes too long, or in rare circumstances is never removed, the job may be forced to leave the job queue by using the **-forcex** option. This forcibly removes jobs that are in the "X" state without attempting to finish any clean up at the remote scheduler.

## Options

**-help** Display usage information

**-version** Display version information

**-pool *centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's hostname and an optional port number

**-name *scheddname*** Send the command to a machine identified by *scheddname*

**-addr** ***"<a.b.c.d:port>"*** Send the command to a machine located at *"<a.b.c.d:port>"*

**-debug** Causes debugging information to be sent to `stderr` based on the value of the configuration variable `TOOL_DEBUG`

**-forcex** Force the immediate local removal of jobs in the 'X' state (only affects jobs already being removed)

***cluster*** Remove all jobs in the specified cluster

***cluster.process*** Remove the specific job in the cluster

***user*** Remove jobs belonging to specified user

**-constraint** ***expression*** Remove all jobs which match the job ClassAd expression constraint

**-all** Remove all the jobs in the queue

## General Remarks

When removing a PVM universe job, you should always remove the entire job cluster. (In the PVM universe, each PVM job is assigned its own cluster number, and each machine class is assigned a "process" number in the job's cluster.) Removing a subset of the machine classes for a PVM job is not supported.

Use the *-forcex* argument with caution, as it will remove jobs from the local queue immediately, but can "orphan" parts of the job that are running remotely and haven't yet been stopped or removed.

## Examples

To remove all jobs of a user named Mary that are not currently running:

```
% condor_rm Mary -constraint Activity!=\"Busy\"
```

Note that quotation marks must be escaped with the backslash characters for most shells.

## Exit Status

*condor_rm* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_run*

Submit a shell command-line as a Condor job

## Synopsis

***condor_run*** [**-u** *universe*] *"shell command"*

## Description

*condor_run* bundles a shell command line into a Condor job and submits the job. The *condor_run* command waits for the Condor job to complete, writes the job's output to the terminal, and exits with the exit status of the Condor job. No output appears until the job completes.

Enclose the shell command line in double quote marks, so it may be passed to *condor_run* without modification. *condor_run* will not read input from the terminal while the job executes. If the shell command line requires input, redirect the input from a file, as illustrated by the example

```
% condor_run "myprog < input.data"
```

*condor_run* jobs rely on a shared file system for access to any necessary input files. The current working directory of the job must be accessible to the machine within the Condor pool where the job runs.

Specialized environment variables may be used to specify requirements for the machine where the job may run.

**CONDOR_ARCH** Specifies the architecture of the required platform. Values will be the same as the `Arch` machine ClassAd attribute.

**CONDOR_OPSYS** Specifies the operating system of the required platform. Values will be the same as the `OpSys` machine ClassAd attribute.

**CONDOR_REQUIREMENTS** Specifies any additional requirements for the Condor job. It is recommended that the value defined for `CONDOR_REQUIREMENTS` be enclosed in parenthesis.

When one or more of these environment variables is specified, the job is submitted with:

```
Requirements = $CONDOR_REQUIREMENTS && Arch == $CONDOR_ARCH && \
   OpSys == $CONDOR_OPSYS
```

Without these environment variables, the job receives the default requirements expression, which requests a machine of the same platform as the machine on which *condor_run* is executed.

All environment variables set when *condor_run* is executed will be included in the environment of the Condor job.

*condor_run* removes the Condor job from the queue and deletes its temporary files, if *condor_run* is killed before the Condor job completes.

## Options

**-u *universe*** Submit the job under the specified universe. The default is vanilla. While any universe may be specified, only the vanilla, standard, scheduler, and local universes result in a submit description file that may work properly.

## Examples

*condor_run* may be used to compile an executable on a different platform. As an example, first set the environment variables for the required platform:

```
% setenv CONDOR_ARCH "SUN4u"
% setenv CONDOR_OPSYS "SOLARIS28"
```

Then, use *condor_run* to submit the compilation as in the following three examples.

```
% condor_run "f77 -O -o myprog myprog.f"
```

or

```
% condor_run "make"
```

or

```
% condor_run "condor_compile cc -o myprog.condor myprog.c"
```

## Files

*condor_run* creates the following temporary files in the user's working directory. The placeholder ¡pid¿ is replaced by the process id of *condor_run*.

**.condor_run.<pid>** A shell script containing the shell command line.

**.condor_submit.<pid>** The submit description file for the job.

**.condor_log.<pid>** The Condor job's log file; it is monitored by *condor_run*, to determine when the job exits.

**.condor_out.<pid>** The output of the Condor job before it is output to the terminal.

**.condor_error.<pid>** Any error messages for the Condor job before they are output to the terminal.

*condor_run* removes these files when the job completes. However, if *condor_run* fails, it is possible that these files will remain in the user's working directory, and the Condor job may remain in the queue.

## General Remarks

*condor_run* is intended for submitting simple shell command lines to Condor. It does not provide the full functionality of *condor_submit*. Therefore, some *condor_submit* errors and system failures may not be handled correctly.

All processes specified within the single shell command line will be executed on the single machine matched with the job. Condor will not distribute multiple processes of a command line pipe across multiple machines.

*condor_run* will use the shell specified in the `SHELL` environment variable, if one exists. Otherwise, it will use */bin/sh* to execute the shell command-line.

By default, *condor_run* expects Perl to be installed in `/usr/bin/perl`. If Perl is installed in another path, ask the Condor administrator to edit the path in the *condor_run* script, or explicitly call Perl from the command line:

```
% perl path-to-condor/bin/condor_run "shell-cmd"
```

## Exit Status

*condor_run* exits with a status value of 0 (zero) upon complete success. The exit status of *condor_run* will be non-zero upon failure. The exit status in the case of a single error due to a system call will be the error number (`errno`) of the failed call.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_stats*

Display historical information about the Condor pool

## Synopsis

***condor_stats*** [**-f** *filename*] [**-orgformat**] [**-pool** *centralmanagerhostname[:portnumber]*] [**time-range**] *query-type*

## Description

*condor_stats* displays historic information about a Condor pool. Based on the type of information requested, a query is sent to the *condor_collector* daemon, and the information received is displayed using the standard output. If the **-f** option is used, the information will be written to a file instead of to standard output. The **-pool** option can be used to get information from other pools, instead of from the local (default) pool. The *condor_stats* tool is used to query resource information (single or by platform), submitter and user information, and checkpoint server information. If a time range is not specified, the default query provides information for the previous 24 hours. Otherwise, information can be retrieved for other time ranges such as the last specified number of hours, last week, last month, or a specified date range.

The information is displayed in columns separated by tabs. The first column always represents the time, as a percentage of the range of the query. Thus the first entry will have a value close to 0.0, while the last will be close to 100.0. If the **-orgformat** option is used, the time is displayed as number of seconds since the Unix epoch. The information in the remainder of the columns depends on the query type.

Note that logging of pool history must be enabled in the *condor_collector* daemon, otherwise no information will be available.

One query type is required. If multiple queries are specified, only the last one takes effect.

## Time Range Options

**-lastday** Get information for the last day.

**-lastweek** Get information for the last week.

**-lastmonth** Get information for the last month.

**-lasthours *n*** Get information for the n last hours.

**-from *m d y*** Get information for the time since the beginning of the specified date. A start date prior to the Unix epoch causes *condor_stats* to print its usage information and quit.

**-to *m d y*** Get information for the time up to the beginning of the specified date, instead of up to now. A finish date in the future causes *condor_stats* to print its usage information and quit.

## Query Type Arguments

The query types that do not list all of a category require further specification as given by an argument.

**-resourcequery *hostname*** A single resource query provides information about a single machine. The information also includes the keyboard idle time (in seconds), the load average, and the machine state.

**-resourcelist** A query of a single list of resources to provide a list of all the machines for which the *condor_collector* daemon has historic information within the query's time range.

**-resgroupquery *arch/opsys — "Total"*** A query of a specified group to provide information about a group of machines based on their platform (operating system and architecture). The architecture is defined by the machine ClassAd `Arch`, and the operating system is defined by the machine ClassAd `OpSys`. The string "Total" ask for information about all platforms.

The columns displayed are the number of machines that are unclaimed, matched, claimed, preempting, and in the owner state.

**-resgrouplist** Queries for a list of all the group names for which the *condor_collector* has historic information within the query's time range.

**-userquery *email_address/submit_machine*** Query for a specific submitter on a specific machine. The information displayed includes the number of running jobs and the number of idle jobs. An example argument appears as

```
-userquery jondoe@sample.com/onemachine.sample.com
```

**-userlist** Queries for the list of all submitters for which the *condor_collector* daemon has historic information within the query's time range.

**-usergroupquery *email_address — "Total"*** Query for all jobs submitted by the specific user, regardless of the machine they were submitted from, or all jobs. The information displayed includes the number of running jobs and the number of idle jobs.

**-usergrouplist** Queries for the list of all users for which the *condor_collector* has historic information within the query's time range.

**-ckptquery *hostname*** Query about a checkpoint server given its host name. The information displayed includes the number of Mbytes received, Mbytes sent, average receive bandwidth (in Kbytes/sec), and average send bandwidth (in Kbytes/sec).

**-ckptlist** Query for the entire list of checkpoint servers for which the *condor_collector* has historic information in the query's time range.

## Options

**-f *filename*** Write the information to a file instead of the standard output.

**-pool *centralmanagerhostname[:portnumber]*** Contact the specified central manager instead of the local one.

**-orgformat** Display the information in an alternate format for timing, which presents timestamps since the Unix epoch. This argument only affects the display of *resoursequery*, *resgroupquery*, *userquery*, *usergroupquery*, and *ckptquery*.

## Exit Status

*condor_stats* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_status*

Display status of the Condor pool

## Synopsis

***condor_status*** [**-debug**] [*help options*] [*query options*] [*display options*] [*custom options*] [*hostname . . .*]

## Description

*condor_status* is a versatile tool that may be used to monitor and query the Condor pool. The *condor_status* tool can be used to query resource information, submitter information, checkpoint server information, and daemon master information. The specific query sent and the resulting information display is controlled by the query options supplied. Queries and display formats can also be customized.

The options that may be supplied to *condor_status* belong to five groups:

- **Help options** provide information about the *condor_status* tool.
- **Query options** control the content and presentation of status information.
- **Display options** control the display of the queried information.
- **Custom options** allow the user to customize query and display information.
- **Host options** specify specific machines to be queried

At any time, only one *help option,* one *query option* and one *custom option* may be specified. Any number of *custom* and *host options* may be specified.

## Options

**-debug** Causes debugging information to be sent to `stderr` based on the value of the configuration variable `TOOL_DEBUG`

**-help** (Help option) Display usage information

**-diagnose** (Help option) Print out query ad without performing query

**-avail** (Query option) Query *condor_startd* ads and identify resources which are available

**-claimed** (Query option) Query *condor_startd* ads and print information about claimed resources

**-ckptsrvr** (Query option) Query *condor_ckpt_server* ads and display checkpoint server attributes

**-direct *hostname*** (Query option) Go directly to the given hostname to get the ads to display

**-java** (Query option) Display only Java-capable resources.

**-master** (Query option) Query *condor_master* ads and display daemon master attributes

**-pool *centralmanagerhostname[:portnumber]*** (Query option) Query the specified central manager using an optional port number. (*condor_status* queries `COLLECTOR_HOST` by default)

**-schedd** (Query option) Query *condor_schedd* ads and display attributes

**-negotiator** (Query option) Query *condor_negotiator* ads and display attributes

**-server** (Query option) Query *condor_startd* ads and display resource attributes

**-startd** (Query option) Query *condor_startd* ads

**-state** (Query option) Query *condor_startd* ads and display resource state information

**-submitters** (Query option) Query ads sent by submitters and display important submitter attributes

**-any** (Query option) Query all ads and display their type, target type, and name

**-cod** (Display option) Display only machine ClassAds that have COD claims. Information displayed includes the claim ID, the owner of the claim, and the state of the COD claim.

**-verbose** (Display option) Display entire ClassAds. Implies that totals will not be displayed.

**-long** (Display option) Display entire ClassAds (same as **-verbose**)

**-total** (Display option) Display totals only

**-xml** (Display option) Display entire ClassAds, in xml format. The xml format is fully defined at http://www.cs.wisc.edu/condor/classad/refman/.

**-expert** (Display option) Display shortened error messages

**-sort *attr*** (Display option) Display entries in ascending order based on the value of the named attribute

**-constraint *const*** (Custom option) Add constraint expression. See section 4.1 for details on writing expressions.

**-format *fmt attr*** (Custom option) Display attribute or expression *attr* in format *fmt*. To display the attribute or expression the format must contain a single `printf(3)` style conversion specifier. Attributes must be from the resource ClassAd. Expressions are ClassAd expressions and may refer to attributes in the resource ClassAd. If the attribute is not present in a given ClassAd and cannot be parsed as an expression then the format option will be silently skipped. The conversion specifier must match the type of the attribute or expression. %s is suitable for strings such as `Name`, %d for integers such as `LastHeardFrom`, and %f for floating point numbers such as `LoadAvg`. An incorrect format will result in undefined behavior. Do not use more than one conversion specifier in a given format. More than one conversion specifier will result in undefined behavior. To output multiple attributes repeat the **-format** option once for each desired attribute. Like `printf(3)` style formats, you can include other text that will be reproduced directly. You can specify a format without any conversion specifiers, but you must still give an attribute. You can include \n to add specify a line break.

## General Remarks

- The default output from *condor_status* is formatted to be human readable, not script readable. In an effort to make the output fit within 80 characters, values in some fields might be truncated. Furthermore, the Condor Project can (and does) change the formatting of this default output as we see fit. Therefore, any script that is attempting to parse data from *condor_status* is strongly encouraged to use the **-format** option (described above).
- The information obtained from *condor_startd* and *condor_schedd* daemons may sometimes appear to be inconsistent. This is normal since *condor_startd* and *condor_schedd* daemons update the Condor manager at different rates, and since there is a delay as information propagates through the network and the system.

- Note that the `ActivityTime` in the `Idle` state is *not* the amount of time that the machine has been idle. See the section on *condor_startd* states in the *Administrator's Manual* for more information.
- When using *condor_status* on a pool with SMP machines, you can either provide the hostname, in which case you will get back information about all virtual machines that are represented on that host, or you can list specific virtual machines by name. See the examples below for details.
- If you specify host names, without domains, Condor will automatically try to resolve those host names into fully qualified host names for you. This also works when specifying specific nodes of an SMP machine. In this case, everything after the "@" sign is treated as a hostname and that is what is resolved.
- You can use the **-direct** option in conjunction with almost any other set of options. However, at this time, the only daemon that will allow direct queries for its ad(s) is the *condor_startd*. So, the only options currently not supported with **-direct** are **-schedd** and **-master**. Most other options use startd ads for their information, so they work seamlessly with **-direct**. The only other restriction on **-direct** is that you may only use 1 **-direct** option at a time. If you want to query information directly from multiple hosts, you must run *condor_status* multiple times.
- Unless you use the local hostname with **-direct**, *condor_status* will still have to contact a collector to find the address where the specified daemon is listening. So, using a **-pool** option in conjunction with **-direct** just tells *condor_status* which collector to query to find the address of the daemon you want. The information actually displayed will still be retrieved directly from the daemon you specified as the argument to **-direct**.

## Examples

Example 1 To view information from all nodes of an SMP machine, use only the hostname. For example, if you had a 4-CPU machine, named `vulture.cs.wisc.edu`, you might see

```
% condor_status vulture

Name          OpSys       Arch   State      Activity   LoadAv Mem   ActvtyTime

vm1@vulture.c SOLARIS26   INTEL  Owner      Idle       0.020  128   0+00:57:13
vm2@vulture.c SOLARIS26   INTEL  Claimed    Busy       1.006  128   0+01:16:03
vm3@vulture.c SOLARIS26   INTEL  Claimed    Busy       0.978  128   0+03:32:53
vm4@vulture.c SOLARIS26   INTEL  Claimed    Busy       1.001  128   0+02:21:07

                     Machines Owner Claimed Unclaimed Matched Preempting

     INTEL/SOLARIS26        4     0       4         0       0          0

               Total        4     0       4         0       0          0
```

Example 2 To view information from a specific nodes of an SMP machine, specify the node directly. You do this by providing the name of the virtual machine. This has the form `vm#@hostname`. For example:

```
% condor_status vm2@vulture

Name          OpSys       Arch   State      Activity   LoadAv Mem   ActvtyTime

vm2@vulture.c SOLARIS26   INTEL  Claimed    Busy       1.006  128   0+01:16:03

                     Machines Owner Claimed Unclaimed Matched Preempting

     INTEL/SOLARIS26        1     0       1         0       0          0

                Total        1     0       1         0       0          0
```

Constraint option examples

To use the constraint option to see all machines with the `OpSys` of `"LINUX"`, use

```
% condor_status -constraint OpSys==\"LINUX\"
```

Note that quotation marks must be escaped with the backslash characters for most shells.

To see all machines that are currently in the Idle state, use

```
% condor_status -constraint State==\"Idle\"
```

To see all machines that are bench marked to have a MIPS rating of more than 750, use

```
% condor_status -constraint 'Mips>750'
```

-cod option example

The **-cod** option displays the status of COD claims within a given Condor pool.

```
Name        ID   ClaimState TimeInState RemoteUser JobId Keyword
astro.cs.wi COD1 Idle        0+00:00:04 wright
chopin.cs.w COD1 Running     0+00:02:05 wright     3.0   fractgen
chopin.cs.w COD2 Suspended   0+00:10:21 wright     4.0   fractgen

               Total  Idle  Running  Suspended  Vacating  Killing
 INTEL/LINUX       3     1        1          1         0        0
       Total       3     1        1          1         0        0
```

## Exit Status

*condor_status* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_store_cred*

securely stash user's password

## Synopsis

***condor_store_cred*** [**-help**]

***condor_store_cred*** *add* [**-c | -u** *username* ][**-p** *password*] [**-n** *machinename*] [**-f** *filename*]

***condor_store_cred*** *delete* [**-c | -u** *username* ][**-n** *machinename*]

***condor_store_cred*** *query* [**-c | -u** *username* ][**-n** *machinename*]

## Description

On a Windows machine, *condor_store_cred* stores the password of a user/domain pair securely in the Windows registry. Using this stored password, Condor is able to run jobs with the user ID of the submitting user. In addition, Condor uses this password to acquire the submitting user's credentials when writing output or log files. The password is stored in the same manner as the system does when setting or changing account passwords. When *condor_store_cred* is invoked, it contacts the *condor_schedd* daemon to carry out the requested operations on behalf of the user. This is necessary since registry keys are accessible only by the Windows SYSTEM account, not by administrators or other users.

On a Unix machine, *condor_store_cred* is used to manage the pool password, placed in a file specified by the `SEC_PASSWORD_FILE` configuration variable, and for use in password authentication among Condor daemons.

The password is stashed in a persistent manner; it is maintained across system reboots.

The *add* argument stores the current user's password securely in the registry. The user is prompted to enter the password twice for confirmation, and characters are not echoed. If there is already a password stashed, the old password will be overwritten by the new password.

The *delete* deletes the current password, if it exists.

The *query* reports whether the password is stored or not.

## Options

**-c** Apply the option to the pool password.

**-f *filename*** For Unix machines only, generates a pool password file named *filename* that may be used with the PASSWORD authentication method.

**-help** Displays a brief summary of command options.

**-n *machinename*** Apply the command on the given machine.

**-p *password*** Stores given password, rather than prompting.

**-u *username*** Specify the user name.

## Exit Status

*condor_store_cred* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_submit*

Queue jobs for execution under Condor

## Synopsis

***condor_submit*** [**-verbose**] [**-name** *schedd_name*] [**-remote** *schedd_name*] [**-pool** *pool_name*] [**-disable**] [**-password** *passphrase*] [**-debug**] [**-append** *command* ...][**-spool**] [*submit description file*]

## Description

*condor_submit* is the program for submitting jobs for execution under Condor. *condor_submit* requires a submit description file which contains commands to direct the queuing of jobs. One submit description file may contain specifications for the queuing of many Condor jobs at once. A single invocation of *condor_submit* may cause one or more clusters. A cluster is a set of jobs specified in the submit description file between **queue** commands for which the executable is not changed. It is advantageous to submit multiple jobs as a single cluster because:

- Only one copy of the checkpoint file is needed to represent all jobs in a cluster until they begin execution.
- There is much less overhead involved for Condor to start the next job in a cluster than for Condor to start a new cluster. This can make a big difference when submitting lots of short jobs.

Multiple clusters may be specified within a single submit description file. Each cluster must specify a single executable.

The job ClassAd attribute `ClusterId` identifies a cluster. See section 2.5.2 for specifics on this attribute.

Note that submission of jobs from a Windows machine requires a stashed password to allow Condor to impersonate the user submitting the job. To stash a password, use the *condor_store_cred* command. See the manual page at page 714 for details.

*SUBMIT DESCRIPTION FILE COMMANDS*

Each submit description file describes one cluster of jobs to be placed in the Condor execution pool. All jobs in a cluster must share the same executable, but they may have different input and output files, and different program arguments. The submit description file is the only command-line argument to *condor_submit*.

The submit description file must contain one *executable* command and at least one *queue* command. All of the other commands have default actions.

The commands which can appear in the submit description file are numerous. They are listed here in alphabetical order by category.

*BASIC COMMANDS*

**arguments =** <**argument_list**> List of arguments to be supplied to the program on the command line. In the Java Universe, the first argument must be the name of the class containing `main`.

There are two permissible formats for specifying arguments. The new syntax supports uniform quoting of spaces within arguments; the old syntax supports spaces in arguments only in special circumstances.

In the old syntax, arguments are delimited (separated) by space characters. Double-quotes must be escaped with a backslash (i.e. put a backslash in front of each double-quote).

Further interpretation of the argument string differs depending on the universe and operating system. On Windows, your argument string is simply passed verbatim (other than the backslash in front of double-quotes) to the windows application. Most Windows applications will allow you to put spaces within an argument value by surrounding the argument with double-quotes. In the grid universe, arguments are passed verbatim (other than the backslash in front of double-quotes) into the RSL string used by Globus. See section 5.3.2 for further details. In all other cases, there is no further interpretation of the arguments.

Example:

```
arguments = one \"two\" 'three'
```

Produces in unix vanilla universe:

```
argument 1: one
argument 2: "two"
argument 3: 'three'
```

Here are the rules for using the new syntax:

1. Put double quotes around the entire argument string. This distinguishes the new syntax from the old, because these double-quotes are not escaped with backslashes, as required in the old syntax. Any literal double-quotes within the string must be escaped by repeating them.
2. Use whitespace (e.g. spaces or tabs) to separate arguments.
3. To put any whitespace in an argument, you must surround the space and as much of the surrounding argument as you like with single-quotes.
4. To insert a literal single-quote, you must repeat it anywhere inside of a single-quoted section.

Example:

```
arguments = "one ""two"" 'spacey ''quoted'' argument'"
```

Produces:

```
argument 1: one
argument 2: "two"
argument 3: spacey 'quoted' argument
```

Notice that in the new syntax, backslash has no special meaning. This is for the convenience of Windows users.

**environment** = <**parameter_list**> List of environment variables.

There are two different formats for specifying the environment variables: the old format and the new format. The old format is retained for backward-compatibility. It suffers from a platform-dependent syntax and the inability to insert some special characters into the environment.

The new syntax for specifying environment values:

1. Put double quote marks around the entire argument string. This distinguishes the new syntax from the old. The old syntax does not have double quote marks around it. Any literal double quote marks within the string must be escaped by repeating the double quote mark.
2. Each environment entry has the form

   ```
   <name>=<value>
   ```

3. Use whitespace (space or tab characters) to separate environment entries.
4. To put any whitespace in an environment entry, surround the space and as much of the surrounding entry as desired with single quote marks.
5. To insert a literal single quote mark, repeat the single quote mark anywhere inside of a section surrounded by single quote marks.

Example:

```
environment = "one=1 two=""2"" three='spacey ''quoted'' value'"
```

Produces the following environment entries:

```
one=1
two="2"
three=spacey 'quoted' value
```

Under the old syntax, there are no double quote marks surrounding the environment specification. Each environment entry remains of the form

```
<name>=<value>
```

Under Unix, list multiple environment entries by separating them with a semicolon (;). Under Windows, separate multiple entries with a vertical bar (| ). There is no way to insert a literal semicolon under Unix or a literal vertical bar under Windows. Note that spaces are accepted, but rarely desired, characters within parameter names and values, because they are treated as literal characters, not separators or ignored whitespace. Place spaces within the parameter list only if required.

A Unix example:

```
environment = one=1;two=2;three="quotes have no 'special' meaning"
```

This produces the following:

```
one=1
two=2
three="quotes have no 'special' meaning"
```

**error** = <**pathname**> A path and file name used by Condor to capture any error messages the program would normally write to the screen (that is, this file becomes `stderr`). If not specified, the default value of `/dev/null` is used for submission to a Unix machine. If not specified, error messages are ignored for submission to a Windows machine. More than one job should not use the same error file, since this will cause one job to overwrite the errors of another. The error file and the output file should not be the same file as the outputs will overwrite each other or be lost. For grid universe jobs, **error** may be a URL that the Globus tool *globus_url_copy* understands.

**executable** = <**pathname**> An optional path and a required file name of the executable file for this job cluster. Only one **executable** command within a submit description file is guaranteed to work properly. More than one often works.

If no path or a relative path is used, then the executable file is presumed to be relative to the current working directory of the user as the *condor_submit* command is issued.

If submitting into the standard universe (the default), then the named executable must have been re-linked with the Condor libraries (such as via the *condor_compile* command). If submitting into the vanilla universe, then the named executable need not be re-linked and can be any process which can run in the background (shell scripts work fine as well). If submitting into the Java universe, then the argument must be a compiled `.class` file.

**getenv** = <**True** | **False**> If **getenv** is set to `True`, then *condor_submit* will copy all of the user's current shell environment variables at the time of job submission into the job ClassAd. The job will therefore execute with the same set of environment variables that the user had at submit time. Defaults to `False`.

**input** = <**pathname**> Condor assumes that its jobs are long-running, and that the user will not wait at the terminal for their completion. Because of this, the standard files which normally access the terminal, (`stdin`, `stdout`, and `stderr`), must refer to files. Thus, the file name specified with **input** should contain any keyboard input the program requires (that is, this file becomes `stdin`). If not specified, the default value of `/dev/null` is used for submission to

a Unix machine. If not specified, input is ignored for submission to a Windows machine. For grid universe jobs, **input** may be a URL that the Globus tool *globus_url_copy* understands.

Note that this command does *not* refer to the command-line arguments of the program. The command-line arguments are specified by the **arguments** command.

**log =** <**pathname**> Use **log** to specify a file name where Condor will write a log file of what is happening with this job cluster. For example, Condor will place a log entry into this file when and where the job begins running, when the job produces a checkpoint, or moves (migrates) to another machine, and when the job completes. Most users find specifying a **log** file to be handy; its use is recommended. If no **log** entry is specified, Condor does not create a log for this cluster.

**log_xml =** <**True** | **False**> If **log_xml** is `True`, then the log file will be written in ClassAd XML. If not specified, XML is not used. Note that the file is an XML fragment; it is missing the file header and footer. Do not mix XML and non-XML within a single file. If multiple jobs write to a single log file, ensure that all of the jobs specify this option in the same way.

**notification =** <**Always** | **Complete** | **Error** | **Never**> Owners of Condor jobs are notified by e-mail when certain events occur. If defined by *Always*, the owner will be notified whenever the job produces a checkpoint, as well as when the job completes. If defined by *Complete* (the default), the owner will be notified when the job terminates. If defined by *Error*, the owner will only be notified if the job terminates abnormally. If defined by *Never*, the owner will not receive e-mail, regardless to what happens to the job. The statistics included in the e-mail are documented in section 2.6.7 on page 57.

**notify_user =** <**email-address**> Used to specify the e-mail address to use when Condor sends e-mail about a job. If not specified, Condor defaults to using the e-mail address defined by

```
job-owner@UID_DOMAIN
```

where the configuration variable `UID_DOMAIN` is specified by the Condor site administrator. If `UID_DOMAIN` has not been specified, Condor sends the e-mail to:

```
job-owner@submit-machine-name
```

**output =** <**pathname**> The **output** file captures any information the program would ordinarily write to the screen (that is, this file becomes `stdout`). If not specified, the default value of `/dev/null` is used for submission to a Unix machine. If not specified, output is ignored for submission to a Windows machine. Multiple jobs should not use the same output file, since this will cause one job to overwrite the output of another. The output file and the error file should not be the same file as the outputs will overwrite each other or be lost. For grid universe jobs, **output** may be a URL that the Globus tool *globus_url_copy* understands.

Note that if a program explicitly opens and writes to a file, that file should *not* be specified as the **output** file.

**priority =** <**integer**> A Condor job priority can be any integer, with 0 being the default. Jobs with higher numerical priority will run before jobs with lower numerical priority. Note that

this priority is on a per user basis. One user with many jobs may use this command to order his/her own jobs, and this will have no effect on whether or not these jobs will run ahead of another user's jobs.

**queue [number-of-procs]** Places one or more copies of the job into the Condor queue. The optional argument *number-of-procs* specifies how many times to submit the job to the queue, and it defaults to 1. If desired, any commands may be placed between subsequent **queue** commands, such as new **input**, **output**, **error**, **initialdir**, or **arguments** commands. This is handy when submitting multiple runs into one cluster with one submit description file.

**universe =** <**vanilla** | **standard** | **pvm** | **scheduler** | **local** | **grid** | **mpi** | **java**> Specifies which Condor Universe to use when running this job. The Condor Universe specifies a Condor execution environment. The standard Universe is the default (except where the configuration variable `DEFAULT_UNIVERSE` defines it otherwise), and tells Condor that this job has been re-linked via *condor_compile* with the Condor libraries and therefore supports checkpointing and remote system calls. The **vanilla** Universe is an execution environment for jobs which have not been linked with the Condor libraries. *Note:* Use the **vanilla** Universe to submit shell scripts to Condor. The **pvm** Universe is for a parallel job written with PVM 3.4. The **scheduler** is for a job that should act as a metascheduler. The **grid** universe forwards the job to an external job management system. Further specification of the **grid** universe is done with the **grid_resource** command. The **mpi** universe is for running mpi jobs made with the MPICH package. The **java** Universe is for programs written to the Java Virtual Machine.

*COMMANDS FOR MATCHMAKING*

**rank =** <**ClassAd Float Expression**> A ClassAd Floating-Point expression that states how to rank machines which have already met the requirements expression. Essentially, rank expresses preference. A higher numeric value equals better rank. Condor will give the job the machine with the highest rank. For example,

```
requirements = Memory > 60
rank = Memory
```

asks Condor to find all available machines with more than 60 megabytes of memory and give to the job the machine with the most amount of memory. See section 2.5.2 within the Condor Users Manual for complete information on the syntax and available attributes that can be used in the ClassAd expression.

**requirements =** <**ClassAd Boolean Expression**> The requirements command is a boolean ClassAd expression which uses C-like operators. In order for any job in this cluster to run on a given machine, this requirements expression must evaluate to true on the given machine. For example, to require that whatever machine executes a Condor job has a least 64 Meg of RAM and has a MIPS performance rating greater than 45, use:

```
requirements = Memory >= 64 && Mips > 45
```

Only one requirements command may be present in a submit description file. By default, *condor_submit* appends the following clauses to the requirements expression:

1. Arch and OpSys are set equal to the Arch and OpSys of the submit machine. In other words: unless you request otherwise, Condor will give your job machines with the same architecture and operating system version as the machine running *condor_submit*.
2. Disk >= DiskUsage. The `DiskUsage` attribute is initialized to the size of the executable plus the size of any files specified in a **transfer_input_files** command. It exists to ensure there is enough disk space on the target machine for Condor to copy over both the executable and needed input files. The `DiskUsage` attribute represents the maximum amount of total disk space required by the job in kilobytes. Condor automatically updates the `DiskUsage` attribute approximately every 20 minutes while the job runs with the amount of space being used by the job on the execute machine.
3. (Memory * 1024) >= ImageSize. To ensure the target machine has enough memory to run your job.
4. If Universe is set to Vanilla, FileSystemDomain is set equal to the submit machine's FileSystemDomain.

View the requirements of a job which has already been submitted (along with everything else about the job ClassAd) with the command *condor_q -l*; see the command reference for *condor_q* on page 673. Also, see the Condor Users Manual for complete information on the syntax and available attributes that can be used in the ClassAd expression.

*FILE TRANSFER COMMANDS*

**should_transfer_files =** <**YES** | **NO** | **IF_NEEDED** > The **should_transfer_files** setting is used to define if Condor should transfer files to and from the remote machine where the job runs. The file transfer mechanism is used to run jobs which are not in the standard universe (and can therefore use remote system calls for file access) on machines which do not have a shared file system with the submit machine. **should_transfer_files** equal to *YES* will cause Condor to always transfer files for the job. *NO* disables Condor's file transfer mechanism. *IF_NEEDED* will not transfer files for the job if it is matched with a resource in the same `FileSystemDomain` as the submit machine (and therefore, on a machine with the same shared file system). If the job is matched with a remote resource in a different `FileSystemDomain`, Condor will transfer the necessary files.

If defining **should_transfer_files** you *must* also define **when_to_transfer_output** (described below). For more information about this and other settings related to transferring files, see section 2.5.4 on page 36.

Note that **should_transfer_files** is not supported for jobs submitted to the grid universe.

**stream_error =** <**True** | **False**> If `True`, then `stderr` is streamed back to the machine from which the job was submitted. If `False`, `stderr` is stored locally and transferred back when the job completes. This command is ignored if the job ClassAd attribute `TransferErr` is `False`. The default value is `True` in the grid universe and `False` otherwise.

**stream_input =** <**True** | **False**> If `True`, then `stdin` is streamed from the machine on which the job was submitted. The default value is `False`. The command is only relevant for jobs submitted to the vanilla or java universes, and it is ignored by the grid universe.

**stream_output** = <**True** | **False**> If `True`, then `stdout` is streamed back to the machine from which the job was submitted. If `False`, `stdout` is stored locally and transferred back when the job completes. This command is ignored if the job ClassAd attribute `TransferOut` is `False`. The default value is `True` in the grid universe and `False` otherwise.

**transfer_executable** = <**True** | **False**> This command is applicable to jobs submitted to the grid, vanilla, and MPI universes. If **transfer_executable** is set to `False`, then Condor looks for the executable on the remote machine, and does not transfer the executable over. This is useful for an already pre-staged executable; Condor behaves more like rsh. The default value is `True`.

**transfer_input_files** = < **file1,file2,file...** > A comma-delimited list of all the files to be transferred into the working directory for the job before the job is started. By default, the file specified in the **executable** command and any file specified in the **input** command (for example, `stdin`) are transferred.

Only the transfer of files is available; the transfer of subdirectories is not supported.

For more information about this and other settings related to transferring files, see section 2.5.4 on page 36.

**transfer_output_files** = < **file1,file2,file...** > This command forms an explicit list of output files to be transferred back from the temporary working directory on the execute machine to the submit machine. Most of the time, there is no need to use this command. Other than for grid universe jobs, if **transfer_output_files** is not specified, Condor will automatically transfer back all files in the job's temporary working directory which have been modified or created by the job. This is usually the desired behavior. Explicitly listing output files is typically only done when the job creates many files, and the user wants to keep a subset of those files. If there are multiple files, they must be delimited with commas. WARNING: Do not specify **transfer_output_files** in the submit description file unless there is a really good reason – it is best to let Condor figure things out by itself based upon what the job produces.

For grid universe jobs, to have files other than standard output and standard error transferred from the execute machine back to the submit machine, do use **transfer_output_files**, listing all files to be transferred. These files are found on the execute machine in the working directory of the job.

For more information about this and other settings related to transferring files, see section 2.5.4 on page 36.

**transfer_output_remaps** = < **" name = newname ; name2 = newname2 ... "**> This specifies the name (and optionally path) to use when downloading output files from the completed job. Normally, output files are transferred back to the initial working directory with the same name they had in the execution directory. This gives you the option to save them with a different path or name. If you specify a relative path, the final path will be relative to the job's initial working directory.

*name* describes an output file name produced by your job, and *newname* describes the file name it should be downloaded to. Multiple remaps can be specified by separating each with a semicolon. If you wish to remap file names that contain equals signs or semicolons, these special characters may be escaped with a backslash.

**when_to_transfer_output** = < **ON_EXIT** | **ON_EXIT_OR_EVICT** > Setting **when_to_transfer_output** equal to *ON_EXIT* will cause Condor to transfer the job's output files back to the submitting machine only when the job completes (exits on its own).

The *ON_EXIT_OR_EVICT* option is intended for fault tolerant jobs which periodically save their own state and can restart where they left off. In this case, files are spooled to the submit machine any time the job leaves a remote site, either because it exited on its own, or was evicted by the Condor system for any reason prior to job completion. The files spooled back are placed in a directory defined by the value of the `SPOOL` configuration variable. Any output files transferred back to the submit machine are automatically sent back out again as input files if the job restarts.

For more information about this and other settings related to transferring files, see section 2.5.4 on page 36.

*POLICY COMMANDS*

**hold** = <**True** | **False**> If **hold** is set to `True`, then the job will be submitted in the hold state. Jobs in the hold state will not run until released by *condor_release*. Defaults to false.

**leave_in_queue** = <**ClassAd Boolean Expression**> When the ClassAd Expression evaluates to `True`, the job is not removed from the queue upon completion. The job remains in the queue until the user runs *condor_rm* to remove the job from the queue. This allows the user of a remotely spooled job to retrieve output files in cases where Condor would have removed them as part of the cleanup associated with completion. Defaults to `False`.

**on_exit_hold** = <**ClassAd Boolean Expression**> This expression is checked when the job exits and if true, places the job on hold. If false then nothing happens and the `on_exit_remove` expression is checked to determine if that needs to be applied.

For example: Suppose a job is known to run for a minimum of an hour. If the job exits after less than an hour, the job should be placed on hold and an e-mail notification sent, instead of being allowed to leave the queue.

```
on_exit_hold = (CurrentTime - JobStartDate) < (60 * $(MINUTE))
```

This expression places the job on hold if it exits for any reason before running for an hour. An e-mail will be sent to the user explaining that the job was placed on hold because this expression became `True`.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions.

If left unspecified, this will default to `False`.

This expression is available for the vanilla, java, and scheduler universes. It is additionally available, when submitted from a Unix machine, for the standard universe.

**on_exit_remove** = <**ClassAd Boolean Expression**> This expression is checked when the job exits and if true, then it allows the job to leave the queue normally. If false, then the job is placed

back into the Idle state. If the user job runs under the vanilla universe, then the job restarts from the beginning. If the user job runs under the standard universe, then it continues from where it left off, using the last checkpoint.

For example, suppose you have a job that occasionally segfaults, but you know if you run the job again with the same data, chances are that the will finish successfully. This is how you would represent that with `on_exit_remove` (assuming the signal identifier for segmentation fault is 11 on the platform where your job will be running):

```
on_exit_remove = (ExitBySignal == False) || (ExitSignal != 11)
```

This expression will only let the job leave the queue if the job was not killed by a signal (it exited normally on its own) or if it was killed by a signal other than 11 (representing segmentation fault). So, if it was killed by signal 11, it will stay in the job queue. In any other case of the job exiting, the job will leave the queue as it normally would have done.

As another example, if your job should only leave the queue if it exited on its own with status 0, you would use this `on_exit_remove` expression:

```
on_exit_remove = (ExitBySignal == False) && (ExitCode == 0)
```

If the job was killed by a signal or exited with a non-zero exit status, Condor would leave the job in the queue to run again.

If left unspecified, the `on_exit_remove` expression will default to `True`.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions.

This expression is available for the vanilla, java, and scheduler universes. It is additionally available, when submitted from a Unix machine, for the standard universe. Note that the *condor_schedd* daemon, by default, only checks these periodic expressions once every 300 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL` configuration macro.

**periodic_hold** = <**ClassAd Boolean Expression**> This expression is checked periodically at an interval of the number of seconds set by the configuration variable `PERIODIC_EXPR_INTERVAL`. If it becomes true, the job will be placed on hold. If unspecified, the default value is `False`.

See the Examples section for an example of a `periodic_*` expression.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions.

This expression is available for the vanilla, java and grid universes. It is additionally available, when submitted from a Unix machine, for the standard universe. Note that the schedd, by default, only checks periodic expressions once every 300 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL` configuration macro.

**periodic_release** = <**ClassAd Boolean Expression**> This expression is checked periodically at an interval of the number of seconds set by the configuration variable `PERIODIC_EXPR_INTERVAL` while the job is in the Hold state. If the expression becomes `True`, the job will be released.

This expression is available for the vanilla, java, and grid universes. It is additionally available, when submitted from a Unix machine, for the standard universe. Note that the *condor_schedd* daemon, by default, only checks periodic expressions once every 300 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL` configuration macro.

**periodic_remove** = <**ClassAd Boolean Expression**> This expression is checked periodically at an interval of the number of seconds set by the configuration variable `PERIODIC_EXPR_INTERVAL`. If it becomes `True`, the job is removed from the queue. If unspecified, the default value is `False`.

See the Examples section for an example of a `periodic_*` expression.

`periodic_*` expressions take precedence over `on_exit_*` expressions, and `*_hold` expressions take precedence over a `*_remove` expressions. So, the `periodic_remove` expression takes precedent over the `on_exit_remove` expression, if the two describe conflicting actions.

This expression is available for the vanilla, java and grid universes. It is additionally available, when submitted from a Unix machine, for the standard universe. Note that the schedd, by default, only checks periodic expressions once every 300 seconds. The period of these evaluations can be adjusted by setting the `PERIODIC_EXPR_INTERVAL` configuration macro.

*COMMANDS SPECIFIC TO THE STANDARD UNIVERSE*

**allow_startup_script** = <**True** | **False**> If True, a standard universe job will execute a script instead of submitting the job, and the consistency check to see if the executable has been linked using *condor_compile* is omitted. The **executable** command within the submit description file specifies the name of the script. The script is used to do preprocessing before the job is submitted. The shell script ends with an *exec* of the job executable, such that the process id of the executable is the same as that of the shell script. Here is an example script that gets a copy of a machine-specific executable before the *exec*.

```
#! /bin/sh

# get the host name of the machine
$host=`uname -n`

# grab a standard universe executable designed specifically
# for this host
scp elsewhere@cs.wisc.edu:${host} executable

# The PID MUST stay the same, so exec the new standard universe process.
exec executable ${1+"$@"}
```

If this command is not present (defined), then the value defaults to false.

**append_files = file1, file2, ...** If your job attempts to access a file mentioned in this list, Condor will force all writes to that file to be appended to the end. Furthermore, condor_submit will not truncate it. This list uses the same syntax as compress_files, shown above.

This option may yield some surprising results. If several jobs attempt to write to the same file, their output may be intermixed. If a job is evicted from one or more machines during the course of its lifetime, such an output file might contain several copies of the results. This option should be only be used when you wish a certain file to be treated as a running log instead of a precise result.

This option only applies to standard-universe jobs.

**buffer_files** = < **" name** = **(size,block-size) ; name2** = **(size,block-size) ... "** >

**buffer_size** = <**bytes-in-buffer**>

**buffer_block_size** = <**bytes-in-block**> Condor keeps a buffer of recently-used data for each file a job accesses. This buffer is used both to cache commonly-used data and to consolidate small reads and writes into larger operations that get better throughput. The default settings should produce reasonable results for most programs.

These options only apply to standard-universe jobs.

If needed, you may set the buffer controls individually for each file using the buffer_files option. For example, to set the buffer size to 1 Mbyte and the block size to 256 KBytes for the file `input.data`, use this command:

```
buffer_files = "input.data=(1000000,256000)"
```

Alternatively, you may use these two options to set the default sizes for all files used by your job:

```
buffer_size = 1000000
buffer_block_size = 256000
```

If you do not set these, Condor will use the values given by these two configuration file macros:

```
DEFAULT_IO_BUFFER_SIZE = 1000000
DEFAULT_IO_BUFFER_BLOCK_SIZE = 256000
```

Finally, if no other settings are present, Condor will use a buffer of 512 Kbytes and a block size of 32 Kbytes.

**compress_files = file1, file2, ...** If your job attempts to access any of the files mentioned in this list, Condor will automatically compress them (if writing) or decompress them (if reading). The compress format is the same as used by GNU gzip.

The files given in this list may be simple file names or complete paths and may include $*$ as a wildcard. For example, this list causes the file /tmp/data.gz, any file named event.gz, and any file ending in .gzip to be automatically compressed or decompressed as needed:

```
compress_files = /tmp/data.gz, event.gz, *.gzip
```

Due to the nature of the compression format, compressed files must only be accessed sequentially. Random access reading is allowed but is very slow, while random access writing is simply not possible. This restriction may be avoided by using both compress_files and fetch_files at the same time. When this is done, a file is kept in the decompressed state at the execution machine, but is compressed for transfer to its original location.

This option only applies to standard universe jobs.

**fetch_files = file1, file2, ...** If your job attempts to access a file mentioned in this list, Condor will automatically copy the whole file to the executing machine, where it can be accessed quickly. When your job closes the file, it will be copied back to its original location. This list uses the same syntax as compress_files, shown above.

This option only applies to standard universe jobs.

**file_remaps** = < **" name** = **newname ; name2** = **newname2 ... "**> Directs Condor to use a new file name in place of an old one. *name* describes a file name that your job may attempt to open, and *newname* describes the file name it should be replaced with. *newname* may include an optional leading access specifier, `local:` or `remote:`. If left unspecified, the default access specifier is `remote:`. Multiple remaps can be specified by separating each with a semicolon.

This option only applies to standard universe jobs.

If you wish to remap file names that contain equals signs or semicolons, these special characters may be escaped with a backslash.

**Example One:** Suppose that your job reads a file named `dataset.1`. To instruct Condor to force your job to read `other.dataset` instead, add this to the submit file:

```
file_remaps = "dataset.1=other.dataset"
```

**Example Two:** Suppose that your run many jobs which all read in the same large file, called `very.big`. If this file can be found in the same place on a local disk in every machine in the pool, (say `/bigdisk/bigfile`,) you can instruct Condor of this fact by remapping `very.big` to `/bigdisk/bigfile` and specifying that the file is to be read locally, which will be much faster than reading over the network.

```
file_remaps = "very.big = local:/bigdisk/bigfile"
```

**Example Three:** Several remaps can be applied at once by separating each with a semicolon.

```
file_remaps = "very.big = local:/bigdisk/bigfile ; dataset.1 = other.dataset"
```

**local_files = file1, file2, ...** If your job attempts to access a file mentioned in this list, Condor will cause it to be read or written at the execution machine. This is most useful for temporary files not used for input or output. This list uses the same syntax as compress_files, shown above.

```
local_files = /tmp/*
```

This option only applies to standard universe jobs.

**want_remote_io =** <**True** | **False**> This option controls how a file is opened and manipulated in a standard universe job. If this option is true, which is the default, then the *condor_shadow* makes all decisions about how each and every file should be opened by the executing job. This entails a network round trip (or more) from the job to the *condor_shadow* and back again for every single `open()` in addition to other needed information about the file. If set to false, then when the job queries the *condor_shadow* for the first time about how to open a file, the *condor_shadow* will inform the job to automatically perform all of its file manipulation on the local file system on the execute machine and any file remapping will be ignored. This means that there **must** be a shared file system (such as NFS or AFS) between the execute machine and the submit machine and that **ALL** paths that the job could open on the execute machine must be valid. The ability of the standard universe job to checkpoint, possibly to a checkpoint server, is not affected by this attribute. However, when the job resumes it will be expecting the same file system conditions that were present when the job checkpointed.

*COMMANDS FOR THE GRID*

**globus_rematch =** <**ClassAd Boolean Expression**> This expression is evaluated by the *condor_gridmanager* whenever:

1. the **globus_resubmit** expression evaluates to `True`
2. the *condor_gridmanager* decides it needs to retry a submission (as when a previous submission failed to commit)

If **globus_rematch** evaluates to `True`, then *before* the job is submitted again to globus, the *condor_gridmanager* will request that the *condor_schedd* daemon renegotiate with the matchmaker (the *condor_negotiator*). The result is this job will be matched again.

**globus_resubmit =** <**ClassAd Boolean Expression**> The expression is evaluated by the *condor_gridmanager* each time the *condor_gridmanager* gets a job ad to manage. Therefore, the expression is evaluated:

1. when a grid universe job is first submitted to Condor-G
2. when a grid universe job is released from the hold state
3. when Condor-G is restarted (specifically, whenever the *condor_gridmanager* is restarted)

If the expression evaluates to `True`, then any previous submission to the grid universe will be forgotten and this job will be submitted again as a fresh submission to the grid universe. This may be useful if there is a desire to give up on a previous submission and try again. Note that this may result in the same job running more than once. Do not treat this operation lightly.

**globus_rsl =** <**RSL-string**> Used to provide any additional Globus RSL string attributes which are not covered by other submit description file commands or job attributes. Used for **grid universe** jobs, where the grid resource has a **grid-type-string** of **gt2** or **gt3**.

**globus_xml =** <**XML-string**> Used to provide any additional attributes in the GRAM XML job description that Condor writes which are not covered by regular submit description file parameters. Used for grid type **gt4** jobs.

**grid_resource =** <**grid-type-string**> <**grid-specific-parameter-list**> For each **grid-type-string** value, there are further type-specific values that must specified. This submit description file command allows each to be given in a space-separated list. Allowable **grid-type-string** values are **gt2**, **gt3**, **gt4**, **condor**, **nordugrid**, and **unicore**. See section 5.3 for details on the variety of grid types.

For a **grid-type-string** of **condor**, the first parameter is the name of the remote *condor_schedd* daemon. The second parameter is the name of the pool to which the remote *condor_schedd* daemon belongs. See section 5.3.1 for details.

For a **grid-type-string** of **gt2**, the single parameter is the name of the pre-WS GRAM resource to be used. See section 5.3.2 for details.

For a **grid-type-string** of **gt3**, the single parameter is the name of the OSGA GRAM service to be used. See section 5.3.2 for details.

For a **grid-type-string** of **gt4**, the first parameter is the name of the WS GRAM service to be used. The second parameter is the name of WS resource to be used (usually the name of the back-end scheduler). See section 5.3.2 for details.

For a **grid-type-string** of **lsf**, no additional parameters are used. See section 5.3.6 for details.

For a **grid-type-string** of **nordugrid**, the single parameter is the name of the NorduGrid resource to be used. See section 5.3.3 for details.

For a **grid-type-string** of **pbs**, no additional parameters are used. See section 5.3.5 for details.

For a **grid-type-string** of **unicore**, the first parameter is the name of the Unicore Usite to be used. The second parameter is the name of the Unicore Vsite to be used. See section 5.3.4 for details.

**keystore_alias =** <**name**> A string to locate the certificate in a Java keystore file, as used for a **unicore** job.

**keystore_file =** <**pathname**> The complete path and file name of the Java keystore file containing the certificate to be used for a **unicore** job.

**keystore_passphrase_file =** <**pathname**> The complete path and file name to the file containing the passphrase protecting a Java keystore file containing the certificate. Relevant for a **unicore** job.

**MyProxyCredentialName =** <**symbolic name**> The symbolic name that identifies a credential to the *MyProxy* server. This symbolic name is set as the credential is initially stored on the server (using *myproxy-init*).

**MyProxyHost =** <**host**>**:**<**port**> The Internet address of the host that is the *MyProxy* server. The **host** may be specified by either a host name (as in `head.example.com`) or an IP address (of the form 123.456.7.8). The **port** number is an integer.

**MyProxyNewProxyLifetime =** <**number-of-minutes**> The new lifetime (in minutes) of the proxy after it is refreshed.

**MyProxyPassword =** <**password**> The password needed to refresh a credential on the *MyProxy* server. This password is set when the user initially stores credentials on the server (using *myproxy-init*). As an alternative to using **MyProxyPassword** in the submit description file, the password may be specified as a command line argument to *condor_submit* with the *-password* argument.

**MyProxyRefreshThreshold =** <**number-of-seconds**> The time (in seconds) before the expiration of a proxy that the proxy should be refreshed. For example, if **MyProxyRefreshThreshold** is set to the value 600, the proxy will be refreshed 10 minutes before it expires.

**MyProxyServerDN =** <**credential subject**> A string that specifies the expected Distinguished Name (credential subject, abbreviated DN) of the *MyProxy* server. It must be specified when the *MyProxy* server DN does not follow the conventional naming scheme of a host credential. This occurs, for example, when the *MyProxy* server DN begins with a user credential.

**nordugrid_rsl =** <**RSL-string**> Used to provide any additional RSL string attributes which are not covered by regular submit description file parameters. Used when the **universe** is **grid**, and the type of grid system is **nordugrid**.

**transfer_error =** <**True** | **False**> For jobs submitted to the grid universe only. If `True`, then the error output (from `stderr`) from the job is transferred from the remote machine back to the submit machine. The name of the file after transfer is given by the **error** command. If `False`, no transfer takes place (from the remote machine to submit machine), and the name of the file is given by the **error** command. The default value is `True`.

**transfer_input =** <**True** | **False**> For jobs submitted to the grid universe only. If `True`, then the job input (`stdin`) is transferred from the machine where the job was submitted to the remote machine. The name of the file that is transferred is given by the **input** command. If `False`, then the job's input is taken from a pre-staged file on the remote machine, and the name of the file is given by the **input** command. The default value is `True`.

For transferring files other than `stdin`, see **transfer_input_files**.

**transfer_output =** <**True** | **False**> For jobs submitted to the grid universe only. If `True`, then the output (from `stdout`) from the job is transferred from the remote machine back to the submit machine. The name of the file after transfer is given by the **output** command. If `False`, no transfer takes place (from the remote machine to submit machine), and the name of the file is given by the **output** command. The default value is `True`.

For transferring files other than `stdout`, see **transfer_output_files**.

**x509userproxy =** <**full-pathname**> Used to override the default path name for X.509 user certificates. The default location for X.509 proxies is the `/tmp` directory, which is generally a local file system. Setting this value would allow Condor to access the proxy in a shared file system (for example, AFS). Condor will use the proxy specified in the submit description file first. If nothing is specified in the submit description file, it will use the environment variable X509_USER_CERT. If that variable is not present, it will search in the default location.

**x509userproxy** is relevant when the **universe** is **grid**, and the type of grid system is one of **gt2**, **gt3**, **gt4**, or **nordugrid**.

*COMMANDS FOR PARALLEL, JAVA, SCHEDULER, and PVM UNIVERSES*

**hold_kill_sig** = <**signal-number**> For the scheduler universe only, **signal-number** is the signal delivered to the job when the job is put on hold with *condor_hold*. **signal-number** may be either the platform-specific name or value of the signal. If this command is not present, the value of **kill_sig** is used.

**jar_files** = <**file_list**> Specifies a list of additional JAR files to include when using the Java universe. JAR files will be transferred along with the executable and automatically added to the classpath.

**java_vm_args** = <**argument_list**> Specifies a list of additional arguments to the Java VM itself, When Condor runs the Java program, these are the arguments that go before the class name. This can be used to set VM-specific arguments like stack size, garbage-collector arguments and initial property values.

**machine_count** = <**min..max**> | <**max**> For the PVM universe, both *min* and *max* or just *max* may be defined. If **machine_count** is specified, Condor will not start the job until it can simultaneously supply the job with *min* machines. Condor will continue to try to provide up to *max* machines, but will not delay starting of the job to do so. If the job is started with fewer than *max* machines, the job will be notified via a usual PvmHostAdd notification as additional hosts come on line.

For the parallel (and therefore, the mpi) universe, a single value (*max*) is required. It is neither a maximum or minimum, but the number of machines to be dedicated toward running the job.

**remove_kill_sig** = <**signal-number**> For the scheduler universe only, **signal-number** is the signal delivered to the job when the job is removed with *condor_rm*. **signal-number** may be either the platform-specific name or value of the signal. This example shows it both ways for a Linux signal:

```
remove_kill_sig = SIGUSR1
remove_kill_sig = 10
```

If this command is not present, the value of **kill_sig** is used.

*ADVANCED COMMANDS*

**copy_to_spool** = <**True** | **False**> If **copy_to_spool** is set to `True`, then *condor_submit* will copy the executable to the local spool directory before running it on a remote host. Oftentimes this can be quite time consuming and unnecessary. By setting it to `False`, *condor_submit* will skip this step. The default is `False` for **grid** universe jobs or when the **-spool** or **-remote** options are used; the default is `True` for all other jobs.

**coresize** = <**size**> Should the user's program abort and produce a core file, **coresize** specifies the maximum size in bytes of the core file which the user wishes to keep. If **coresize** is not specified in the command file, the system's user resource limit "coredumpsize" is used. This limit is not used in HP-UX and DUX operating systems.

**deferral_time =** <**Unix Epoch Timestamp**> This option allows a job to begin execution at a specific time instead of executing as soon as it arrives at the execution machine. The deferral time is an expression that must evaluate to a Unix Epoch timestamp (the number of seconds elapsed since 00:00:00 on January 1, 1970, Coordinated Universal Time). A job using this option will be delayed for execution by the *condor_starter* until the deferral time arrives. If the job misses its execution time, that is, if the deferral time is in the past, the job will be aborted and removed from the queue. The time that the job will start at is based on the execution machine's system clock.

The following example will set a job to run at exactly on January 1st, 2006 at 12:00 pm:

```
DeferralTime = 1136138400
```

This example will cause a job to always wait 60 seconds after it arrives at the execution machine before executing:

```
DeferralTime = (CurrentTime + 60)
```

To allow for jobs to run even if the deferral time is missed, please refer to **deferral_window**.

Please note that scheduler universe jobs are unable to use this feature because they are not executed by the *condor_starter*. A scheduler job will fail to be submitted if *condor_deferral_time* is defined in its submission file.

**deferral_window =** <**number-of-seconds**> The deferral window is used in conjunction with the **deferral_time** command to allow jobs that miss their execution time to run anyway. The window is the number of seconds in the past Condor is willingly to run a job if the deferral time is missed.

In the example below, the **deferral_time** always evaluate to 60 seconds in the past from the current time, but the job is still allowed to execute because the **deferral_window** is 120 seconds:

```
DeferralWindow = 120
DeferralTime   = (CurrentTime - 60)
```

**image_size =** <**size**> This command tells Condor the maximum virtual image size to which you believe your program will grow during its execution. Condor will then execute your job only on machines which have enough resources, (such as virtual memory), to support executing your job. If you do not specify the image size of your job in the description file, Condor will automatically make a (reasonably accurate) estimate about its size and adjust this estimate as your program runs. If the image size of your job is underestimated, it may crash due to inability to acquire more address space, e.g. malloc() fails. If the image size is overestimated, Condor may have difficulty finding machines which have the required resources. *size* must be in kbytes, e.g. for an image size of 8 megabytes, use a *size* of 8000.

**initialdir =** <**directory-path**> Used to give jobs a directory with respect to file input and output. Also provides a directory (on the machine from which the job is submitted) for the user log, when a full path is not specified.

For vanilla or MPI universe jobs where there is a shared file system, it is the current working directory on the machine where the job is executed.

For vanilla, grid, or MPI universe jobs where file transfer mechanisms are utilized (there is *not* a shared file system), it is the directory on the machine from which the job is submitted where the input files come from, and where the job's output files go to.

For standard universe jobs, it is the directory on the machine from which the job is submitted where the *condor_shadow* daemon runs; the current working directory for file input and output accomplished through remote system calls.

For scheduler universe jobs, it is the directory on the machine from which the job is submitted where the job runs; the current working directory for file input and output with respect to relative path names.

Note that the path to the executable is *not* relative to **initialdir**; if it is a relative path, it is relative to the directory in which the *condor_submit* command is run.

**job_lease_duration =** <**number-of-seconds**> For vanilla and java universe jobs only, the duration (in seconds) of a job lease. The default value is undefined. See section 2.14.4 for details of job leases.

**kill_sig =** <**signal-number**> When Condor needs to kick a job off of a machine, it will send the job the signal specified by **signal-number**. **signal-number** needs to be an integer which represents a valid signal on the execution machine. For jobs submitted to the standard universe, the default value is the number for `SIGTSTP` which tells the Condor libraries to initiate a checkpoint of the process. For jobs submitted to the vanilla universe, the default is `SIGTERM` which is the standard way to terminate a program in Unix.

**match_list_length =** <**integer value**> Defaults to the value zero (0). When **match_list_length** is defined with an integer value greater than zero (0), attributes are inserted into the job ClassAd. The maximum number of attributes defined is given by the integer value. The job ClassAds introduced are given as

```
LastMatchName0 = "most-recent-Name"
LastMatchName1 = "next-most-recent-Name"
```

The value for each introduced ClassAd is given by the value of the `Name` attribute from the machine ClassAd of a previous execution (match). As a job is matched, the definitions for these attributes will roll, with `LastMatchName1` becoming `LastMatchName2`, `LastMatchName0` becoming `LastMatchName1`, and `LastMatchName0` being set by the most recent value of the `Name` attribute.

An intended use of these job attributes is in the requirements expression. The requirements can allow a job to prefer a match with either the same or a different resource than a previous match.

**max_job_retirement_time =** <**integer expression**> An integer-valued expression (in seconds) that does nothing unless the machine that runs the job has been configured to provide retirement time (see section 3.5.9). Retirement time is a grace period given to a job to finish naturally when a resource claim is about to be preempted. No kill signals are sent during a retirement time. The default behavior in many cases is to take as much retirement time as the machine offers, so this command will rarely appear in a submit description file.

When a resource claim is to be preempted, this expression in the submit file specifies the maximum run time of the job (in seconds, since the job started). This expression has no effect, if it is greater than the maximum retirement time provided by the machine policy. If the resource claim is *not* preempted, this expression and the machine retirement policy are irrelevant. If the resource claim *is* preempted and the job finishes sooner than the maximum time, the claim closes gracefully and all is well. If the resource claim is preempted and the job does *not* finish in time, the usual preemption procedure is followed (typically a soft kill signal, followed by some time to gracefully shut down, followed by a hard kill signal).

Standard universe jobs and any jobs running with **nice_user** priority have a default **max_job_retirement_time** of 0, so no retirement time is utilized by default. In all other cases, no default value is provided, so the maximum amount of retirement time is utilized by default.

Setting this expression does not affect the job's resource requirements or preferences. For a job to only run on a machine with a minimum , or to preferentially run on such machines, explicitly specify this in the requirements and/or rank expressions.

**nice_user =** <**True** | **False**> Normally, when a machine becomes available to Condor, Condor decides which job to run based upon user and job priorities. Setting **nice_user** equal to `True` tells Condor not to use your regular user priority, but that this job should have last priority among all users and all jobs. So jobs submitted in this fashion run only on machines which no other non-nice_user job wants — a true "bottom-feeder" job! This is very handy if a user has some jobs they wish to run, but do not wish to use resources that could instead be used to run other people's Condor jobs. Jobs submitted in this fashion have "nice-user." pre-appended in front of the owner name when viewed from *condor_q* or *condor_userprio*. The default value is **False**.

**noop_job =** <**ClassAd Boolean Expression**> When this boolean expression is `True`, the job is immediately removed from the queue, and Condor makes no attempt at running the job. The log file for the job will show a job submitted event and a job terminated event, along with an exit code of 0, unless the user specifies a different signal or exit code.

**noop_job_exit_code =** <**return value**> When **noop_job** is in the submit description file and evaluates to `True`, this command allows the job to specify the return value as shown in the job's log file job terminated event. If not specified, the job will show as having terminated with status 0. This overrides any value specified with **noop_job_exit_signal**.

**noop_job_exit_signal =** <**signal number**> When **noop_job** is in the submit description file and evaluates to `True`, this command allows the job to specify the signal number that the job's log event will show the job having terminated with.

**remote_initialdir =** <**directory-path**> The path specifies the directory in which the job is to be executed on the remote machine. This is currently supported in all universes except for the standard universe.

**rendezvousdir =** <**directory-path**> Used to specify the shared file system directory to be used for file system authentication when submitting to a remote scheduler. Should be a path to a preexisting directory.

+<**attribute**> = <**value**> A line which begins with a '+' (plus) character instructs *condor_submit* to insert the following *attribute* into the job ClasssAd with the given *value*.

In addition to commands, the submit description file can contain macros and comments:

**Macros** Parameterless macros in the form of `$(macro_name)` may be inserted anywhere in Condor submit description files. Macros can be defined by lines in the form of

```
<macro_name> = <string>
```

Three pre-defined macros are supplied by the submit description file parser. The third of the pre-defined macros is only relevant to MPI universe jobs. The `$(Cluster)` macro supplies the value of the `ClusterId` job ClassAd attribute, and the `$(Process)` macro supplies the value of the `ProcId` job ClassAd attribute. These macros are intended to aid in the specification of input/output files, arguments, etc., for clusters with lots of jobs, and/or could be used to supply a Condor process with its own cluster and process numbers on the command line. The `$(Process)` macro should not be used for PVM jobs. The `$(Node)` macro is defined only for MPI universe jobs. It is a unique value assigned for the duration of the job that essentially identifies the machine on which a program is executing.

If the dollar sign ($) is desired as a literal character, then use

```
$(DOLLAR)
```

In addition to the normal macro, there is also a special kind of macro called a *substitution macro* that allows the substitution of a ClassAd attribute value defined on the resource machine itself (gotten after a match to the machine has been made) into specific commands within the submit description file. The substitution macro is of the form:

```
$$(attribute)
```

A common use of this macro is for the heterogeneous submission of an executable:

```
executable = povray.$$(opsys).$$(arch)
```

Values for the `opsys` and `arch` attributes are substituted at match time for any given resource. This allows Condor to automatically choose the correct executable for the matched machine.

An extension to the syntax of the substitution macro provides an alternative string to use if the machine attribute within the substitution macro is undefined. The syntax appears as:

```
$$(attribute:string_if_attribute_undefined)
```

An example using this extended syntax provides a path name to a required input file. Since the file can be placed in different locations on different machines, the file's path name is given as an argument to the program.

```
argument = $$(input_file_path:/usr/foo)
```

On the machine, if the attribute `input_file_path` is not defined, then the path `/usr/foo` is used instead.

A further extension to the syntax of the substitution macro allows the evaluation of a ClassAd expression to define the value. As all substitution macros, the expression is evaluated after a match has been made. Therefore, the expression may refer to machine attributes by prefacing them with the scope resolution prefix `TARGET.`, as specified in section 4.1.2. To place a ClassAd expression into the substitution macro, square brackets are added to delimit the expression. The syntax appears as:

```
$$([ClassAd expression])
```

The environment macro, $ENV, allows the evaluation of an environment variable to be used in setting a submit description file command. The syntax used is

```
$ENV(variable)
```

An example submit description file command that uses this functionality evaluates the submittor's home directory in order to set the path and file name of a log file:

```
log = $ENV(HOME)/jobs/logfile
```

The environment variable is evaluated when the submit description file is processed.

The $RANDOM_CHOICE macro allows a random choice to be made from a given list of parameters at submission time. For an expression, if some randomness needs to be generated, the macro may appear as

```
$RANDOM_CHOICE(0,1,2,3,4,5,6)
```

When evaluated, one of the parameters values will be chosen.

**Comments** Blank lines and lines beginning with a pound sign ('#') character are ignored by the submit description file parser.

## Options

**-verbose** Verbose output - display the created job ClassAd

**-name *schedd_name*** Submit to the specified *condor_schedd*. Use this option to submit to a *condor_schedd* other than the default local one.

**-remote *schedd_name*** Submit to the specified *condor_schedd*, spooling all required input files over the network connection. This option is equivalent to using both **-name** and **-spool**.

**-pool *pool_name*** Look in the specified pool for the *condor_schedd* to submit to. This option is used with **-name** or **-remote**.

**-disable** Disable file permission checks.

**-password *passphrase*** Specify a password to the *MyProxy* server.

**-debug** Cause debugging information to be sent to `stderr`, based on the value of the configuration variable `SUBMIT_DEBUG`.

**-append *command*** Augment the commands in the submit description file with the given command. This command will be considered to immediately precede the Queue command within the submit description file, and come after all other previous commands. The submit description file is not modified. Multiple commands are specified by using the **-append** option multiple times. Each new command is given in a separate **-append** option. Commands with spaces in them will need to be enclosed in double quote marks.

**-spool** Spool all required input files, user log, and proxy over the connection to the *condor_schedd*. After submission, modify local copies of the files without affecting your jobs. Any output files for completed jobs need to be retrieved with *condor_transfer_data*.

***submit description file*** The pathname to the submit description file. If this optional argument is missing or equal to "-", then the commands are taken from standard input.

## Exit Status

*condor_submit* will exit with a status value of 0 (zero) upon success, and a non-zero value upon failure.

## Examples

- Submit Description File Example 1: This example queues three jobs for execution by Condor. The first will be given command line arguments of *15* and *2000*, and it will write its standard output to `foo.out1`. The second will be given command line arguments of *30* and *2000*, and it will write its standard output to `foo.out2`. Similarly the third will have arguments of

*45* and *6000*, and it will use `foo.out3` for its standard output. Standard error output (if any) from all three programs will appear in `foo.error`.

```
####################
#
# submit description file
# Example 1: queuing multiple jobs with differing
# command line arguments and output files.
#
####################

Executable     = foo
Universe       = standard

Arguments      = 15 2000
Output  = foo.out1
Error   = foo.err1
Queue

Arguments      = 30 2000
Output  = foo.out2
Error   = foo.err2
Queue

Arguments      = 45 6000
Output  = foo.out3
Error   = foo.err3
Queue
```

- Submit Description File Example 2: This submit description file example queues 150 runs of program *foo* which must have been compiled and linked for Sun workstations running Solaris 8. Condor will not attempt to run the processes on machines which have less than 32 Megabytes of physical memory, and it will run them on machines which have at least 64 Megabytes, if such machines are available. Stdin, stdout, and stderr will refer to `in.0`, `out.0`, and `err.0` for the first run of this program (process 0). Stdin, stdout, and stderr will refer to `in.1`, `out.1`, and `err.1` for process 1, and so forth. A log file containing entries about where and when Condor runs, takes checkpoints, and migrates processes in this cluster will be written into file `foo.log`.

```
####################
#
# Example 2: Show off some fancy features including
# use of pre-defined macros and logging.
#
####################

Executable     = foo
Universe       = standard
Requirements   = Memory >= 32 && OpSys == "SOLARIS28" && Arch =="SUN4u"
Rank           = Memory >= 64
Image_Size     = 28 Meg

Error   = err.$(Process)
Input   = in.$(Process)
```

```
Output  = out.$(Process)
Log = foo.log

Queue 150
```

- Command Line example: The following command uses the **-append** option to add two commands before the job(s) is queued. A log file and an error log file are specified. The submit description file is unchanged.

  ```
  condor_submit -a "log = out.log" -a "error = error.log" mysubmitfile
  ```

  Note that each of the added commands is contained within quote marks because there are space characters within the command.

- `periodic_remove` example: A job should be removed from the queue, if the total suspension time of the job is more than half of the run time of the job.
  Including the command

  ```
  periodic_remove = CumulativeSuspensionTime >
                    ((RemoteWallClockTime - CumulativeSuspensionTime) / 2.0)
  ```

  in the submit description file causes this to happen.

## General Remarks

- For security reasons, Condor will refuse to run any jobs submitted by user root (UID = 0) or by a user whose default group is group wheel (GID = 0). Jobs submitted by user root or a user with a default group of wheel will appear to sit forever in the queue in an idle state.
- All pathnames specified in the submit description file must be less than 256 characters in length, and command line arguments must be less than 4096 characters in length; otherwise, *condor_submit* gives a warning message but the jobs will not execute properly.
- Somewhat understandably, behavior gets bizarre if the user makes the mistake of requesting multiple Condor jobs to write to the same file, and/or if the user alters any files that need to be accessed by a Condor job which is still in the queue. For example, the compressing of data or output files before a Condor job has completed is a common mistake.
- To disable checkpointing for Standard Universe jobs, include the line:

  ```
  +WantCheckpoint = False
  ```

  in the submit description file before the queue command(s).

## See Also

Condor User Manual

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_submit_dag*

Manage and queue jobs within a specified DAG for execution on remote machines

## Synopsis

***condor_submit_dag*** [**-no_submit**] [**-verbose**] [**-force**] [**-maxidle** *NumberOfJobs*] [**-maxjobs** *NumberOfJobs*] [**-dagman** *DagmanExecutable*] [**-maxpre** *NumberOfPREscripts*] [**-maxpost** *NumberOfPOSTscripts*] [**-storklog** *LogFileName*] [**-notification** *value*] [**-noeventchecks**] [**-allowlogerror**] [**-r** *schedd_name*] [**-debug** *level*] [**-usedagdir**] [**-outfile_dir** *directory*] *DAGInputFile1* [*DAGInputFile2 . . .DAGInputFileN* ]

## Description

*condor_submit_dag* is the program for submitting a DAG (directed acyclic graph) of jobs for execution under Condor. The program enforces the job dependencies defined in one or more *DAGInputFile*s. Each *DAGInputFile* contains commands to direct the submission of jobs implied by the nodes of a DAG to Condor. See the Condor User Manual, section 2.11 for a complete description.

## Options

**-no_submit** Produce the Condor submit description file for DAGMan, but do not submit DAGMan as a Condor job.

**-verbose** Cause *condor_submit_dag* to give verbose error messages.

**-force** Require *condor_submit_dag* to overwrite the files that it produces, if the files already exist. Note that `dagman.out` will be appended to, not overwritten.

**-maxidle *NumberOfJobs*** Sets the maximum number of idle jobs allowed before *condor_dagman* stops submitting more jobs. Once idle jobs start to run, *condor_dagman* will resume submitting jobs. *NumberOfJobs* is a positive integer. If the option is omitted, the number of idle jobs is unlimited.

**-maxjobs *NumberOfJobs*** Sets the maximum number of jobs within the DAG that will be submitted to Condor at one time. *NumberOfJobs* is a positive integer. If the option is omitted, the default number of jobs is unlimited.

**-dagman *DagmanExecutable*** Allows the specification of an alternate *condor_dagman* executable to be used instead of the one found in the user's path. This must be a fully qualified path.

**-maxpre *NumberOfPREscripts*** Sets the maximum number of PRE scripts within the DAG that may be running at one time. *NumberOfPREScripts* is a positive integer. If this option is omitted, the default number of PRE scripts is unlimited.

**-maxpost *NumberOfPOSTscripts*** Sets the maximum number of POST scripts within the DAG that may be running at one time. *NumberOfPOSTScripts* is a positive integer. If this option is omitted, the default number of POST scripts is unlimited.

**-log *LogFileName*** Deprecated option; do not use.

**-storklog *LogFileName*** Sets the file name for the Stork log for data placement jobs.

**-notification *value*** Sets the e-mail notification for DAGMan itself. This information will be used within the Condor submit description file for DAGMan. This file is produced by *condor_submit_dag*. See **notification** within the section of submit description file commands in the *condor_submit* manual page on page 716 for specification of *value*.

**-noeventchecks** This argument is no longer used; it is now ignored. Its functionality is now implemented by the `DAGMAN_ALLOW_EVENTS` configuration macro (see section 3.3.23).

**-allowlogerror** This optional argument has *condor_dagman* try to run the specified DAG, even in the case of detected errors in the user log specification.

**-r *schedd_name*** Submit to a remote schedd. The jobs will be submitted to the schedd on the specified remote host. On Unix systems, the Condor administrator for you site must override the default AUTHENTICATION_METHODS configuration setting to enable remote file system (FS_REMOTE) authentication.

**-debug *level*** Passes the the *level* of debugging output desired to *condor_dagman*. *level* is an integer, with values of 0-7 inclusive, where 7 is the most verbose output. A default value of 3 is passed to *condor_dagman* when not specified with this option. See the *condor_dagman* manual page on page 635 for detailed descriptions of these values.

**-usedagdir** This optional argument has causes *condor_dagman* to run each specified DAG as if *condor_submit_dag* had been run in the directory containing that DAG file. This option is most useful when running multiple DAGs in a single *condor_dagman*.

**-outfile_dir *directory*** Specifies the directory in which the `.dagman.out` file will be written. The *directory* may be specified relative to the current working directory as *condor_submit_dag* is executed, or specified with an absolute path. Without this option, the `.dagman.out` file is placed in the same directory as the first DAG input file listed on the command line.

## See Also

Condor User Manual

## Exit Status

*condor_submit_dag* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Examples

To run a single DAG:

```
% condor_submit_dag diamond.dag
```

To run a DAG when it has already been run and the output files exist:

```
% condor_submit_dag -force diamond.dag
```

To run a DAG, limiting the number of idle node jobs in the DAG to a maximum of five:

```
% condor_submit_dag -maxidle 5 diamond.dag
```

To run a DAG, limiting the number of concurrent PRE scripts to 10 and the number of concurrent POST scripts to five:

```
% condor_submit_dag -maxpre 10 -maxpost 5 diamond.dag
```

To run two DAGs, each of which is set up to run in its own directory:

```
% condor_submit_dag -usedagdir dag1/diamond1.dag dag2/diamond2.dag
```

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_transfer_data*

transfer spooled data

## Synopsis

***condor_transfer_data*** [**-help** | **-version**]

***condor_transfer_data*** [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname* ]| [**-addr** *"<a.b.c.d:port>"*] *cluster. . .*| *cluster.process. . .*| *user. . .* | **-constraint** *expression . . .*

***condor_transfer_data*** [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname* ]| [**-addr** *"<a.b.c.d:port>"*] **-all**

## Description

*condor_transfer_data* causes Condor to transfer spooled data. It is meant to be used in conjunction with the **-spool** option of *condor_submit*, as in

```
condor_submit -spool mysubmitfile
```

Submission of a job with the **-spool** option causes Condor to spool all input files, the user log, and any proxy across a connection to the machine where the *condor_schedd* daemon is running. After spooling these files, the machine from which the job is submitted may disconnect from the network or modify its local copies of the spooled files.

When the job finishes, the job has `JobStatus` = 4, meaning that the job has completed. The output of the job is spooled, and *condor_transfer_data* retrieves the output of the completed job.

## Options

**-help** Display usage information

**-version** Display version information

**-pool *centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's hostname and an optional port number

**-name *scheddname*** Send the command to a machine identified by *scheddname*

**-addr** ***"<a.b.c.d:port>"*** Send the command to a machine located at *"<a.b.c.d:port>"*

***cluster*** Transfer spooled data belonging to the specified cluster

***cluster.process*** Transfer spooled data belonging to a specific job in the cluster

***user*** Transfer spooled data belonging to the specified user

**-constraint** ***expression*** Transfer spooled data for jobs which match the job ClassAd expression constraint

***-all*** Transfer all spooled data

## Exit Status

*condor_transfer_data* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_updates_stats*

Display output from *condor_status*

## Synopsis

***condor_updates_stats*** [**--help** | **-h**] | [**--version**]

***condor_updates_stats*** [**--long** | **-l**] [**--history=**<**min**>**-**<**max**>] [**--interval=**<**seconds**>] [**--notime**] [**--time**] [**--summary** | **-s**]

## Description

*condor_updates_stats* parses the output from *condor_status*, and it displays the information relating to update statistics in a useful format. The statistics are displayed with the most recent update first; the most recent update is numbered with the smallest value.

The number of historic points that represent updates is configurable on a per-source basis. See `COLLECTOR_DAEMON_HISTORY_SIZE` in section 3.3.17.

## Options

**—help** Display usage information and exit.

**-h** Same as **—help**.

**—version** Display Condor version information and exit.

**—long** All update statistics are displayed. Without this option, the statistics are condensed.

**-l** Same as **—long**.

**—history=**<**min**>**-**<**max**> Sets the range of update numbers that are printed. By default, the entire history is displayed. To limit the range, the minimum and/or maximum number may be specified. If a minimum is not specified, values from 0 to the maximum are displayed. If the maximum is not specified, all values after the minimum are displayed. When both minimum and maximum are specified, the range to be displayed includes the endpoints as well as all values in between. If no = sign is given, command-line parsing fails, and usage information is displayed. If an = sign is given, with no minimum or maximum values, the default of the

entire history is displayed.

**—interval=**<**seconds**> The assumed update interval, in seconds. Assumed times for the the updates are displayed, making the use of the **—time** option together with the **—interval** option redundant.

**—notime** Do not display assumed times for the the updates. If more than one of the options **—notime** and **—time** are provided, the final one within the command line parsed determines the display.

**—time** Display assumed times for the the updates. If more than one of the options **—notime** and **—time** are provided, the final one within the command line parsed determines the display.

**—summary** Display only summary information, not the entire history for each machine.

**-s** Same as **—summary**.

## Exit Status

*condor_updates_stats* will exit with a status value of 0 (zero) upon success, and it will exit with a nonzero value upon failure.

## Examples

Assuming the default of 128 updates kept, and assuming that the update interval is 5 minutes, *condor_updates_stats* displays:

```
$ condor_status -l host1 | condor_updates_stats --interval=300
(Reading from stdin)
*** Name/Machine = 'HOST1.cs.wisc.edu' MyType = 'Machine' ***
 Type: Main
   Stats: Total=2277, Seq=2276, Lost=3 (0.13%)
     0 @ Mon Feb 16 12:55:38 2004: Ok
  ...
    28 @ Mon Feb 16 10:35:38 2004: Missed
    29 @ Mon Feb 16 10:30:38 2004: Ok
  ...
   127 @ Mon Feb 16 02:20:38 2004: Ok
```

Within this display, update numbered 27, which occurs later in time than the missed update numbered 28, is Ok. Each change in state, in reverse time order, displays in this condensed version.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_userlog*

Display and summarize job statistics from job log files.

## Synopsis

***condor_userlog*** [**-help**] [**-total** | **-raw**] [**-debug**] [**-evict**] [**-j** *cluster* | *cluster.proc*] [**-all**] [**-hostname**] *logfile . . .*

## Description

*condor_userlog* parses the information in job log files and displays summaries for each workstation allocation and for each job. See the manual page for *condor_submit* on page 716 for instructions for specifying that Condor write a log file for your jobs.

If **-total** is not specified, *condor_userlog* will first display a record for each workstation allocation, which includes the following information:

**Job** The cluster/process id of the Condor job.

**Host** The host where the job ran. By default, the host's IP address is displayed. If **-hostname** is specified, the hostname will be displayed instead.

**Start Time** The time (month/day hour:minute) when the job began running on the host.

**Evict Time** The time (month/day hour:minute) when the job was evicted from the host.

**Wall Time** The time (days+hours:minutes) for which this workstation was allocated to the job.

**Good Time** The allocated time (days+hours:min) which contributed to the completion of this job. If the job exited during the allocation, then this value will equal "Wall Time." If the job performed a checkpoint, then the value equals the work saved in the checkpoint during this allocation. If the job did not exit or perform a checkpoint during this allocation, the value will be 0+00:00. This value can be greater than 0 and less than "Wall Time" if the application completed a periodic checkpoint during the allocation but failed to checkpoint when evicted.

**CPU Usage** The CPU time (days+hours:min) which contributed to the completion of this job.

*condor_userlog* will then display summary statistics per host:

**Host/Job** The IP address or hostname for the host.

**Wall Time** The workstation time (days+hours:minutes) allocated by this host to the jobs specified in the query. By default, all jobs in the log are included in the query.

**Good Time** The time (days+hours:minutes) allocated on this host which contributed to the completion of the jobs specified in the query.

**CPU Usage** The CPU time (days+hours:minutes) obtained from this host which contributed to the completion of the jobs specified in the query.

**Avg Alloc** The average length of an allocation on this host (days+hours:minutes).

**Avg Lost** The average amount of work lost (days+hours:minutes) when a job was evicted from this host without successfully performing a checkpoint.

**Goodput** This percentage is computed as Good Time divided by Wall Time.

**Util.** This percentage is computed as CPU Usage divided by Good Time.

*condor_userlog* will then display summary statistics per job:

**Host/Job** The cluster/process id of the Condor job.

**Wall Time** The total workstation time (days+hours:minutes) allocated to this job.

**Good Time** The total time (days+hours:minutes) allocated to this job which contributed to the job's completion.

**CPU Usage** The total CPU time (days+hours:minutes) which contributed to this job's completion.

**Avg Alloc** The average length of a workstation allocation obtained by this job in minutes (days+hours:minutes).

**Avg Lost** The average amount of work lost (days+hours:minutes) when this job was evicted from a host without successfully performing a checkpoint.

**Goodput** This percentage is computed as Good Time divided by Wall Time.

**Util.** This percentage is computed as CPU Usage divided by Good Time.

Finally, *condor_userlog* will display a summary for all hosts and jobs.

## Options

**-help** Get a brief description of the supported options

**-total** Only display job totals

**-raw** Display raw data only

**-debug** Debug mode

**-j** Select a specific cluster or cluster.proc

**-evict** Select only allocations which ended due to eviction

**-all** Select all clusters and all allocations

**-hostname** Display hostname instead of IP address

## General Remarks

Since the Condor job log file format does not contain a year field in the timestamp, all entries are assumed to occur in the current year. Allocations which begin in one year and end in the next will be silently ignored.

## Exit Status

*condor_userlog* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention:

Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_userprio*

Manage user priorities

## Synopsis

***condor_userprio*** [**-pool** *centralmanagerhostname[:portnumber]*] [**-all**] [**-usage**] [**-setprio** *username value*] [**-setfactor** *username value*] [**-setaccum** *username value*] [**-setbegin** *username value*] [**-setlast** *username value*] [**-resetusage** *username*] [**-resetall**] [**-delete** *username*] [**-getreslist** *username*] [**-allusers**] [**-activefrom** *month day year*] [**-l**]

## Description

*condor_userprio* with no arguments, lists the active users (see below) along with their priorities, in increasing priority order. The -all option can be used to display more detailed information about each user, which includes the following columns:

**Effective Priority** The effective priority value of the user, which is used to calculate the user's share when allocating resources. A lower value means a higher priority, and the minimum value (highest priority) is 0.5. The effective priority is calculated by multiplying the real priority by the priority factor.

**Real Priority** The value of the real priority of the user. This value follows the user's resource usage.

**Priority Factor** The system administrator can set this value for each user, thus controlling a user's effective priority relative to other users. This can be used to create different classes of users.

**Res Used** The number of resources currently used (e.g. the number of running jobs for that user).

**Accumulated Usage** The accumulated number of resource-hours used by the user since the usage start time.

**Usage Start Time** The time since when usage has been recorded for the user. This time is set when a user job runs for the first time. It is reset to the present time when the usage for the user is reset (with the -resetusage or -resetall options).

**Last Usage Time** The most recent time a resource usage has been recorded for the user.

The -usage option displays the username, accumulated usage, usage start time and last usage time for each user, sorted on accumulated usage.

The -setprio, -setfactor options are used to change a user's real priority and priority factor. The -setaccum option sets a user's accumulated usage. The -setbegin, -setlast options are used to change a user's begin usage time and last usage time. The -setaccum option sets a user's accumulated usage.

The -resetusage and -resetall options are used to reset the accumulated usage for users. The usage start time is set to the current time when the accumulated usage is reset. These options require administrator privileges.

By default only users for whom usage was recorded in the last 24 hours or whose priority is greater than the minimum are listed. The -activefrom and -allusers options can be used to display users who had some usage since a specified date, or ever. The summary line for last usage time will show this date.

The -getreslist option is used to display the resources currently used by a user. The output includes the start time (the time the resource was allocated to the user), and the match time (how long has the resource been allocated to the user).

Note that when specifying user names on the command line, the name must include the UID domain (e.g. user@uid-domain - exactly the same way user names are listed by the userprio command).

The -pool option can be used to contact a different central-manager instead of the local one (the default).

For security purposes (authentication and authorization), this command requires an administrator's level of access. See section 3.6.1 on page 261 for further explanation.

## Options

**-pool *centralmanagerhostname[:portnumber]*** Contact specified *centralmanagerhostname* with an optional port number instead of the local central manager. This can be used to check other pools. NOTE: The hostname (and optionally port) specified refer to the hostname (and port) of the *condor_negotiator* to query for user priorities. This is slightly different than most Condor tools that support -pool, which expect the hostname (and optionally port) of the *condor_collector*, instead.

**-all** Display detailed information about each user.

**-usage** Display usage information for each user.

**-setprio *username value*** Set the real priority of the specified user to the specified value.

**-setfactor *username value*** Set the priority factor of the specified user to the specified value.

**-setaccum *username value*** Set the accumulated usage of the specified user to the specified floating point value.

**-setbegin *username value*** Set the begin usage time of the specified user to the specified value.

**-setlast *username value*** Set the last usage time of the specified user to the specified value.

**-resetusage *username*** Reset the accumulated usage of the specified user to zero.

**-resetall** Reset the accumulated usage of all the users to zero.

**-delete *username*** Remove the specified *username* from Condor's accounting.

**-getreslist *username*** Display all the resources currently allocated to the specified user.

**-allusers** Display information for all the users who have some recorded accumulated usage.

**-activefrom *month day year*** Display information for users who have some recorded accumulated usage since the specified date.

**-l** Show the class-ad which was received from the central-manager in long format.

## Exit Status

*condor_userprio* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_vacate*

Vacate jobs that are running on the specified hosts

## Synopsis

***condor_vacate*** [**-help** | **-version**]

***condor_vacate*** [**-graceful** | **-fast**] [**-debug**] [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *hostname* ]| [**-addr** *"<a.b.c.d:port>"*] . . . [ | **-all**]

## Description

*condor_vacate* causes Condor to checkpoint any running jobs on a set of machines and force the jobs to vacate the machine. The job(s) remains in the submitting machine's job queue.

Given the (default) **-graceful** option, a job running under the standard universe will first produce a checkpoint and then the job will be killed. Condor will then restart the job somewhere else, using the checkpoint to continue from where it left off. A job running under the vanilla universe is killed, and Condor restarts the job from the beginning somewhere else. *condor_vacate* has no effect on a machine with no Condor job currently running.

There is generally no need for the user or administrator to explicitly run *condor_vacate*. Condor takes care of jobs in this way automatically following the policies given in configuration files.

## Options

**-help** Display usage information

**-version** Display version information

**-graceful** Inform the job to checkpoint, then soft-kill it.

**-fast** Hard-kill jobs instead of checkpointing them

**-debug** Causes debugging information to be sent to `stderr` based on the value of the configuration variable `TOOL_DEBUG`

**-pool *centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's hostname and an optional port number

**-name *hostname*** Send the command to a machine identified by *hostname*

***hostname*** Send the command to a machine identified by *hostname*

**-addr *"<a.b.c.d:port>"*** Send the command to a machine's master located at *"<a.b.c.d:port>"*

***"<a.b.c.d:port>"*** Send the command to a machine located at *"<a.b.c.d:port>"*

**-all** Send the command to all machines in the pool

## Exit Status

*condor_vacate* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Examples

To send a *condor_vacate* command to two named machines:

```
% condor_vacate  robin cardinal
```

To send the *condor_vacate* command to a machine within a pool of machines other than the local pool, use the **-pool** option. The argument is the name of the central manager for the pool. Note that one or more machines within the pool must be specified as the targets for the command. This command sends the command to a the single machine named **cae17** within the pool of machines that has **condor.cae.wisc.edu** as its central manager:

```
% condor_vacate -pool condor.cae.wisc.edu -name cae17
```

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_vacate_job*

vacate jobs in the Condor queue from the hosts where they are running

## Synopsis

***condor_vacate_job*** [**-help** | **-version**]

***condor_vacate_job*** [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname* ]| [**-addr** *"<a.b.c.d:port>"*] [**-fast**] *cluster...*| *cluster.process...*| *user...* | **-constraint** *expression ...*

***condor_vacate_job*** [**-pool** *centralmanagerhostname[:portnumber]* | **-name** *scheddname* ]| [**-addr** *"<a.b.c.d:port>"*] [**-fast**] **-all**

## Description

*condor_vacate_job* finds one or more jobs from the Condor job queue and vacates them from the host(s) where they are currently running. The jobs remain in the job queue and return to the idle state.

A job running under the standard universe will first produce a checkpoint and then the job will be killed. Condor will then restart the job somewhere else, using the checkpoint to continue from where it left off. A job running under any other universe will be sent a soft kill signal (SIGTERM by default, or whatever is defined as the `SoftKillSig` in the job ClassAd), and Condor will restart the job from the beginning somewhere else.

If the **-fast** option is used, the job(s) will be immediately killed, meaning that standard universe jobs will not be allowed to checkpoint, and the job will have to revert to the last checkpoint or start over from the beginning.

If the **-name** option is specified, the named *condor_schedd* is targeted for processing. If the **-addr** option is used, the *condor_schedd* at the given address is targeted for processing. Otherwise, the local *condor_schedd* is targeted. The jobs to be vacated are identified by one or more job identifiers, as described below. For any given job, only the owner of the job or one of the queue super users (defined by the `QUEUE_SUPER_USERS` macro) can vacate the job.

Using *condor_vacate_job* on jobs which are not currently running has no effect.

## Options

**-help** Display usage information

**-version** Display version information

**-pool *centralmanagerhostname[:portnumber]*** Specify a pool by giving the central manager's hostname and an optional port number

**-name *scheddname*** Send the command to a machine identified by *scheddname*

**-addr *"<a.b.c.d:port>"*** Send the command to a machine located at *"<a.b.c.d:port>"*

***cluster*** Vacate all jobs in the specified cluster

***cluster.process*** Vacate the specific job in the cluster

***user*** Vacate jobs belonging to specified user

**-constraint *expression*** Vacate all jobs which match the job ClassAd expression constraint

***-all*** Vacate all the jobs in the queue

***-fast*** Perform a fast vacate and hard kill the jobs

## General Remarks

Do not confuse *condor_vacate_job* with *condor_vacate*. *condor_vacate* is given a list of hosts to vacate, regardless of what jobs happen to be running on them. Only machine owners and administrators have permission to use *condor_vacate* to evict jobs from a given host. *condor_vacate_job* is given a list of job to vacate, regardless of which hosts they happen to be running on. Only the owner of the jobs or queue super users have permission to use *condor_vacate_job*.

When vacating a PVM universe job, you should always vacate the entire job cluster. (In the PVM universe, each PVM job is assigned its own cluster number, and each machine class is assigned a "process" number in the job's cluster.) Vacating a subset of the machine classes for a PVM job is not supported using *condor_vacate_job*. To vacate individual nodes in a PVM computation, you must use *condor_vacate* and target the specific hosts.

## Examples

To vacate job 23.0:

```
% condor_vacate_job 23.0
```

To vacate all jobs of a user named Mary:

```
% condor_vacate_job mary
```

To vacate all standard universe jobs owned by Mary:

```
% condor_vacate_job -constraint 'JobUniverse == 1 && Owner == "mary"'
```

Note that the entire constraint, including the quotation marks, must be enclosed in single quote marks for most shells.

## Exit Status

*condor_vacate_job* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_version*

print Condor version and platform information

## Synopsis

***condor_version*** [**-help**]

***condor_version*** [**-arch**] [**-opsys**] [**-syscall**]

## Description

With no arguments, *condor_version* prints the currently installed Condor version number and platform information.

## Options

**help** Print usage information

**arch** Print this machine's ClassAd value for `Arch`

**opsys** Print this machine's ClassAd value for `OpSys`

**syscall** Get any requested version and/or platform information from the `libcondorsyscall.a` that this Condor pool is configured to use, instead of using the values that are compiled into the tool itself. This option may be used in combination with any other options to modify where the information is coming from.

## Exit Status

*condor_version* will exit with a status value of 0 (zero) upon success, and it should never exit with a failing value.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *condor_wait*

Wait for jobs to finish

## Synopsis

***condor_wait*** [**-help** | **-version**]

***condor_wait*** [**-debug**] [**-wait** *seconds*] [**-num** *number-of-jobs*] *log-file* [**job ID**]

## Description

*condor_wait* watches a user log file (created with the **log** command within a submit description file) and returns when one or more jobs from the log have completed or aborted.

Because *condor_wait* expects to find at least one job submitted event in the log file, at least one job must have been successfully submitted with *condor_submit* before *condor_wait* is executed.

*condor_wait* will wait forever for jobs to finish, unless a shorter wait time is specified.

## Options

**-help** Display usage information

**-version** Display version information

**-debug** Show extra debugging information.

**-wait *seconds*** Wait no more than the integer number of *seconds.* The default is unlimited time.

**-num *number-of-jobs*** Wait for the integer *number-of-jobs* jobs to end. The default is all jobs in the log file.

**log file** The name of the log file to watch for information about the job.

**job ID** A specific job or set of jobs to watch. If the **job ID** is only the job ClassAd attribute `ClusterId`, then *condor_wait* waits for all jobs with the given `ClusterId`. If the **job ID** is a pair of the job ClassAd attributes, given by `ClusterId.ProcId`, then *condor_wait* waits for the specific job with this **job ID**. If this option is not specified, all jobs that exist in

the log file when *condor_wait* is invoked will be watched.

## General Remarks

*condor_wait* is an inexpensive way to test or wait for the completion of a job or a whole cluster, if you are trying to get a process outside of Condor to synchronize with a job or set of jobs.

It can also be used to wait for the completion of a limited subset of jobs, via the **-num** option.

## Examples

```
condor_wait logfile
```

This command waits for all jobs that exist in `logfile` to complete.

```
condor_wait logfile 40
```

This command waits for all jobs that exist in `logfile` with a job ClassAd attribute `ClusterId` of 40 to complete.

```
condor_wait -num 2 logfile
```

This command waits for any two jobs that exist in `logfile` to complete.

```
condor_wait logfile 40.1
```

This command waits for job 40.1 that exists in `logfile` to complete.

```
condor_wait -wait 3600 logfile 40.1
```

This waits for job 40.1 to complete by watching `logfile`, but it will not wait more than one hour (3600 seconds).

## Exit Status

*condor_wait* exits with 0 if and only if the specified job or jobs have completed or aborted. *condor_wait* returns 1 if unrecoverable errors occur, such as a missing log file, if the job does not exist in the log file, or the user-specified waiting time has expired.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *filelock_midwife*

create an artifact of the creation of a process

## Synopsis

***filelock_midwife* -help**

***filelock_midwife*** [**–file** *filename*] *program* [*programargs*]

## Description

*filelock_midwife* starts a given *program,* while creating an artifact of the program's birth. At a later time the *filelock_undertaker* can examine the artifact to determine whether the program is still running, or whether the program has exited. *filelock_midwife* accomplishes this by obtaining a file lock on the given artifact file before starting the program.

Warning: *filelock_midwife* will not work on NFS unless the separate file lock server is running.

## Options

**–file *filename*** The *filename* to use for the artifact file. The file `lock.file` is the default file used when this option is not specified.

***program* [*programargs*]** Forks a process and executes *program* with *programargs* as command-line arguments (when specified).

## Exit Status

*filelock_midwife* will exit with a status of 0 (zero) upon success, and non-zero otherwise.

## See Also

*uniq_pid_midwife* (on page 792), *filelock_undertaker* (on page 772).

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *filelock_undertaker*

determine whether a process has exited

## Synopsis

***filelock_undertaker* -help**

***filelock_undertaker*** [**–file** *filename*] [**–block**]

## Description

*filelock_undertaker* can examine an artifact file created by *filelock_midwife* and determine whether the program started by the *midwife* has exited. It does this by attempting to acquire a file lock.

Be warned that this will not work on NFS unless the separate file lock server is running.

## Options

**–block** If the process has not exited, block until it does.

**–file *filename*** The name of the artifact file. created by *filelock_midwife*. The file `lock.file` is the default file used when this option is not specified.

## Exit Status

*filelock_undertaker* will exit with a status of 0 (zero) if the monitored process has exited, with a status of 1 (one) if the monitored process has definitely not exited, with a status of 2 if it is uncertain whether the process has exited (this is generally due to a failure by the *filelock_midwife*), or with any other value for program failure.

## See Also

*uniq_pid_undertaker* (on page 794), *filelock_midwife* (on page 770).

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *install_release*

install an arbitrary software release into a named directory

## Synopsis

***install_release*** [**-help**]

***install_release*** [**-f**] [**-basedir** *directory*] [**-log** *filename*] [**-wget**] [**-globuslocation** *directory*] [**-o** *otherfile1...*] *package*

## Description

*install_release* installs an arbitrary software release into a named directory. In addition it creates a log of the installed files for easy uninstallation. This program can install packages of type tar, gzip, or gzip'ed tar. The installation package can be located on a mounted file system, an http server, an ftp server, or a grid ftp server.

## Options

**-basedir *directory*** The directory where the package should be installed. When not specified, the directory defaults to the current working directory.

**-f** Forcefully overwrite files if they exist.

**-globuslocation *directory*** This program does not come prepackaged with *globus-url-copy* or the supporting libraries. If globus is not installed in the `/opt/globus` directory, the user must specify the installation location of globus using this option.

**-help** Display brief usage information and exit.

**-log *filename*** The file name for the installation log.

**-o *otherfile1...*** A space-separated list of files that will be installed along with the installation package. The files will only be copied. No extraction or decompression will be performed on these files. These files will be logged in the installation log.

***package*** The full path to the installation package. Locations on file systems can be specified without the `file:` prefix, but other locations must prefix with the appropriate protocol (`http:`, `ftp:`, or `gsiftp:`).

**-wget** This program defaults to using *globus-url-copy* to fetch the installation package. This option specifies that this program should use *wget* for http and ftp requests and Perl's copy function for file system requests. *wget* must be installed on the machine and must be in the user's path.

## Exit Status

*install_release* will exit with a status value of 0 (zero) upon success, and non-zero otherwise.

## See Also

*cleanup_release* (on page 596)

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *stork_q*

Query active Stork data placement jobs.

## Synopsis

***stork_q*** [**-help** | **-version**]

***stork_q*** [**-debug**] [**-name** *server_specification*]

## Description

*stork_q* prints the entire queue of active Stork jobs. Output is provided in the syntax of the ClassAd language. See http://www.cs.wisc.edu/condor/classad for information on the ClassAd language. Completed jobs are not printed. If the **-name** option is specified, the named *stork_server* is targeted for processing. Otherwise, the local *stork_server* is targeted.

## Options

**-help** Display usage information

**-version** Display version information

**-debug** Show extra debugging information.

**-name *server_specification*** Specification of *stork_server* using *machinename:port* or <a.b.c.d:port>.

## Exit Status

*stork_q* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *stork_list_cred*

list all credentials of a user stored on the *stork_credd* daemon on behalf of Stork

## Synopsis

***stork_list_cred*** [**-help** | **-version**]

***stork_list_cred*** [**-debug**] **-n** *hostname:portnumber*

## Description

*stork_list_cred* lists all credentials stored on the *stork_credd* daemon of the user issuing the *stork_list_cred* command. Two fields appear for each item listed: the unique name for the credential, and the string "X.509". As more credential types are handled by the *stork_credd* daemon, this second field with identify the credential type.

## Options

**-help** Display usage information

**-version** Display version information

**-debug** Show extra debugging information.

**-n *hostname:portnumber*** Identify the *stork_credd* daemon by host name and port number.

## Exit Status

*stork_list_cred* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *stork_rm*

remove a Stork job

## Synopsis

***stork_rm*** [**-help** | **-version**]

***stork_rm*** [**-debug**] [**-name** *server_specification*] *job-id*

## Description

*stork_rm* removes a Stork job from the queue, using the required command-line argument to identify the Stork job. *stork_rm* removes a single job from the Stork job queue. If the **-name** option is specified, the named *stork_server* is targeted for processing. Otherwise, the local *stork_server* is targeted. The job to be removed is identified by the job id returned by *stork_submit*.

## Options

**-help** Display usage information

**-version** Display version information

**-debug** Show extra debugging information.

**-name *server_name*** Name of *stork_server*.

**-name *server_specification*** Specification of *stork_server* using *machinename:port* or <a.b.c.d:port>.

## Exit Status

*stork_rm* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *stork_rm_cred*

cause the *stork_credd* daemon to remove a credential

## Synopsis

***stork_rm_cred*** [**-help** | **-version**]

***stork_rm_cred*** [**-debug**] [**-N** *credential-name*] **-n** *hostname:portnumber*

## Description

*stork_rm_cred* removes a credential from the *stork_credd* daemon.

## Options

**-help** Display usage information

**-version** Display version information

**-debug** Show extra debugging information.

**-n *hostname:portnumber*** Identify the *stork_credd* daemon by host name and port number.

**-N *credential-name*** The unique name of the credential to be removed. When not specified, the *stork_credd* daemon uses the string ”DEFAULT” as the unique name.

## Exit Status

*stork_rm_cred* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *stork_store_cred*

store a credential on the *stork_credd* daemon for use by Stork

## Synopsis

***stork_store_cred*** [**-help** | **-version**]

***stork_store_cred*** [**-debug**] [**-N** *credential-name*] [**-m** *[user@]hostname[:portnumber]*] [**-D** *proxy-server-name*] [**-S**] **-n** *hostname:portnumber* **-f** *filename* **-t x509**

## Description

*stork_store_cred* stores a credential to a *stork_credd* daemon.

The required **-t** option identifies that the credential is an X.509 certificate.

## Options

**-help** Display usage information

**-version** Display version information

**-debug** Show extra debugging information.

**-n *hostname:portnumber*** Identify the *stork_credd* daemon by host name and port number.

**-f *filename*** A full path and file name where the credential is stored.

**-N *credential-name*** A unique name used by the *stork_credd* daemon to identify the credential. When not specified, the *stork_credd* daemon uses the string ”DEFAULT” as the unique name.

**-m *[user@]hostname[:portnumber]*** An identification of the *MyProxy* server.

**-D *proxy-server-name*** The distinguished name of the *MyProxy* server.

**-S** Read the *MyProxy* password from standard input.

## Exit Status

*stork_store_cred* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *stork_status*

print a Stork job's status

## Synopsis

***stork_status*** [**-help** | **-version**]

***stork_status*** [**-debug**] [**-name** *server_specification*] *job-id*

## Description

*stork_status* prints information about a Stork job. Jobs may be current or completed. If the **-name** option is specified, the named *stork_server* is targeted for processing. Otherwise, the local *stork_server* is targeted. The job to be removed is identified by the job id (as returned by *stork_submit*). The information is printed as a ClassAd. See http://www.cs.wisc.edu/condor/classad for information on the ClassAd language.

## Options

**-help** Display usage information

**-version** Display version information

**-debug** Show extra debugging information.

**-name *server_name*** Name of *stork_server*.

**-name *server_specification*** Specification of *stork_server* using *machinename:port* or <a.b.c.d:port>.

## Exit Status

*stork_status* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *stork_submit*

submit a Stork job

## Synopsis

***stork_submit*** [**-help** | **-version**]

***stork_submit*** [**-debug**] [**-stdin**] [**-name** *server_specification*] [**-lognotes** *"prose"*] *submit-descpription-file*

## Description

*stork_submit* is used to submit a Stork data placement job. Upon job submission, an integer identifier is assigned to the submission, and it is printed to standard output. This job identifier is required by other commands that manage Stork jobs.

The name of the Stork submit description file is the single, required, command-line argument. Stork places no constraints on the submit description file name. See section 2.12, for a more complete description of Stork.

*STORK SUBMIT DESCRIPTION FILE COMMANDS* Stork submit description files use ClassAd syntax, different from Condor submit file syntax. See http://www.cs.wisc.edu/condor/classad/.

**dap_type = transfer;** Required command identifying that there will be a transfer from source to destination.

**arguments = "**<**argument_list**>**";** List of arguments to be supplied to the module on the command line. Arguments are delimited (separated) by space characters.

**input = "**<**pathname**>**";** Stork assumes that its jobs are long-running, and that the user will not wait at the terminal for their completion. Because of this, the standard files which normally access the terminal, (`stdin`, `stdout`, and `stderr`), must refer to files. Thus, the file name specified with **input** should contain any keyboard input the program requires (that is, this file becomes `stdin`). If not specified, the default value of `/dev/null` is used for submission to a Unix machine.

*stork_submit* will prepend the current working directory if the pathname is relative (does not start with a / character). This implies that the submit directory must be shared between *stork_submit* and the Stork server host, when using relative paths. All local file paths passed to Stork must be valid on the Stork server host.

Note that this command does *not* refer to the command-line arguments of the program. The command-line arguments are specified by the **arguments** command.

**output = "**<**pathname**>**";** The **output** file name will capture any information the program would normally write to the screen (that is, this file becomes `stdout`). If not specified, the default value of `/dev/null` is used for submission to a Unix machine. Multiple jobs should not use the same output file, since this will cause one job to overwrite the output of another. The output file and the error file should not be the same file as the outputs will overwrite each other or be lost.

Note that if your program explicitly opens and writes to a file, that file should *not* be specified as the output file.

*stork_submit* will prepend the current working directory if the pathname is relative (does not start with a / character). This implies that the submit directory must be shared between *stork_submit* and the Stork server host, when using relative paths. All local file paths passed to Stork must be valid on the Stork server host.

**err = "**<**pathname**>**";** The **err** file name will capture any error messages the program would normally write to the screen (that is, this file becomes `stderr`). If not specified, the default value of `/dev/null` is used for submission to a Unix machine. More than one job should not use the same error file, since this will cause one job to overwrite the errors of another. The error file and the output file should not be the same file as the outputs will overwrite each other or be lost.

*stork_submit* will prepend the current working directory if the pathname is relative (does not start with a / character). This implies that the submit directory must be shared between *stork_submit* and the Stork server host, when using relative paths. All local file paths passed to Stork must be valid on the Stork server host.

**log = "**<**pathname**>**";** Use **log** to specify a file name where Stork will write a log file of what is happening with this job cluster. For example, Stork will log into this file when and where the job begins running, when the job is checkpointed and/or migrated, when the job completes, etc. Most users find specifying a **log** file to be very handy; its use is recommended. If no **log** entry is specified, Stork does not create a log for this job.

*stork_submit* will prepend the current working directory if the pathname is relative (does not start with a / character). This implies that the submit directory must be shared between *stork_submit* and the Stork server host, when using relative paths. All local file paths passed to Stork must be valid on the Stork server host. **log** file paths should not use NFS file systems.

**log_xml = "True";** | **"False";** If **log_xml** is true, then the log file will be written in ClassAd XML. If it isn't specified, XML is not used. Note that it's an XML fragment, and is missing the file header and footer. Also note that you should never mix XML and non-XML in a single file: if multiple jobs write to a single log file, it is up to you to make sure that all of them specify (or don't specify) this option in the same way.

**src_url =** <**protocol-name:URL**> A (required) URL to identify the data source, as well as the protocol to be used at the source. `file:///` URLs must refer to valid paths on the Stork server host.

**dest_url =** <**protocol-name:URL**> A (required) URL to identify the data destination, as well as the protocol to be used at the destination. `file:///` URLs must refer to valid paths on the Stork server host.

**x509proxy =** <**path-to-proxy**> The path to and file name of an X.509 proxy when needed for GSI authentication. A value of **"default"** directs Stork to search in the standard Globus GSI proxy locations.

**cred_name =** <**credential-handle**> Alternatively, a X.509 proxy may be managed via the Stork credential manager. The **cred_name** specifies the name by which the credential was stored in the credential manager. See the *stork_store_cred* manual on page 784.

**alt_protocols =** <**sourceprotocol-desinationprotocol, sourceprotocol-desinationprotocol, ...**> A comma separated list of alternative protocol pairings to be used when a data transfer fails. For each pair, the protocol to use at the source of the transfer is followed by a - (dash) and the protocol to be used at the destination of the transfer. The list is used (together with the original **src_url** and **dest_url** protocols) in a round robin fashion. The source and destination URLs are unchanged; only the protocols to be used are changed.

## Options

**-help** Display usage information

**-version** Display version information

**-debug** Show extra debugging information.

**-stdin** Read commands from `stdin` instead of from a file.

**-name *server_specification*** Specification of *stork_server* using *machinename:port* or <a.b.c.d:port>.

**-lognotes *"prose"*** The string given within quote marks is appended to the data placement ClassAd before the job is submitted.

## Exit Status

*stork_submit* will exit with a status value of 0 (zero) upon success, and it will exit with the value 1 (one) upon failure.

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *uniq_pid_midwife*

create an artifact of the creation of a process

## Synopsis

***uniq_pid_midwife*** [**--noblock**] [**--file** *filename*] [**--precision** *seconds*] *program* [*programargs*]

## Description

*uniq_pid_midwife* starts a given program, while creating an artifact of the program's birth. At a later time the *uniq_pid_undertaker* can examine the artifact to determine whether the program is still running or whether it has exited. *uniq_pid_midwife* accomplishes this by recording an enforced unique process identifier to the artifact.

## Options

**--file** ***filename*** The *filename* to use for the artifact file. Defaults to `pid.file`.

**--precision** ***seconds*** The precision the operating system is expected to have in regards to process creation times. Defaults to an operating system specific value. The default is the best choice in most cases.

**--noblock** Exit after the program has been confirmed, typically 3 times the precision. Defaults to block until the program exits.

***program*** **[*programargs*]** Forks a process and executes *program* with *programargs* as command-line arguments (when specified).

## Exit Status

*uniq_pid_midwife* will exit with a status of 0 (zero) upon success, and non-zero otherwise.

## See Also

*uniq_pid_undertaker* (on page 794), *filelock_midwife* (on page 770).

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# *uniq_pid_undertaker*

determine whether a process has exited

## Synopsis

***uniq_pid_undertaker*** [**--block**] [**--file** *file*] [**--precision** *seconds*]

## Description

*uniq_pid_undertaker* can examine an artifact file created by *uniq_pid_midwife* and determine whether the program started by the *midwife* has exited.

## Options

**--block** If the process has not exited, block until it does.

**--file *file*** The name of the *uniq_pid_midwife* created artifact file. Defaults to `pid.file`.

**--precision *seconds*** Uses *seconds* as the precision range within which the operating system will provide a process's birthday. Defaults to an operating system specific value. Only use this option if the same *seconds* value was provided to *uniq_pid_midwife*.

## Exit Status

*uniq_pid_undertaker* will exit with a status of 0 (zero) if the monitored process has exited, with a status of 1 (one) if the monitored process has definitely not exited, with a status of 2 if it is uncertain whether the process has exited (this is generally due to a failure by the *uniq_pid_midwife*), or with any other value for program failure.

## See Also

*uniq_pid_midwife* (on page 792), *filelock_undertaker* (on page 772).

## Author

Condor Team, University of Wisconsin–Madison

## Copyright

Copyright © 1990-2007 Condor Team, Computer Sciences Department, University of Wisconsin-Madison, Madison, WI. All Rights Reserved. No use of the Condor Software Program is authorized without the express consent of the Condor Team. For more information contact: Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

U.S. Government Rights Restrictions: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of Commercial Computer Software-Restricted Rights at 48 CFR 52.227-19, as applicable, Condor Team, Attention: Professor Miron Livny, 7367 Computer Sciences, 1210 W. Dayton St., Madison, WI 53706-1685, (608) 262-0856 or miron@cs.wisc.edu.

See the *Condor Version 6.8.6 Manual* for additional notices.

# INDEX


$ENV
    in configuration file, 144
    in submit description file, 737
$RANDOM_CHOICE()
    in configuration, 144
    in submit description file, 737
$RANDOM_INTEGER()
    in configuration, 144, 469
_CONDOR_SCRATCH_DIR, 45

ACCOUNTANT_LOCAL_DOMAIN macro, 196, 573
accounting
    by group, 219
ActivationTimer macro, 250
activities and state figure, 239
activity
    of a machine, 238
    transitions, 239–250
    transitions summary, 248
ActivityTimer macro, 250
ADD_WINDOWS_FIREWALL_EXCEPTION macro, 168
administrator
    e-mail to, 118
administrator's manual, 104–367
AFS
    interaction with, 101
AfterHours macro, 255
agents
    condor_shadow, 17
ALIVE_INTERVAL macro, 170, 183
ALL_DEBUG macro, 154
ALLOW_* macro, 486
ALLOW_* macros macro, 280
ALLOW_ADMIN_COMMANDS macro, 168
ALLOW_ADMINISTRATOR macro, 278
ALLOW_CONFIG macro, 278
ALLOW_DAEMON macro, 495
ALLOW_FORCE_RM macro, 506
ALLOW_NEGOTIATOR macro, 278
ALLOW_OWNER macro, 278
ALLOW_READ macro, 278
ALLOW_REMOTE_SUBMIT macro, 184
ALLOW_WRITE macro, 278
ALTERNATE_STARTER_1 macro, 170
API
    Command line, 403
    Condor GAHP, 404
    DRMAA, 402
    Perl module, 404
    Web Service, 396
APPEND_PREF_STANDARD macro, 191
APPEND_PREF_VANILLA macro, 191
APPEND_RANK macro, 190
APPEND_RANK_STANDARD macro, 190, 191, 320
APPEND_RANK_VANILLA macro, 190, 191
APPEND_REQ_STANDARD macro, 190, 320
APPEND_REQ_VANILLA macro, 190
APPEND_REQUIREMENTS macro, 190
ARCH macro, 143, 340
Args
    job ClassAd attribute, 28, 226
argv[0]
    Condor use of, 103
AUTH_SSL_CLIENT_CADIR macro, 208, 271

AUTH_SSL_CLIENT_CAFILE macro, 208, 270
AUTH_SSL_CLIENT_CERTFILE macro, 208, 270
AUTH_SSL_CLIENT_KEYFILE macro, 208, 270
AUTH_SSL_SERVER_CADIR macro, 208, 270
AUTH_SSL_SERVER_CAFILE macro, 208, 270
AUTH_SSL_SERVER_CERTFILE macro, 208, 270
AUTH_SSL_SERVER_KEYFILE macro, 208, 270
authentication, 265–274
    GSI, 267
    Kerberos, 271
    Kerberos principal, 271
    SSL, 270
    using a file system, 273
    using a remote file system, 273
    Windows, 273
authorization
    for security, 278
available platforms, 5

Backfill, 359
    BOINC Configuration in Condor, 362
    BOINC Installation, 361
    BOINC Overview, 360
    Defining Condor policy, 360
    Overview, 359
BACKFILL_SYSTEM macro, 173, 360
BackgroundLoad macro, 250
batch system, 12
BIN macro, 146
BIND_ALL_INTERFACES macro, 157, 309, 525, 543
BOINC_Arguments macro, 363
BOINC_Environment macro, 363
BOINC_Error macro, 363
BOINC_Executable macro, 362, 363
BOINC_InitialDir macro, 362–364
BOINC_Output macro, 363
BOINC_Owner macro, 362, 363
BOINC_Universe macro, 363
C_GAHP_LOG macro, 201, 415
C_GAHP_WORKER_THREAD_LOG macro, 201
central manager, 104, 105
    installation issues, 110
certificate
    X.509, 267
CERTIFICATE_MAPFILE macro, 208, 274
checkpoint, 2, 3, 17, 380
    compression, 381
    implementation, 380
    library interface, 383
    periodic, 3, 380, 469
    stand alone, 381
checkpoint image, 17
checkpoint server, 105
    configuration of, 317
    installation, 315–320
    multiple servers, 318
Chirp, 63
    Chirp.jar, 64
    ChirpClient, 64
    ChirpInputStream, 63
    ChirpOutputStream, 63
CKPT_SERVER_DEBUG macro, 317
CKPT_SERVER_DIR macro, 163, 317
CKPT_SERVER_HOST macro, 162, 301, 314, 318
CKPT_SERVER_LOG macro, 317
CkptArch
    job ClassAd attribute, 28, 226
CkptOpSys
    job ClassAd attribute, 28, 227
CLAIM_WORKLIFE macro, 170, 258, 540
ClassAd, 2, 4, 13, 368–379
    attributes, 13, 370
    expression functions, 370
    expression operators, 370, 377
    expression syntax, 369
    job, 13
    job attributes, 28, 226
    machine, 13
    machine attributes, 23, 222
    machine example, 14
    scope of evaluation, MY., 376
    scope of evaluation, TARGET., 376
ClassAd attribute

CurrentTime, 28, 226
rank, 23, 378
rank examples, 33
requirements, 23, 378
ClassAd functions, 370
ceiling(), 372
floor(), 372
ifThenElse(), 371
int(), 371
interval(), 375
isBoolean(), 371
isError(), 371
isInteger(), 371
isReal(), 371
isString(), 371
isUndefined(), 371
random(), 372
real(), 372
regexp(), 375
regexps(), 375
round(), 372
size(), 373
strcat(), 373
strcmp(), 373
stricmp(), 373
string(), 372
stringListAve(), 374
stringListIMember(), 374
stringListMax(), 374
stringListMember(), 374
stringListMin(), 374
stringListRegexpMember(), 375
stringListSize(), 374
stringListSum(), 374
substr(), 373
time(), 375
toLower(), 373
toUpper(), 373
ClassAd job attribute
AccountingGroup, 219
Args, 28, 226
CkptArch, 28, 226
CkptOpSys, 28, 227
ClusterId, 28, 227, 736
Cmd, 28, 227
CompletionDate, 28, 227
CumulativeSuspensionTime, 28, 227
CurrentHosts, 28, 227
EnteredCurrentStatus, 28, 227
ExecutableSize, 29, 227
ExitBySignal, 29, 227
ExitCode, 29, 227
ExitSignal, 29, 227
ExitStatus, 29, 227
HoldKillSig, 29, 228
HoldReason, 30, 228
HoldReasonCode, 29, 227
HoldReasonSubCode, 29, 228
ImageSize, 30, 228
JobLeaseDuration, 30, 228
JobPrio, 30, 228
JobStartDate, 30, 228
JobStatus, 30, 228
JobUniverse, 30, 228
LastCheckpointPlatform, 30, 229
LastCkptServer, 30, 229
LastCkptTime, 30, 229
LastMatchTime, 30, 229
LastRejMatchReason, 31, 229
LastRejMatchTime, 31, 229
LastSuspensionTime, 31, 229
LastVacateTime, 31, 229
LocalSysCpu, 31, 229
LocalUserCpu, 31, 230
MaxHosts, 31, 230
MaxJobRetirementTime, 31, 230
MinHosts, 31, 230
NiceUser, 31, 230
NumCkpts, 31, 230
NumGlobusSubmits, 31, 230
NumJobMatches, 31, 230
NumRestarts, 32, 230
NumSystemHolds, 32, 230
Owner, 32, 230
ProcId, 32, 230
QDate, 32, 230
ReleaseReason, 32, 230
RemoteIwd, 32, 230
RemoteSysCpu, 32, 230
RemoteUserCpu, 32, 230
RemoteWallClockTime, 32, 231
RemoveKillSig, 32, 231

StreamErr, 32, 231
StreamOut, 32, 231
TotalSuspensions, 33, 231
TransferErr, 33, 231
TransferExecutable, 33, 231
TransferIn, 33, 231
TransferOut, 33, 231
ClassAd machine attribute
Activity, 23, 222
Arch, 24, 222
AvailSince, 181
AvailTime, 181
AvailTimeEstimate, 181
CheckpointPlatform, 24, 223
ClockDay, 24, 223
ClockMin, 24, 223
CondorLoadAvg, 24, 223
ConsoleIdle, 24, 223
Cpus, 24, 223
CurrentRank, 24, 223
Disk, 24, 223
EnteredCurrentActivity, 25, 223
FileSystemDomain, 25, 223
KeyboardIdle, 25, 223
KFlops, 25, 223
LastAvailInterval, 181
LastHeardFrom, 25, 223
LoadAvg, 25, 223
Machine, 25, 224
MaxJobRetirementTime, 26, 224
Memory, 25, 224
Mips, 25, 224
MyType, 25, 224
Name, 25, 224
OpSys, 25, 224
Requirements, 26, 224
StartdIpAddr, 26, 224
State, 26, 224
TargetType, 26, 225
UidDomain, 26, 225
VirtualMachineID, 26, 225
VirtualMemory, 27, 225
ClassAd machine attribute (in Claimed State)
ClientMachine, 27, 225
PreemptingOwner, 27, 225
PreemptingRank, 27, 226
PreemptingUser, 27, 225
RemoteOwner, 27, 225
RemoteUser, 27, 225
TotalClaimRunTime, 27, 226
TotalClaimSuspendTime, 27, 226
TotalJobRunTime, 27, 226
TotalJobSuspendTime, 27, 226
ClassAd machine attribute (when running)
JobId, 28, 226
JobStart, 28, 226
LastPeriodicCheckpoint, 28, 226
`CLASSAD_LIFETIME` macro, 192
cleanup_release, 596
`CLIENT_TIMEOUT` macro, 193
clipped platform
availability, 5
definition of, 5
clock skew, 478
cluster
definition, 28, 227
`Cluster` macro, 528, 736
ClusterId
job ClassAd attribute, 28, 227, 736
`CM_IP_ADDR` macro, 150
Cmd
job ClassAd attribute, 28, 227
COD
attributes, 386
ClusterId, 388
ProcID, 388
authorizing users, 385
condor_cod tool, 389
condor_cod_activate command, 388, 392
condor_cod_deactivate command, 394
condor_cod_release command, 394
condor_cod_request command, 390
condor_cod_resume command, 393
condor_cod_suspend command, 393
defining an application, 385
defining applications
Job ID, 388
defining attributes by configuration, 388
introduction, 384
limitations, 395
managing claims, 389
optional attributes, 386

Args, 387
Env, 387
Err, 387
In, 386
JarFiles, 387
JobUniverse, 387
KillSig, 387
Out, 387
StarterUserLog, 387
StarterUserLogUseXML, 387
overview, 385
required attributes, 386
Cmd, 386
IWD, 386
Owner, 386
COD (Computing on Demand), 384–395
COLLECTOR_ADDRESS_FILE macro, 296
COLLECTOR_ADDRESS_FILE macro, 156
COLLECTOR_CLASS_HISTORY_SIZE macro, 195
COLLECTOR_DAEMON_HISTORY_SIZE macro, 195, 600, 748
COLLECTOR_DAEMON_STATS macro, 195
COLLECTOR_DEBUG macro, 196
COLLECTOR_HOST macro, 145, 296, 313, 479, 582, 676, 709
COLLECTOR_NAME macro, 193, 582
COLLECTOR_QUERY_WORKERS macro, 195, 530
COLLECTOR_REQUIREMENTS macro, 192
COLLECTOR_SOCKET_BUFSIZE macro, 194
COLLECTOR_SOCKET_CACHE_SIZE macro, 158, 194, 315
COLLECTOR_TCP_SOCKET_BUFSIZE macro, 194, 501
compilers
supported with condor_compile, 6
CompletionDate
job ClassAd attribute, 28, 227
COMPRESS_PERIODIC_CKPT macro, 188
COMPRESS_VACATE_CKPT macro, 188
Computing On Demand
Defining Applications
Job ID, 388
Optional attributes, 386
Required attributes, 386
Computing on Demand (see COD), 384
Condor
binaries, 461
configuration, 138
contact information, 9, 489
contributions, 7
default policy, 250
distribution, 461
downloading, 461
FAQ, 461–489
flocking, 412
Frequently Asked Questions, 461–489
getting, 461
limitations, under UNIX, 4
mailing lists, 9, 489
new versions, notification of, 489
overview, 1–4
Personal, 464
platforms available, 5
pool, 104
PVM applications, 65
resource allocation, 13
resource management, 2
shared functionality in daemons, 320
source code, 462
universe, 16
Unix administrator, 110
user manual, 12–103
Condor commands
condor_advertise, 598
condor_check_userlogs, 602
condor_checkpoint, 604
condor_chirp, 607
condor_cod, 610
condor_cold_start, 136, 613
condor_cold_stop, 136, 617
condor_compile, 59, 620
list of supported compilers, 6
condor_config_bind, 623
condor_config_val, 625
condor_configure, 629
condor_convert_history, 633
condor_dagman, 635
condor_fetchlog, 639
condor_findhost, 642

condor_glidein, 440, 645
condor_history, 653
condor_hold, 52, 656
condor_master, 659
condor_master_off, 661
condor_off, 663
condor_on, 666
condor_preen, 669
condor_prio, 52, 58, 671
condor_q, 16, 50, 53, 673
condor_qedit, 681
condor_reconfig, 683
condor_reconfig_schedd, 686
condor_release, 52, 688
condor_reschedule, 691
condor_restart, 694
condor_rm, 16, 51, 697
condor_run, 700
condor_stats, 704
condor_status, 14, 16, 23, 49, 50, 708
condor_store_cred, 714
condor_submit, 16, 20, 102, 716
condor_submit_dag, 742
condor_transfer_data, 746
condor_updates_stats, 748
condor_userprio, 58, 755
condor_vacate, 759
condor_vacate_job, 762
condor_version, 765
condor_wait, 767
really slow; why?, 479
stork_list_cred, 778
stork_q, 776
stork_rm, 780
stork_rm_cred, 782
stork_status, 786
stork_store_cred, 784
stork_submit, 788
Condor daemon
*condor_had*, 325
*condor_master* manual page, 659
command line arguments, 321
*condor_ckpt_server*, 107
*condor_collector*, 107
*condor_kbdd*, 107
*condor_master*, 106
*condor_negotiator*, 107
*condor_schedd*, 106
condor_shadow, 17, 102
*condor_shadow*, 107
*condor_startd*, 106, 222
*condor_starter*, 106
descriptions, 106
Condor GAHP, 414
Condor-C, 414–418
configuration, 414
job submission, 415
limitations, 418
Condor-G, 418–432
GASS, 419
GRAM, 419
GSI, 418
job arguments, 427
job submission, 420
limitations, 432
proxy, 420
X.509 certificate, 420
Condor-PVM, 65
CONDOR_ADMIN macro, 148
condor_advertise command, 598
condor_check_userlogs command, 602
condor_checkpoint command, 604
condor_chirp command, 607
condor_ckpt_server daemon, 107
condor_cod command, 610
condor_cold_start, 613
condor_cold_stop, 617
condor_collector, 314
condor_collector daemon, 107
*condor_compile*, 473
condor_compile command, 620
list of supported compilers, 6
CONDOR_CONFIG macro, 133
condor_config_bind command, 623
condor_config_val command, 625
*condor_configure*, 115
condor_configure command, 629
condor_convert_history command, 633
condor_dagman command, 635
CONDOR_DEVELOPERS macro, 10, 193
CONDOR_DEVELOPERS_COLLECTOR
macro, 10, 193

condor_fetchlog command, 639
condor_findhost command, 642
CONDOR_GAHP macro, 201, 415
condor_glidein command, 645
condor_history command, 653
condor_hold command, 656
CONDOR_HOST macro, 145, 300
CONDOR_IDS
environment variable, 111, 149
CONDOR_IDS macro, 111, 148, 290
*condor_init* script, 116
*condor_install* script, 116
CONDOR_JOB_POLL_INTERVAL macro, 200
condor_kbdd daemon, 107
condor_master command, 659
condor_master daemon, 106
condor_master_off command, 661
condor_negotiator daemon, 107
condor_off command, 663
condor_on command, 666
condor_preen command, 669
condor_prio command, 671
condor_q command, 673
condor_qedit command, 681
condor_reconfig command, 683
condor_reconfig_schedd command, 686
condor_release command, 688
condor_reschedule command, 691
condor_restart command, 694
condor_rm command, 697
condor_run command, 700
condor_schedd daemon, 106
receiving signal 25, 486
CONDOR_SERVER_PRINCIPAL macro, 271
condor_shadow, 17, 50
condor_shadow daemon, 107
CONDOR_SSH_KEYGEN macro, 74
CONDOR_SSHD macro, 74
condor_startd daemon, 106
*condor_startd* daemon, 222
condor_starter daemon, 106
condor_stats command, 704
condor_status command, 708
condor_store_cred command, 714
condor_submit command, 716
condor_submit_dag, 79
condor_submit_dag command, 742
CONDOR_SUPPORT_EMAIL macro, 148
condor_transfer_data command, 746
condor_updates_stats command, 748
condor_userprio command, 755
condor_vacate command, 759
condor_vacate_job command, 762
condor_version command, 765
CONDOR_VIEW_HOST macro, 145
CONDOR_VM, 45
condor_wait command, 767
CondorView
Client, 133
Client installation, 134
configuration, 345
Server, 344
use of *crontab* program, 135
configuration, 138
checkpoint server configuration variables, 162
Condor-wide configuration variables, 145
condor_collector configuration variables, 192
condor_gridmanager configuration variables, 199
condor_master configuration variables, 163
condor_negotiator configuration variables, 196
condor_preen configuration variables, 192
condor_schedd configuration variables, 182
condor_shadow configuration variables, 187
condor_shadow.pvm configuration variables, 188
condor_startd configuration variables, 168
condor_starter configuration variables, 188
condor_submit configuration variables, 190
daemon logging configuration variables,

151
DaemonCore configuration variables, 155
DAGMan configuration variables, 203
example, 234
for flocking, 412
for glidein, 441
grid and glidein configuration variables, 203
grid_monitor configuration variables, 202
high availability configuration variables, 208
IsOwner expression, 240
network-related configuration variables, 157
pre-defined macros, 142
Quill configuration variables, 212
RANK, 234
security configuration variables, 206
shared file system configuration variables, 159
SMP machines, 351
START expression, 222
startd policy, 221
configuration file
$ENV definition, 144
evaluation order, 138
macro definitions, 139
macros, 143
pre-defined macros, 142
subsystem names, 142
configuration files
location, 113, 120
configuration macro
<SUBSYS>_<LEVEL>_LOG, 155
<SUBSYS>_ADDRESS_FILE, 156, 295
<SUBSYS>_ARGS, 164
<SUBSYS>_ATTRS, 156
<SUBSYS>_DEBUG, 152
<SUBSYS>_ENABLE_SOAP_SSL, 214
<SUBSYS>_EXPRS, 156
<SUBSYS>_LOCK, 152
<SUBSYS>_LOG, 152
<SUBSYS>_SOAP_SSL_PORT, 214
<SUBSYS>_TIMEOUT_MULTIPLIER, 158
<SUBSYS>, 164
ACCOUNTANT_LOCAL_DOMAIN, 196, 573
ADD_WINDOWS_FIREWALL_EXCEPTION, 168
ALIVE_INTERVAL, 170, 183
ALLOW_* macros, 280
ALLOW_*, 486
ALLOW_ADMIN_COMMANDS, 168
ALLOW_DAEMON, 495
ALLOW_FORCE_RM, 506
ALLOW_REMOTE_SUBMIT, 184
ALL_DEBUG, 154
ALTERNATE_STARTER_1, 170
APPEND_PREF_STANDARD, 191
APPEND_PREF_VANILLA, 191
APPEND_RANK_STANDARD, 190, 191, 320
APPEND_RANK_VANILLA, 190, 191
APPEND_RANK, 190
APPEND_REQUIREMENTS, 190
APPEND_REQ_STANDARD, 190, 320
APPEND_REQ_VANILLA, 190
ARCH, 143, 340
AUTH_SSL_CLIENT_CADIR, 208, 271
AUTH_SSL_CLIENT_CAFILE, 208, 270
AUTH_SSL_CLIENT_CERTFILE, 208, 270
AUTH_SSL_CLIENT_KEYFILE, 208, 270
AUTH_SSL_SERVER_CADIR, 208, 270
AUTH_SSL_SERVER_CAFILE, 208, 270
AUTH_SSL_SERVER_CERTFILE, 208, 270
AUTH_SSL_SERVER_KEYFILE, 208, 270
ActivationTimer, 250
ActivityTimer, 250
AfterHours, 255
BACKFILL_SYSTEM, 173, 360
BIND_ALL_INTERFACES, 157, 309, 525, 543

BIN, 146
BOINC_Arguments, 363
BOINC_Environment, 363
BOINC_Error, 363
BOINC_Executable, 362, 363
BOINC_InitialDir, 362–364
BOINC_Output, 363
BOINC_Owner, 362, 363
BOINC_Universe, 363
BackgroundLoad, 250
CERTIFICATE_MAPFILE, 208, 274
CKPT_SERVER_DEBUG, 317
CKPT_SERVER_DIR, 163, 317
CKPT_SERVER_HOST, 162, 301, 314, 318
CKPT_SERVER_LOG, 317
CLAIM_WORKLIFE, 170, 258, 540
CLASSAD_LIFETIME, 192
CLIENT_TIMEOUT, 193
CM_IP_ADDR, 150
COLLECTOR_ADDRESS_FILE, 296
COLLECTOR_CLASS_HISTORY_SIZE, 195
COLLECTOR_DAEMON_HISTORY_SIZE, 195, 600, 748
COLLECTOR_DAEMON_STATS, 195
COLLECTOR_DEBUG, 196
COLLECTOR_HOST, 145, 296, 313, 479, 582, 676, 709
COLLECTOR_NAME, 193, 582
COLLECTOR_QUERY_WORKERS, 195, 530
COLLECTOR_REQUIREMENTS, 192
COLLECTOR_SOCKET_BUFSIZE, 194
COLLECTOR_SOCKET_CACHE_SIZE, 158, 194, 315
COLLECTOR_TCP_SOCKET_BUFSIZE, 194, 501
COMPRESS_PERIODIC_CKPT, 188
COMPRESS_VACATE_CKPT, 188
CONDOR_ADMIN, 148
CONDOR_CONFIG, 133
CONDOR_DEVELOPERS_COLLECTOR, 10, 193
CONDOR_DEVELOPERS, 10, 193
CONDOR_GAHP, 201, 415
CONDOR_HOST, 145, 300
CONDOR_IDS, 111, 148, 290
CONDOR_JOB_POLL_INTERVAL, 200
CONDOR_SERVER_PRINCIPAL, 271
CONDOR_SSHD, 74
CONDOR_SSH_KEYGEN, 74
CONDOR_SUPPORT_EMAIL, 148
CONDOR_VIEW_HOST, 145
CONSOLE_DEVICES, 121, 171, 341
CONTINUE, 169
COUNT_HYPERTHREAD_CPUS, 172
CPUBusyTime, 251
CPUBusy, 251
CPUIdle, 251
CPUIsBusy, 251
CREATE_CORE_FILES, 150
CREDD_CACHE_LOCALLY, 199, 448
CREDD_HOST, 199, 448, 498, 509
CRED_CHECK_INTERVAL, 215
CRED_INDEX_FILE, 215
CRED_STORE_DIR, 215
CRED_SUPER_USERS, 215
C_GAHP_LOG, 201, 415
C_GAHP_WORKER_THREAD_LOG, 201
Cluster, 528, 736
ContinueIdleTime, 250
DAEMONNAME_ENVIRONMENT, 164
DAEMON_LIST, 117, 163, 317, 341, 345, 500, 541, 659
DAGMAN_ABORT_DUPLICATES, 206
DAGMAN_ALLOW_EVENTS, 204
DAGMAN_CONDOR_RM_EXE, 205, 551
DAGMAN_CONDOR_SUBMIT_EXE, 205, 551
DAGMAN_DEBUG, 205
DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION, 204
DAGMAN_LOG_ON_NFS_IS_ERROR, 206
DAGMAN_MAX_JOBS_IDLE, 204
DAGMAN_MAX_JOBS_SUBMITTED, 204
DAGMAN_MAX_SUBMITS_PER_INTERVAL, 203, 586
DAGMAN_MAX_SUBMIT_ATTEMPTS, 203, 586

DAGMAN_MUNGE_NODE_NAMES, 94, 204
DAGMAN_PENDING_REPORT_INTERVAL, 206
DAGMAN_PROHIBIT_MULTI_JOBS, 205, 515
DAGMAN_RESET_RETRIES_UPON_RESCUE, 554
DAGMAN_RETRY_NODE_FIRST, 203, 536
DAGMAN_RETRY_SUBMIT_FIRST, 203
DAGMAN_STARTUP_CYCLE_DETECT, 203
DAGMAN_STORK_RM_EXE, 205, 551
DAGMAN_STORK_SUBMIT_EXE, 205, 551
DAGMAN_SUBMIT_DELAY, 203, 588
DC_DAEMON_LIST, 163
DEFAULT_CRED_EXPIRE_THRESHOLD, 215
DEFAULT_DOMAIN_NAME, 150, 301, 482, 520
DEFAULT_IO_BUFFER_BLOCK_SIZE, 191
DEFAULT_IO_BUFFER_SIZE, 191
DEFAULT_PRIO_FACTOR, 196, 573
DEFAULT_RANK_STANDARD, 191
DEFAULT_RANK_VANILLA, 191
DEFAULT_UNIVERSE, 190, 721
DELEGATE_JOB_GSI_CREDENTIALS, 207, 496, 519
DENY_*, 486
DEPLOYMENT_RECOMMENDED_DIRS, 137
DEPLOYMENT_RECOMMENDED_EXECS, 137
DEPLOYMENT_RELEASE_DIR, 137
DEPLOYMENT_REQUIRED_DIRS, 137
DEPLOYMENT_REQUIRED_EXECS, 137
DISCONNECTED_KEYBOARD_IDLE_BOOST, 174, 351
DedicatedScheduler, 173, 355
EMAIL_DOMAIN, 150
ENABLE_BACKFILL, 173, 360, 362
ENABLE_GRID_MONITOR, 202, 432, 522
ENABLE_HISTORY_ROTATION, 149, 506
ENABLE_PERSISTENT_CONFIG, 155, 524, 571
ENABLE_RUNTIME_CONFIG, 155, 524, 571
ENABLE_SOAP_SSL, 214
ENABLE_SOAP, 213
ENABLE_USERLOG_LOCKING, 152
ENABLE_WEB_SERVER, 213
ENCRYPT_EXECUTE_DIRECTORY, 208
ENV, 144
EVICT_BACKFILL, 173, 360, 523
EXECUTE_LOGIN_IS_DEDICATED, 160, 293, 518
EXECUTE, 25, 147, 223
EXEC_TRANSFER_ATTEMPTS, 188
FILESYSTEM_DOMAIN, 119, 143, 161, 300
FILE_LOCK_VIA_MUTEX, 152
FLOCK_COLLECTOR_HOSTS, 186, 412
FLOCK_FROM, 413
FLOCK_NEGOTIATOR_HOSTS, 185, 413
FLOCK_TO, 412
FS_REMOTE_DIR, 207, 273
FULL_HOSTNAME, 142
GAHP_ARGS, 201
GAHP, 201
GLIDEIN_SERVER_URLS, 203, 440
GLITE_LOCATION, 201, 434
GLOBUS_GATEKEEPER_TIMEOUT, 201
GRACEFUL_SHUTDOWN_TIMEOUT, 537
GRIDMANAGER_CHECKPROXY_INTERVAL, 199
GRIDMANAGER_CONNECT_FAILURE_RETRY_COUNT, 201, 551
GRIDMANAGER_CONTACT_SCHEDD_DELAY, 200
GRIDMANAGER_EMPTY_RESOURCE_DELAY, 200, 520

GRIDMANAGER_GAHP_CALL_TIMEOUT, 201
GRIDMANAGER_GLOBUS_COMMIT_TIMEOUT, 201, 586
GRIDMANAGER_JOB_PROBE_INTERVAL, 200
GRIDMANAGER_LOG, 199, 415
GRIDMANAGER_MAX_JOBMANAGERS_PER_RESOURCE, 200, 522
GRIDMANAGER_MAX_PENDING_REQUESTS, 201
GRIDMANAGER_MAX_PENDING_SUBMITS_PER_RESOURCE, 200
GRIDMANAGER_MAX_PENDING_SUBMITS, 200
GRIDMANAGER_MAX_PENDING_SUBMIT_PER_RESOURCE, 539
GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE, 200, 539
GRIDMANAGER_MINIMUM_PROXY_TIME, 199
GRIDMANAGER_RESOURCE_PROBE_DELAY, 200
GRIDMANAGER_RESOURCE_PROBE_INTERVAL, 200, 551
GRIDMAP, 207, 269
GRID_MONITOR_HEARTBEAT_TIMEOUT, 202, 589
GRID_MONITOR_NO_STATUS_TIMEOUT, 496
GRID_MONITOR_RETRY_DURATION, 202, 589
GRID_MONITOR, 202, 432
GROUP_AUTOREGROUP_<groupname>, 199
GROUP_AUTOREGROUP, 199, 221
GROUP_NAMES, 198, 221
GROUP_PRIO_FACTOR_<groupname>, 198, 221
GROUP_QUOTA_<groupname>, 198
GSI_DAEMON_CERT, 207, 268
GSI_DAEMON_DIRECTORY, 207, 268, 269
GSI_DAEMON_KEY, 207, 268
GSI_DAEMON_NAME, 207, 496
GSI_DAEMON_PROXY, 207, 268
GSI_DAEMON_TRUSTED_CA_DIR, 207, 268, 541
GT2_GAHP, 201
GT3_GAHP, 201
GT4_GAHP, 202
HAD_ARGS, 210
HAD_CONNECTION_TIMEOUT, 210
HAD_DEBUG, 211
HAD_LIST, 210
HAD_LOG, 211
HAD_UPDATE_INTERVAL, 211
HAD_USE_PRIMARY, 210
HAD_USE_REPLICATION, 211, 327
HAD, 211
HA_<SUBSYS>_LOCK_HOLD_TIME, 209
HA_<SUBSYS>_LOCK_URL, 209
HA_<SUBSYS>_POLL_PERIOD, 210
HA_LOCK_HOLD_TIME, 209
HA_LOCK_URL, 209
HA_POLL_PERIOD, 209
HIGHPORT, 157, 297, 530, 531, 536, 557, 578
HISTORY, 149, 542
HOLD_JOB_IF_CREDENTIAL_EXPIRES, 199
HOSTALLOW. . ., 155
HOSTALLOW_*, 486
HOSTALLOW_ADMINISTRATOR, 130, 312
HOSTALLOW_CONFIG, 312
HOSTALLOW_DAEMON, 495
HOSTALLOW_NEGOTIATOR_SCHEDD, 413
HOSTALLOW_NEGOTIATOR, 313, 540
HOSTALLOW_READ, 129, 287
HOSTALLOW_WRITE, 110, 129, 133, 441, 511, 586
HOSTALLOW, 155
HOSTDENY_*, 486
HOSTDENY, 155
HOSTNAME, 142
HighLoad, 250
IGNORE_NFS_LOCK_ERRORS, 162
INCLUDE, 146, 564
INVALID_LOG_FILES, 669

IN_HIGHPORT, 158, 297, 530, 531
IN_LOWPORT, 157, 297, 530, 531
IP_ADDRESS, 142
IS_OWNER, 169
IS_VALID_CHECKPOINT_PLATFORM, 169
JAVA_CLASSPATH_ARGUMENT, 181
JAVA_CLASSPATH_DEFAULT, 182
JAVA_CLASSPATH_SEPARATOR, 182
JAVA_EXTRA_ARGUMENTS, 182, 366, 557
JAVA_MAXHEAP_ARGUMENT, 181, 366
JAVA, 181, 365, 426, 427
JOB_INHERITS_STARTER_ENVIRONMENT, 190, 501
JOB_IS_FINISHED_INTERVAL, 183
JOB_PROXY_OVERRIDE_FILE, 508, 520
JOB_RENICE_INCREMENT, 188, 232
JOB_START_COUNT, 183, 564
JOB_START_DELAY, 183, 564, 574
KBDD, 117
KEEP_POOL_HISTORY, 194, 345, 521
KERBEROS_MAP_FILE, 271
KILLING_TIMEOUT, 246, 249
KILL, 169, 259
KeyboardBusy, 251
LD_LIBRARY_PATH, 537
LIBEXEC, 146, 187, 564
LIB, 146
LOCAL_CONFIG_DIR, 148, 518, 524
LOCAL_CONFIG_FILE, 113, 142, 147, 340, 342, 469, 555
LOCAL_DIR, 111, 114, 146
LOCAL_ROOT_CONFIG_FILE, 113, 140, 142, 148
LOCK, 112, 116, 149
LOG_ON_NFS_IS_ERROR, 191
LOG, 146, 150, 171, 322
LOWPORT, 157, 297, 530, 531, 536, 557, 578
LSF_GAHP, 202, 434
LastCkpt, 250
MAIL, 148, 341, 591
MASTER_<SUBSYS>_CONTROLLER, 210
MASTER_<name>_BACKOFF_CEILING, 166
MASTER_<name>_BACKOFF_CONSTANT, 165
MASTER_<name>_BACKOFF_FACTOR, 166
MASTER_<name>_RECOVER_FACTOR, 166
MASTER_ADDRESS_FILE, 168
MASTER_ATTRS, 168
MASTER_BACKOFF_CEILING, 166
MASTER_BACKOFF_CONSTANT, 165, 536
MASTER_BACKOFF_FACTOR, 166
MASTER_CHECK_INTERVAL, 192
MASTER_CHECK_NEW_EXEC_INTERVAL, 165
MASTER_DEBUG, 168
MASTER_HAD_BACKOFF_CONSTANT, 327
MASTER_HA_LIST, 208, 324
MASTER_INSTANCE_LOCK, 168
MASTER_NAME, 167, 659
MASTER_NEW_BINARY_DELAY, 165
MASTER_RECOVER_FACTOR, 166
MASTER_UPDATE_INTERVAL, 165
MATCH_TIMEOUT, 237, 243, 248
MAXJOBRETIREMENTTIME, 170, 541
MAX_<SUBSYS>_<LEVEL>_LOG, 155
MAX_<SUBSYS>_LOG, 152, 155, 508
MAX_ACCOUNTANT_DATABASE_SIZE, 196
MAX_CKPT_SERVER_LOG, 317
MAX_CLAIM_ALIVES_MISSED, 170, 183
MAX_C_GAHP_LOG, 201
MAX_DAGMAN_LOG, 90, 205
MAX_DISCARDED_RUN_TIME, 163, 316
MAX_HAD_LOG, 211
MAX_HISTORY_LOG, 149
MAX_HISTORY_ROTATIONS, 149, 506
MAX_JOBS_RUNNING, 50, 183, 297
MAX_JOB_QUEUE_LOG_ROTATIONS, 149, 529, 530
MAX_NUM_CPUS, 172, 525

MAX_REPLICATION_LOG, 212
MAX_SCHEDD_LOG, 289
MAX_SHADOW_EXCEPTIONS, 183
MAX_SOAP_TRANSACTION_DURATION, 543
MAX_TRANSFER_LIFETIME, 211
MAX_VIRTUAL_MACHINE_TYPES, 175
MEMORY, 172
MPI_CONDOR_RSH_PATH, 187, 564
MYPROXY_GET_DELEGATION, 213, 431
MachineBusy, 251, 255
MaxSuspendTime, 251
MaxVacateTime, 251
NEGOTIATE_ALL_JOBS_IN_CLUSTER, 186, 219
NEGOTIATOR_ADDRESS_FILE, 295
NEGOTIATOR_CONSIDER_PREEMPTION, 199, 258
NEGOTIATOR_CYCLE_DELAY, 196, 559
NEGOTIATOR_DEBUG, 198
NEGOTIATOR_DISCOUNT_SUSPENDED_RESOURCES, 196
NEGOTIATOR_HOST, 145, 560, 582
NEGOTIATOR_INTERVAL, 196
NEGOTIATOR_MATCHLIST_CACHING, 507, 542
NEGOTIATOR_MAX_TIME_PER_SUBMITTER, 198
NEGOTIATOR_POST_JOB_RANK, 197, 568, 569
NEGOTIATOR_PRE_JOB_RANK, 197, 568
NEGOTIATOR_SOCKET_CACHE_SIZE, 197, 297
NEGOTIATOR_TIMEOUT, 196
NEGOTIATOR_USE_NONBLOCKING_STARTD_CONTACT, 159, 515
NETWORK_INTERFACE, 157, 299, 301, 543
NETWORK_MAX_PENDING_CONNECTS, 151
NET_REMAP_ENABLE, 159
NET_REMAP_INAGENT, 159
NET_REMAP_ROUTE, 159
NET_REMAP_SERVICE, 159
NICE_USER_PRIO_FACTOR, 196, 216
NODE, 72, 76
NONBLOCKING_COLLECTOR_UPDATE, 158, 515
NORDUGRID_GAHP, 202
NO_DNS, 150, 520
NUM_CPUS, 172
NUM_VIRTUAL_MACHINES_TYPE_<N>, 175, 350
NUM_VIRTUAL_MACHINES, 175, 350
Node, 736
NonCondorLoadAvg, 250
OBITUARY_LOG_LENGTH, 165
OPSYS, 143, 340, 561, 563
OUT_HIGHPORT, 158, 297, 530, 531
OUT_LOWPORT, 158, 297, 530, 531
PASSWD_CACHE_REFRESH, 151
PBS_GAHP, 202, 434
PERIODIC_CHECKPOINT, 380, 469
PERIODIC_EXPR_INTERVAL, 186, 545
PERIODIC_MEMORY_SYNC, 188
PERSISTENT_CONFIG_DIR, 155, 524, 571
PID, 143
POLLING_INTERVAL, 170, 244
POOL_HISTORY_DIR, 194, 345, 582
POOL_HISTORY_MAX_STORAGE, 194, 345
POOL_HISTORY_SAMPLING_INTERVAL, 194
PPID, 143
PREEMPTION_RANK, 198, 569
PREEMPTION_REQUIREMENTS, 59, 197, 199, 217, 676
PREEMPT, 169, 259
PREEN_ADMIN, 192, 669
PREEN_ARGS, 164
PREEN_INTERVAL, 165
PREEN, 164
PRIORITY_HALFLIFE, 58, 196, 216, 217
PUBLISH_OBITUARIES, 165
PVMD, 188

PVMGS, 188
ParallelSchedulingGroup, 187, 357–359
Process, 528, 736
QUERY_TIMEOUT, 193
QUEUE_ALL_USERS_TRUSTED, 184, 514
QUEUE_CLEAN_INTERVAL, 184
QUEUE_SUPER_USERS, 185
QUILL_ADDRESS_FILE, 213, 334
QUILL_ARGS, 212
QUILL_DB_IP_ADDR, 212, 333
QUILL_DB_NAME, 212, 333
QUILL_DB_QUERY_PASSWORD, 213, 334
QUILL_ENABLED, 212
QUILL_HISTORY_CLEANING_INTERVAL, 212, 333
QUILL_HISTORY_DURATION, 213, 333
QUILL_IS_REMOTELY_QUERYABLE, 213, 334
QUILL_LOG, 212
QUILL_MANAGE_VACUUM, 213, 333
QUILL_NAME, 212, 333
QUILL_POLLING_PERIOD, 212, 333
QUILL_SHOULD_REINDEX, 499
QUILL, 212
Q_QUERY_TIMEOUT, 151
RANDOM_CHOICE(), 144
RANDOM_INTEGER(), 144, 469
RANK_FACTOR, 357
RANK, 169, 357
RELEASE_DIR, 113, 146, 341
REMOTE_PRIO_FACTOR, 196, 216
REPLICATION_ARGS, 211
REPLICATION_DEBUG, 212
REPLICATION_INTERVAL, 211
REPLICATION_LIST, 211
REPLICATION_LOG, 212
REPLICATION, 212
REQUEST_CLAIM_TIMEOUT, 183
REQUIRE_LOCAL_CONFIG_FILE, 148, 590
RESERVED_DISK, 25, 149, 223
RESERVED_MEMORY, 172
RESERVED_SWAP, 149, 472
RESERVE_AFS_CACHE, 161
RUNBENCHMARKS, 173
Requirements, 182, 538
SBIN, 146
SCHEDD_ADDRESS_FILE, 185
SCHEDD_ASSUME_NEGOTIATOR_GONE, 186, 543
SCHEDD_ATTRS, 185
SCHEDD_BACKUP_SPOOL, 187
SCHEDD_DEBUG, 185, 554
SCHEDD_EXECUTE, 185
SCHEDD_HOST, 145
SCHEDD_INTERVAL, 183, 574
SCHEDD_LOCK, 185
SCHEDD_LOG, 472
SCHEDD_NAME, 167, 185
SCHEDD_PREEMPTION_RANK, 187, 358
SCHEDD_PREEMPTION_REQUIREMENTS, 187, 358
SCHEDD_ROUND_ATTR_<xxxx>, 186
SCHEDD_ROUND_ATTR, 504
SCHED_UNIV_RENICE_INCREMENT, 184
SECONDARY_COLLECTOR_LIST, 168
SEC_*_AUTHENTICATION_METHODS, 206
SEC_*_AUTHENTICATION, 206
SEC_*_CRYPTO_METHODS, 206
SEC_*_ENCRYPTION, 206
SEC_*_INTEGRITY, 206
SEC_*_NEGOTIATION, 206
SEC_CLAIMTOBE_INCLUDE_DOMAIN, 510
SEC_DEBUG_PRINT_KEYS, 586
SEC_DEFAULT_AUTHENTICATION_METHODS, 314
SEC_DEFAULT_SESSION_DURATION, 207
SEC_PASSWORD_FILE, 208, 272, 521
SEC_TCP_SESSION_TIMEOUT, 208
SETTABLE_ATTRS. . ., 156
SETTABLE_ATTRS_PERMISSION-LEVEL, 289
SETTABLE_ATTRS, 156, 288

SHADOW_ALLOW_UNSAFE_REMOTE_EXEC, 558
SHADOW_DEBUG, 187
SHADOW_LOCK, 187
SHADOW_LOG, 54
SHADOW_PVM, 182
SHADOW_QUEUE_UPDATE_INTERVAL, 187
SHADOW_RENICE_INCREMENT, 184
SHADOW_SIZE_ESTIMATE, 184
SHADOW, 182
SHELL, 702
SHUTDOWN_FAST_TIMEOUT, 165
SHUTDOWN_GRACEFUL_TIMEOUT, 156
SIGNIFICANT_ATTRIBUTES, 219, 534, 535
SLOW_CKPT_SPEED, 188
SOAP_LEAVE_IN_QUEUE, 213, 398
SOAP_SSL_CA_DIR, 214
SOAP_SSL_CA_FILE, 214
SOAP_SSL_DH_FILE, 214
SOAP_SSL_SERVER_KEYFILE_PASSWORD, 214
SOAP_SSL_SERVER_KEYFILE, 214
SOFT_UID_DOMAIN, 119, 160, 290
SPOOL, 147, 575, 582
STARTD_ADDRESS_FILE, 171
STARTD_AD_REEVAL_EXPR, 535
STARTD_ATTRS, 171, 289
STARTD_AVAIL_CONFIDENCE, 181
STARTD_CLAIM_ID_FILE, 171, 551
STARTD_COMPUTE_AVAIL_STATS, 181
STARTD_CRON_<ModuleName>_ARGS, 178
STARTD_CRON_<ModuleName>_CWD, 178
STARTD_CRON_<ModuleName>_ENV, 178
STARTD_CRON_<ModuleName>_EXECUTABLE, 177
STARTD_CRON_<ModuleName>_KILL, 178
STARTD_CRON_<ModuleName>_MODE, 177
STARTD_CRON_<ModuleName>_OPTIONS, 178
STARTD_CRON_<ModuleName>_PERIOD, 177
STARTD_CRON_<ModuleName>_PREFIX, 177
STARTD_CRON_<ModuleName>_RECONFIG, 177
STARTD_CRON_AUTOPUBLISH, 176, 539
STARTD_CRON_CONFIG_VAL, 176
STARTD_CRON_JOBLIST, 176, 534
STARTD_CRON_JOBS, 179, 534
STARTD_CRON_NAME, 175
STARTD_CRON, 581
STARTD_DEBUG, 171, 289
STARTD_EXPRS, 319, 359
STARTD_HAS_BAD_UTMP, 171
STARTD_JOB_EXPRS, 171
STARTD_MAX_AVAIL_PERIOD_SAMPLES, 181
STARTD_NAME, 172
STARTD_NOCLAIM_SHUTDOWN, 173
STARTD_RESOURCE_PREFIX, 174, 525
STARTD_SHOULD_WRITE_CLAIM_ID_FILE, 171, 551
STARTD_VM_EXPRS, 174, 258, 568
STARTER_ALLOW_RUNAS_OWNER, 448
STARTER_CHOOSES_CKPT_SERVER, 163, 318
STARTER_DEBUG, 189
STARTER_INITIAL_UPDATE_INTERVAL, 588
STARTER_JOB_ENVIRONMENT, 190, 568
STARTER_LOCAL_LOGGING, 189
STARTER_UPDATE_INTERVAL, 189
STARTER, 170
START_BACKFILL, 173, 360, 362, 523
START_DAEMONS, 165
START_LOCAL_UNIVERSE, 182, 538
START_MASTER, 165
START_SCHEDULER_UNIVERSE, 182, 538

START, 169, 173, 356
STATE_FILE, 211
STORK_ENVIRONMENT, 537
STORK_MAXDELAY_INMINUTES, 214
STORK_MAX_NUM_JOBS, 214
STORK_MAX_RETRY, 98, 214
STORK_MODULE_DIR, 215
STORK_TMP_CRED_DIR, 214
SUBMIT_DEBUG, 154
SUBMIT_EXPRS, 191, 523, 528
SUBMIT_MAX_PROCS_IN_CLUSTER, 192, 514
SUBMIT_SEND_RESCHEDULE, 191, 588
SUBMIT_SKIP_FILECHECK, 191
SUBSYSTEM, 142
SUSPEND, 169, 475
SYSAPI_GET_LOADAVG, 151
SYSTEM_PERIODIC_HOLD, 539
SYSTEM_PERIODIC_RELEASE, 539
SYSTEM_PERIODIC_REMOVE, 539
StartIdleTime, 250
StateTimer, 250
TCP_UPDATE_COLLECTORS, 158
TILDE, 142
TOOL_DEBUG, 154
TOUCH_LOG_INTERVAL, 152, 520
TRUNC_<SUBSYS>_<LEVEL>_LOG_ON_OPEN, 155
TRUNC_<SUBSYS>_LOG_ON_OPEN, 152, 155
TRUST_UID_DOMAIN, 160, 533
UID_DOMAIN, 119, 144, 159, 290, 291, 300, 720
UNAME_ARCH, 143
UNAME_OPSYS, 143
UNICORE_GAHP, 202
UPDATE_COLLECTOR_WITH_TCP, 158, 194, 315, 581
UPDATE_INTERVAL, 170, 176, 241, 537
USERNAME, 143
USER_JOB_WRAPPER, 189, 483, 581
USE_AFS, 162
USE_CKPT_SERVER, 163, 317, 318
USE_NFS, 161
USE_VISIBLE_DESKTOP, 190, 449
VACATE, 259
VALID_COD_USERS, 385
VALID_LOG_FILES, 192, 322, 669
VALID_SPOOL_FILES, 192, 669
VIRTUAL_MACHINES_CONNECTED_TO_CONSOLE, 174, 351
VIRTUAL_MACHINES_CONNECTED_TO_KEYBOARD, 174, 351
VIRTUAL_MACHINE_TYPE_<N>, 175, 349
VMP_HOST_MACHINE, 347
VMP_VM_LIST, 347
VMx_USER, 160, 293
WALL_CLOCK_CKPT_INTERVAL, 184
WANT_SUSPEND, 169
WANT_VACATE, 169, 259
WEB_ROOT_DIR, 213
WINDOWS_FIREWALL_FAILURE_RETRY, 168
WorkHours, 255
ALLOW_ADMINISTRATOR, 278
ALLOW_CONFIG, 278
ALLOW_NEGOTIATOR, 278
ALLOW_OWNER, 278
ALLOW_READ, 278
ALLOW_WRITE, 278
COLLECTOR_ADDRESS_FILE, 156
DENY_ADMINISTRATOR, 278
DENY_CONFIG, 278
DENY_NEGOTIATOR, 278
DENY_OWNER, 278
DENY_READ, 278
DENY_WRITE, 278
MAXJOBRETIREMENTTIME, 257
NEGOTIATOR_ADDRESS_FILE, 156
SEC_ADMINISTRATOR_AUTHENTICATION_METHODS, 266
SEC_ADMINISTRATOR_AUTHENTICATION, 265
SEC_ADMINISTRATOR_CRYPTO_METHODS, 276
SEC_ADMINISTRATOR_ENCRYPTION, 275
SEC_ADMINISTRATOR_INTEGRITY, 277

SEC_CLIENT_AUTHENTICATION_METHODS, 266
SEC_CLIENT_AUTHENTICATION, 265
SEC_CLIENT_CRYPTO_METHODS, 276
SEC_CLIENT_ENCRYPTION, 275
SEC_CLIENT_INTEGRITY, 276
SEC_CONFIG_AUTHENTICATION_METHODS, 266
SEC_CONFIG_AUTHENTICATION, 265
SEC_CONFIG_CRYPTO_METHODS, 276
SEC_CONFIG_ENCRYPTION, 275
SEC_CONFIG_INTEGRITY, 277
SEC_DAEMON_AUTHENTICATION_METHODS, 266
SEC_DAEMON_AUTHENTICATION, 265
SEC_DAEMON_CRYPTO_METHODS, 276
SEC_DAEMON_ENCRYPTION, 275
SEC_DEFAULT_AUTHENTICATION_METHODS, 266
SEC_DEFAULT_AUTHENTICATION, 265
SEC_DEFAULT_CRYPTO_METHODS, 276
SEC_DEFAULT_ENCRYPTION, 275
SEC_DEFAULT_INTEGRITY, 276, 277
SEC_NEGOTIATOR_AUTHENTICATION_METHODS, 266
SEC_NEGOTIATOR_AUTHENTICATION, 265
SEC_NEGOTIATOR_CRYPTO_METHODS, 276
SEC_NEGOTIATOR_ENCRYPTION, 275
SEC_NEGOTIATOR_INTEGRITY, 277
SEC_OWNER_AUTHENTICATION_METHODS, 266
SEC_OWNER_AUTHENTICATION, 265
SEC_OWNER_CRYPTO_METHODS, 276
SEC_OWNER_ENCRYPTION, 275
SEC_OWNER_INTEGRITY, 277
SEC_READ_AUTHENTICATION_METHODS, 266
SEC_READ_AUTHENTICATION, 265
SEC_READ_CRYPTO_METHODS, 276
SEC_READ_ENCRYPTION, 275
SEC_READ_INTEGRITY, 277
SEC_WRITE_AUTHENTICATION_METHODS, 266
SEC_WRITE_AUTHENTICATION, 265
SEC_WRITE_CRYPTO_METHODS, 276
SEC_WRITE_ENCRYPTION, 275
SEC_WRITE_INTEGRITY, 277
Console activity, 127
CONSOLE_DEVICES macro, 121, 171, 341
CONTINUE macro, 169
ContinueIdleTime macro, 250
contrib module
    CondorView client, 133
COUNT_HYPERTHREAD_CPUS macro, 172
CPU activity, 127
CPUBusy macro, 251
CPUBusyTime macro, 251
CPUIdle macro, 251
CPUIsBusy macro, 251
crashes, 486
CREATE_CORE_FILES macro, 150
CRED_CHECK_INTERVAL macro, 215
CRED_INDEX_FILE macro, 215
CRED_STORE_DIR macro, 215
CRED_SUPER_USERS macro, 215
CREDD_CACHE_LOCALLY macro, 199, 448
CREDD_HOST macro, 199, 448, 498, 509
crontab program, 135
CumulativeSuspensionTime
    job ClassAd attribute, 28, 227
current working directory, 294
CurrentHosts
    job ClassAd attribute, 28, 227
cwd
    of jobs, 294

daemon
    *condor_had*, 325
    *condor_ckpt_server*, 107
    *condor_collector*, 107
    *condor_kbdd*, 107

*condor_master*, 106
*condor_negotiator*, 107
*condor_schedd*, 106
*condor_shadow*, 107
*condor_startd*, 106, 221, 222
*condor_starter*, 106
running as root, 102
DAEMON_LIST macro, 117, 163, 317, 341, 345, 500, 541, 659
daemoncore, 320–323
command line arguments, 321
Unix signals, 321
DAEMONNAME_ENVIRONMENT macro, 164
daemons
descriptions, 106
DAGMan, 79–95
ABORT-DAG-ON, 86
DAGs within DAGs, 92
describing dependencies, 84
dot, 91
example submit description file, 88
File Paths in DAGs, 94
input file example, 80
Job Entry (names node of DAG), 81
job submission, 89
POST script, 82
PRE and POST scripts, 82
PRE script, 82
rescue DAG, 91
RETRY of failed nodes, 85
Single submission of multiple, independent DAGs, 93
submit description file with, 87
VARS (macro for submit description file), 85
visualizing DAGs, 91
DAGMAN_ABORT_DUPLICATES macro, 206
DAGMAN_ALLOW_EVENTS macro, 204
DAGMAN_CONDOR_RM_EXE macro, 205, 551
DAGMAN_CONDOR_SUBMIT_EXE macro, 205, 551
DAGMAN_DEBUG macro, 205
DAGMAN_IGNORE_DUPLICATE_JOB_EXECUTION macro, 204
DAGMAN_LOG_ON_NFS_IS_ERROR macro, 206
DAGMAN_MAX_JOBS_IDLE macro, 204
DAGMAN_MAX_JOBS_SUBMITTED macro, 204
DAGMAN_MAX_SUBMIT_ATTEMPTS macro, 203, 586
DAGMAN_MAX_SUBMITS_PER_INTERVAL macro, 203, 586
DAGMAN_MUNGE_NODE_NAMES macro, 94, 204
DAGMAN_PENDING_REPORT_INTERVAL macro, 206
DAGMAN_PROHIBIT_MULTI_JOBS macro, 205, 515
DAGMAN_RESET_RETRIES_UPON_RESCUE macro, 554
DAGMAN_RETRY_NODE_FIRST macro, 203, 536
DAGMAN_RETRY_SUBMIT_FIRST macro, 203
DAGMAN_STARTUP_CYCLE_DETECT macro, 203
DAGMAN_STORK_RM_EXE macro, 205, 551
DAGMAN_STORK_SUBMIT_EXE macro, 205, 551
DAGMAN_SUBMIT_DELAY macro, 203, 588
DC_DAEMON_LIST macro, 163
dedicated scheduling, 355
DedicatedScheduler macro, 173, 355
DEFAULT_CRED_EXPIRE_THRESHOLD macro, 215
DEFAULT_DOMAIN_NAME macro, 150, 301, 482, 520
DEFAULT_IO_BUFFER_BLOCK_SIZE macro, 191
DEFAULT_IO_BUFFER_SIZE macro, 191
DEFAULT_PRIO_FACTOR macro, 196, 573
DEFAULT_RANK_STANDARD macro, 191
DEFAULT_RANK_VANILLA macro, 191
DEFAULT_UNIVERSE macro, 190, 721
DELEGATE_JOB_GSI_CREDENTIALS macro, 207, 496, 519
DENY_* macro, 486
DENY_ADMINISTRATOR macro, 278
DENY_CONFIG macro, 278
DENY_NEGOTIATOR macro, 278
DENY_OWNER macro, 278

DENY_READ macro, 278
DENY_WRITE macro, 278
Deployment commands
cleanup_release, 596
condor_cold_start, 613
condor_cold_stop, 617
filelock_midwife, 770
filelock_undertaker, 772
install_release, 774
uniq_pid_midwife, 792
uniq_pid_undertaker, 794
deployment commands, 136
DEPLOYMENT_RECOMMENDED_DIRS macro, 137
DEPLOYMENT_RECOMMENDED_EXECS macro, 137
DEPLOYMENT_RELEASE_DIR macro, 137
DEPLOYMENT_REQUIRED_DIRS macro, 137
DEPLOYMENT_REQUIRED_EXECS macro, 137
directed acyclic graph (DAG), 79
Directed Acyclic Graph Manager (DAGMan), 79
DISCONNECTED_KEYBOARD_IDLE_BOOST macro, 174, 351
disk space requirement
execute directory, 111
log directory, 112
spool directory, 111
all versions, 114
Condor files, 112
distributed ownership
of machines, 2
Distributed Resource Management Application API (DRMAA), 402
dot, 91
download, 108
DRMAA (Distributed Resource Management Application API), 402
dynamic deployment, 136
configuration, 136
relevance to grid computing, 441

effective user priority (EUP), 216
EMAIL_DOMAIN macro, 150
ENABLE_BACKFILL macro, 173, 360, 362
ENABLE_GRID_MONITOR macro, 202, 432, 522
ENABLE_HISTORY_ROTATION macro, 149, 506
ENABLE_PERSISTENT_CONFIG macro, 155, 524, 571
ENABLE_RUNTIME_CONFIG macro, 155, 524, 571
ENABLE_SOAP macro, 213
ENABLE_SOAP_SSL macro, 214
ENABLE_USERLOG_LOCKING macro, 152
ENABLE_WEB_SERVER macro, 213
ENCRYPT_EXECUTE_DIRECTORY macro, 208
EnteredCurrentStatus
job ClassAd attribute, 28, 227
ENV macro, 144
environment variables, 45
_CONDOR_SCRATCH_DIR, 45
CONDOR_CONFIG, 485
CONDOR_IDS, 111, 149
CONDOR_VM, 45
copying current environment, 719
in submit description file, 737
setting, for a job, 718
X509_USER_PROXY, 45, 521, 553
EVICT_BACKFILL macro, 173, 360, 523
EXEC_TRANSFER_ATTEMPTS macro, 188
ExecutableSize
job ClassAd attribute, 29, 227
execute machine, 105
EXECUTE macro, 25, 147, 223
EXECUTE_LOGIN_IS_DEDICATED macro, 160, 293, 518
execution environment, 45
ExitBySignal
job ClassAd attribute, 29, 227
ExitCode
job ClassAd attribute, 29, 227
ExitSignal
job ClassAd attribute, 29, 227
ExitStatus
job ClassAd attribute, 29, 227
expression

IS_VALID_CHECKPOINT_PLATFORM, 233
IsOwner, 240
RANK, 234
START, 232

FAQ, 461–489
Condor on Windows machines, 479
installing Condor, 461
file
locking, 4, 18
memory-mapped, 4, 18
read only, 4, 18
submit description, 20
write only, 4, 18
file system
AFS, 101
NFS, 102
file transfer
large files (¿2Gbytes), 7
file transfer mechanism, 36, 722
missing files, 479
FILE_LOCK_VIA_MUTEX macro, 152
filelock_midwife, 770
filelock_undertaker, 772
FILESYSTEM_DOMAIN macro, 119, 143, 161, 300
FLOCK_COLLECTOR_HOSTS macro, 186, 412
FLOCK_FROM macro, 413
FLOCK_NEGOTIATOR_HOSTS macro, 185, 413
FLOCK_TO macro, 412
flocking, 412
Frequently Asked Questions, 461–489
FS_REMOTE_DIR macro, 207, 273
FULL_HOSTNAME macro, 142

GAHP (Grid ASCII Helper Protocol), 414
GAHP macro, 201
GAHP_ARGS macro, 201
GASS (Global Access to Secondary Storage), 419
GCB (Generic Connection Brokering), 301
broker, 303
Condor client configuration, 308
GCB routing table configuration, 309
GCB routing table syntax and examples, 310
inagent, 303
security implications, 312
GCB broker
configuration, 304
how to spawn the broker, 306
ports 65432 and 65430, 304
glidein, 440, 484
configuration, 441
GLIDEIN_SERVER_URLS macro, 203, 440
GLITE_LOCATION macro, 201, 434
Globus
gatekeeper errors, 484
GLOBUS_GATEKEEPER_TIMEOUT macro, 201
GRACEFUL_SHUTDOWN_TIMEOUT macro, 537
GRAM (Grid Resource Allocation and Management), 419
grid computing
Condor-C, 414
FAQs, 484
glidein, 440
Grid Monitor, 431
matchmaking, 434
submitting jobs to gt2, 420
submitting jobs to gt3, 425
submitting jobs to gt4, 426
submitting jobs to NorduGrid, 432
submitting jobs to PBS, 433
submitting jobs to Platform LSF, 434
submitting jobs to Unicore, 433
GRID_MONITOR macro, 202, 432
GRID_MONITOR_HEARTBEAT_TIMEOUT macro, 202, 589
GRID_MONITOR_NO_STATUS_TIMEOUT macro, 496
GRID_MONITOR_RETRY_DURATION macro, 202, 589
GRIDMANAGER_CHECKPROXY_INTERVAL macro, 199
GRIDMANAGER_CONNECT_FAILURE_RETRY_COUNT macro, 201, 551

GRIDMANAGER_CONTACT_SCHEDD_DELAY macro, 200
GRIDMANAGER_EMPTY_RESOURCE_DELAY macro, 200, 520
GRIDMANAGER_GAHP_CALL_TIMEOUT macro, 201
GRIDMANAGER_GLOBUS_COMMIT_TIMEOUT macro, 201, 586
GRIDMANAGER_JOB_PROBE_INTERVAL macro, 200
GRIDMANAGER_LOG macro, 199, 415
GRIDMANAGER_MAX_JOBMANAGERS_PER_RESOURCE macro, 200, 522
GRIDMANAGER_MAX_PENDING_REQUESTS macro, 201
GRIDMANAGER_MAX_PENDING_SUBMIT_PER_RESOURCE macro, 539
GRIDMANAGER_MAX_PENDING_SUBMITS macro, 200
GRIDMANAGER_MAX_PENDING_SUBMITS_PER_RESOURCE macro, 200
GRIDMANAGER_MAX_SUBMITTED_JOBS_PER_RESOURCE macro, 200, 539
GRIDMANAGER_MINIMUM_PROXY_TIME macro, 199
GRIDMANAGER_RESOURCE_PROBE_DELAY macro, 200
GRIDMANAGER_RESOURCE_PROBE_INTERVAL macro, 200, 551
GRIDMAP macro, 207, 269
GROUP_AUTOREGROUP macro, 199, 221
GROUP_AUTOREGROUP_<groupname> macro, 199
GROUP_NAMES macro, 198, 221
GROUP_PRIO_FACTOR_<groupname> macro, 198, 221
GROUP_QUOTA_<groupname> macro, 198
groups
  accounting, 219
  quotas, 220
GSI (Grid Security Infrastructure), 418
GSI_DAEMON_CERT macro, 207, 268
GSI_DAEMON_DIRECTORY macro, 207, 268, 269
GSI_DAEMON_KEY macro, 207, 268
GSI_DAEMON_NAME macro, 207, 496
GSI_DAEMON_PROXY macro, 207, 268
GSI_DAEMON_TRUSTED_CA_DIR macro, 207, 268, 541
GT2_GAHP macro, 201
GT3_GAHP macro, 201
GT4_GAHP macro, 202

HA_<SUBSYS>_LOCK_HOLD_TIME macro, 209
HA_<SUBSYS>_LOCK_URL macro, 209
HA_<SUBSYS>_POLL_PERIOD macro, 210
HA_LOCK_HOLD_TIME macro, 209
HA_LOCK_URL macro, 209
HA_POLL_PERIOD macro, 209
HAD macro, 211
HAD_ARGS macro, 210
HAD_CONNECTION_TIMEOUT macro, 210
HAD_DEBUG macro, 211
HAD_LIST macro, 210
HAD_LOG macro, 211
HAD_UPDATE_INTERVAL macro, 211
HAD_USE_PRIMARY macro, 210
HAD_USE_REPLICATION macro, 211, 327
heterogeneous pool
  submitting a job to, 45
High Availability, 323
  of central manager, 324
  of job queue, 323
  sample configuration, 327
High-Performance Computing (HPC), 1
High-Throughput Computing (HTC), 1
HighLoad macro, 250
HIGHPORT macro, 157, 297, 530, 531, 536, 557, 578
HISTORY macro, 149, 542
HOLD_JOB_IF_CREDENTIAL_EXPIRES macro, 199
HoldKillSig
  job ClassAd attribute, 29, 228
HoldReason
  job ClassAd attribute, 30, 228
HoldReasonCode
  job ClassAd attribute, 29, 227
HoldReasonSubCode
  job ClassAd attribute, 29, 228
host certificate, 268

HOSTALLOW macro, 155
HOSTALLOW... macro, 155
HOSTALLOW_* macro, 486
HOSTALLOW_ADMINISTRATOR macro, 130, 312
HOSTALLOW_CONFIG macro, 312
HOSTALLOW_DAEMON macro, 495
HOSTALLOW_NEGOTIATOR macro, 313, 540
HOSTALLOW_NEGOTIATOR_SCHEDD macro, 413
HOSTALLOW_READ macro, 129, 287
HOSTALLOW_WRITE macro, 110, 129, 133, 441, 511, 586
HOSTDENY macro, 155
HOSTDENY_* macro, 486
HOSTNAME macro, 142
HPC (High-Performance Computing), 1
HTC (High-Throughput Computing), 1

IGNORE_NFS_LOCK_ERRORS macro, 162
ImageSize, 477
  job ClassAd attribute, 30, 228
IN_HIGHPORT macro, 158, 297, 530, 531
IN_LOWPORT macro, 157, 297, 530, 531
INCLUDE macro, 146, 564
install_release, 774
installation
  checkpoint server, 315
  CondorView Client, 134
  download, 108
  Java, 365
  running as root, 110
  scripts, 116
  using Linux RPMs, 132
  Windows, 123–132
  with *condor_configure*, 115
INVALID_LOG_FILES macro, 669
IP_ADDRESS macro, 142
IS_OWNER macro, 169
IS_VALID_CHECKPOINT_PLATFORM macro, 169
IsOwner expression, 240

Java, 17, 60, 365
JAVA macro, 181, 365, 426, 427
Java Virtual Machine, 17, 60, 365
JAVA_CLASSPATH_ARGUMENT macro, 181
JAVA_CLASSPATH_DEFAULT macro, 182
JAVA_CLASSPATH_SEPARATOR macro, 182
JAVA_EXTRA_ARGUMENTS macro, 182, 366, 557
JAVA_MAXHEAP_ARGUMENT macro, 181, 366
job
  analysis, 53
  batch ready, 15
  completion, 57
  credential error on Windows, 481
  dependencies within, 79
  exiting with signal 9 $Unix$, 471
  exiting with status 128 $NT$, 482
  heterogeneous submit, 45
  image size, 477
  lease, 103
  log events, 54
  multiple data sets, 2
  not running, 53
  not running, why?, 471
  preparation, 15
  priority, 52, 58
  state, 30, 50, 52, 228
  submission using a shared file system, 35
  submission without a shared file system, 36
  submitting, 20
  universe, 30, 228
job ID
  cluster identifier, 28, 227, 736
  process identifier, 32, 230
  use in *condor_wait*, 767
Job monitor, 99
JOB_INHERITS_STARTER_ENVIRONMENT macro, 190, 501
JOB_IS_FINISHED_INTERVAL macro, 183
JOB_PROXY_OVERRIDE_FILE macro, 508, 520
JOB_RENICE_INCREMENT macro, 188, 232
JOB_START_COUNT macro, 183, 564

JOB_START_DELAY macro, 183, 564, 574
JobLeaseDuration
job ClassAd attribute, 30, 228
JobPrio
job ClassAd attribute, 30, 228
JobStartDate
job ClassAd attribute, 30, 228
JobStatus
job ClassAd attribute, 30, 228
JobUniverse
job ClassAd attribute, 30, 228
JVM, 17, 60, 365

KBDD macro, 117
KEEP_POOL_HISTORY macro, 194, 345, 521
Kerberos authentication, 271
KERBEROS_MAP_FILE macro, 271
KeyboardBusy macro, 251
KILL macro, 169, 259
KILLING_TIMEOUT macro, 246, 249

large file transfer support, 7
LastCheckpointPlatform
job ClassAd attribute, 30, 229
LastCkpt macro, 250
LastCkptServer
job ClassAd attribute, 30, 229
LastCkptTime
job ClassAd attribute, 30, 229
LastMatchTime
job ClassAd attribute, 30, 229
LastRejMatchReason
job ClassAd attribute, 31, 229
LastRejMatchTime
job ClassAd attribute, 31, 229
LastSuspensionTime
job ClassAd attribute, 31, 229
LastVacateTime
job ClassAd attribute, 31, 229
LD_LIBRARY_PATH macro, 537
LIB macro, 146
LIBEXEC macro, 146, 187, 564
linking
dynamic, 5, 18
static, 5, 18
Linux
keyboard and mouse activity, 444, 476
local universe, 20
LOCAL_CONFIG_DIR macro, 148, 518, 524
LOCAL_CONFIG_FILE macro, 113, 142, 147, 340, 342, 469, 555
LOCAL_DIR macro, 111, 114, 146
LOCAL_ROOT_CONFIG_FILE macro, 113, 140, 142, 148
LocalSysCpu
job ClassAd attribute, 31, 229
LocalUserCpu
job ClassAd attribute, 31, 230
LOCK macro, 112, 116, 149
log files
event descriptions, 54
LOG macro, 146, 150, 171, 322
LOG_ON_NFS_IS_ERROR macro, 191
LOWPORT macro, 157, 297, 530, 531, 536, 557, 578
LSF, 434
LSF_GAHP macro, 202, 434

machine
central manager, 105
checkpoint server, 105
execute, 105
owner, 104
submit, 105
machine activity, 238
Backfill, 239
Benchmarking, 238
Busy, 238
Idle, 238
Killing, 239
Retiring, 239
Suspended, 238
transitions, 239–250
transitions summary, 248
Unclaimed, 238
Vacating, 239
machine ClassAd, 14
machine state, 235
Backfill, 236
Claimed, 235
Matched, 235

Owner, 235
Preempting, 235
transitions, 239–250
transitions summary, 248
Unclaimed, 235
machine state and activities figure, 239
MachineBusy macro, 251, 255
macro
in configuration file, 139
in submit description file, 736
predefined, 71, 76
subsystem names, 142
MAIL macro, 148, 341, 591
mailing lists, 9, 489
master-worker paradigm, 66
MASTER_<name>_BACKOFF_CEILING macro, 166
MASTER_<name>_BACKOFF_CONSTANT macro, 165
MASTER_<name>_BACKOFF_FACTOR macro, 166
MASTER_<name>_RECOVER_FACTOR macro, 166
MASTER_<SUBSYS>_CONTROLLER macro, 210
MASTER_ADDRESS_FILE macro, 168
MASTER_ATTRS macro, 168
MASTER_BACKOFF_CEILING macro, 166
MASTER_BACKOFF_CONSTANT macro, 165, 536
MASTER_BACKOFF_FACTOR macro, 166
MASTER_CHECK_INTERVAL macro, 192
MASTER_CHECK_NEW_EXEC_INTERVAL macro, 165
MASTER_DEBUG macro, 168
MASTER_HA_LIST macro, 208, 324
MASTER_HAD_BACKOFF_CONSTANT macro, 327
MASTER_INSTANCE_LOCK macro, 168
MASTER_NAME macro, 167, 659
MASTER_NEW_BINARY_DELAY macro, 165
MASTER_RECOVER_FACTOR macro, 166
MASTER_UPDATE_INTERVAL macro, 165
MATCH_TIMEOUT macro, 237, 243, 248
matchmaking, 2
negotiation algorithm, 217
on the Grid, 434
MAX_<SUBSYS>_<LEVEL>_LOG macro, 155
MAX_<SUBSYS>_LOG macro, 152, 155, 508
MAX_ACCOUNTANT_DATABASE_SIZE macro, 196
MAX_C_GAHP_LOG macro, 201
MAX_CKPT_SERVER_LOG macro, 317
MAX_CLAIM_ALIVES_MISSED macro, 170, 183
MAX_DAGMAN_LOG macro, 90, 205
MAX_DISCARDED_RUN_TIME macro, 163, 316
MAX_HAD_LOG macro, 211
MAX_HISTORY_LOG macro, 149
MAX_HISTORY_ROTATIONS macro, 149, 506
MAX_JOB_QUEUE_LOG_ROTATIONS macro, 149, 529, 530
max_job_retirement_time, 735
MAX_JOBS_RUNNING macro, 50, 183, 297
MAX_NUM_CPUS macro, 172, 525
MAX_REPLICATION_LOG macro, 212
MAX_SCHEDD_LOG macro, 289
MAX_SHADOW_EXCEPTIONS macro, 183
MAX_SOAP_TRANSACTION_DURATION macro, 543
MAX_TRANSFER_LIFETIME macro, 211
MAX_VIRTUAL_MACHINE_TYPES macro, 175
MaxHosts
job ClassAd attribute, 31, 230
MaxJobRetirementTime
job ClassAd attribute, 31, 230
MAXJOBRETIREMENTTIME macro, 170, 541
MAXJOBRETIREMENTTIME macro, 257
MaxSuspendTime macro, 251
MaxVacateTime macro, 251
MEMORY macro, 172
migration, 2, 3
MinHosts
job ClassAd attribute, 31, 230
MPI application
under the dedicated scheduler, 73, 355
MPI universe, 74–79

MPI_CONDOR_RSH_PATH macro, 187, 564
multiple network interfaces, 298
MY., ClassAd scope resolution prefix, 376
MYPROXY_GET_DELEGATION macro, 213, 431

NEGOTIATE_ALL_JOBS_IN_CLUSTER macro, 186, 219
negotiation, 217
  by group, 220
NEGOTIATOR_ADDRESS_FILE macro, 295
NEGOTIATOR_ADDRESS_FILE macro, 156
NEGOTIATOR_CONSIDER_PREEMPTION macro, 199, 258
NEGOTIATOR_CYCLE_DELAY macro, 196, 559
NEGOTIATOR_DEBUG macro, 198
NEGOTIATOR_DISCOUNT_SUSPENDED_RESOURCES macro, 196
NEGOTIATOR_HOST macro, 145, 560, 582
NEGOTIATOR_INTERVAL macro, 196
NEGOTIATOR_MATCHLIST_CACHING macro, 507, 542
NEGOTIATOR_MAX_TIME_PER_SUBMITTER macro, 198
NEGOTIATOR_POST_JOB_RANK macro, 197, 568, 569
NEGOTIATOR_PRE_JOB_RANK macro, 197, 568
NEGOTIATOR_SOCKET_CACHE_SIZE macro, 197, 297
NEGOTIATOR_TIMEOUT macro, 196
NEGOTIATOR_USE_NONBLOCKING_STARTD_CONTACT macro, 159, 515
NET_REMAP_ENABLE macro, 159
NET_REMAP_INAGENT macro, 159
NET_REMAP_ROUTE macro, 159
NET_REMAP_SERVICE macro, 159
network, 4, 18, 294
network interfaces
  multiple, 298
NETWORK_INTERFACE macro, 157, 299, 301, 543
NETWORK_MAX_PENDING_CONNECTS macro, 151
NFS
  interaction with, 102
nice job, 59
NICE_USER_PRIO_FACTOR macro, 196, 216
NiceUser
  job ClassAd attribute, 31, 230
NICs, 298
NIS
  Condor must be dynamically linked, 478
NO_DNS macro, 150, 520
NODE macro, 72, 76
Node macro, 736
NONBLOCKING_COLLECTOR_UPDATE macro, 158, 515
NonCondorLoadAvg macro, 250
NorduGrid, 432
NORDUGRID_GAHP macro, 202
NUM_CPUS macro, 172
NUM_VIRTUAL_MACHINES macro, 175, 350
NUM_VIRTUAL_MACHINES_TYPE_<N> macro, 175, 350
NumCkpts
  job ClassAd attribute, 31, 230
NumGlobusSubmits
  job ClassAd attribute, 31, 230
NumJobMatches
  job ClassAd attribute, 31, 230
NumRestarts
  job ClassAd attribute, 32, 230
NumSystemHolds
  job ClassAd attribute, 32, 230

OBITUARY_LOG_LENGTH macro, 165
opportunistic scheduling, 355
OPSYS macro, 143, 340, 561, 563
OUT_HIGHPORT macro, 158, 297, 530, 531
OUT_LOWPORT macro, 158, 297, 530, 531
overview, 1–4
Owner
  job ClassAd attribute, 32, 230
owner
  of directories, 111

parallel universe, 20, 70–79
  running MPI applications, 73
Parallel Virtual Machine (PVM), 65

ParallelSchedulingGroup macro, 187, 357–359
PASSWD_CACHE_REFRESH macro, 151
PBS (Portable Batch System), 433
PBS_GAHP macro, 202, 434
PERIODIC_CHECKPOINT macro, 380, 469
PERIODIC_EXPR_INTERVAL macro, 186, 545
PERIODIC_MEMORY_SYNC macro, 188
Perl module, 404
  examples, 406
permission denied, 486
PERSISTENT_CONFIG_DIR macro, 155, 524, 571
Personal Condor, 126, 464
PID macro, 143
platform-specific information
  AIX, 459
  AIX 5.1L, 459
  AIX 5.2L, 459
  Linux, 443
  Linux keyboard and mouse activity, 444
  Mac OS 10.3, 459
  Mac OS 10.4, 459
  Mac OS 10.4 for Intel, 459
  Macintosh OS X, 458
  Red Hat 6.x, 444
  Red Hat 7.x, 444
  Red Hat 8.x, 444
  Red Hat 9.x, 445
  Red Hat Fedora 1, 2, and 3, 445
  Windows, 445–458
platforms supported, 5
policy
  at UW-Madison, 253
  default with Condor, 250
  desktop/non-desktop, 255
  disabling preemption, 257
  suspending jobs instead of evicting them, 258
  time of day, 254
  version differences, 259
POLLING_INTERVAL macro, 170, 244
pool of machines, 104
POOL_HISTORY_DIR macro, 194, 345, 582
POOL_HISTORY_MAX_STORAGE macro, 194, 345
POOL_HISTORY_SAMPLING_INTERVAL macro, 194
port usage, 295
  conflicts, 298
  FAQ on communication errors, 465
  firewalls, 296
  multiple collectors, 298
  nonstandard ports for central managers, 295
PPID macro, 143
PREEMPT macro, 169, 259
preemption
  desktop/non-desktop, 255
  disabling, 257
  priority, 58, 217
  vacate, 59
PREEMPTION_RANK macro, 198, 569
PREEMPTION_REQUIREMENTS macro, 59, 197, 199, 217, 676
PREEN macro, 164
PREEN_ADMIN macro, 192, 669
PREEN_ARGS macro, 164
PREEN_INTERVAL macro, 165
priority
  by group, 219
  in machine allocation, 215
  nice job, 59
  of a job, 52, 58
  of a user, 58
PRIORITY_HALFLIFE macro, 58, 196, 216, 217
process
  definition for a submitted job, 32, 230
Process macro, 528, 736
ProcId
  job ClassAd attribute, 32, 230
proxy, 420
  renewal with *MyProxy*, 429
PUBLISH_OBITUARIES macro, 165
PVM
  machine class, 67
  master-worker paradigm, 66
  PvmHostSuspend and PvmHostResume notifications, 67

submit description file, 68
PVM (Parallel Virtual Machine), 65–70
PVMD macro, 188
PVMGS macro, 188

Q_QUERY_TIMEOUT macro, 151
QDate
job ClassAd attribute, 32, 230
QUERY_TIMEOUT macro, 193
QUEUE_ALL_USERS_TRUSTED macro, 184, 514
QUEUE_CLEAN_INTERVAL macro, 184
QUEUE_SUPER_USERS macro, 185
Quill, 330
QUILL macro, 212
QUILL_ADDRESS_FILE macro, 213, 334
QUILL_ARGS macro, 212
QUILL_DB_IP_ADDR macro, 212, 333
QUILL_DB_NAME macro, 212, 333
QUILL_DB_QUERY_PASSWORD macro, 213, 334
QUILL_ENABLED macro, 212
QUILL_HISTORY_CLEANING_INTERVAL macro, 212, 333
QUILL_HISTORY_DURATION macro, 213, 333
QUILL_IS_REMOTELY_QUERYABLE macro, 213, 334
QUILL_LOG macro, 212
QUILL_MANAGE_VACUUM macro, 213, 333
QUILL_NAME macro, 212, 333
QUILL_POLLING_PERIOD macro, 212, 333
QUILL_SHOULD_REINDEX macro, 499
quotas
for a group, 220

RANDOM_CHOICE() macro
use in submit description file, 737
RANDOM_CHOICE() macro, 144
RANDOM_INTEGER() macro, 144, 469
rank attribute, 23
examples, 33, 378
RANK macro, 169, 357
RANK_FACTOR macro, 357
real user priority (RUP), 216
recovery from crashes, 486
RELEASE_DIR macro, 113, 146, 341
ReleaseReason
job ClassAd attribute, 32, 230
remote system call, 2, 3, 17
condor_shadow, 17, 50, 102
REMOTE_PRIO_FACTOR macro, 196, 216
RemoteIwd
job ClassAd attribute, 32, 230
RemoteSysCpu
job ClassAd attribute, 32, 230
RemoteUserCpu
job ClassAd attribute, 32, 230
RemoteWallClockTime
job ClassAd attribute, 32, 231
RemoveKillSig
job ClassAd attribute, 32, 231
REPLICATION macro, 212
REPLICATION_ARGS macro, 211
REPLICATION_DEBUG macro, 212
REPLICATION_INTERVAL macro, 211
REPLICATION_LIST macro, 211
REPLICATION_LOG macro, 212
REQUEST_CLAIM_TIMEOUT macro, 183
REQUIRE_LOCAL_CONFIG_FILE macro, 148, 590
requirements attribute, 23, 378
automatic extensions, 472
Requirements macro, 182, 538
RESERVE_AFS_CACHE macro, 161
RESERVED_DISK macro, 25, 149, 223
RESERVED_MEMORY macro, 172
RESERVED_SWAP macro, 149, 472
resource
management, 2
offer, 3
owner, 104
request, 3
RPM installation on Linux, 132
RunBenchmarks expression, 242
RUNBENCHMARKS macro, 173
running a job
at certain times of day, 469
on a different architecture, 45
on only certain machines, 467
only at night, 469
running multiple programs, 22

SBIN macro, 146
scalability
using the Grid Monitor, 431
SCHED_UNIV_RENICE_INCREMENT macro, 184
SCHEDD_ADDRESS_FILE macro, 185
SCHEDD_ASSUME_NEGOTIATOR_GONE macro, 186, 543
SCHEDD_ATTRS macro, 185
SCHEDD_BACKUP_SPOOL macro, 187
SCHEDD_DEBUG macro, 185, 554
SCHEDD_EXECUTE macro, 185
SCHEDD_HOST macro, 145
SCHEDD_INTERVAL macro, 183, 574
SCHEDD_LOCK macro, 185
SCHEDD_LOG macro, 472
SCHEDD_NAME macro, 167, 185
SCHEDD_PREEMPTION_RANK macro, 187, 358
SCHEDD_PREEMPTION_REQUIREMENTS macro, 187, 358
SCHEDD_ROUND_ATTR macro, 504
SCHEDD_ROUND_ATTR_<XXXX> macro, 186
scheduler universe, 20
scheduling
dedicated, 70, 75, 355
opportunistic, 355
SDK
Chirp, 63
SEC_*_AUTHENTICATION macro, 206
SEC_*_AUTHENTICATION_METHODS macro, 206
SEC_*_CRYPTO_METHODS macro, 206
SEC_*_ENCRYPTION macro, 206
SEC_*_INTEGRITY macro, 206
SEC_*_NEGOTIATION macro, 206
SEC_ADMINISTRATOR_AUTHENTICATION macro, 265
SEC_ADMINISTRATOR_AUTHENTICATION_METHODS macro, 266
SEC_ADMINISTRATOR_CRYPTO_METHODS macro, 276
SEC_ADMINISTRATOR_ENCRYPTION macro, 275
SEC_ADMINISTRATOR_INTEGRITY macro, 277
SEC_CLAIMTOBE_INCLUDE_DOMAIN macro, 510
SEC_CLIENT_AUTHENTICATION macro, 265
SEC_CLIENT_AUTHENTICATION_METHODS macro, 266
SEC_CLIENT_CRYPTO_METHODS macro, 276
SEC_CLIENT_ENCRYPTION macro, 275
SEC_CLIENT_INTEGRITY macro, 276
SEC_CONFIG_AUTHENTICATION macro, 265
SEC_CONFIG_AUTHENTICATION_METHODS macro, 266
SEC_CONFIG_CRYPTO_METHODS macro, 276
SEC_CONFIG_ENCRYPTION macro, 275
SEC_CONFIG_INTEGRITY macro, 277
SEC_DAEMON_AUTHENTICATION macro, 265
SEC_DAEMON_AUTHENTICATION_METHODS macro, 266
SEC_DAEMON_CRYPTO_METHODS macro, 276
SEC_DAEMON_ENCRYPTION macro, 275
SEC_DEBUG_PRINT_KEYS macro, 586
SEC_DEFAULT_AUTHENTICATION macro, 265
SEC_DEFAULT_AUTHENTICATION_METHODS macro, 314
SEC_DEFAULT_AUTHENTICATION_METHODS macro, 266
SEC_DEFAULT_CRYPTO_METHODS macro, 276
SEC_DEFAULT_ENCRYPTION macro, 275
SEC_DEFAULT_INTEGRITY macro, 276, 277
SEC_DEFAULT_SESSION_DURATION macro, 207
SEC_NEGOTIATOR_AUTHENTICATION macro, 265
SEC_NEGOTIATOR_AUTHENTICATION_METHODS macro, 266
SEC_NEGOTIATOR_CRYPTO_METHODS macro, 276

SEC_NEGOTIATOR_ENCRYPTION macro, 275
SEC_NEGOTIATOR_INTEGRITY macro, 277
SEC_OWNER_AUTHENTICATION macro, 265
SEC_OWNER_AUTHENTICATION_METHODS macro, 266
SEC_OWNER_CRYPTO_METHODS macro, 276
SEC_OWNER_ENCRYPTION macro, 275
SEC_OWNER_INTEGRITY macro, 277
SEC_PASSWORD_FILE macro, 208, 272, 521
SEC_READ_AUTHENTICATION macro, 265
SEC_READ_AUTHENTICATION_METHODS macro, 266
SEC_READ_CRYPTO_METHODS macro, 276
SEC_READ_ENCRYPTION macro, 275
SEC_READ_INTEGRITY macro, 277
SEC_TCP_SESSION_TIMEOUT macro, 208
SEC_WRITE_AUTHENTICATION macro, 265
SEC_WRITE_AUTHENTICATION_METHODS macro, 266
SEC_WRITE_CRYPTO_METHODS macro, 276
SEC_WRITE_ENCRYPTION macro, 275
SEC_WRITE_INTEGRITY macro, 277
SECONDARY_COLLECTOR_LIST macro, 168
security
  in Condor, 260–294
  access levels, 261
  authentication, 265
  authorization, 278
  based on user authorization, 278
  encryption, 275
  host-based, 282
  integrity, 276
  running jobs as user nobody, 292
  sample daemon-side configuration, 277
  sessions, 281
  unified map file, 274
sessions, 281
SETTABLE_ATTRS macro, 156, 288
SETTABLE_ATTRS. . . macro, 156
SETTABLE_ATTRS_PERMISSION-LEVEL macro, 289
shadow, 17
SHADOW macro, 182
SHADOW_ALLOW_UNSAFE_REMOTE_EXEC macro, 558
SHADOW_DEBUG macro, 187
SHADOW_LOCK macro, 187
SHADOW_LOG macro, 54
SHADOW_PVM macro, 182
SHADOW_QUEUE_UPDATE_INTERVAL macro, 187
SHADOW_RENICE_INCREMENT macro, 184
SHADOW_SIZE_ESTIMATE macro, 184
shared file system
  submission of jobs, 35
  submission of jobs without one, 36
SHELL macro, 702
SHUTDOWN_FAST_TIMEOUT macro, 165
SHUTDOWN_GRACEFUL_TIMEOUT macro, 156
signal, 4, 18
  SIGTSTP, 4, 18
  SIGUSR2, 4, 18
SIGNIFICANT_ATTRIBUTES macro, 219, 534, 535
Simple Object Access Protocol(SOAP), 396
skew in timing information, 478
SLOW_CKPT_SPEED macro, 188
SMP machines
  configuration, 347–354
SOAP
  Web Service API, 396
SOAP_LEAVE_IN_QUEUE macro, 213, 398
SOAP_SSL_CA_DIR macro, 214
SOAP_SSL_CA_FILE macro, 214
SOAP_SSL_DH_FILE macro, 214
SOAP_SSL_SERVER_KEYFILE macro, 214
SOAP_SSL_SERVER_KEYFILE_PASSWORD macro, 214
SOFT_UID_DOMAIN macro, 119, 160, 290
Software Developer's Kit
  Chirp, 63
Solaris26, 473
Solaris27, 473

SPOOL macro, 147, 575, 582
START expression, 222
START macro, 169, 173, 356
START_BACKFILL macro, 173, 360, 362, 523
START_DAEMONS macro, 165
START_LOCAL_UNIVERSE macro, 182, 538
START_MASTER macro, 165
START_SCHEDULER_UNIVERSE macro, 182, 538
startd
configuration, 221
STARTD_AD_REEVAL_EXPR macro, 535
STARTD_ADDRESS_FILE macro, 171
STARTD_ATTRS macro, 171, 289
STARTD_AVAIL_CONFIDENCE macro, 181
STARTD_CLAIM_ID_FILE macro, 171, 551
STARTD_COMPUTE_AVAIL_STATS macro, 181
STARTD_CRON macro, 581
STARTD_CRON_<ModuleName>_ARGS macro, 178
STARTD_CRON_<ModuleName>_CWD macro, 178
STARTD_CRON_<ModuleName>_ENV macro, 178
STARTD_CRON_<ModuleName>_EXECUTABLE macro, 177
STARTD_CRON_<ModuleName>_KILL macro, 178
STARTD_CRON_<ModuleName>_MODE macro, 177
STARTD_CRON_<ModuleName>_OPTIONS macro, 178
STARTD_CRON_<ModuleName>_PERIOD macro, 177
STARTD_CRON_<ModuleName>_PREFIX macro, 177
STARTD_CRON_<ModuleName>_RECONFIG macro, 177
STARTD_CRON_AUTOPUBLISH macro, 176, 539
STARTD_CRON_CONFIG_VAL macro, 176
STARTD_CRON_JOBLIST macro, 176, 534
STARTD_CRON_JOBS macro, 179, 534
STARTD_CRON_NAME macro, 175
STARTD_DEBUG macro, 171, 289
STARTD_EXPRS macro, 319, 359
STARTD_HAS_BAD_UTMP macro, 171
STARTD_JOB_EXPRS macro, 171
STARTD_MAX_AVAIL_PERIOD_SAMPLES macro, 181
STARTD_NAME macro, 172
STARTD_NOCLAIM_SHUTDOWN macro, 173
STARTD_RESOURCE_PREFIX macro, 174, 525
STARTD_SHOULD_WRITE_CLAIM_ID_FILE macro, 171, 551
STARTD_VM_EXPRS macro, 174, 258, 568
STARTER macro, 170
STARTER_ALLOW_RUNAS_OWNER macro, 448
STARTER_CHOOSES_CKPT_SERVER macro, 163, 318
STARTER_DEBUG macro, 189
STARTER_INITIAL_UPDATE_INTERVAL macro, 588
STARTER_JOB_ENVIRONMENT macro, 190, 568
STARTER_LOCAL_LOGGING macro, 189
STARTER_UPDATE_INTERVAL macro, 189
StartIdleTime macro, 250
state
of a machine, 235
transitions, 239–250
transitions summary, 248
state and activities figure, 239
STATE_FILE macro, 211
StateTimer macro, 250
status
of queued jobs, 50
Stork, 95–99
jobs under DAGMan, 99
submit description file, 96
Stork commands
stork_list_cred, 778
stork_q, 776
stork_rm, 780
stork_rm_cred, 782
stork_status, 786
stork_store_cred, 784
stork_submit, 788

Stork submit commands
alt_protocols, 790
cred_name, 790
dap_type, 788
dest_url, 789
src_url, 789
x509proxy, 790
STORK_ENVIRONMENT macro, 537
stork_list_cred command, 778
STORK_MAX_NUM_JOBS macro, 214
STORK_MAX_RETRY macro, 98, 214
STORK_MAXDELAY_INMINUTES macro, 214
STORK_MODULE_DIR macro, 215
stork_q command, 776
stork_rm command, 780
stork_rm_cred command, 782
stork_status command, 786
stork_store_cred command, 784
stork_submit command, 788
STORK_TMP_CRED_DIR macro, 214
StreamErr
job ClassAd attribute, 32, 231
StreamOut
job ClassAd attribute, 32, 231
submit commands
$ENV macro, 737
$RANDOM_CHOICE() macro, 737
allow_startup_script, 726
append_files, 727
arguments, 717
buffer_block_size, 727
buffer_files, 727
buffer_size, 727
compress_files, 727
copy_to_spool, 732
coresize, 732
deferral_time, 733
deferral_window, 733
environment, 718
error, 719
executable, 719
fetch_files, 728
file_remaps, 728
getenv, 719
globus_rematch, 729
globus_resubmit, 729
globus_rsl, 729
globus_xml, 729
grid_resource, 421, 425, 426, 730
hold, 724
hold_kill_sig, 732
image_size, 733
initialdir, 734
input, 719
jar_files, 732
java_vm_args, 732
job_lease_duration, 734
keystore_alias, 730
keystore_file, 730
keystore_passphrase_file, 730
kill_sig, 734
leave_in_queue, 724
local_files, 728
log, 720, 789
log_xml, 720
machine_count, 732
match_list_length, 734
max_job_retirement_time, 735
MyProxyCredentialName, 730
MyProxyHost, 730
MyProxyNewProxyLifetime, 730
MyProxyPassword, 731
MyProxyRefreshThreshold, 731
MyProxyServerDN, 731
nice_user, 735
noop_job, 735
noop_job_exit_code, 735
noop_job_exit_signal, 735
nordugrid_rsl, 731
notification, 720
notify_user, 720
on_exit_hold, 724
on_exit_remove, 724
output, 720
periodic_hold, 725
periodic_release, 726
periodic_remove, 726
priority, 720
queue, 721
rank, 33, 44, 721
remote_initialdir, 735

remove_kill_sig, 732
rendezvousdir, 736
requirements, 23, 43, 721
should_transfer_files, 37, 722
stream_error, 722
stream_input, 722
stream_output, 723
transfer_error, 731
transfer_executable, 723
transfer_input, 731
transfer_input_files, 723
transfer_output, 731
transfer_output_files, 723
transfer_output_remaps, 723
universe, 721
want_remote_io, 729
when_to_transfer_output, 37, 479, 724
x509userproxy, 731
submit description file, 20
contents of, 20
examples, 21–23
for PVM application, 68
grid universe, 421
submit machine, 105
submit-only installation, 121
SUBMIT_DEBUG macro, 154
SUBMIT_EXPRS macro, 191, 523, 528
SUBMIT_MAX_PROCS_IN_CLUSTER macro, 192, 514
SUBMIT_SEND_RESCHEDULE macro, 191, 588
SUBMIT_SKIP_FILECHECK macro, 191
substitution macro
in submit description file, 736
<SUBSYS> macro, 164
<SUBSYS>_ADDRESS_FILE macro, 156
<SUBSYS>_ARGS macro, 164
<SUBSYS>_ATTRS macro, 156
<SUBSYS>_DEBUG macro, 152
<SUBSYS>_DEBUG macro levels
D_ACCOUNTANT, 154
D_ALL, 152
D_CKPT, 154
D_COMMAND, 153
D_DAEMONCORE, 153
D_FDS, 154
D_FULLDEBUG, 153
D_HOSTNAME, 154
D_JOB, 153
D_KEYBOARD, 153
D_LOAD, 153
D_MACHINE, 153
D_MATCH, 153
D_NETWORK, 153
D_PID, 154
D_PRIV, 153
D_PROCFAMILY, 154
D_PROTOCOL, 154
D_SECURITY, 154
D_SYSCALLS, 153
<SUBSYS>_ENABLE_SOAP_SSL macro, 214
<SUBSYS>_EXPRS macro, 156
<SUBSYS>_<LEVEL>_LOG macro, 155
<SUBSYS>_LOCK macro, 152
<SUBSYS>_LOG macro, 152
<SUBSYS>_SOAP_SSL_PORT macro, 214
<SUBSYS>_TIMEOUT_MULTIPLIER macro, 158
SUBSYSTEM macro, 142
subsystem names, 142
supported platforms, 5
SUSPEND macro, 169, 475
SYSAPI_GET_LOADAVG macro, 151
SYSTEM_PERIODIC_HOLD macro, 539
SYSTEM_PERIODIC_RELEASE macro, 539
SYSTEM_PERIODIC_REMOVE macro, 539

TARGET., ClassAd scope resolution prefix, 376
TCP, 314
sending updates, 314
TCP_UPDATE_COLLECTORS macro, 158
thread
kernel-level, 4, 18
user-level, 4, 18
TILDE macro, 142
timing information
incorrect, 478
TOOL_DEBUG macro, 154
TotalSuspensions
job ClassAd attribute, 33, 231

TOUCH_LOG_INTERVAL macro, 152, 520
TransferErr
job ClassAd attribute, 33, 231
TransferExecutable
job ClassAd attribute, 33, 231
TransferIn
job ClassAd attribute, 33, 231
TransferOut
job ClassAd attribute, 33, 231
transferring files, 36, 479
TRUNC_<SUBSYS>_<LEVEL>_LOG_ON_OPEN macro, 155
TRUNC_<SUBSYS>_LOG_ON_OPEN macro, 152, 155
TRUST_UID_DOMAIN macro, 160, 533

UDP, 314
lost datagrams, 314
UID
effective, 290
potential risk running jobs as user nobody, 292
real, 289
UID_DOMAIN macro, 119, 144, 159, 290, 291, 300, 720
UIDs in Condor, 289–294
UNAME_ARCH macro, 143
UNAME_OPSYS macro, 143
Unicore, 433
UNICORE_GAHP macro, 202
uniq_pid_midwife, 792
uniq_pid_undertaker, 794
universe, 16
Grid, 17, 19
grid, 415, 440
grid, grid type gt2, 420
grid, grid type gt3, 425
grid, grid type gt4, 426
Java, 19
java, 17
job attribute definitions, 30, 228
local, 20
MPI, 17
parallel, 17, 20
PVM, 17, 19
scheduler, 20
standard, 17
vanilla, 17, 19
Unix
alarm, 4, 18
exec, 4, 18
flock, 4, 18
fork, 4, 18
large files, 5, 19
lockf, 4, 18
mmap, 4, 18
pipe, 4, 18
semaphore, 4, 18
shared memory, 4, 18
sleep, 4, 18
socket, 4, 18
system, 4, 18
timer, 4, 18
Unix administrator, 110
Unix daemon
running as root, 102
Unix directory
execute, 111
lock, 112
log, 112
spool, 111
Unix installation
download, 108
scripts, 116
submit-only, 121
Unix user
condor, 111
root, 110
UPDATE_COLLECTOR_WITH_TCP macro, 158, 194, 315, 581
UPDATE_INTERVAL macro, 170, 176, 241, 537
upgrade
version 6.4.x to version 6.6.x, 462
upgrading
items to be aware of, 491
USE_AFS macro, 162
USE_CKPT_SERVER macro, 163, 317, 318
USE_NFS macro, 161
USE_VISIBLE_DESKTOP macro, 190, 449
user
priority, 58

user condor
home directory not found, 478
user manual, 12–103
user nobody
potential security risk with jobs, 292
user priority, 215
effective (EUP), 216
real (RUP), 216
USER_JOB_WRAPPER macro, 189, 483, 581
USERNAME macro, 143

vacate, 59
VACATE macro, 259
VALID_COD_USERS macro, 385
VALID_LOG_FILES macro, 192, 322, 669
VALID_SPOOL_FILES macro, 192, 669
vanilla jobs
cycling between suspended and unsuspended, 474
VARS, 85
viewing
log files, 99
virtual machine
configuration, 347
running Condor jobs under, 346
VIRTUAL_MACHINE_TYPE_<N> macro, 175, 349
VIRTUAL_MACHINES_CONNECTED_TO_CONSOLE macro, 174, 351
VIRTUAL_MACHINES_CONNECTED_TO_KEYBOARD macro, 174, 351
VMP_HOST_MACHINE macro, 347
VMP_VM_LIST macro, 347
VMx_USER macro, 160, 293

WALL_CLOCK_CKPT_INTERVAL macro, 184
WANT_SUSPEND macro, 169
WANT_VACATE macro, 169, 259
Web Service API, 396
file transfer, 398
job submission, 396
transactions, 396
WEB_ROOT_DIR macro, 213
Windows
Condor daemon names, 132
installation, 123–132
initial file size, 124
location of files, 126
Personal Condor, 126
preparation, 124
required disk space, 125
unattended install, 128
manual install, 130
out of desktop heap, 483
release notes, 446
starting the Condor service, 131
WINDOWS_FIREWALL_FAILURE_RETRY macro, 168
WorkHours macro, 255

X509_USER_PROXY, 45, 521, 553