Netcool/OMNIbus
Version 7 Release 4

Probe and Gateway Guide

SC14-7530-02

Netcool/OMNIbus
Version 7 Release 4

Probe and Gateway Guide

SC14-7530-02

Note
Before using this information and the product it supports, read the information in "Notices" on page 251.

This edition applies to version 7, release 4 of IBM Tivoli Netcool/OMNIbus (product number 5724-S44) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

About this publication

Tivoli Netcool/OMNIbus is a service level management (SLM) system that delivers real-time, centralized monitoring of complex networks and IT domains.

The *IBM Tivoli Netcool/OMNIbus Probe and Gateway Guide* contains introductory and reference information about probes, including probe rules file syntax, properties and command-line options, error messages, and troubleshooting techniques. This publication also contains introductory and reference information about gateways, including gateway commands, command-line options, and error messages.

Intended audience

This publication is intended for both users and administrators who need to configure and use probes and gateways.

Probes and gateways are part of Tivoli Netcool/OMNIbus, and it is assumed that you understand how Tivoli Netcool/OMNIbus works.

What this publication contains

This publication contains the following sections:

- Chapter 1, "About probes," on page 1

  Provides information about probes, their architecture, components, and modes of operation.
- Chapter 2, "Probe rules file syntax," on page 19

  Describes the syntax of the rules file that defines how the probe must process event data to create a meaningful Tivoli Netcool/OMNIbus alert.
- Chapter 3, "Probe rules file customizations," on page 67

  Describes the customizations that can be applied to probe rules files to extend the functionality of probes.
- Chapter 4, "Running probes," on page 85

  Describes how to run probes.
- Chapter 5, "Remotely administering probes," on page 91

  Describes how to remotely manage probes using the probe HTTP interface.
- Chapter 6, "Common probe properties and command-line options," on page 113

  Describes the properties and command-line options that are common to all probes and TSMs.
- Chapter 7, "Netcool MIB Manager," on page 129

  Provides information about using Netcool MIB Manager to parse Simple Network Management Protocol (SNMP) management information base (MIB) files.
- Chapter 8, "About gateways," on page 159

  Provides information about gateways, their modes of operation, and gateway components.
- Chapter 9, "Configuring gateways," on page 167

  Provides information about configuring gateways.

- Chapter 10, “Running gateways,” on page 201

  Describes how to run gateways.
- Appendix A, “Probe error messages and troubleshooting techniques,” on page 203

  Provides information on probe error messages and troubleshooting.
- Appendix B, “Common gateway error messages,” on page 215

  Provides information on gateway error messages.
- Appendix C, “Regular expressions,” on page 223

  Provides reference information on regular expressions.
- Appendix D, “ObjectServer tables and data types,” on page 233

  Provides reference information on relevant ObjectServer tables.

Publications

This section lists publications in the Tivoli Netcool/OMNIbus library and related documents. The section also describes how to access Tivoli publications online and how to order Tivoli publications.

Your Tivoli Netcool/OMNIbus library

The following documents are available in the Tivoli Netcool/OMNIbus library:

- *IBM Tivoli Netcool/OMNIbus Installation and Deployment Guide*, SC14-7526

  Includes installation and upgrade procedures for Tivoli Netcool/OMNIbus, and describes how to configure security and component communications. The publication also includes examples of Tivoli Netcool/OMNIbus architectures and describes how to implement them.
- *IBM Tivoli Netcool/OMNIbus Administration Guide*, SC14-7527

  Describes how to perform administrative tasks using the Tivoli Netcool/OMNIbus Administrator GUI, command-line tools, and process control. The publication also contains descriptions and examples of ObjectServer SQL syntax and automations.
- *IBM Tivoli Netcool/OMNIbus Web GUI Administration and User's Guide*, SC14-7528

  Describes how to perform administrative and event visualization tasks using the Tivoli Netcool/OMNIbus Web GUI.
- *IBM Tivoli Netcool/OMNIbus User's Guide*, SC14-7529

  Provides an overview of the desktop tools and describes the operator tasks related to event management using these tools.
- *IBM Tivoli Netcool/OMNIbus Probe and Gateway Guide*, SC14-7530

  Contains introductory and reference information about probes and gateways, including probe rules file syntax and gateway commands.
- *IBM Tivoli Monitoring for Tivoli Netcool/OMNIbus Agent User's Guide*, SC14-7532

  Describes how to install the health monitoring agent for Tivoli Netcool/OMNIbus and contains reference information about the agent.
- *IBM Tivoli Netcool/OMNIbus Event Integration Facility Reference*, SC14-7533

  Describes how to develop event adapters that are tailored to your network environment and the specific needs of your enterprise. This publication also describes how to filter events at the source.
- *IBM Tivoli Netcool/OMNIbus Error Messages Guide*, SC14-7534

  Describes system messages in Tivoli Netcool/OMNIbus and how to respond to those messages.

- *IBM Tivoli Netcool/OMNIbus Web GUI Administration API (WAAPI) User's Guide*, SC22-7535

  Shows how to administer the Tivoli Netcool/OMNIbus Web GUI using the XML application programming interface named WAAPI
- *IBM Tivoli Netcool/OMNIbus ObjectServer HTTP Interface Reference Guide*, SC27-5613Describes the URIs and common behaviors of the Application Programming Interface (API) that is called the ObjectServer HTTP Interface. Describes how to enable the API and provides examples of JSON payloads, and HTTP requests and responses.
- *IBM Tivoli Netcool/OMNIbus ObjectServer OSLC Interface Reference Guide*, SC27-5613Describes the services, resources, and common behaviors of the Open Services for Lifecycle Collaboration (OSLC) Application Programming Interface (API) that is called the ObjectServer OSLC Interface. Describes how to enable the API and provides examples of service provider definitions, RDF/XML payloads, and HTTP requests and responses.

Accessing terminology online

The IBM Terminology Web site consolidates the terminology from IBM product libraries in one convenient location. You can access the Terminology Web site at the following Web address:

http://www.ibm.com/software/globalization/terminology

Accessing publications online

IBM posts publications for this and all other Tivoli products, as they become available and whenever they are updated, to the Tivoli Information Center Web site at:

http://publib.boulder.ibm.com/infocenter/tivihelp/v3r1/index.jsp

Note: If you print PDF documents on other than letter-sized paper, set the option in the **File** > **Print** window that allows Adobe Reader to print letter-sized pages on your local paper.

Ordering publications

You can order many Tivoli publications online at the following Web site:

http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss

You can also order by telephone by calling one of these numbers:

- In the United States: 800-879-2755
- In Canada: 800-426-4968

In other countries, contact your software account representative to order Tivoli publications. To locate the telephone number of your local representative, perform the following steps:

1. Go to the following Web site:

   http://www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss
2. Select your country from the list and click **Go**. The Welcome to the IBM Publications Center page is displayed for your country.

3. On the left side of the page, click **About this site** to see an information page that includes the telephone number of your local representative.

Accessibility

Accessibility features help users with a physical disability, such as restricted mobility or limited vision, to use software products successfully.

With this product, you can use assistive technologies to hear and navigate the interface. You can also use the keyboard instead of the mouse to operate some features of the graphical user interface.

Tivoli technical training

For Tivoli technical training information, refer to the following IBM Tivoli Education Web site:

http://www.ibm.com/software/tivoli/education

Support information

If you have a problem with your IBM software, you want to resolve it quickly. IBM provides the following ways for you to obtain the support you need:

Online
Go to the IBM Software Support site at http://www.ibm.com/software/support/probsub.html and follow the instructions.

IBM Support Assistant
The IBM Support Assistant (ISA) is a free local software serviceability workbench that helps you resolve questions and problems with IBM software products. The ISA provides quick access to support-related information and serviceability tools for problem determination. To install the ISA software, go to http://www.ibm.com/software/support/isa.

Documentation
If you have a suggestion for improving the content or organization of this guide, send it to the Tivoli Netcool/OMNIbus Information Development team at:

mailto://L3MMDOCS@uk.ibm.com

Conventions used in this publication

This publication uses several conventions for special terms and actions and operating system-dependent commands and paths.

Typeface conventions

This publication uses the following typeface conventions:

Bold

- Lowercase commands and mixed case commands that are otherwise difficult to distinguish from surrounding text
- Interface controls (check boxes, push buttons, radio buttons, spin buttons, fields, folders, icons, list boxes, items inside list boxes,

multicolumn lists, containers, menu choices, menu names, tabs, property sheets), labels (such as **Tip:** and **Operating system considerations:**)
- Keywords and parameters in text

Italic
- Citations (examples: titles of publications, diskettes, and CDs)
- Words defined in text (example: a nonswitched line is called a *point-to-point* line)
- Emphasis of words and letters (words as words example: "Use the word *that* to introduce a restrictive clause."; letters as letters example: "The LUN address must start with the letter *L*.")
- New terms in text (except in a definition list): a *view* is a frame in a workspace that contains data
- Variables and values you must provide: ... where *myname* represents....

Monospace
- Examples and code examples
- File names, programming keywords, and other elements that are difficult to distinguish from surrounding text
- Message text and prompts addressed to the user
- Text that the user must type
- Values for arguments or command options

Operating system-dependent variables and paths

This publication uses the UNIX convention for specifying environment variables and for directory notation.

When using the Windows command line, replace **$***variable* with **%***variable***%** for environment variables, and replace each forward slash (**/**) with a backslash (\) in directory paths. For example, on UNIX systems, the $NCHOME environment variable specifies the path of the Netcool® home directory. On Windows systems, the %NCHOME% environment variable specifies the path of the Netcool home directory. The names of environment variables are not always the same in the Windows and UNIX environments. For example, %TEMP% in Windows environments is equivalent to $TMPDIR in UNIX environments.

If you are using the bash shell on a Windows system, you can use the UNIX conventions.

Operating system-specific directory names

Where Tivoli Netcool/OMNIbus files are identified as located within an *arch* directory under NCHOME, *arch* is a variable that represents your operating system directory, as shown in the following table.

Table 1. Directory names for the arch variable

Directory name represented by *arch*	**Operating system**
`aix5`	AIX® systems
`hpux11hpia`	HP-UX Itanium-based systems
`linux2x86`	Red Hat Linux and SUSE systems
`linux2s390`	Linux for System z®

Table 1. Directory names for the arch variable (continued)

Directory name represented by *arch*	**Operating system**
`solaris2`	Solaris systems
`win32`	Windows systems

Fix pack information

Information that is applicable only to the fix pack versions of Tivoli Netcool/OMNIbus are prefaced with a graphic. For example, if a set of instructions is preceded by the graphic Fix Pack 1 , it means that the instructions can only be performed if you installed fix pack 1 of your installed version of Tivoli Netcool/OMNIbus. In the release notes, descriptions of known problems that are prefaced with Fix Pack 1 are solved in fix pack 1, and so on.

Note: Fix packs are distributed separately for the server components and the Web GUI component.

Chapter 1. About probes

Probes and Telco Service Managers (TSMs) connect to an event source, detect and acquire event data, and forward the data to the ObjectServer as alerts. TSMs operate in the same manner as probes but have some additional functions. Probes and TSMs use the logic specified in a rules file to manipulate the event elements before converting them into fields of an alert in the ObjectServer alerts.status table.

The following figure shows how probes fit into the Tivoli Netcool/OMNIbus architecture.

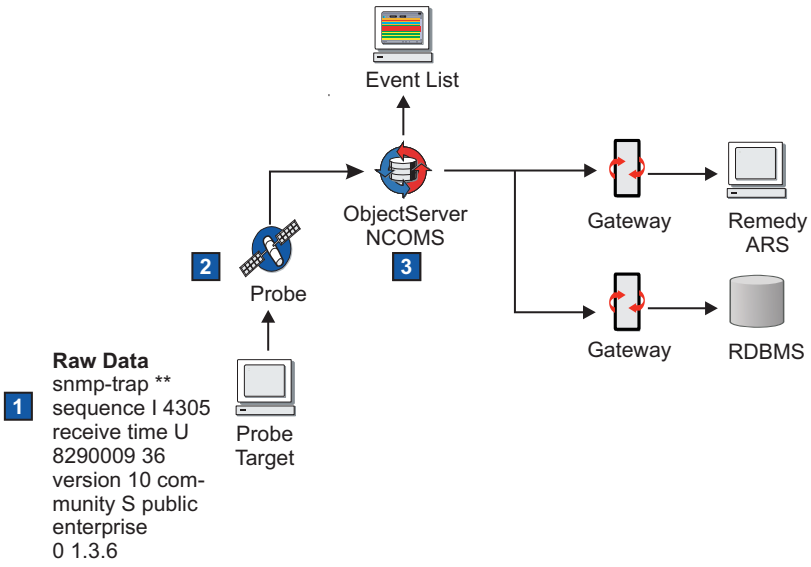

Figure 1. Event processing in Tivoli Netcool/OMNIbus

The flow of event data is as follows:

1. Event data is generated by the probe target.
2. The probe tokenizes the event data, adds extra information to the event, and assigns values to the fields in the ObjectServer alerts.status table. The probe then forwards the processed data to the ObjectServer as an alert.
3. The ObjectServer stores and manages alerts, which can be displayed in the event list, and optionally forwarded to one or more gateways.

Note: The information in this publication is generic to all probes. For probe-specific information, see the individual probe publications in the IBM Tivoli Network Management Information Center:

1. Go to http://publib.boulder.ibm.com/infocenter/tivihelp/v8r1/index.jsp.
2. Expand the *IBM Tivoli Netcool/OMNIbus* node in the navigation pane on the left.
3. Expand the *Tivoli Netcool/OMNIbus probes and TSMs* node.
4. Look for the relevant publication.

Related concepts:

"Types of probes"
Each probe is uniquely designed to acquire event data from a specific source. However, probes can be categorized based on how they acquire events.

Probe registration

When a probe connects to the ObjectServer, it registers information about itself in the registry.probes table.

The registry.probes table tracks dynamic runtime information about probes. The probe controls what data is entered into the table.

If you have two or more instances of a probe running on one computer, and each instance has the same name, only one instance will be registered in the registry.probes table. To enable registration of all the instances of a probe running on the same computer, you must use unique values for each probe's **Name** property.

Note: When a probe is connected to the ObjectServer through a proxy server, the connection ID of the probe can change over time and it might therefore be registered incorrectly. This is because the proxy server optimizes its ObjectServer connections and dynamically shuffles probe connections around. However, the connection ID stored in the registry.probes table remains the same. It is not updated when a probe is moved to another connection on the same proxy server.

A workaround for this problem is to not use a proxy server in multitiered deployments.

If you use the `kill -9` command to stop a probe process, and the probe is connected through a proxy server, the existing probe data in the registry.probes table is retained and is not refreshed when the probe is restarted. This problem does not arise when a probe process is stopped using the `kill` command.

Related reference:

"registry.probes table" on page 247
The registry.probes table is used to track dynamic runtime information about probes. When a probe connects to the ObjectServer, it registers information about itself in the registry.probes table. The probe controls what data is entered into the table.

Types of probes

Each probe is uniquely designed to acquire event data from a specific source. However, probes can be categorized based on how they acquire events.

The types of probes are:

- Device
- Log file
- Database
- API
- CORBA
- Miscellaneous

The probe type is determined by the method in which the probe detects events. For example, the Probe for Agile ATM Switch Management detects events

produced by a device (an ATM switch), but it acquires events from a log file, not directly from the switch. Therefore, this probe is classed as a log file probe and not a device probe. Likewise, the Probe for Oracle obtains event data from a database table, and is therefore classed as a database probe.

Device probes

A device probe acquires events by connecting to a remote device, such as an ATM switch.

Device probes often run on a separate machine to the one they are probing, and connect to the target machine through a network link, modem, or physical cable. Some device probes can use more than one method to connect to the target machine.

After connecting to the target machine, the probe detects events and forwards them to the ObjectServer. Some device probes are passive and wait to detect an event before forwarding it to the ObjectServer. Other device probes are more active and issue commands to the target device in order to acquire events.

Log file probes

A log file probe acquires events by reading a log file that is created by the target system.

For example, the Probe for Heroix RoboMon Element Manager reads the Heroix RoboMon Element Manager event file.

Most log file probes run on the machine where the log file resides; this is not necessarily the same machine as the target system. The target system appends events to the log file. Periodically, the probe opens the log file, acquires and processes the events stored in it, and forwards the relevant events to the ObjectServer as alerts. You can configure how often the probe checks the log file for new events, and how events are processed.

Database probes

A database probe acquires events from a single database table; the *source* table. Depending on the configuration, any change (insert, update, or delete) to a row of the source table can produce an event.

For example, the Probe for Oracle acquires data from transactions logged in an Oracle database table.

When a database probe starts, it creates a temporary logging table and adds a trigger to the source table. When a change is made to the source table, the trigger forwards the event to the logging table. Periodically, the events stored in the logging table are forwarded to the ObjectServer as alerts, and the contents of the logging table are discarded. You can configure how often the probe checks the logging table for new events.

Attention: Existing triggers on the source table might be overwritten when the probe is installed.

Database probes treat each row of the source table as a single entity. Even if only one field of a row in the source table changes, all of the fields of that row are forwarded to the logging table, and from there to the ObjectServer. If a row in the

source table is deleted, the probe forwards the contents of the row before it was deleted. If a row in the source table is inserted or updated, the probe forwards the contents of the row after the insert or update action.

API probes

An API probe acquires events through the application programming interface (API) of another application.

For example, the Probe for Sun Management Center uses the Sun Management Center Java™ API to connect remotely to the Sun Management Center.

API probes use specially-designed libraries to acquire events from another application or management system. These libraries contain functions that connect to the target system and manage the retrieval of events. The API probes call these functions, which connect to the target system and return any events to the probe. The probe processes these events and forwards them to the ObjectServer as alerts.

CORBA probes

The Common Object Request Broker Architecture (CORBA) allows distributed systems to be defined independently of a specific programming language. CORBA probes use CORBA interfaces to connect to the data source, which is usually an Element Management System (EMS).

Equipment vendors publish the details of their specific CORBA interface as Interface Definition Language (IDL) files. These IDL files are used to create the CORBA client and server applications. A specific probe is required for each specific CORBA interface. Some CORBA probes use the IBM Object Request Broker (ORB) to communicate with other vendor ORBs. The IBM ORB is supplied with Tivoli Netcool/OMNIbus. Some CORBA probes use ORBs from other vendors. See the probe documentation for details of which ORB is required.

Most CORBA probes are written using Java, and require specific Java components to be installed to run the probe, as described in the individual publications for these probes. Probes written in Java use the following additional processes:

- The **probe-nco-p-nonnative** probe, which enables probes written in Java to communicate with the standard probe C library (libOpl)
- Java runtime libraries

Miscellaneous probes

All of the miscellaneous probes have characteristics that differentiate them from the other types of probes, and from each other. Each of these probes carries out a specialized task that requires it to work in a unique way.

For example, the Email Probe connects to the mail server, retrieves e-mails, processes them, deletes them, and then disconnects. This is useful on a workstation that does not have sufficient resources to permit an SMTP server and associated local mail delivery system to be kept resident and continuously running.

Another example of a probe in the miscellaneous category is the Ping Probe. It is used for general purpose applications on UNIX operating systems and does not require any special hardware. You can use the Ping Probe to monitor any device that supports the ICMP protocol, such as switches, routers, PCs, and UNIX hosts.

Probe components

A probe has the following primary components: an executable file, a properties file, and a rules file.

Some probes have additional components. When additional components are provided, they are described in the individual probe publications.

Executable file

The executable file is the core of a probe. This file connects to the event source, acquires and processes events, and forwards the events to the ObjectServer as alerts.

On 32-bit UNIX and Linux operating systems, probes are installed to the `$NCHOME/omnibus/probes/`*arch* directory.

On 64-bit UNIX and Linux operating systems, probes are installed to the `$NCHOME/omnibus/platform/`*arch*`/probes64` directory.

On Windows operating systems, probes are installed to the `%NCHOME%\omnibus\probes\win32` directory.

On all operating systems, you must use the `nco_p_*` wrapper scripts in `$NCHOME/omnibus/probes/` to start probes, for example:

```
$NCHOME/omnibus/probes/nco_p_ping
```

When the probe starts, it obtains information about how to configure its environment from its properties and rules files. The probe uses this configuration information to customize the data that it forwards to the ObjectServer.

Related concepts:

"Properties file"
Probe properties define the environment in which the probe runs.

"Rules file" on page 7
The rules file defines how the probe processes event data to create a meaningful alert. For each alert, the rules file also creates an identifier that uniquely identifies the problem source.

Properties file

Probe properties define the environment in which the probe runs.

For example, the **Server** property specifies the ObjectServer to which the probe forwards alerts. Probe properties files are stored in the following directory:

```
$NCHOME/omnibus/probes/arch
```

Where *arch* represents the operating system directory.

Properties files are identified by the `.props` file extension. For example, the properties file for the Ping Probe is: `$NCHOME/omnibus/probes/solaris2/ping.props`.

Properties files consist of name-value pairs separated by a colon, as shown in the following example:

```
Server : "NCOMS"
```

In this name-value pair, **Server** is the name of the property and NCOMS is the value to which the property is set. String values must be enclosed in double quotation marks (" "). Other values do not require quotation marks.

Important: If you change probe properties, you might need to restart the probe for the change to take effect. Most probe properties are read only when the probe initializes. If you change a probe property while a probe is running, the change is stored as a variable in the probe rules file. You cannot force a probe to save the changed or new value of the property while the probe is running.

Related concepts:

"Executable file" on page 5
The executable file is the core of a probe. This file connects to the event source, acquires and processes events, and forwards the events to the ObjectServer as alerts.

Chapter 4, "Running probes," on page 85
When running a probe, you can specify properties in a properties file or options at the command line to configure settings for the probe.

Probe property types

Probe properties can be divided into two categories: common properties and probe-specific properties.

Common properties are relevant to all probes. For example, the **Server** property is a common property, because every probe needs to know which ObjectServer to send alerts to.

Probe-specific properties vary by probe. Some probes do not have any specific properties, but most have additional properties that relate to the environment in which they run. For example, the Ping Probe has a **Pingfile** property that specifies the name of a file containing a list of the machines to be pinged.

Probe-specific properties are described in the individual probe publications.

Related reference:

Chapter 6, "Common probe properties and command-line options," on page 113
A number of properties and command-line options are common to all probes and TSMs.

Probe property versus probe command-line option usage

Each probe property has a corresponding command-line option.

For example, the **Server** property is set in the properties file as follows:

```
Server : "NCOMS"
```

You can also set this property on the command line by using the `-server` command-line option as follows:

```
$OMNIHOME/probes/nco_p_probename -server NCOMS
```

The command-line option overrides the property when both are set. For example, if the property sets the server to NCOMS and the command-line option sets the server to STWO, the value STWO is used for the ObjectServer name.

Related reference:

Chapter 6, “Common probe properties and command-line options,” on page 113
A number of properties and command-line options are common to all probes and TSMs.

Rules file

The rules file defines how the probe processes event data to create a meaningful alert. For each alert, the rules file also creates an identifier that uniquely identifies the problem source.

Rules files work as follows:

1. The probe acquires raw data from the event source and breaks it down into tokens. Each token represents a piece of event data.
2. The probe parses these tokens to elements.
3. The probe processes these elements according to the rules in the rules file, to assign values to the fields in the ObjectServer. These field values are the event details in the form that is used by the ObjectServer.
4. The probe forwards the field values to the ObjectServer as an alert.

In the rules file, elements are identified by the `$` symbol. For example, `$Node` is an element that contains the node name of the event source. Field values are identified by the `@` symbol. For example, `@Node` can be a field value that contains the node name of the event source.

When the probe processes the elements, the `@Identifier` field is assigned a unique value. Duplicate alerts, that is, events that have the same value in the `@Identifier` field, are correlated, so that they are displayed in the event list only once.

Local rules files are stored in the `$NCHOME/omnibus/probes/`*arch* directory, where *arch* is the operating system directory.

The rules file is identified by the `.rules` file extension. For example, the rules file for the Ping Probe that runs on Solaris is: `$NCHOME/omnibus/probes/solaris2/ping.rules`

A probe reads its rules file only at startup, or if the probe is forced to reread its rules file. Rules files can be cached, so that if a probe is unable to read its rules file, the probe attempts to read the rules from the cached file.

You can use a Web address to specify a rules file on a remote server that is accessible by using HTTP. This method allows all rules files to be sourced for each probe from a central point. You can use a suitable configuration management tool, such as CVS, at the central point to enable version management of all rules files.

Related concepts:

"Executable file" on page 5
The executable file is the core of a probe. This file connects to the event source, acquires and processes events, and forwards the events to the ObjectServer as alerts.

"Probe architecture" on page 9
The function of a probe is to acquire information from an event source and forward it to the ObjectServer. Probes use *tokens* and *elements*, and apply rules, to transform event source data into a format that the ObjectServer can recognize.

Chapter 4, "Running probes," on page 85
When running a probe, you can specify properties in a properties file or options at the command line to configure settings for the probe.

Related tasks:

"Rereading the rules file" on page 60
Because probes read the rules file only on startup, you must force the probe to reread the rules whenever you make changes to it. The probe processes the reread request only on receipt of a new event. If the probe is idle or is already processing an event, it will not reread the rules file until a new event is received.

"Enabling caching of probe rules files" on page 61
To ensure that a probe is always able to read a valid set of rules when the probe is started, enable the caching of the rules file. By default, rules file caching is disabled.

Related reference:

Chapter 2, "Probe rules file syntax," on page 19
The rules file defines how the probe should process event data to create a meaningful Tivoli® Netcool/OMNIbus alert. The rules file also creates an identifier for each alert to uniquely identify the problem source, so that repeated events can be deduplicated.

Naming conventions for probe component files

Each probe has an abbreviated name that is used to identify the probe executable file and other associated files.

The naming conventions used for probe file names are shown in the following table.

Table 2. Naming conventions for probe file names

Probe file type	File name and location
Properties file	`$NCHOME/omnibus/probes/`*`arch`*`/`*`probename`*`.props`
Rules file	`$NCHOME/omnibus/probes/`*`arch`*`/`*`probename`*`.rules`

In these paths:

- *arch* represents the operating system directory on which the probe is installed; for example, `solaris2` on a Solaris system.
- *probename* represents the abbreviated probe name.

For example, the abbreviated name for SunNet Manager is `snmlog` and the Probe for SunNet Manager properties file is named `snmlog.props`.

The rules file is named `snmlog.rules`.

Probe architecture

The function of a probe is to acquire information from an event source and forward it to the ObjectServer. Probes use *tokens* and *elements*, and apply rules, to transform event source data into a format that the ObjectServer can recognize.

The following figure shows how probes use rules to process the event data that is acquired from the event source.

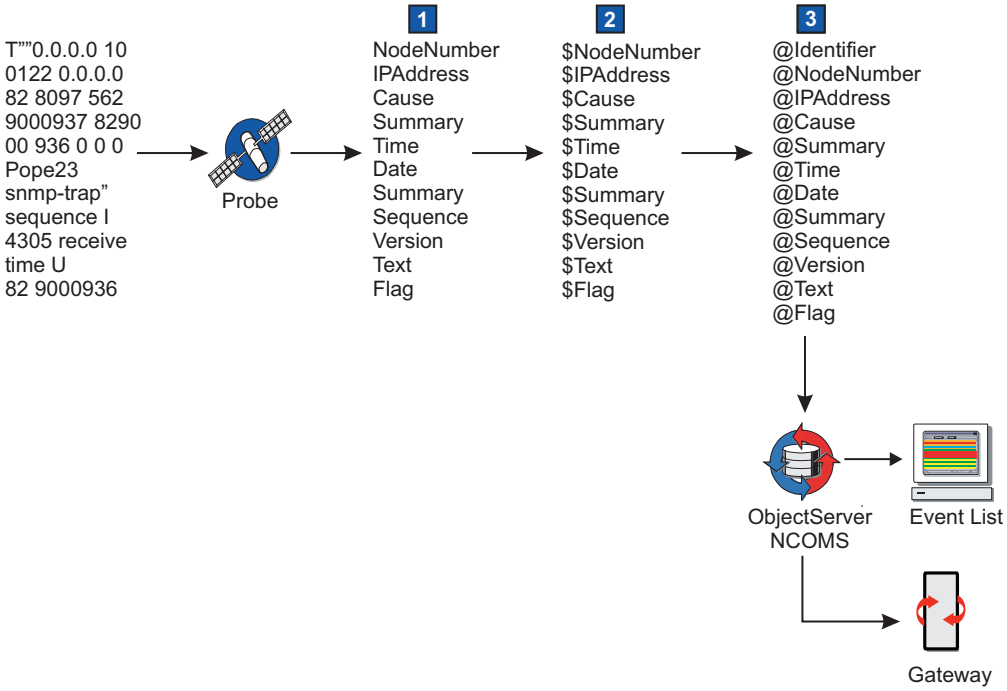

Figure 2. Event mapping using rules

The processing stages are as follows:

1. The raw event data that a probe acquires cannot be sent directly to the ObjectServer. The probe breaks the event data into tokens. Each token represents a piece of event data.
2. The probe then parses these tokens into elements and processes the elements according to the rules in the rules file. Elements are identified in the rules file by the $ symbol. For example, $Node is an element containing the node name of the event source.
3. Elements are used to assign values to ObjectServer fields, which are indicated by the @ symbol. The field values contain the event details in a form that the ObjectServer can understand. Fields make up the alerts that are forwarded to the ObjectServer, where they are stored and managed in the alerts.status table, and displayed in the event list.

   The Identifier field is also generated by the rules file.

Related concepts:

"How unique identifiers are constructed for events"
The Identifier field (@Identifier) uniquely identifies a problem source. Like other ObjectServer fields, the Identifier field is constructed from the tokens that the probe acquires from the event stream according to the rules in the rules file.

"Rules file" on page 7
The rules file defines how the probe processes event data to create a meaningful alert. For each alert, the rules file also creates an identifier that uniquely identifies the problem source.

How unique identifiers are constructed for events

The Identifier field (@Identifier) uniquely identifies a problem source. Like other ObjectServer fields, the Identifier field is constructed from the tokens that the probe acquires from the event stream according to the rules in the rules file.

The Identifier field allows the ObjectServer to correlate alerts so that duplicate alerts are displayed in the event list only once. Instead of inserting a new alert, the alert is *reinserted*; that is, the existing alert is updated. These updates are configurable. For example, the Tally field (@Tally) is typically incremented to keep track of the number of times that the event occurs.

It is essential that the identifier identifies repeated events appropriately. The following identifier is not specific enough, because any events with the same manager and node are treated as duplicates:

```
@Identifier=@Manager+@Node
```

If the identifier is too specific, the ObjectServer cannot correlate and deduplicate repeated events. For example, an identifier that contains a time value prevents correct deduplication.

The following identifier correctly identifies repeated events in a typical environment:

```
@Identifier=@Node+" "+@AlertKey+" "+@AlertGroup+" "+@Type+" "+@Agent+" "+@Manager
```

Event deduplication with probes

Deduplication is managed by the ObjectServer, but can be configured in the probe rules file. This enables you to set deduplication rules on a per-event basis. You can specify which fields of an alert are to be updated if the alert is deduplicated using the `update` function.

Related concepts:

"Probe architecture" on page 9
The function of a probe is to acquire information from an event source and forward it to the ObjectServer. Probes use *tokens* and *elements*, and apply rules, to transform event source data into a format that the ObjectServer can recognize.

Related reference:

Chapter 2, "Probe rules file syntax," on page 19
The rules file defines how the probe should process event data to create a meaningful Tivoli Netcool/OMNIbus alert. The rules file also creates an identifier for each alert to uniquely identify the problem source, so that repeated events can be deduplicated.

"Update on deduplication function" on page 45
The ObjectServer manages the deduplication process, but you can also configure this process in the probe rules file. Use the `update` function to specify which fields of an alert are to be updated if the alert is deduplicated. This allows deduplication rules to be set on a per-alert basis.

Modes of operation of probes

You can configure probes to operate in a variety of modes, including store-and-forward mode, raw capture mode, secure mode, and peer-to-peer failover mode.

Store-and-forward mode for probes

Probes can continue to run if the target ObjectServer is down. During this period, the probe switches to *store* mode. The probe reverts to *forward* mode when the ObjectServer is functional again.

Automatic store and forward

By default, the store-and-forward mode is active only after a connection to the ObjectServer is established, used, and then lost. If the ObjectServer is not running when the probe starts, the store-and-forward mode is not triggered, and the probe terminates.

You can set the probe to run in *automatic* store-and-forward mode. In this mode, the probe goes straight into store mode if the ObjectServer is not running, on condition that the probe was connected to the ObjectServer at least once before. Enable automatic store-and-forward mode by using the `-autosaf` command-line option or the **`AutoSAF`** property.

Note: If the probe is trying to connect to a virtual pair of ObjectServers and both of the ObjectServers are down, the probe checks the **`AutoSAF`** property setting. If automatic store-and-forward is enabled, the probe begins to store events in the store-and-forward file; otherwise, the probe terminates.

Legacy store and forward

When the probe detects that the ObjectServer is not present (usually because it cannot forward an alert to the ObjectServer), the probe switches to store mode. In this mode, the probe writes all of the messages that it would normally send to the ObjectServer to a store-and-forward file. This file name is constructed by using the value of the **`SAFFileName`** property. A *.servername* extension is automatically appended to the **`SAFFileName`** value, where *servername* is the name of the

ObjectServer to which the probe is attempting to send alerts. If the probe is configured to send alerts to multiple ObjectServers, individual store-and-forward files are therefore created for each ObjectServer.

If corrupted records are identified in a store-and-forward file, these records are ignored and the probe will forward only the valid records to the ObjectServer. You can indicate whether to automatically save a file that contains corrupted records for future diagnosis by using the **KeepLastBrokenSAF** property. If you set this property to 1, the store-and-forward file that contains corrupted records is renamed *SAFFileName*.*servername*.broken. In this file name, *SAFFileName* is the value of the **SAFFileName** property and *servername* is the name of the ObjectServer to which the probe is attempting to send alerts. Any previous .broken file is overwritten.

You can also use the **StoreSAFRejects** property to continuously save the individual corrupted store-and-forward records for analysis. If the **StoreSAFRejects** is set to 1, the corrupted records are continuously saved to a *SAFFileName*.*servername*.rejected file.

Tip: The **SAFFileName**.*servername*.rejected file has an unlimited size, and must be manually deleted when no longer needed.

Legacy store and forward can be configured by using the properties in the following sample. **StoreAndForward** must be set to 1 for legacy store and forward. The other properties display default values that can be changed.

```
StoreAndForward:1
SAFFileName:'$OMNIHOME/var/SAF'
MaxSAFFileSize:1024
SAFPoolSize:3
```

Circular store and forward

Run the probe in circular store-and-forward mode to minimize event loss during failover and failback. In this mode, the probe stores all the alerts that it generates while it is connected to the ObjectServer. These alerts are stored in rolling store-and-forward files that roll over after a time interval set by the **RollSAFInterval** property. Set the **RollSAFInterval** property to a value that is equal to, or greater than, the granularity of the ObjectServer.

The circular store-and-forward files are named *SAFFileName*.*servername* and *SAFFileName*.*servername*_1.

When the probe is disconnected from the ObjectServer, the probe stores the timestamp of the last successful event and the ObjectServer name in a file. The file is named in the format *SAFFilename*.DisconnectionTime. This file is stored in the same directory as the store-and-forward files. If a backup ObjectServer is available for failover, the probe uses the disconnection time and the value of the **RollSAFInterval** property to calculate which events to send to the backup ObjectServer. For example, if the **RollSAFInterval** property is set to 90 seconds, all events from 90 seconds before the time of disconnection are replayed to the backup ObjectServer. Consequently, some events might be duplicated in the backup ObjectServer. The probe resends events that might already be sent to the primary ObjectServer, but were replicated in the backup ObjectServer before the primary ObjectServer went down.

If the probe is unable to connect to an ObjectServer, the probe automatically switches its handling of rolling store-and-forward files to the legacy

store-and-forward behavior. The probe starts storing all events in a pool of store-and-forward files, where the size of the pool is defined by the **SAFPoolSize** property, and the maximum file size is defined by the **MaxSAFFileSize** property. During this time, the **RollSAFInterval** property is not used to roll over the store-and-forward files; instead, each file rolls over when it reaches the size that is specified by **MaxSAFFileSize**.

Circular-store-and-forward can be configured by using the following properties. **StoreAndForward** must be set to 2 for circular store and forward; the other properties display default values that can be changed.

```
StoreAndForward:2
SAFFileName:'$OMNIHOME/var/SAF'
MaxSAFFileSize:1024
SAFPoolSize:3
RollSAFInterval:90
```

Summary of store-and-forward behavior

The following table summarizes how the ObjectServer status, and the combination of **AutoSAF** and **StoreAndForward** properties affect the behavior of probes.

Table 3. Store-and-forward summary

ObjectServer status before probe startup	AutoSAF property	StoreAndForward property	Expected result
ObjectServer down	0	0	The probe does not start.
ObjectServer down	0	1	The probe does not start.
ObjectServer down	1	0	The probe starts writing events into the store-and-forward file. When the ObjectServer comes up, the probe forwards the store-and-forward file events and then stops writing events to the store-and-forward file. If the ObjectServer gets disconnected later, events are not stored.
ObjectServer down	1	1	The probe starts writing events into the store-and-forward file. When the ObjectServer comes up, the probe forwards the store-and-forward file events. If the ObjectServer gets disconnected later, the probe stores events in store-and-forward files; these events are forwarded on reconnection.

Table 3. Store-and-forward summary (continued)

ObjectServer status before probe startup	AutoSAF property	StoreAndForward property	Expected result
ObjectServer up	No effect of property	0	The probe forwards events in existing store-and-forward files but it does not store any new events in store-and-forward files. If a probe loses its connection to the ObjectServer, it terminates in a controlled way. Any events that the probe received but did not transmit to the ObjectServer are lost. A probe that has multiple ObjectServer connections cannot run in store-and-forward mode `0` and is automatically switched to store-and-forward mode 1.
ObjectServer up	No effect of property	1	The probe forwards events in the store-and-forward files to the connected ObjectServer, and stores new events in the store-and-forward files only when disconnected from the ObjectServer.
ObjectServer up	No effect of property	2	The probe forwards events in the store-and-forward files to the connected ObjectServer, and stores new events in the rolling store-and-forward files when connected. The probe stores all events in a pool of files when disconnected.

Related reference:

"Multithreaded processing of alert data" on page 52
When a probe rules file is processed, multithreaded processing is used by default to apply probe rules to the raw event data that is acquired from the event source, and to send the generated alerts to the registered ObjectServers. Note that this multithreaded processing is different from the multithreaded or single-threaded event capture that is implemented in some classes of probes.

Chapter 6, "Common probe properties and command-line options," on page 113
A number of properties and command-line options are common to all probes and TSMs.

Raw capture mode for probes

You can use the raw capture mode to save the complete stream of event data acquired by a probe into a file, without any processing by the rules file. This can be useful for auditing, recording, or debugging the operation of a probe.

The captured data is in a format that can be replayed by the Standard Input Probe. See the publication for the Standard Input Probe for further information. You can access this publication as follows from the IBM Tivoli Network Management Information Center (http://publib.boulder.ibm.com/infocenter/tivihelp/v8r1/index.jsp):

1. Expand the *IBM Tivoli Netcool/OMNIbus* node in the navigation pane on the left.

2. Expand the *Tivoli Netcool/OMNIbus probes and TSMs* node.
3. Go to the *Universal* node.

To enable the raw capture mode, use the `-raw` command-line option or the **RawCapture** property. You can also set the **RawCapture** property in the rules file, so that you can send the raw event data to a file only when certain conditions are met.

Replay the raw captured data, using the Standard Input probe. A possible syntax is as follows:

```
cat <raw_capture_filename> | $OMNIHOME/probes/nco_p_stdin –server <server>
```

For example:

```
cat opt/Omnibus/var/mttrapd.cap | /opt/Omnibus/probes/nco_p_stdin -server NCOMS
```

The **RawCaptureFile**, **RawCaptureFileAppend**, and **MaxRawFileSize** properties also control the operation of the raw capture mode.

Related reference:

"Changing the value of the RawCapture property in the rules file" on page 22
Most probes read properties once at startup, so changing probe properties after startup does not usually affect probe behavior. However, you can set the **RawCapture** property in the rules file, so that you can send the raw event data to a file only when certain conditions are met.

Chapter 6, "Common probe properties and command-line options," on page 113
A number of properties and command-line options are common to all probes and TSMs.

Secure mode for probes

You can run the ObjectServer in secure mode. When you start the ObjectServer using the `-secure` command-line option, the ObjectServer authenticates probe, gateway, and proxy server connections by requiring a user name and password.

When a connection request is sent, the ObjectServer issues an authentication message. The probe, gateway, or proxy server must respond with the correct user name and password combination.

If the ObjectServer is not running in secure mode, probe, gateway, and proxy server connection requests are not authenticated.

Before running a probe that connects to a secure ObjectServer or proxy server, ensure that the **AuthUserName** and **AuthPassword** properties are set in the probe properties file, with values for the user name and password. If the user name and password combination is incorrect, the ObjectServer issues an error message and rejects the connection.

When in FIPS 140–2 mode, the password can either be specified in plain text, or can be encrypted with the **nco_aes_crypt** utility. If you are encrypting passwords by using **nco_aes_crypt** in FIPS 140–2 mode, you must specify `AES_FIPS` as the encryption algorithm.

When in non-FIPS 140–2 mode, the password can be encrypted with the **nco_g_crypt** or **nco_aes_crypt** utilities. If you are encrypting passwords by using **nco_aes_crypt** in non-FIPS 140–2 mode, you can specify either `AES_FIPS` or `AES` as

the encryption algorithm. Use AES only if you need to maintain compatibility with passwords that were encrypted using the tools provided in versions earlier than Tivoli Netcool/OMNIbus V7.2.1.

For further information about using the **nco_aes_crypt** utility, see the *IBM Tivoli Netcool/OMNIbus Installation and Deployment Guide*.

Related reference:

Chapter 6, "Common probe properties and command-line options," on page 113
A number of properties and command-line options are common to all probes and TSMs.

Peer-to-peer failover mode for probes

Two instances of a probe can run simultaneously in a peer-to-peer failover relationship. One instance is designated as the master. The other instance acts as a slave and is on hot standby. If the master instance fails, the slave instance is activated.

Note: Peer-to-peer failover is not supported for all probes. Probes that list the **Mode**, **PeerHost**, and **PeerPort** properties when you run the command `$OMNIHOME/probes/nco_p_probename -dumpprops` support peer-to-peer failover.

To set up a peer-to-peer failover relationship:

- For the master instance, set the **Mode** property to `master` and the **PeerHost** property to the network element name of the slave.
- For the slave instance, set the **Mode** property to `slave` and the **PeerHost** property to the network element name of the master.
- For both instances, set the **PeerPort** property to the port through which the master and slave communicate.

The master instance sends a heartbeat poll to the slave instance at the time interval specified by the **BeatInterval** property. The slave instance caches all the alert data it receives and deletes all alert data from the cache each time a heartbeat is received from the master instance. If the slave instance receives no heartbeat in the time period defined by the sum of the values of the **BeatInterval** and **BeatThreshold** properties (**BeatInterval** + **BeatThreshold**), the slave instance assumes that the master is no longer active, and forwards all alerts in the cache to the ObjectServer. The slave instance continues to forward all alerts until it receives another heartbeat from the original master instance. The timeout period while waiting for heartbeats is 1 second. So there can be a maximum delay of (**BeatInterval** + **BeatThreshold** + 1) seconds before the slave instance forwards its cached alerts. All alerts in the cache are sent.

The **BeatInterval** setting that is defined for the master instance takes precedence; the slave instance ignores its local **BeatInterval** setting.

To disable the peer-to-peer failover relationship, run a single instance of the probe with the **Mode** property set to `standard`. This is the default setting.

The failover mode of probes running in a peer-to-peer failover relationship is set in the properties files.

You can also switch the mode of a probe between master and slave in the rules file. There is a delay of up to one second before the mode change takes effect. This

can result in duplicate events if two probe instances are switching from `standard` mode to `master` or `slave`; however, no data is lost.

When the two probe instances running in store-and-forward mode are connected to a failover pair of ObjectServers, the master instance sends alerts to the primary ObjectServer. If the primary ObjectServer fails, the master instance of the probe fails over and starts sending alerts in its store-and-forward file to the backup ObjectServer. If the master instance of the probe fails, the slave instance takes over. If the slave instance fails to connect to the ObjectServer, the slave then creates a store-and-forward file for storing alert data. When the master instance is reactivated, any store-and-forward files in the master instance are deleted to prevent old alerts from being resent.

Example: Setting the peer-to-peer failover mode in the properties files

Example properties file values for the master are as follows:

```
PeerPort: 9999
PeerHost: "slavehost"
Mode: "master"
```

Example properties file values for the slave are as follows:

```
PeerPort: 9999
PeerHost: "masterhost"
Mode: "slave"
```

Example: Setting the peer-to-peer failover mode in the rules file

To switch a probe instance to become the master, use the rules file syntax:

```
%Mode = "master"
```

Chapter 2. Probe rules file syntax

The rules file defines how the probe should process event data to create a meaningful Tivoli Netcool/OMNIbus alert. The rules file also creates an identifier for each alert to uniquely identify the problem source, so that repeated events can be deduplicated.

Related concepts:

"How unique identifiers are constructed for events" on page 10
The Identifier field (@Identifier) uniquely identifies a problem source. Like other ObjectServer fields, the Identifier field is constructed from the tokens that the probe acquires from the event stream according to the rules in the rules file.

"Rules file" on page 7
The rules file defines how the probe processes event data to create a meaningful alert. For each alert, the rules file also creates an identifier that uniquely identifies the problem source.

Rules file development guidelines

Use the following guidelines to develop rules files with a consistent format and structure:

- Rules files must be of production quality and not require any additional modification.
- Rules files must not result in any additional modifications to the ObjectServer. That is, there must be no additional event fields other than those provided by Tivoli Netcool/OMNIbus.
- The basic structure of the rules files must be both easily maintainable and easily extendible, therefore enabling the addition of event handling for new devices without affecting existing rules.
- The basic textual-conventions used for the rules files must be consistent and therefore ensure that all newly created rules files share a common format.
- The rules files must be clearly documented to allow each event to be recognized without the need for any additional documentation.
- The events formatted by the rules files must be deduplicated properly by the Tivoli Netcool/OMNIbus ObjectServer. The Identifier field (@Identifier) must be set correctly to enable the granularity of deduplication to be directly controlled.
- The events formatted by the rules files must be compatible, whenever possible, with the ObjectServer's GenericClear Automation.
- Always make a backup copy of a rules file before modifying it. Save the file as a `.rules.date` file. For example, `snmp.rules.070131`. You need this file in case you have to perform a rollback.
- Any changes to the rules file must be commented out with a number sign (#) at the beginning of the line.
- Use the `details` ($*) function only when debugging rules files or writing rules files. After using `details` ($*) for long periods of time, the ObjectServer tables become very large and the performance of the ObjectServer suffers.
- The SWITCH and CASE constructs are processed more efficiently, and must therefore be used in preference to the IF and ELSE statements.
- Use lookup tables wherever possible. When multiple values are linked to a single key, use a multi-column lookup table. Lookup tables must be defined

within an external file based table, specified with a `.lookup` file extension. This enables clear identification of the lookup tables. Additionally, the lookup tables must be the first elements of the rules files that are read by the probe. That is, in the basic rules file, locate the lookup file `include` statement at the top.

- Matching pairs of problem and resolution events must have identical `@AlertGroup` and `@AlertKey` values, and appropriate `@Type` and `@Severity` values.
- A Resolution event must have a severity alert of `1` (indeterminate) and a type of `2` (resolution). Do not set the severity of resolution events to `0` (Clear). This would prevent events being processed by the ObjectServer's GenericClear Automation.
- Rules file must be created in the same character encoding that is used by the locale of the probe's runtime environment. The character encoding must be supported by the International Components for Unicode (ICU) libraries.

Related tasks:

"Testing rules files" on page 58
You can test the syntax of a rules file by using the Probe Rules Syntax Checker, **`nco_p_syntax`**. This is more efficient than running the probe to test that the syntax of the rules file is correct.

Elements, fields, properties, and arrays in rules files

A probe takes an event stream and parses it into elements. Event elements are processed by the probe based on the logic in the rules file. Elements are assigned to fields and forwarded to the ObjectServer, where they are inserted as alerts into the alerts.status table.

The Identifier field, used by the ObjectServer for deduplication, is also created based on the logic in the rules file.

Elements are indicated by the `$` symbol in the rules file. For example, $Node is an element containing the node name of the event source. You can assign elements to ObjectServer fields, indicated by the `@` symbol in the rules file.

Note: The normal format for referring to elements works only if the name of the element contains only letters, digits, and underscores. If a probe dynamically generates element names, it is possible to generate elements that contain other characters. You can refer to elements such as these by putting the element name inside parentheses; for example, $(strange=name).

Related concepts:

"How unique identifiers are constructed for events" on page 10
The Identifier field (@Identifier) uniquely identifies a problem source. Like other ObjectServer fields, the Identifier field is constructed from the tokens that the probe acquires from the event stream according to the rules in the rules file.

Assigning values to ObjectServer fields

You can assign values to ObjectServer fields by direct assignment, concatenation, or by adding text.

Examples are as follows:

- Direct assignment example: `@Node = $Node`
- Concatenation example: `@Summary = $Summary + $Group`
- Adding text example: `@Summary = $Node + "has problem" + $Summary`

You can express numeric values in decimal or hexadecimal form. The following statements, which set the Class field to `100`, are equivalent:

- `@Class=100`
- `@Class=0x64`

In addition to assigning elements to fields, you can use processing statements, operators, and functions to manipulate these values in rules files before assigning them.

Tip: Elements are stored as strings, so you must use the `int` function to convert elements into integers before performing numeric operations.

Related reference:

"Math functions" on page 40
You can use math functions to perform numeric operations on elements. Elements are stored as strings, so you must use these functions to convert elements into integers before performing numeric operations.

Assigning temporary elements in rules files

You can create a temporary element in a rules file by assigning it to an expression.

For example:

```
$tempelement = "message"
```

An element, `$tempelement`, is created and assigned the string value `message`.

If you refer to an element that has not been initialized in this way, the element is set to the null string (`""`).

The following example creates the element `$b` and sets it to `setnow`:

```
$b="setnow"
```

The following example then sets the element `$a` to `setnow`:

```
$a=$b
```

In the following example, temporary elements are used to extract information from a `Summary` element, which has the string value: `The Port is down on Port 1 Board 2`.

```
$temp1 = extract ($Summary, "Port ([0-9]+)")
$temp2 = extract ($Summary, "Board ([0-9]+)")
@AlertKey = $temp1 + "." + $temp2
```

The `extract` function is used to assign values to temporary elements `temp1` and `temp2`. Then these elements are concatenated (using the `+` concatenate operator)

with a `.` separating them, and assigned to the AlertKey field. After these statements are run, the AlertKey field has the value `1.2`.

Related reference:

"String functions" on page 36
You can use string functions to manipulate string elements, typically field or element names.

"Math and string operators" on page 33
You can use math operators to add, subtract, divide, and multiply numeric operands in expressions. You can use string operators to manipulate character strings.

Assigning property values to fields

You can assign the value of a probe property, as defined in the properties file or on the command line, to a field value. A property is represented by a `%` symbol in the rules file.

For example, you can add the following statement to your rules file:

```
@Summary = "Server = " + %Server
```

In this example, when the rules file is processed, the probe searches for a property named **`Server`**. If the property is found, its value is concatenated to the text string and assigned to the Summary field. If the property is not found, a null string (`""`) is assigned.

Assigning values to properties

You can assign values to a property in the rules file. If the property does not exist, it is created.

For example, you can create a property called **`Counter`** to keep track of the number of events that have been processed as follows:

```
if (match(%Counter,""))
 {%Counter = 1}
else {%Counter = int(%Counter) + 1}
```

These properties retain their values across events and when the rules file is re-read.

Changing the value of the RawCapture property in the rules file

Most probes read properties once at startup, so changing probe properties after startup does not usually affect probe behavior. However, you can set the **`RawCapture`** property in the rules file, so that you can send the raw event data to a file only when certain conditions are met.

The setting for the raw capture mode takes effect for the current event.

For example:

```
# Start rules processing
%RawCapture=0

if (condition) {
 # Send the current event to the raw capture file
 %RawCapture=1
}
```

You can enable raw capture mode globally by setting the `-raw` command-line option or the **`RawCapture`** property in the probe properties file.

Related concepts:

"Raw capture mode for probes" on page 14
You can use the raw capture mode to save the complete stream of event data acquired by a probe into a file, without any processing by the rules file. This can be useful for auditing, recording, or debugging the operation of a probe.

Related reference:

"Control statements in rules files"
The IF, SWITCH, FOREACH, and BREAK statements provide control flow for processing rules files.

Using arrays

You must define arrays at the start of a rules file, before any processing statements.

Tip: You must also define tables, and target ObjectServers, before any processing statements.

To define an array, use the following syntax:

```
array node_arr
```

Arrays are one dimensional. Each time an assignment is made for a key value that already exists, the previous value is overwritten. For example:

```
node_arr["myhost"] = "a"
node_arr["yourhost"] = "b"
node_arr["myhost"] = "c"
```

After the preceding statements are run, there are two items in the `node_arr` array. The item with the key `myhost` is set to `c`, and the item with the key `yourhost` is set to `b`. You can make assignments using probe elements, for example:

```
node_arr[$Node] = "d"
```

Note: Array values are persistent until a probe is restarted. If you force the probe to re-read the rules file by issuing a `kill -HUP pid` command on the probe process ID, the array values are maintained.

Related reference:

"Lookup table operations" on page 43
Lookup tables provide a way to add extra information in an event. A lookup table consists of a list of keys and values.

"Sending alerts to alternative ObjectServers and tables" on page 47
The `registertarget`, `genevent`, `settarget`, and `setdefaulttarget` functions enable you to send alerts to one or more ObjectServers, and to define the distribution of alerts across the ObjectServers.

Control statements in rules files

The IF, SWITCH, FOREACH, and BREAK statements provide control flow for processing rules files.

FOREACH statement

Use the FOREACH statement to write statements in the probe rules file language that iterate through lists of event elements or table entries.

Syntax

The syntax of the FOREACH statement is as follows:

```
foreach (iterator in list)
{
 statements
}
```

In this syntax, *iterator* represents the item to be identified by the statement. *iterator* can consist of any combination of printable ASCII characters, and must start with a letter. For example: *letter* {*letter*|*digit*}.

list represents the elements to be processed in the rules file. *list* can be a comma-delimited list of elements or an array. You can use $* to instruct the loop to process all elements. The statement processes the elements in the *list* in a non-determinstic order.

statements represents valid probe rules file statements or functions that are to be applied to the elements in *list*.

You can nest FOREACH statements inside each other. The FOREACH statement supports IF and SWITCH statements in the body of a loop.

In a statement the *iterator* represents the current loop item. Referencing its value depends on the type of list that is being processed.

When looping through a list of elements, note the following information:

- You must prefix the *iterator* with $ to reference the current element.
- If used on its own, the *iterator* represents the name of the current element.

When looping through an array, note the following information:

- You must substitute the *iterator* for the key in referencing an array item.
- On its own, the *iterator* represents a string that is the key to the current array item.

Supported probe rules file functions

All probe rules file functions are supported in a FOREACH statement.

Restrictions

You cannot use the FOREACH statement to iterate through properties, fields, or columns. You also cannot use the statement to iterate through a combination of arrays and elements in the same loop.

If you use the `details()` function in a FOREACH loop, only the result of the last-executed `details()` function are stored in the ObjectServer. Additionally, because the FOREACH statement processes the elements in the rules file in a non-determinstic order, it cannot be predicted which element is stored.

Examples of the looping function

Use these examples of the FOREACH looping statement to help you deploy the function in your Tivoli Netcool/OMNIbus environment.

Example 1: Looping through all elements

The following example shows how to use the $* wildcard to process all elements in a loop:

```
foreach ( e in $* )
{
 log( INFO, "The value of $" + e + " = " + $e)
}
```

This statement would return messages for all elements. Each message is similar to the following example:

```
Information: I-UNK-000-000: The value of $DateString = 12/04/10 16:39:50.
```

Example 2: Looping through a comma delimited list

The following example shows how to convert the $Node, $Agent, and $Group tokens to lower case:

```
foreach ( e in $Node, $Agent, $Group )
{
 $e = lower( $e )
}
```

Example 3: Loop through entries in an array

In the following example, an array called “names” is defined at the top of the rules file. During an iteration of the loop the `key` contains the key to the current entry in “names”. The result of this loop is that each entry in “names” is prefixed with `XX`.

```
array names
....
foreach ( key in names )
{
 log( INFO, "Before: The value of names[" + key + "] = " + names[key])
 names[key] = "XX" + names[key]
}
```

Example 4: Using IF with BREAK

The following example loops through all the elements until an element is found that has a value prefixed with `http://`. The URL field is set with this value and the execution breaks out of the loop.

```
foreach (e in $* )
{
 if ( nmatch( $e, "http://" ) )
 {
  @URL = $e
  break
 }
}
```

For more information, see “BREAK statement” on page 30.

Example 5: Using IF inside a FOREACH loop on SNMP OID fields

The following example proposes a solution for handling differences between SNMP V1 and V2:

1. In the first loop, *x* always contains the name of the current element. If the name begins with OID, the value of the element is added to the working array "oids."
2. The key for an entry is prefixed with "OID" followed by a number which is one less than the number used in the name of the original element.
3. The original element is removed.

The result is that the OID elements are now all moved to the left by one element, that is `$OID1 = $OID2, $OID2 = $OID3`, and so on.

Note: This example does not provide a complete solution for handling SNMP V2 and V1 traps.

```
# Declare an empty array
array oids

# Loop through all elements.
foreach ( x in $* )
{
 # Find elements whose names start with 'OID'
 if( nmatch( x, "OID" ) )
 {
 # Extract the OID number from the element name
 # Save the element value in the 'oids' array.
 # oids[ "OID1" ] = $OID2
 # oids[ "OID2" ] = $OID3 etc.
 $n=extract( x, "OID([0-9]+)" )
 if( int( $n ) > 1 )
  {
  $n=int($n)-1
  oids["OID"+$n]=$x
  }
 # Delete original OID element
 remove( $x )
 }
}

# Create new 'OID' elements
foreach ( x in oids )
{
    $x=oids[x]
}
clear (oids)
```

Example 6: Using IF inside a FOREACH loop to handle EIF elements

The following example shows how to use the FOREACH statement to remove single quotation marks (') surrounding any elements in EIF messages:

```
foreach ( e in $* )
{
 if(regmatch($e, "^'.*'$"))
 {
  $e = extract($e, "^'(.*)'$")
  log(DEBUG,"Colons removed from Token " + $e)
 }
}
```

Example 7: Nested loops

The following example shows how to translate elements that contain encoded Octet strings (dot-separated integers) and translate the strings to ASCII text:

```
array octets
table Ascii2Txt =
{
 {"0",""},
 {"9","    "},
 {"32"," "},
 {"33","!"},
. . .
 {"125","}"},
 {"126","~"}
}

foreach ( e in $* )
{
 $n = split( $e, octets, "." )
 $e = ""
 foreach ( n in octets )
 {
  $e = $e + lookup( octets[n], Ascii2Txt )
 }
 clear( octets )
}
```

Example 8: Using the FOREACH statement to parse name-value elements

In the following example, the contents of $input represent a set of name-value pairs separated by semi-colons (;). The example creates new elements from the name-value pairs.

```
array pairs
array values
$input="foo=blah;wibble=wobble"
$num = split( $input, pairs, ";" )
foreach ( t in pairs )
{
 $n = extract( pairs[t], "(.*)=" )
 $v = extract( pairs[t], ".*=(.*)" )
 values[ $n ] = $v
}
remove( $n )
remove( $v )
foreach ( t in values )
{
  $t = values[t]
}
```

Example 9: Using the FOREACH statement to load name-value pairs into the @ExtendedAttr field

To create name-values pairs of all current elements and load them into the @ExtendedAttr field, use the following statement:

```
@ExtendedAttr = nvp_add($*)
```

For only a subset of elements, use a statement as shown in the following example:

```
foreach ( e in $interface, $network, $ipaddr, $netmask, $gateway )
{
 @ExtendedAttr = nvp_add( @ExtendedAttr, e, $e )
}
```

In this example, the statement makes use of the fact that `e` represents the name of the element, and `$e` represents the value of the element. This example would populate the @ExtendedAttr field with data similar to the following sample:

```
interface="eth0";network="178.268.2.0";ipaddr="178.268.2.64";
netmask="233.233.233.0";gateway="178.268.2.1"
```

Example 10: Using the FOREACH statement to selectively remove name-value pairs from the @ExtendedAttr field

Similarly to "Example 9: Using the FOREACH statement to load name-value pairs into the @ExtendedAttr field" on page 27, the FOREACH statement can be used to remove selected name-value pairs from the @ExtendedAttr field, as shown in the following example:

```
@ExtendedAttr = nvp_add( $* )
foreach ( e in $network, $netmask )
{
 @ExtendedAttr = nvp_remove( @ExtendedAttr, e )
}
```

Example 11: Using the SWITCH and BREAK statements in a FOREACH loop

The following example shows how to use a BREAK statement in a SWITCH statement to terminate the processing of a FOREACH loop:

```
foreach ( x in $*)
{
 switch( $x ):
 {
  case "1":
   statements
  case "2":
   statements
  case "3":
   statements
  default:
   log (ERROR, "Unexpected element $" + x + " = " + $x)
   break
 }
}
```

For more information, see "SWITCH statement" on page 29 and "BREAK statement" on page 30.

Related reference:

"Rules file examples" on page 63
These examples show typical rules file segments.

IF statement

A condition is a combination of expressions and operations that resolve to either `TRUE` or `FALSE`. The IF statement allows conditional running of a set of one or more assignment statements by running only the rules for the condition that is `TRUE`.

The IF statement has the following syntax:

```
if (condition) {
 rules
} [ else if (condition) {
 rules
```

```
} ... ]
 [ else (condition) {
 rules
} ]
```

You can combine conditions into increasingly complex conditions using the logical AND operator (&&), which is true only if *all* of its inputs are true, and OR operator (||), which is true if *any* of its inputs are true. For example:

```
if match ($Enterprise, "Acme") && match ($trap-type, "Link-Up")
{ @Summary = "Acme Link Up on " + @Node }
```

Related reference:

"Logical operators" on page 35
You can use logical operators on Boolean values to form expressions that resolve to TRUE or FALSE.

"String functions" on page 36
You can use string functions to manipulate string elements, typically field or element names.

SWITCH statement

A SWITCH statement transfers control to a set of one or more rules assignment statements depending on the value of an expression.

The SWITCH statement has the following syntax:

```
switch (expression) {
case "stringliteral":
 rules
case "stringliteral":
 rules
...
default:
 [rules]
}
```

The *expression* can be any valid expression. For example:

```
switch($node)
```

The *stringliteral* can be any string value. For example:

```
case "jupiter":
```

You can have more than one *stringliteral* separated by the pipe (|) symbol. For example:

```
case "jupiter" | "mars" | "venus":
```

This case runs if the expression matches any of the specified strings.

The SWITCH statement tests for exact matches only. Wherever possible, use this statement instead of an IF statement because SWITCH statements are processed more efficiently and therefore run more quickly.

Any rules in the DEFAULT case are run if no other case is matched. Each SWITCH statement must contain a default case, even if there are no rules associated with it. There is no fall through from one case to another.

The behaviour of a BREAK statement in a SWITCH statement case is identical to the behaviour of a BREAK statement inside an IF statement. If the SWITCH statement is inside the body of a loop statement then the process will exit the loop.

If the SWITCH statement is not part of a loop body then the rules processing of the event is terminated at that point and the event is sent on to the ObjectServer.

BREAK statement

Use the BREAK statement in conjunction with the FOREACH statement to break out of the processing of a loop before the loop is completely processed.

The behavior of the BREAK statement is as follows:

- If the BREAK statement is contained in a FOREACH statement, when the BREAK statement is processed, processing of the FOREACH loop is terminated immediately. Processing continues with the next statement after the FOREACH statement. If the statement contains nested FOREACH statements, only the innermost loop containing the BREAK statement is exited.
- If the BREAK statement is outside of a FOREACH statement, the BREAK statement terminates the processing of the rules for the current event. No more rules are processed after the BREAK but the event is still sent to the ObjectServer (unlike the `discard` function).

For additional information, see "Examples of the looping function" on page 25.

Embedding multiple rules files in a rules file

You can include a number of secondary rules files in your main rules file by using the `include` statement.

The format is as follows:

```
include "rulesfile"
```

Specify the path to the rules file as an absolute or relative path. Relative paths start from the current rules file directory. You can use environment variables in the path, as follows:

```
if(match(@Manager, "ProbeWatch"))
 { include "$OMNIHOME/probes/solaris2/probewatch.rules" }
else ...
```

If you want to include a remote probe rules file that is stored on an IPv6 Web server, use square brackets [] to delimit the IPv6 address in the web address. For example:

```
include "http://[fed0::7887:234:5edf:fe65:348]:8080/probewatch.rules"
```

Rules file functions and operators

You can use operators and functions to manipulate elements in rules files before assigning them to ObjectServer fields.

The following table lists the rules file operators.

Table 4. Rules file operators

Operators	Description	Further details
*, /, -, +	Perform math and string operations.	"Math and string operators" on page 33
&, |, ^, >>, <<	Perform bitwise operations.	"Bit manipulation operators" on page 34

Table 4. Rules file operators (continued)

Operators	Description	Further details
==, !=, <>, <, >, <=, >=	Perform comparison operations.	"Comparison operators" on page 35
NOT (also !), AND (also &&), OR (also ||), XOR (also ^)	Perform logical (Boolean) operations.	"Logical operators" on page 35

The following table lists the rules file functions.

Table 5. Rules file functions

Function name	Description	Further details
`charcount`	Returns the number of characters in a string.	"String functions" on page 36
`clear`	Removes the elements of an array.	"String functions" on page 36
`datetotime`	Converts a string into a time data type.	"Date and time functions" on page 41
`details`	Adds information to the alerts.details table.	"Details function" on page 45
`discard`	Deletes an entire event.	"Elements and event functions" on page 36
`exists`	Tests for the existence of an element.	"Existence function" on page 35
`expand`	This function is deprecated and must not be used as the regular expression argument in the `regmatch` or `extract` functions. Instead of using `expand`, contain the regular expression in single quotes, for example: `'[\n\r]'`	"String functions" on page 36
`extract`	Returns the part of a string (which can be a field, element, or string expression) that matches the parenthesized section of the regular expression.	"String functions" on page 36
`genevent`	Enables you to: • Create and send an alert from a rules file to a target ObjectServer. • Send the same alert to more than one ObjectServer or table.	"Sending alerts to alternative ObjectServers and tables" on page 47
`getdate`	Returns the current date as a date data type.	"Date and time functions" on page 41
`getenv`	Returns the value of an environment variable.	"Host and process utility functions" on page 42
`geteventcount`	Returns the number of events in the event window.	"Monitoring probe loads" on page 55
`gethostaddr`	Returns the IP address of the host by using a naming service.	"Host and process utility functions" on page 42
`gethostname`	Returns the name of the host by using a naming service.	"Host and process utility functions" on page 42

Table 5. Rules file functions (continued)

Function name	Description	Further details
`getload`	Measures the load on the ObjectServer.	"Monitoring probe loads" on page 55
`getpid`	Returns the process ID of a running probe.	"Host and process utility functions" on page 42
`getplatform`	Returns the operating system platform the probe is running on.	"Host and process utility functions" on page 42
`hostname`	Returns the name of the host on which the probe is running.	"Host and process utility functions" on page 42
`int`	Converts a numeric value into an integer.	"Math functions" on page 40
`length`	Returns the number of bytes in a string.	"String functions" on page 36
`log`	Enables you to log messages.	"Message logging functions" on page 46
`lookup`	Uses a lookup table to map additional information to an alert.	"Lookup table operations" on page 43
`lower`	Converts an expression to lowercase.	"String functions" on page 36
`ltrim`	Removes white space from the left of an expression.	"String functions" on page 36
`match`	Tests for an exact string match.	"String functions" on page 36
`nmatch`	Tests for a string match at the beginning of a specified string.	"String functions" on page 36
`nvp_add`	Enables probes to generate events that contain extended attributes, which are supplied as name-value pairs.	"String functions" on page 36
`nvp_remove`	Used with extended attributes. Removes specified keys from a name-value pair string, and returns the new name-value pair string.	"String functions" on page 36
`printable`	Converts any non-printable characters in an expression to a space character.	"String functions" on page 36
`real`	Converts a numeric value into a real number.	"Math functions" on page 40
`recover`	Recovers a discarded event.	"Elements and event functions" on page 36
`regmatch`	Performs full regular expression matching of a value in a regular expression in a string.	"String functions" on page 36
`regreplace`	Uses regular expressions to perform search and replace operations on strings.	"Search and replace function" on page 53
`remove`	Removes an element from an event.	"Elements and event functions" on page 36

Table 5. Rules file functions (continued)

Function name	Description	Further details
`registertarget`	Registers an ObjectServer so alerts can be sent to multiple ObjectServers.	"Sending alerts to alternative ObjectServers and tables" on page 47
`rtrim`	Removes white space from the right of an expression.	"String functions" on page 36
`scanformat`	Converts an expression according to the available formats, similar to the `scanf` family of routines in C.	"String functions" on page 36
`setlog`	Enables you to set the message log level.	"Message logging functions" on page 46
`settarget`, `setdefaulttarget`	Sets the ObjectServer to which alerts are sent.	"Sending alerts to alternative ObjectServers and tables" on page 47
`service`	Sets the status of a service.	"Service function" on page 54
`split`	Separates a string into elements of an array.	"String functions" on page 36
`substr`	Extracts a substring from an expression.	"String functions" on page 36
`table`	Defines a lookup table.	"Lookup table operations" on page 43
`timetodate`	Converts a time value into a string data type.	"Date and time functions" on page 41
`toBase(`*`numeric`*`,`*`numeric`*`)`	Converts a decimal numeric value into a different base.	"Math functions" on page 40
`update`	Indicates which fields are updated when an alert is deduplicated.	"Update on deduplication function" on page 45
`updateload`	Updates the load statistics for the ObjectServer.	"Monitoring probe loads" on page 55
`upper`	Converts an expression to uppercase.	"String functions" on page 36

Math and string operators

You can use math operators to add, subtract, divide, and multiply numeric operands in expressions. You can use string operators to manipulate character strings.

The following table describes the math operators supported in rules files.

Table 6. Math operators

Operator	Description	Example
`*` `/`	Operators used to multiply (`*`) or divide (`/`) two operands.	`$eventid=int($eventid)*2`

Table 6. Math operators (continued)

Operator	Description	Example
+ -	Operators used to add (+) or subtract (-) two operands.	`$eventid=int($eventid)+1`

The following table describes the string operator supported in rules files.

Table 7. String operator

Operator	Description	Example
+	Concatenates two or more strings.	`@field = $element1 + "message" + $element2`

Bit manipulation operators

You can use bitwise operators to manipulate integer operands in expressions.

The following table describes the bitwise operators supported in rules files.

Table 8. Bitwise operators

Operator	Description	Example
`& | ^`	Bitwise `AND` (`&`), `OR` (`|`), and `XOR` (`^`). The results are determined bit-by-bit.	`$result1 = int($number1) & int($number2)`
`>> <<`	Shifts bits right (`>>`) or left (`<<`).	`$result2 = int($number3) >> 1`

These operators manipulate the bits in integer expressions. For example, in the statement:

```
$result2 = int($number3) >> 1
```

If `number3` has the value `17`, `result2` resolves to `8`, as shown:

```
  16 8 4 2 1
>>  1 0 0 0 1
    0 1 0 0 0
```

Note: The bits do not wrap around. When they drop off one end, they are replaced on the other end by a `0`.

Bitwise operators only work with integer expressions. Elements are stored as strings, so you must use the `int` math function to convert elements into integers before performing these operations.

For more information about the bitwise operators supported in ObjectServer SQL, see the *IBM Tivoli Netcool/OMNIbus Administration Guide*.

Related reference:

"Math functions" on page 40
You can use math functions to perform numeric operations on elements. Elements are stored as strings, so you must use these functions to convert elements into integers before performing numeric operations.

Comparison operators

You can use comparison operators to test numeric values for equality and inequality.

The following table describes the comparison operators supported in rules files.

Table 9. Comparison operators

Operator	Description	Example
==	Tests for equality.	`int($eventid) == 5`
!= <>	Tests for inequality.	`int($eventid) != 0`
< > <= >=	Tests for greater than (>), less than (<), greater than or equal to (>=), or less than or equal to (<=).	`int($eventid) <=30`

Logical operators

You can use logical operators on Boolean values to form expressions that resolve to `TRUE` or `FALSE`.

The following table describes the logical operators supported in rules files.

Table 10. Logical operators

Operator	Description	Example
NOT (also `!`)	A NOT expression negates the input value, and is TRUE only if its input is FALSE.	`if NOT(Severity=0)`
AND (also `&&`)	An AND expression is true only if all of its inputs are TRUE.	`if match($Enterprise,"Acme") && match($trap-type,"Link-Up")`
OR (also `||`)	An OR expression is TRUE if any of its inputs are TRUE.	`if match($Enterprise,"Acme") \|\| match($Enterprise,"Bo")`
XOR (also `^`)	An XOR expression is TRUE if either of its inputs, but not both, are TRUE.	`if match($Enterprise,"Acme") XOR match($Enterprise,"Bo")`

Existence function

You can use the `exists` function to test for the existence of an element.

Use the following syntax:

```
exists ($element)
```

The function returns `TRUE` if the element was created for this particular event; otherwise it returns `FALSE`.

Elements and event functions

You can use functions to remove elements from an event, discard an entire event, and recover a discarded event.

The following table describes these functions.

Table 11. Deleting elements or events

Function	Description	Example
`discard`	Deletes an entire event. **Note:** This must be in a conditional statement; otherwise, all events are discarded.	`if match(@Node,"testnode") { discard }`

Fix Pack 1 `discarded`	Tests whether the current alert is flagged for discarding. This condition returns `TRUE` if the alert is flagged for discarding. If the alert is not flagged for discarding, `FALSE` is returned.	`if( discarded )`
		`{`
		`log( DEBUG, "Alert from Node=["+@Node+"] has been marked for discard" )`
		`}`

`recover`	Recovers a discarded event.	`if match(@Node,"testnode") { recover }`
`remove(`*`element_name`*`)`	Removes the element from the event.	`remove(`*`test_element`*`)`

String functions

You can use string functions to manipulate string elements, typically field or element names.

The following table describes the string functions supported in rules files.

Table 12. String functions

Function	Description	Example
`charcount(`*`expression`*`)`	Returns the number of characters in a string. **Note:** When using single byte characters, this will be the same as the number returned by the `length()` function. When using multi-byte characters, this number can differ from that returned by the `length()` function.	`$NumChar = charcount($Node)`
`clear`	Removes the elements of an array.	`clear(`*`array_name`*`)`

Table 12. String functions (continued)

Function	Description	Example

`expand("string")`	Returns the string (which must be a literal string) with escape sequences expanded. Possible expansions are:
	
	`\"` - double quote
	
	`\NNN` - octal value of `NNN`
	
	`\\` - backslash
	
	`\a` - alert (BEL)
	
	`\b` - backspace
	
	`\e` - escape (033 octal)
	
	`\f` - form feed
	
	`\n` - new line
	
	`\r` - carriage return
	
	`\t` - horizontal tab
	
	`\v` - vertical tab
	
	This function cannot be used as the regular expression argument in the `regmatch` or `extract` functions.
	Note: This function is deprecated and must not be used as the regular expression argument in the `regmatch` or `extract` functions. Instead of using `expand`, contain the regular expression in single quotes, for example:
	
	`'[\n\r]'`
		`log(debug, expand("Rules file with embedded \\\""))`
		
		sends the following to the log:
		
		`Sun Oct 21 19:56:15 2001 Debug: Rules file with embedded \"`

`extract(string, "regexp")`	Returns the part of the string (which can be a field, element, or string expression) that matches the parenthesized section of the regular expression.	`extract ($expr,"ab([0-9]+)cd")` If `$expr` is `"ab123cd"` then the value returned is `123`.
`length(expression)`	Returns the number of bytes in a string.	`$NodeLength = length($Node)`
`lower(expression)`	Converts an expression to lowercase.	`$Node = lower($Node)`
`ltrim(expression)`	Removes white space from the left of an expression.	`$TrimNode = ltrim($Node)`
`match(expression, "string")`	`TRUE` if the expression value matches the string exactly.	`if match($Node, "New")`
`nmatch(expression, "string")`	`TRUE` if the expression starts with the specified string.	`if nmatch($Node, "New")`

Table 12. String functions (continued)

Function	Description	Example

nvp_add(*string_nvp*, *$name*, *$value* [, *$name2*, *$value2*,]*)
 	
nvp_add($*)
	Creates or updates a name-value pair string of extended attributes. Multiple name-value pairs can be supplied for the string. Variables and their values can be added to, or replaced in, the name-value pair string.
	
	Creates a name-value pair string of all variables and their values when called as nvp_add($*).
		if (int($PercentFull) > 95)
		{
		@Severity = 5
		@ExtendedAttr = nvp_add(@ExtendedAttr, "PercentFull", $PercentFull, "Disk", $Disk)
		}
		
		If $PercentFull is 97 and $Disk is /dev/sfa1, @ExtendedAttr will be (assuming it was initially empty):
		PercentFull="97";Disk="/dev/sfa1"

nvp_remove(*string_nvp*, *string_key1* [, *string_key2*, [...]])	Used with extended attributes. Removes specified keys from a name-value pair string, and returns the new name-value pair string. Useful where the list of extended attributes to include is longer than the list of attributes to exclude. You can use nvp_add($*) to include all variables and their values, and then use nvp_remove to remove specific ones.	$interface = "eth0" $network = "178.268.2.0" $ipaddr = "178.268.2.64" $netmask = "233.233.233.0" $gateway = "178.268.2.1" @ExtendedAttr = nvp_add($*) @ExtendedAttr = nvp_remove(@ExtendedAttr, "network", "netmask") This results in @ExtendedAttr being: interface="eth0";ipaddr="178.268.2.64";gateway="178.268.2.1"
printable(*expression*)	Converts any non-printable characters in the given expression into a space character.	$Print = printable($Node)
regmatch(*expression*, "*regexp*")	Full regular expression matching.	if (regmatch($enterprise, "^Acme Config:[0-9]"))
regreplace(*expression*, "*regexp*", *string* [, *count*])	Uses a regular expression and a substitution string to perform a search and replace operation on an input string expression.	$result = regreplace($input, "([%'])", "")
rtrim(*expression*)	Removes white space from the right of an expression.	$TrimNode = rtrim($Node)

Table 12. String functions (continued)

Function	Description	Example

`scanformat(`*`expression`*`, "string")`	Converts the expression according to the following formats, similar to the `scanf` family of routines in C. Conversion specifications are:
	
	`%%` - literal `%`; do not interpret
	
	`%d` - matches an optionally signed decimal integer
	
	`%u` - same as `%d`; no check is made for sign
	
	`%o` - matches an optionally signed octal number
	
	`%x` - matches an optionally signed hexadecimal number
	
	`%i` - matches an optionally signed integer
	
	`%e`, `%f`, `%g` - matches an optionally signed floating point number
	
	`%s` - matches a string terminated by white space or end of string
		`$element = "Lou is up in 15 seconds"`
		
		`[$node, $state, $time] = scanformat($element, "%s is %s in %d seconds")`
		
		This sets `$node`, `$state`, and `$time` to `Lou`, `up`, and `15`, respectively.

`num_returned_fields` `= split("`*`string`*`", `*`destination_array`*`, `*`"field_separator"`*`)`	Separates the specified string into elements of the destination array.
	
	The field separator separates the elements. The field separator itself is not returned. If you specify multiple characters in the field separator, when any combination of one or more of the characters is found in the string, a separation will occur.
	
	Regular expressions are not allowed in the string or field separator.
		`$num_elements=split("bilbo: frodo:gandalf",names,":")`
		
		creates an array with three entries:
		
		`names[1] = bilbo`
		
		`names[2] = frodo`
		
		`names[3] = gandalf`
		
		`num_elements` is set to 3.
		
		You must define the `names` array at the start of the rules file, before any processing statements.

`substr(`*`expression`*`,n, len)`	Extracts a substring, starting at the position specified in the second parameter, for the number of characters specified by the third parameter.	`$Substring = substr($Node,2,10)` extracts 10 characters from the second position of the `$Node` element
`upper(`*`expression`*`)`	Converts an expression to uppercase.	`$Node = upper($Node)`

Related concepts:

Appendix C, "Regular expressions," on page 223
Tivoli Netcool/OMNIbus supports the use of regular expressions in search queries that you perform on ObjectServer data. Regular expressions are sequences of *atoms* that are made up of normal characters and metacharacters.

Related reference:

"Using arrays" on page 23
You must define arrays at the start of a rules file, before any processing statements.

"Search and replace function" on page 53
Use the `regreplace` function to perform search and replace operations on strings by using regular expressions.

Math functions

You can use math functions to perform numeric operations on elements. Elements are stored as strings, so you must use these functions to convert elements into integers before performing numeric operations.

The following table describes the math functions supported in rules files.

Table 13. Math functions

Function	Description	Example
`int(`*`numeric`*`)`	Converts a numeric value into an integer.	`if int($PercentFull) > 80`
`real(`*`numeric`*`)`	Converts a numeric value into a real number.	`@DiskSpace= (real($diskspace)/ real($total))*100`

`toBase(`*`base`*`,`*`value`*`)`	Converts a decimal numeric value into a different base.	`toBase(2,16)` returns `10000`
		
		`toBase(16,14)` returns `E`
		
		`toBase(16,$a)` returns the value of the element `$a` converted into base 16

Example: Setting the severity of an alert based on available disk space

In the following example, the severity of an alert that monitors disk space usage is set based on the amount of available disk space.

```
if (int($PercentFull) > 80 && int($PercentFull) <=85)
{
 @Severity=2
}
else if (int($PercentFull)) > 85 && int($PercentFull) <=90)
{
 @Severity=3
}
else if (int($PercentFull > 90 && int($PercentFull) <=95)
{
 @Severity=4
}
else if (int($PercentFull) > 95)
{
 @Severity=5
}
```

Example: Calculating the amount of disk space

The percentage of disk space is not always provided in the event stream. You can calculate the percentage of disk space in the rules file as follows:

```
if (int($total) > 0)
{
 @DiskSpace=(100*int($diskspace))/int($total)
}
```

This can also be calculated using the `real` function:

```
if (int($total) > 0)
{
 @DiskSpace=(real($diskspace)/real($total))*100
}
```

You can then set the severity of the alert, as shown in the preceding example.

Date and time functions

You can use date and time functions to obtain the current time, or to perform date and time conversions.

Times are specified in UNIX time (as the number of elapsed seconds since midnight on 1st January 1970 UTC). The following table describes the date and time functions supported in rules files.

Table 14. Date and time functions

Function	Description	Rules file example
`datetotime(`*`string`*`, `*`conversion_specification`*`)`	Converts a textual representation of a timestamp into UNIX epoch time (that is, the number of seconds since 00:00:00 1 Jan 1970 UTC).	`$Date = datetotime("Tue Dec 19 18:33:11 GMT+00:00 2000", "EEE MMM dd HH:mm:ss vv yyyy")`
`getdate`	Takes no arguments and returns the current date as a date.	`$tempdate = getdate`

`timetodate(`*`UTC`*`, `*`conversion_specification`*`)`	Converts a time value into a string.	`@Summary = "Occurred at " + timetodate ($StateChange, "HH:mm:ss, MM/dd/yy")`
		
		`$Time2 = timetodate (@EventTime, 'EEE MMM dd HH:mm:ss yyyy')`

The *conversion_specification* parameter of the `datetotime` and `timetodate` functions is the date and time format in which you want the conversion to be expressed. The following table provides examples of the input and output data that matches some possible date and time formats.

Note: POSIX date and time formats were deprecated with Tivoli Netcool/OMNIbus V7.3 and replaced by Locale Data Markup Language (LDML) date and time patterns. LDML date and time patterns are defined at http://userguide.icu-project.org/formatparse/datetime.

Table 15. Some LDML date and time formats and matching input and output

Format	Example input and output data
`MM/dd/yy`	`02/29/12`
`MMMM`	`December`

Table 15. Some LDML date and time formats and matching input and output (continued)

Format	Example input and output data
d/M/yyyy H:m:s	19/12/2000 18:33:11
EEE MMM dd HH:mm:ss ZZZZ yyyy	Tue Dec 19 17:33:11 GMT 2000
yyyy-MM-dd:hh:mma vv	2009-03-28:02:00PM PT
EEE MMM dd HH:mm:ss yyyy ZZZ	Sun Jan 15 08:30:00 2006 +0500

For more information about the LDML date and time formats used in Tivoli Netcool/OMNIbus, see the *IBM Tivoli Netcool/OMNIbus Installation and Deployment Guide*.

Host and process utility functions

You can use utility functions to obtain information about the environment in which the probe is running.

The following table describes the host and process functions supported in rules files.

Table 16. Host and process utility functions

Function	Description	Example
getenv(*string*)	Returns the value of a specified environment variable.	$My_OMNIHOME = getenv("OMNIHOME")
gethostaddr(*string*)	Returns the IP address of the host using a naming service (for example, DNS or /etc/hosts). The argument can be a string containing a host name or an IP address. If the host cannot be looked up, the original value is returned. **Note:** DNS lookup (and other similar services) can take an appreciable amount of time which can severely impact the performance of the probe. You should consider instead using a lookup table in the rules file, and only use gethostaddr if the host is not in the table.	@Summary = $Summary + " Node: " + $Node + " Address: " + gethostaddr($Node)
gethostname(*string*)	Returns the name of the host using a naming service (for example, DNS or /etc/hosts). The argument can be a string containing a host name or IP address. If the host cannot be looked up, the original value is returned. **Note:** DNS lookup (and other similar services) can take an appreciable amount of time which can severely impact the performance of the probe. You should consider instead using a lookup table in the rules file, and only use gethostname if the host is not in the table.	@Summary = $Summary + " Node: " + $Node + " Name: " + gethostname($Node)
getpid()	Returns the process ID of the running probe.	$My_PID = getpid()

Table 16. Host and process utility functions (continued)

Function	Description	Example
`getplatform()`	Returns the operating system platform the probe is running under. One of the following values is returned: `linux2x86`, `solaris2`, `hpux11`, `aix5`, or `win32`.	`log(INFO, "Netcool Platform = " + getplatform())`
`hostname()`	Returns the name of the host on which the probe is running.	`$My_Hostname = hostname()`

Lookup table operations

Lookup tables provide a way to add extra information in an event. A lookup table consists of a list of keys and values.

You define a lookup table using the `table` function, and access the table using the `lookup` function.

The `lookup` function evaluates the expression in the keys of the named table and returns the associated value. If the key is not found, an empty string is returned. The `lookup` function has the following syntax:

`lookup(`*`expression`*`,`*`tablename`*`)`

You can create a lookup table in the rules file or in a separate file.

Note: If a lookup table file has multiple columns, every row must have the same number of columns. Any rows that do not have the correct number of columns are discarded. In single column mode, only the first tab is significant; all later tabs are read as part of the single value defined on that row.

Defining lookup tables in the rules file

You can create a lookup table directly in the rules file.

About this task

Lookup table definitions must be located at the start of a rules file, *after* all `registertarget` statements, but *before* any processing statements. A lookup table can have multiple columns. You can also define multiple lookup tables in a rules file. For changes to the lookup table to take effect, the probe must be forced to re-read the rules file.

To create a lookup table:

Procedure

1. Open the rules file for the probe.
2. Following the `registertarget` statements, add the relevant table definition entry for a lookup table with the name *tablename*:
   a. To create the lookup table with a list of keys and values, use the following format:

      ```
      table tablename={{"key","value"},{"key","value"}...}
      ```

   b. To create the lookup table with multiple columns, use the following format:

      ```
      table tablename={{"key1", "value1", "value2", "value3"},
       {"key2", "val1", "val2", "val3"}}
      ```

c. To create the lookup table and specify a default option to handle an event that does not match any of the key values in the table, use the following format:

```
table tablename=
{{"key1", "value1", "value2", "value3"},
{"key2", "val1", "val2", "val3"}}
default = {"defval1", "defval2", "defval3"}
```

Note: The default statement must follow the specific table definition.

Example

For example, to create a lookup table named dept, which matches a node name to the department that the node is in, add the following line to the rules file:

```
table dept={{"node1","Technical"},{"node2","Finance"}}
```

You can access this lookup table in the rules file as follows:

```
@ExtraChar=lookup(@Node,dept)
```

This example uses the @Node field as the key. If the value of the @Node field matches a key in the table, @ExtraChar is set to the corresponding value.

You can obtain values from a multiple value lookup table as follows:

```
[@Summary, @AlertKey, $error_code] = lookup("key1", tablename)
```

Related tasks:

"Rereading the rules file" on page 60
Because probes read the rules file only on startup, you must force the probe to reread the rules whenever you make changes to it. The probe processes the reread request only on receipt of a new event. If the probe is idle or is already processing an event, it will not reread the rules file until a new event is received.

Defining lookup tables in a separate file

You can create the table in a separate file, as an alternative to creating the lookup table directly in the rules file.

If you are specifying a single value, the file must be in the format:

```
key[TAB]value
key[TAB]value
```

For multiple values, the format is:

```
key1[TAB]value1[TAB]value2[TAB]value3
key2[TAB]val1[TAB]val2[TAB]val3
```

You can specify a default option to handle an event that does not match any of the key values in a table. The default statement must follow the specific table definition. The following example is for a table in a separate file:

```
table dept="$OMNIHOME/probes/solaris2/Dept"
default = {"defval1", "defval2", "defval3"}
```

For example, to create a table in which the node name is matched to the department that the node is in, use the following format:

```
node1[TAB]"Technical"
node2[TAB]"Finance"
```

Specify the path to the lookup table file as an absolute or relative path. Relative paths start from the current rules file directory. You can use environment variables in the path. For example:

```
table dept="$OMNIHOME/probes/solaris2/Dept"
```

You can then use this lookup table in the rules file as follows:

```
@ExtraChar=lookup(@Node,dept)
```

You can also control how the probe processes external lookup tables with the **LookupTableMode** property. This property determines how errors are handled when external lookup tables do not have the same number of values on each line.

Update on deduplication function

The ObjectServer manages the deduplication process, but you can also configure this process in the probe rules file. Use the `update` function to specify which fields of an alert are to be updated if the alert is deduplicated. This allows deduplication rules to be set on a per-alert basis.

The `update` function can enable update on deduplication for fields that are not set to be updated in the deduplication trigger. You cannot use the `update` function to override the deduplication trigger to prevent fields from being updated.

The `update` function has the following syntax:

```
update(fieldname [, TRUE | FALSE ] )
```

If set to `TRUE`, update on deduplication is enabled. If set to `FALSE`, update on deduplication is disabled. The default is `FALSE`.

For example, to ensure that the Severity field is updated on deduplication, add the following entry to the rules file:

```
update(@Severity)
```

The following example shows how to disable update on deduplication in the rules file for a previously-enabled field:

```
update(@Severity, FALSE)
```

If, in the deduplication trigger, the field is set to be updated, setting the `update` function to `FALSE` has no effect.

Details function

Details are extra elements created by a probe to display alert information that is not stored in a field of the alerts.status table. Alerts do not have detail information unless this information is added.

Detail elements are stored in the ObjectServer details table called alerts.details. To view details, double-click an alert and select **Details**.

You can add information to the details table by using the `details` function. The detail information is added when an alert is inserted, but not if it is deduplicated.

The following example adds the elements $a and $b to the alerts.details table:

```
details($a,$b)
```

The following example adds all of the alert information to the alerts.details table:

```
details($*)
```

Attention: You must only use $* when you are debugging or writing rules files. After using $* for long periods of time, the ObjectServer tables become very large and the performance of the ObjectServer suffers.

Example: Using the details function

In this example, the $Summary element is compared to the strings `Incoming` and `Backup`. If there is no match, the @Summary field is set to the string `Please see details`, and all of the information for the alert is added to the details table:

```
if (match($Summary, "Incoming"))
{
 @Summary = "Received a call"
}
else if(match($Summary, "Backup"))
{
 @Summary = "Attempting to back up"
}
else
{
 @Summary = "Please see details"
 details($*)
}
```

Message logging functions

You can use the `log` function to log messages during rules processing. You can also set a log level using the setlog function, and only messages equal to, or above, that level are logged.

There are five log levels: `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `FATAL`, in order of increasing severity. For example, if you set the log level to `WARNING`, only `WARNING`, `ERROR`, and `FATAL` messages are logged, but if you set the logging to `ERROR`, then only `ERROR` and `FATAL` messages are logged.

Log function

The `log` function sends a message to the log file.

The syntax is:

```
log([ DEBUG | INFO | WARNING | ERROR | FATAL ],"string")
```

Note: When a `FATAL` message is logged, the probe terminates.

Setlog function

The setlog function sets the minimum level at which messages are logged during rules processing. By default, the level for logging is `WARNING` and above.

The syntax is:

```
setlog([ DEBUG | INFO | WARNING | ERROR | FATAL ])
```

Example: Message logging

The following lines show a sequence of logging functions that are in the rules file:

```
setlog(WARNING)
log(DEBUG,"A debug message")
log(WARNING,"A warning message")
setlog(ERROR)
log(WARNING,"Another warning message")
log(ERROR,"An error message")
```

This produces log output of:

```
A warning message
An error message
```

The DEBUG level message is not logged, because the logging setting is set higher than DEBUG. The second WARNING level message is not logged, because the preceding `setlog` function has set the log level higher than WARNING.

Sending alerts to alternative ObjectServers and tables

The `registertarget`, `genevent`, `settarget`, and `setdefaulttarget` functions enable you to send alerts to one or more ObjectServers, and to define the distribution of alerts across the ObjectServers.

Registering target ObjectServers and setting targets for alerts

To register an ObjectServer, and an alerts table in that ObjectServer to which you want to send events (referred to as the *target ObjectServer*), use the `registertarget` function. If required, you can use this function to specify several target ObjectServers and corresponding alerts tables. Use the `setdefaulttarget` function to specify a different default target ObjectServer and table. To specify an alternative target ObjectServer that is not a default, use the `settarget` function.

Register all target ObjectServers at the start of the rules file, before any processing statements or lookup tables.

In each target ObjectServer, each alert is usually sent to only one alerts table. Optionally, the alert can also be sent to a corresponding details table. Exceptions occur when the `genevent` function is used.

Format of the registertarget function

The format for the `registertarget` function is as follows:

```
target = registertarget(servername, backupservername,"databasetable"
 [, "detailstable" ] )
```

In this statement:

- *target* is a label to identify the target ObjectServer. This label must be unique among all the registered target ObjectServers. A label might denote the type or distribution of alerts to send to the ObjectServer. Examples include NCOMSalerts, FloodProtectionActiveAlert, HighAlerts, StatsInfoAlert. The target is used as the argument in `genevent`, `settarget`, and `setdefaulttarget` functions.
- *servername* is the name of the target ObjectServer.
- *backupservername* is the name of the backup target ObjectServer in the failover pair, if a failover pair is configured.
- *databasetable* is the name of a valid table in any database into which you want the alert data to be inserted. Enclose this value in double quotation marks (" ").

- *detailstable* is the name of a valid details table in any database into which you want the alert data to be inserted. Enclose this value in double quotation marks (" ").

The *servername* and *backupservername* values can have either of the following formats:

- Strings enclosed in double quotation marks (" "), for example "Server" or "ServerBackup".
- The value of the property preceded by the percent sign (%), for example %Server or %ServerBackup.

To omit the backup ObjectServer, specify *backupservername* as an empty string: `""`.

Usage guidelines

The `registertarget` function requires the same user authorization for all the referenced ObjectServers. When connecting to multiple secure ObjectServers, a probe uses the credentials set by its **AuthUserName** and **AuthPassword** properties for all the target ObjectServers. It is also necessary that the user account has adequate authorization permissions for all the target tables that the probe sends event data to.

The first `registertarget` statement in a probe rules file defines the default target ObjectServer. This ObjectServer supersedes any target ObjectServer that you specify by running the probe with the `-server` command-line option. For example, if a probe rules file has a single `registertarget` statement that registers TEST1 as the default target ObjectServer, and you then run the probe with `-server` option set to TEST2, the alerts are sent to TEST1.

If you want additional detail information to be inserted for the alert, use the `details` function to specify this detail information in the rules file.

If a rules file is cached, the names of the target ObjectServers are always written to the cache file as strings. If you registered target ObjectServers by the field names %Server and %ServerBackup, the values of these fields are also resolved to the cache file as strings. For example, if the target ObjectServer is called NCOMS, the `registertarget` statement `alerts = registertarget( %Server,"", "alerts.status","alerts.details" )` is written to the cache file as `alerts = registertarget( "NCOMS","","alerts.status","alerts.details" )`.

Example 1: Registering multiple target ObjectServers

The following example shows the `registertarget` function used to register multiple target ObjectServers. In this example, the alerts are inserted in the alerts.status table of the TEST1 ObjectServer, unless the settings are overridden by a subsequent call to the `setdefaulttarget` or `settarget` functions.

```
DefaultAlerts = registertarget( "TEST1", "", "alerts.status" )
HighAlerts = registertarget( "TEST2", "", "alerts.status" )
ClearAlerts = registertarget( "TEST3", "", "alerts.status" )
London = registertarget( "NCOMS", "NCOMSBACK", "alerts.london" )
MasterStats = registertarget( %Server, %ServerBackup, "master.stats" )
```

Example 2: Specifying a different default target ObjectServer

The following example shows the `setdefaulttarget` command used to change the default target ObjectServer for alerts that have specific severities.

```
# When an event of Major severity or higher comes in,
# set the default ObjectServer to TEST2
if(int(@Severity) > 3)
 { setdefaulttarget(HighAlerts) }
```

Example 3: Specifying an alternative target ObjectServer

The following example shows the `settarget` command used to change the target ObjectServer for alerts that have a specific severity.

```
# Send all clear events to TEST3
if (int(@Severity) = 0)
 { settarget(ClearAlerts) }
```

Related tasks:

"Enabling caching of probe rules files" on page 61
To ensure that a probe is always able to read a valid set of rules when the probe is started, enable the caching of the rules file. By default, rules file caching is disabled.

Related reference:

"Details function" on page 45
Details are extra elements created by a probe to display alert information that is not stored in a field of the alerts.status table. Alerts do not have detail information unless this information is added.

"Lookup table operations" on page 43
Lookup tables provide a way to add extra information in an event. A lookup table consists of a list of keys and values.

"Sending alerts to multiple ObjectServers and tables"
If you want to send the same alert to more than one registered ObjectServer or to more than one table, you must use the genevent function.

Sending alerts to multiple ObjectServers and tables

If you want to send the same alert to more than one registered ObjectServer or to more than one table, you must use the genevent function.

Some usage scenarios for the `genevent` function are as follows:

- You want to configure the probe rules file to detect an event flood condition and temporarily suppress alert data that is being sent to the ObjectServer. You can use the `genevent` function to send the ObjectServer an informational alert at the start of the event flood and when the event flood finishes.
- You require high priority processing alerts and low priority informational alerts to be separated at source and handled differently. You can use the `genevent` function to send the high priority alerts to a high priority ObjectServer, and to an ObjectServer that correlates and archives all alerts. You can also send the low priority alerts only to the ObjectServer that correlates and archives all alerts.
- You require statistical analysis of incoming alert data, but do not want to increase the load on the ObjectServer receiving the events. You can use the `genevent` function to send statistical information that is derived from the incoming alert data to another ObjectServer for analysis at a later stage.
- You want to duplicate all alert data across two or more ObjectServers so that the ObjectServers can perform different operations on the data. You additionally want to eliminate the overhead of running a unidirectional gateway between the ObjectServers. You can use the `genevent` function to send the alert data to all the ObjectServers. Note, however, that this type of usage is *not* intended as a replacement to the use of a gateway in a failover pair because the duplicated alerts will not be correctly associated with each other.

The format for the genevent function is as follows:

```
genevent(target[, column_identifier, column_value, ...])
```

In this statement:

- *target* is the value that you specified for *target* in the relevant registertarget statement.
- *column_identifier* and *column_value* represent name-value pairs, where *column_identifier* is a valid ObjectServer field in the table where the alert is to be inserted, and *column_value* is the data value that you want to insert. The *column_identifier* value must be prefixed with the @ symbol to denote an ObjectServer field; for example, @Summary. The *column_value* can be a static value, or an expression that is resolved when the rules file is processed; for example $Summary + $Group. If *column_value* is a string value, it must be enclosed in double quotation marks.

  Tip: When specifying *column_value*, use a data type that is appropriate for the ObjectServer field. From Netcool/OMNIbus Administrator, you can use the Databases, Tables and Columns pane (which is used to add or edit table columns) to verify the data types assigned to fields. Alternatively, you can use the ObjectServer SQL DESCRIBE command.

  Note: When the rules file is processed, data type conversion is attempted on a column value if there is a mismatch between the column identifier and the specified data type. If the conversion is unsuccessful, a non-fatal error is logged and the event is not generated.

If you want to send the current alert data to a target ObjectServer, and automatically insert all available data into the relevant fields, omit the column identifiers and values from the genevent statement as follows. You might find this format useful if you want to send a duplicate of the current alert data to more than one ObjectServer. With this format, note also that the alert data includes only those fields that have been set up above the genevent statement in the rules file.

```
genevent(target)
```

Typically include the column identifiers and values in the genevent(*target*) statement if you want to populate a specific subset of the fields in the target ObjectServer. For example:

```
genevent(StatusAlerts, @Node, $Node, @Summary, "Condition X has occurred")
```

To view examples for the genevent function, see the sample secondary rules file that is provided to support the detection of event floods and anomalous event rates. This file, called flood.rules, is available in the $NCHOME/omnibus/extensions/eventflood directory. (The flood.rules file must be used in conjunction with the accompanying configuration rules file called flood.config.rules.)

Sending detail information and service status to targets

You can use genevent statements to send detail information and service status under the following conditions:

- If a registertarget statement specifies a details table to which detail information should be sent, a genevent statement that sends alerts to the same target will also send the detail information to the details table specified in the registertarget statement. This condition is true only if the details statement precedes the genevent statement.

- If you use the service function to define the status of a service, and the service statement precedes the `genevent` statement, the `genevent` statement will send the status information to its target ObjectServer.
- If more than one details or service statement precedes or follows a `genevent` statement, only the information from the last details or service statement directly above the `genevent` statement will be sent to the target. Information that is generated by any of the other details or service statements is associated with the main alert only, and is sent only to the relevant targets defined in the `registertarget` statements.

In the following example, the `genevent` statement adds the elements `$c` and `$d` to the alerts.details table in the TEST2 ObjectServer. For the host being monitored, a marginal service status is also assigned to each alert, when viewed from the Services window, which is available from the Conductor or event list.

```
DefaultAlerts = registertarget( "TEST1", "", "alerts.status" )
HighAlerts = registertarget( "TEST2", "", "alerts.status" "alerts.details")
...
details ($a,$b)
...
details ($c,$d)
...
service($host, bad)
...
service($host, marginal)
...
genevent(HighAlerts)
...
details ($y,$z)
...
service($host, good)
```

Related concepts:

"Detecting event floods and anomalous event rates" on page 67
Event floods can cause ObjectServer outages, and can lead to extended periods where there is no visibility of network events. An unusually low or high rate of receipt of events can also be indicative of a problem or change in the source, which needs to be addressed.

Related reference:

"Registering target ObjectServers and setting targets for alerts" on page 47
To register an ObjectServer, and an alerts table in that ObjectServer to which you want to send events (referred to as the *target ObjectServer*), use the `registertarget` function. If required, you can use this function to specify several target ObjectServers and corresponding alerts tables. Use the `setdefaulttarget` function to specify a different default target ObjectServer and table. To specify an alternative target ObjectServer that is not a default, use the `settarget` function.

"Details function" on page 45
Details are extra elements created by a probe to display alert information that is not stored in a field of the alerts.status table. Alerts do not have detail information unless this information is added.

"Service function" on page 54
Use the `service` function to define the status of a service before alerts are forwarded to the ObjectServer. The status changes the color of the alert when it is displayed in the event list and Service windows.

Multithreaded processing of alert data

When a probe rules file is processed, multithreaded processing is used by default to apply probe rules to the raw event data that is acquired from the event source, and to send the generated alerts to the registered ObjectServers. Note that this multithreaded processing is different from the multithreaded or single-threaded event capture that is implemented in some classes of probes.

In multithreaded mode, a single thread is used for rules file processing, and individual threads are used for communicating with each registered ObjectServer. The rules file processing thread applies the rules to the incoming data, establishes connections to the relevant ObjectServers, and sends the processed results to the appropriate communication thread. The communication thread transforms the processed data into SQL INSERT statements and sends them to the ObjectServer.

If required, you can switch from multithreaded processing to single-threaded processing by setting the **SingleThreadedComms** property to `TRUE`. In single-threaded mode, a single rules file processing and communication thread is used.

With multithreaded processing, alerts are simultaneously sent to the different ObjectServers. If required, you can use the single-threaded mode to enforce the order in which alerts are sent to the ObjectServers; this order is defined by the order in which the `registertarget` statements are listed in the rules file. You can also use the single-threaded mode for debugging, because the order in which events are processed and sent out can be more easily understood.

In multithreaded mode, if buffering is enabled by using the **Buffering** property, a separate text buffer is maintained for each ObjectServer, to temporarily hold data that cannot be immediately processed by the communication thread. If buffering is disabled, the SQL INSERT statements are sent to the ObjectServers as soon as the statements are constructed.

If store-and-forward mode is enabled by using the **StoreAndForward** property and multithreaded processing is in operation, separate store-and-forward files are created to hold the data that cannot be sent to each ObjectServer. The store-and-forward files are stored in `$OMNIHOME/var` directory and are named using the default format **SAFFileName.***servername*, where **SAFFileName** represents the **SAFFileName** property setting and *.servername* is appended to show the ObjectServer name.

Note: When running in multithreaded mode, a probe initially starts in single-threaded mode (by design) before switching to multithreaded mode. This behavior is also observed when a probe re-reads its rules file, and is recorded in the probe log file in debug mode.

Related concepts:

"Store-and-forward mode for probes" on page 11
Probes can continue to run if the target ObjectServer is down. During this period, the probe switches to *store* mode. The probe reverts to *forward* mode when the ObjectServer is functional again.

Related reference:

Chapter 6, "Common probe properties and command-line options," on page 113
A number of properties and command-line options are common to all probes and TSMs.

Search and replace function

Use the `regreplace` function to perform search and replace operations on strings by using regular expressions.

The syntax is as follows:

```
regreplace(input, "regularexpression", "substitution" [,count])
```

Where:

- The *input* is a string expression. The `regreplace` function reads the input string from left to right.
- The *regularexpression* is a string. It cannot be a string expression. You can use parentheses () in the string to specify substrings that require specific matching. You can use multiple sets of parentheses in a string.
- The *substitution* is a string expression that specifies how strings that match *input* are to be written in the result. You can use metacharacters to reference matching substrings (in parentheses), as well as an entire matching string or strings. For example, `\1` matches the first group in the regular expression, `\2` the second, and so on, while `&` or `\0` match the entire string. Characters and strings that do not match the regular expression are copied to the result string.
- The *count* is an optional positive integer expression, and denotes the number of substitutions to be made on matching strings. If you do not provide a value for *count*, the substitutions continue until no more matching strings are found. If *count* is a non-integer expression, it is interpreted as 0, and the *input* is not changed. If *count* is a negative integer, a warning message is entered in the probe log, and the *input* is not changed.

Example: Using search and replace to remove unwanted characters from a string

The following example shows how to use the `regreplace` function to replace underscores (_), percent signs (%), and single quotes (') with a blank string:

```
$result = regreplace("%Node___='foobar27'%" , "([_%']*)", "" )
```

The result of this expression is as follows:

```
$result="Node=foobar27"
```

The following example shows how to use the `regreplace` function to replace carriage return (CR) or line feed (LF) control characters with a blank string:

```
@Summary = regreplace($Summary, '[\n\r]', "")
```

Example: Reordering groups of characters in a string

The following example shows how to match multiple substrings within a string and, in the output, reorder the substrings. The order of substrings in the input string is changed in the output string.

```
regreplace("aba argle aca", "(a.a) (.*) (a.a)", "\3 \2 \1")
```

The regular expression matches the substrings in the following order:

```
\1="aba"
\2="argle"
\3="aca"
```

The substitution string specifies that the matched strings be written in the reverse order to which the input is read. Consequently, the result of this expression is as follows:

```
$result = "aca argle aba"
```

Example: Using metacharacters to match an entire string

The following example shows how to use the metacharacter `&`, which can also be expressed as `\0`, to match the entire string represented by the regular expression:

```
regreplace("aaabbbaaa", "a(b+)a" "_&_")
```

The `&` or `\0` metacharacters match everything that maps to the regular expression, not only the substring in parentheses. In this example, the regular expression matches the following substring in the input: `abbba`. The nonmatching substrings are copied to the output.

The result of this expression is as follows:

```
$result="aa_abbba_aa"
```

Related concepts:

Appendix C, "Regular expressions," on page 223
Tivoli Netcool/OMNIbus supports the use of regular expressions in search queries that you perform on ObjectServer data. Regular expressions are sequences of *atoms* that are made up of normal characters and metacharacters.

Service function

Use the `service` function to define the status of a service before alerts are forwarded to the ObjectServer. The status changes the color of the alert when it is displayed in the event list and Service windows.

The syntax is:

```
service(service_identifier, service_status)
```

The *service_identifier* identifies the monitored service, for example, `$host`.

The following table lists the service status levels.

Table 17. Service function status levels

Service status level	Definition
BAD	The service level agreement is not being met.
MARGINAL	There are some problems with the service.
GOOD	There are no problems with the service.
No Level Defined	The status of the service is unknown.

Example: Service function

If you want a Ping Probe to return a service status for each host it monitors, you can use the `service` function in the rules file to assign a service status to each alert. In the following example, a service status is assigned to each alert based on the value of the `status` element.

```
switch ($status)
{
 case "unreachable":
```

```
  @Severity = "5"
  @Summary = @Node + " is not reachable"
  @Type = 1
  service($host, bad) # Service Entry
 case "alive":
  @Severity = "3"
  @Summary = @Node + " is now alive"
  @Type = 2
  service($host, good) # Service Entry
 case "noaddress":
  @Severity = "2"
  @Summary = @Node + " has no address"
  service($host, marginal) # Service Entry
 case "removed":
  @Severity = "5"
  @Summary = @Node + " has been removed"
  service($host, marginal) # Service Entry
 case "slow":
  @Severity = "2"
  @Summary = @Node + " has not responded within
  trip time"
  service($host, marginal) # Service Entry
 case "newhost":
  @Severity = "1"
  @Summary = @Node + " is a new host"
  service($host, good) # Service Entry
 case "responded":
  @Severity = "0"
  @Summary = @Node + " has responded"
  service($host, good) # Service Entry
  default:
  @Summary = "Ping Probe error details: " + $*
  @Severity = "3"
  service($host, marginal) # Service Entry
}
```

Monitoring probe loads

To monitor load, it is necessary to obtain time measurements and calculate the number of events processed over time. The updateload function takes a time measurement each time it is called, and the getload function returns the load as events per second.

Each time the updateload function runs, the current time stamp, recorded in seconds and microseconds, is added to the beginning of a series of time stamps. The remaining time stamps record the difference in time from the previous time stamp. For example, to take a time measurement and update a property called **load** with a new time stamp:

```
%load = updateload(%load)
```

Tip: Depending on the operating system, differing levels of granularity may be reported in time stamps.

You can specify a maximum time window for which samples are kept, and a maximum number of samples. By default, the time window is one second and the maximum number of samples is 50. You can specify the number of seconds for which load samples are kept and the maximum number of samples in the format:

time_window_in_seconds.max_number_of_samples

For example, to set or reset these values for the `load` property:

```
%load = "2.40"
```

When the number of seconds in the time window is exceeded, any samples outside of that time window are removed. When the number of samples reaches the limit, the oldest measurement is removed.

The getload function calculates the current load, returned as events per second. For example, to calculate the current load and assign it to a temporary element called `current_load`:

```
$current_load = getload(%load)
```

The geteventcount function complements the getload function by returning the total number of events in the event window.

Related reference:

"Rules file examples" on page 63
These examples show typical rules file segments.

Reserved words in the probe rules language

In the probe rules language, certain words are reserved as keywords, and must not be used as variable names or property names within probe rules files.

The following list shows the reserved words:

- and
- array
- bad
- break
- case
- char
- character
- charcount
- clear
- datetime
- datetotime
- debug
- decode
- default
- details
- discard
- Fix Pack 1 discarded
- double
- else
- error
- exists
- exit
- expand
- extract
- false
- fatal
- foreach
- genevent

- getdate
- good
- if
- in
- include
- info
- information
- int
- integer
- len
- length
- log
- lookup
- lower
- ltrim
- marginal
- match
- nmatch
- no
- not
- nvp_add
- nvp_remove
- off
- on
- or
- printable
- real
- recover
- registertarget
- regmatch
- regreplace
- remove
- rtrim
- scanformat
- service
- setdefaultobjectserver
- setdefaulttarget
- setlog
- setobjectserver
- settarget
- split
- string
- substr
- switch
- table

- timetodate
- true
- update
- upper
- warn
- warning
- xor
- yes

Testing rules files

You can test the syntax of a rules file by using the Probe Rules Syntax Checker, **nco_p_syntax**. This is more efficient than running the probe to test that the syntax of the rules file is correct.

About this task

The Probe Rules Syntax Checker is installed with the Probe Support feature of Tivoli Netcool/OMNIbus and is installed in the following directory:

- UNIX Linux `$NCHOME/omnibus/probes`
- Windows `%NCHOME%\omnibus\probes\win32`

Procedure

To run the Probe Rules Syntax Checker, enter the following command:
`nco_p_syntax -rulesfile /rules_file_path/rules_file.rules`
When running this command, use the `-rulesfile` command-line option to specify the full path and file name of the rules file.

Results

The Probe Rules Syntax Checker runs in debug mode by default. You can override this setting with the `-messagelevel` command-line option; for example, `-messagelevel info`.

The probe connects to the ObjectServer, tests the rules file, displays any errors to the screen, and then exits. If no errors are displayed, the syntax of the rules file is correct. For details about the Probe Rules Syntax Checker, see the publication for this probe. You can access this publication as follows from the IBM Tivoli Network Management Information Center (http://publib.boulder.ibm.com/infocenter/tivihelp/v8r1/index.jsp):

1. Expand the *IBM Tivoli Netcool/OMNIbus* node in the navigation pane on the left.
2. Expand the *Tivoli Netcool/OMNIbus probes and TSMs* node.
3. Go to the *Universal* node.

Debugging rules files

When you change the rules file, add new rules, or create lookup tables, it is useful to test the probe by running it in debug mode. Debug mode shows how an event is being parsed by the probe and can uncover any problems with the rules file.

About this task

You can enable debug mode from the command-line interface or by changing the probe properties file. If you need to change the message level of a running probe without stopping the probe, you can use the **kill** command against the probe process ID (PID).

Procedure

- To enable debug mode from the command-line interface, enter the following command:

  `$OMNIHOME/probes/nco_p_`*`probename`* `-messagelevel DEBUG -messagelog STDOUT`

  If you omit the `-messagelog` command-line option, the debug information is sent to the probe log file in the `$OMNIHOME/log` directory rather than to the screen.
- To enable debug mode by using the probe properties file, add the following entries to the file:

  ```
  MessageLevel: "DEBUG"
  MessageLog: "STDOUT"
  ```

  If you omit the **MessageLog** property, the debug information is sent to the probe log file in the `$OMNIHOME/log` directory rather than to the screen.
- To change the message level of a running probe to debug mode, use the `kill -USR2` *`pid`* command on the probe PID.

  Each time you issue the `kill -USR2` *`pid`* command, the message level is cycled.

  For more information, see the man pages for the **ps** and **kill** commands.

  Tip: For JAVA probes, issue the **kill** command on the **nco_p_nonnative** process ID.

What to do next

For changes to the rules file to take effect, force the probe to reread the rules file.

Related tasks:

"Rereading the rules file" on page 60
Because probes read the rules file only on startup, you must force the probe to reread the rules whenever you make changes to it. The probe processes the reread request only on receipt of a new event. If the probe is idle or is already processing an event, it will not reread the rules file until a new event is received.

Rereading the rules file

Because probes read the rules file only on startup, you must force the probe to reread the rules whenever you make changes to it. The probe processes the reread request only on receipt of a new event. If the probe is idle or is already processing an event, it will not reread the rules file until a new event is received.

Procedure

You can force a probe to reread the rules file in the following ways:

- Run the following command on the probe process ID (PID):
  `kill -HUP pid`

  where *pid* is the process ID.

  If the updated rules file contains syntax errors or references to fields that do not exist, the probe logs an error message on receipt of the HUP signal and continues to use the previous version of the rules file.

  For more information, see the **ps** and **kill** man pages.

  Tip: For JAVA probes, issue the command `kill -HUP` on the **nco_p_nonnative** process.
- Issue an **nco_probereloadrules** HTTP command against the probe to reload the rules file.
- Restart the probe.

  Important: If you restart the probe, events might be lost for the time that the probe is stopped.

Related concepts:

"Rules file" on page 7
The rules file defines how the probe processes event data to create a meaningful alert. For each alert, the rules file also creates an identifier that uniquely identifies the problem source.

Related tasks:

"Debugging rules files" on page 59
When you change the rules file, add new rules, or create lookup tables, it is useful to test the probe by running it in debug mode. Debug mode shows how an event is being parsed by the probe and can uncover any problems with the rules file.

"Defining lookup tables in the rules file" on page 43
You can create a lookup table directly in the rules file.

"Reloading rules files (nco_probereloadrules)" on page 99
You can use the **nco_probereloadrules** utility to remotely reload a probe rules file without restarting the probe.

"Enabling caching of probe rules files"
To ensure that a probe is always able to read a valid set of rules when the probe is started, enable the caching of the rules file. By default, rules file caching is disabled.

Related reference:

"Reload the rules file" on page 105
This request accesses the reloadrulesflag attribute of the probe, which is contained in the standard probe C library (libOpl). By setting the value of the attribute, the request triggers the probe to reload its rules file before the next event is processed. A SIGHUP signal on UNIX triggers the same action.

Chapter 6, "Common probe properties and command-line options," on page 113
A number of properties and command-line options are common to all probes and TSMs.

Enabling caching of probe rules files

To ensure that a probe is always able to read a valid set of rules when the probe is started, enable the caching of the rules file. By default, rules file caching is disabled.

When you switch on rules file caching, the probe writes a copy of the rules file to a single cache file when you start the probe. If the file size exceeds 1 GB, the rules file is written to multiple cache files. Each time that the probe successfully reads the rule file, it writes the file to the cache. When the probe is subsequently restarted, it reads the cache file if it cannot read the rules file.

All include files and lookup tables that are referenced in the rules file are written to the cache file inline. If the rules file contains `registertarget` statements that use the %Server and %ServerBackup fields to register target ObjectServers, the property values are resolved as quoted strings in the cache file. If the probe rules parser identifies rows in a lookup table that have duplicate keys, a warning is output. The cache file does not preserve the order of entries from the rules file, so if you receive a warning, verify that the order of entries in the cache file would not change the behavior of the probe.

Procedure

To enable caching:

1. In the probe properties file, set the **CacheRules** property to 1.
2. If you want use a caching file that is different to the default, set the **CacheRulesFile** property to specify the file path and name. Enclose the path in double quotation marks (" ").

Results

The probe rereads the rules file and writes it to the cached file. Each time the probe rereads the rules file, the probe checks the value of the **CacheRules** property and the **CacheRulesFile** property. If the values of these properties are set accordingly, the cache file is updated.

If the size of the cache file exceeds 1 GB, a new cache file is created, which is appended with 1.

Example

If the value of the **CacheRulesFile** property is `"$OMNIHOME/var/rulescache"`, the rules file is written to this cache file until it reaches 1 GB. Then, a new cache file is created, called `$OMNIHOME/var/rulescache1`. If this file exceeds 1 GB, `$OMNIHOME/var/rulescache2` is created, and so on.

Related concepts:

"Rules file" on page 7
The rules file defines how the probe processes event data to create a meaningful alert. For each alert, the rules file also creates an identifier that uniquely identifies the problem source.

Related tasks:

"Rereading the rules file" on page 60
Because probes read the rules file only on startup, you must force the probe to reread the rules whenever you make changes to it. The probe processes the reread request only on receipt of a new event. If the probe is idle or is already processing an event, it will not reread the rules file until a new event is received.

Related reference:

"Registering target ObjectServers and setting targets for alerts" on page 47
To register an ObjectServer, and an alerts table in that ObjectServer to which you want to send events (referred to as the *target ObjectServer*), use the `registertarget` function. If required, you can use this function to specify several target ObjectServers and corresponding alerts tables. Use the `setdefaulttarget` function to specify a different default target ObjectServer and table. To specify an alternative target ObjectServer that is not a default, use the `settarget` function.

Chapter 6, "Common probe properties and command-line options," on page 113
A number of properties and command-line options are common to all probes and TSMs.

Rules file examples

These examples show typical rules file segments.

Example: Enhancing the Summary field

This example rule tests if the `$trap-type` element is `Link-Up`. If it is, the @Summary field is populated with a string made up of `Link up on`, the name of the node from the record being generated, `Port`, and the value of the `$ifIndex` element:

```
if( match($trap-type,"Link-Up") )
{
 @Summary = "Link up on " + @Node + " Port " + $ifIndex
}
```

Example: Populating multiple fields

This example rule is similar to the previous rule except that the `@AlertKey` and `@Severity` fields are also populated:

```
if( match($trap-type, "Link-Up") )
{
 @Summary = "Link up on " + @Node + " Port " + $ifIndex
 @AlertKey = $ifIndex
 @Severity = 4
}
```

Example: Nested IF statements

This example rule first tests if the trap has come from an `Acme` manager, and then tests if it is a `Link-Up`. If both conditions are met, the `@Summary` field is populated with the values of the `@Node` field and `$ifIndex` and `$ifLocReason` elements:

```
if( match($enterprise,"Acme") )
{
 if( match($trap-type, "Link-Up") )
 {
 @Summary= "Acme Link Up on " + @Node + " Port " + $ifIndex +
 " Reason: "+$ifLocReason
 } }
```

Example: Regular expression match

This example rule tests for a line starting with `Acme Configuration:` followed by a single digit:

```
if (regmatch($enterprise,"^Acme Configuration:[0-9]"))
{
 @Summary="Generic configuration change for " + @Node
}
```

Example: Regular expression extract

This example rule tests for a line starting with `Acme Configuration:` followed by a single digit. If the condition is met, it extracts that single digit and places it in the `@Summary` field:

```
if (regmatch($enterprise,"^Acme Configuration:[0-9]"))
{
 @Summary="Acme error "+extract($enterprise,"^Acme Configuration:
 ([0-9])")+" on" + @Node
}
```

Example: Numeric comparisons

This example rule tests the value of an element called `$freespace` as a numeric value by converting it to an integer and performing a numeric comparison:

```
if (int($freespace) < 1024)
{
  @Summary="Less than 1024K free on drive array"
}
```

Example: Simple numeric expressions

This example rule creates an element called `$tmpval`. The value of `$tmpval` is derived from the `$temperature` element, which is converted to an integer and then has 20 subtracted from it. The string element `$tmpval` contains the result of this calculation:

```
$tmpval=int($temperature)-20
```

Example: Strings and numerics in one expression

This example rule creates an element called `$Kilobytes`. The value of `$Kilobytes` is derived from the `$DiskSize` element, which is divided by 1024 before being converted to a string type with the letter `K` appended:

```
$Kilobytes = string(int($DiskSize)/1024) + "K"
```

Example: Using load functions to monitor nodes

This example shows how to measure load for each node that is generating events. If a node is producing more than five events per second, a warning is written to the probe log file. If more than 80 events per second are generated for all nodes being monitored by the probe, events are sent to an alternative ObjectServer and a warning is written to the probe log file.

```
# declare the ObjectServers HIGHLOAD and LOWLOAD
# declare the loads array
LOWLOAD = registertarget( "NCOMS_LOW", "", "alerts.status")
HIGHLOAD = registertarget( "NCOMS_HIGH", "", "alerts.status")
array loads;

# initialize array items with the number of seconds samples may span and
# number of samples to maintain.

if ( match("", loads[@Node]) ){
    loads[@Node] = "2.50"
}
if ( match("" , %general_load) ){
    %general_load="2.50"
}
loads[@Node] = updateload(loads[@Node])
%general_load=updateload(%general_load)
if ( int(getload(loads[@Node]) ) > 5 ){
```

```
    log(WARN, $Node + " is creating more than 5 events per second")
}
if ( int(getload(%general_load)) > 80){
    log(WARN, "Probe is creating more than 80 events per second - switching to HIGHLOAD")
    settarget(HIGHLOAD)
}
```

Related reference:

"Search and replace function" on page 53
Use the `regreplace` function to perform search and replace operations on strings by using regular expressions.

"Examples of the looping function" on page 25
Use these examples of the FOREACH looping statement to help you deploy the function in your Tivoli Netcool/OMNIbus environment.

Chapter 3. Probe rules file customizations

You can extend the functionality of probes by using a number of resources that are provided in the `$NCHOME/omnibus/extensions` directory of your Tivoli Netcool/OMNIbus installation. Sample SQL and probe rules files can be used to customize any probe for event flood detection or anomalous event rates, and for self monitoring by using statistical data that is captured and processed by the probe.

Detecting event floods and anomalous event rates

Event floods can cause ObjectServer outages, and can lead to extended periods where there is no visibility of network events. An unusually low or high rate of receipt of events can also be indicative of a problem or change in the source, which needs to be addressed.

You can configure a probe to detect an event flood condition or anomalous event rates, and to perform remedial actions. Some usage scenarios are as follows:

- When an event flood is detected, you want to discard all further alerts until the event rate falls back below a predefined threshold, which indicates that the event flood is over.
- When an event flood is detected, you want to divert all further alerts to an alternative ObjectServer until the event rate falls below a predefined threshold, which indicates that the event flood is over.
- When an event flood is detected, you want to send an informational alert to the ObjectServer at the start of the event flood, and another informational alert when the event flood finishes.
- When an event flood is detected, you want to forward only major and critical alerts to the primary ObjectServer, and to discard all other alerts or divert them to an alternative ObjectServer until the event flood is deemed to be over.
- When an anomalous rate of receipt of events is detected, you want to send the ObjectServer an informational alert that describes the nature of the anomalous event rate.

When flood control is enabled, and an unusually high or low event rate is detected, the event list can be populated with multiple events for the same issue. These events are not cleared from the event list when the issue is resolved. You can manually delete unwanted entries.

Two secondary rules files are provided that you can use to configure a probe to detect when it is subject to an event flood or other anomalous event rates. These rules files are provided in the `$NCHOME/omnibus/extensions/eventflood` directory. Details of these files are as follows:

- `flood.rules`: This flood rules file contains the event rate calculations and logic to detect event floods and anomalous event rates. This file calculates an average rate of receipt of events for the probe, and then sets upper and lower event rate thresholds as a configurable percentage of this average event rate. The current event rate is compared to these event rate thresholds to determine whether the probe is subject to an anomalous rate of receipt of events. The flood rules file also uses predefined thresholds for a normal event rate and an event flood rate

to determine whether the probe is subject to an event flood. Optional remedial actions are included to generate informational alerts, discard alerts, or divert alerts.

The flood rules file also writes a stream of messages to the probe log file, detailing its processing results. To accommodate these messages, you should consider increasing the maximum size that is currently specified for the log file.

- `flood.config.rules`: This rules file defines configuration elements and their values, which are used within the `flood.rules` file. These elements include defined threshold multipliers and limits, the sampling time, the time windows and maximum number of events allowed for computing average, flood, and anomalous loads, and variables associated with remedial actions.

To configure the probe, you must embed updated copies of these two files within the main probe rules file (a requirement for `flood.config.rules`), or one of your secondary rules files. When the probe rules file is processed, remedial actions are performed, as per your specifications.

Related reference:

"Flood rules file" on page 74
Use the `flood.rules` file to calculate event rates for detecting event floods or an anomalous receipt of events, and to specify remedial actions. This file must be used in conjunction with the flood configuration rules file `flood.config.rules`.

"Flood configuration rules file" on page 71
Use the `flood.config.rules` file to set the configuration variables that are used to detect an event flood or an anomalous event rate. This file must be used in conjunction with the flood rules file `flood.rules`.

Configuring probes to detect event floods and anomalous event rates

You can configure probes to detect an event flood and anomalous event rates by using the secondary rules files that are installed in the `$NCHOME/omnibus/extensions/eventflood` directory.

Procedure

To enable these features for a probe:

1. Go to the `$NCHOME/omnibus/extensions/eventflood` directory.
2. Copy the `flood.config.rules` and `flood.rules` files to a preferred local or remote directory where the primary rules file for the probe or any secondary rules files are stored. Remove the default read-only permissions from the `flood.config.rules` and `flood.rules` files.
3. Edit the `flood.config.rules` and `flood.rules` files as appropriate for your requirements. You can comment out any unrequired sections. For example, if you do not want to discard alerts during an event flood, you can comment out the conditional statement in the `flood.rules` file. In the sample `flood.rules` file, event rates are calculated against all event sources that send data to the probe.
4. Use an `include` statement to embed the `flood.config.rules` file at the very beginning of your set of rules, before any processing statements. This file contains an array declaration and `registertarget` statements, which must be defined at the start of a rules file.

   Note: If you include other rules files that use the `registertarget` command (for example, `tivoli_eif_virtualization_pt1.rules`), you must arrange the

main probe rules file such that all the `registertarget` instructions are placed together at the start of the file, followed by all the array instructions, followed by all the variable declarations. If you do not include these elements in this order, the rules file will produce syntax errors when the probe is run.

5. Use an `include` statement to embed the `flood.rules` file (within the primary rules file or another secondary rules file) in the section that defines your set of rules. If you want to conditionally discard or divert alerts with particular severity levels during the event flood, you must include the `flood.rules` file towards the end of the set of rules, at a position where the severity of the alert has already been determined.
6. After updating the probe rules file with the `include` statements, have the probe re-read the rules file.
7. In the probe properties file, set the **MaxLogFileSize** property to a value that is large enough to accommodate the extra log file messages that are generated when the `flood.rules` file is processed.

Related concepts:

"Properties file" on page 5
Probe properties define the environment in which the probe runs.

"Rules file" on page 7
The rules file defines how the probe processes event data to create a meaningful alert. For each alert, the rules file also creates an identifier that uniquely identifies the problem source.

Related tasks:

"Rereading the rules file" on page 60
Because probes read the rules file only on startup, you must force the probe to reread the rules whenever you make changes to it. The probe processes the reread request only on receipt of a new event. If the probe is idle or is already processing an event, it will not reread the rules file until a new event is received.

Related reference:

"Flood rules file" on page 74
Use the `flood.rules` file to calculate event rates for detecting event floods or an anomalous receipt of events, and to specify remedial actions. This file must be used in conjunction with the flood configuration rules file `flood.config.rules`.

"Flood configuration rules file" on page 71
Use the `flood.config.rules` file to set the configuration variables that are used to detect an event flood or an anomalous event rate. This file must be used in conjunction with the flood rules file `flood.rules`.

"Embedding multiple rules files in a rules file" on page 30
You can include a number of secondary rules files in your main rules file by using the `include` statement.

Chapter 6, "Common probe properties and command-line options," on page 113
A number of properties and command-line options are common to all probes and TSMs.

Protecting the ObjectServer against event floods

Configure event flood control so that the ObjectServer can detect when it is at risk of being overloaded by connected clients and send a message to the clients to instruct them to take remedial action.

About this task

Two metrics are used to calculate the load on the ObjectServer from connected clients. An average is calculated over a specified time period and thresholds are applied to determine whether the ObjectServer should activate flood control. By default, for each metric, the average is calculated every 60 seconds, over a 300 second period. If the average is in excess of the threshold for both metrics then flood control is invoked and an event is raised. These metrics, thresholds, and default values are described in the following table.

Table 18. Metrics and thresholds used to determine when to activate event flood control

Metric	Threshold	Description
Time that is spent by the ObjectServer processing requests from clients	Time that is spent over a 60-second period. The default is 40 seconds.	To calculate the value, the catalog.profiles table is read.
Time that is spent by the ObjectServer in triggers.	Time that is spent over a 60-second period. The default is 30 seconds.	To calculate the value, the catalog.trigger_stats table is read.

By default, flood control mode is deactivated again at the point when processing time falls below both defined thresholds, and flood control has been activated for 300 seconds. A resolution event is raised when flood control is deactivated.

All default values are configurable.

Configuration for event flood protection is supplied in the `$NCHOME/omnibus/extensions/eventflood/flood_control.sql` file. To configure event flood protection, apply this file to the ObjectServer schema. The configuration contains the default values.

Procedure

To configure event flood control:

1. Change to the `$NCHOME/omnibus/extensions/eventflood` directory and copy the `flood_control.sql` file to a preferred location.
2. To change the default values for event flood protection, open the file and edit the content at the points that are described in the following table.

Table 19. Default values in the `flood_control.sql` file to edit to change flood control protection values

Code section	Description
`set window_size = 5;`	The time, in minutes, over which the average values are calculated for the metrics. The default is `5`.
`if( avg_trigger_per >= 0.5 )`	The threshold, expressed as a fraction of a minute, for time that is spent in triggers. The default is `0.5`, that is, 30 seconds in a minute.

Table 19. Default values in the flood_control.sql file to edit to change flood control protection values (continued)

Code section	Description
`if( avg_client_per >= 0.66 )`	The threshold, expressed as a fraction of a minute, for time that spent processing SQL requests from clients. The default is `0.66`, that is, 40 seconds in a minute.
`if( elapsed >= 300 )`	The time, in seconds that must elapse after the flood protection values fell below the defined thresholds before flood protection mode is deactivated.
`'ObjectServer ' + getservername() + ' is currently in flood',`	The text for the event that is raised when the ObjectServer enters flood control. Do not change the `' + getservername() + '` element.
`'ObjectServer ' + getservername() + ' is ending flood control',`	The text for the resolution event that is raised when flood control is deactivated.

3. Apply the configuration for event flood protection to the ObjectServer schema by running the SQL interactive interface and issuing the following command:

   UNIX Linux `$NCHOME/omnibus/bin/nco_sql -user username -password password -server servername < directory_path/flood_control.sql`

   Windows `"%NCHOME%\omnibus\bin\isql" -U username -P password -S servername -i directory_path/flood_control.sql`

   Where *username* is a valid user name, *password* is the corresponding password, *servername* is the name of the ObjectServer, and *directory_path* is the fully qualified directory path to the `.sql` file.

Flood configuration rules file

Use the `flood.config.rules` file to set the configuration variables that are used to detect an event flood or an anomalous event rate. This file must be used in conjunction with the flood rules file `flood.rules`.

The entries in the `flood.config.rules` file, and the actions that you can take to amend the values, are described in the following table. The entries are shown in the order in which they are defined in the file, starting from the top.

Table 20. flood.config.rules file entries

Entry	Description	Action
`DefaultOS = registertarget(%Server, %ServerBackup, "alerts.status")`	This statement registers the default ObjectServer (and backup ObjectServer, if one is configured) as a target for alerts. In the flood rules file, this is the target ObjectServer to which an informational alert is sent when the current event rate from probe sources is unusually high or low, or when an event flood starts and ends. The default table to which the alert is sent is alerts.status.	Change the alerts table name to a preferred valid name.

Table 20. flood.config.rules file entries (continued)

Entry	Description	Action
`#FloodEventOS = registertarget("NCOMS_BK", "", "alerts.status")`	This commented-out line registers an NCOMS_BK backup ObjectServer. In the flood rules file, this is an alternative target ObjectServer to which alerts with particular severity levels can be diverted during an event flood.	Uncomment this line if you want to divert alerts to this ObjectServer when an event flood is detected. Change the ObjectServer name and the alerts table name to preferred valid names.
`array event_rate_array`	This array is defined to hold all the event rate calculation variables. These variables are used throughout the flood rules file.	N/A

`$average_event_rate_time_window`
 	
`$average_event_rate_max_sample_size`
	These elements store values that are used to calculate what is considered to be the average (or normal) rate of receipt of events:
	• The `$average_event_rate_time_window` element defines the maximum time window (in seconds) for which events are kept. This value depicts a rolling time window, which is updated by calling the `updateload` function. The `$average_event_rate_time_window` element also sets the *training period*, which is the length of time the probe runs to determine the average or normal event rate.
	• The `$average_event_rate_max_sample_size` element defines the maximum number of events to keep during the average event rate time window.
	
	In the flood rules file, these elements are used to capture the event count in the last *n* seconds before the current time, and to calculate the average event rate during this period.
		Change the default values as appropriate for your requirements.

`$flood_detection_time_window`
 	
`$flood_detection_max_sample_size`
	These elements store values that are used to calculate the event flood detection rate, in order to determine whether an event flood is imminent:
	• The `$flood_detection_time_window` element defines the maximum time window (in seconds) for which events are kept. This value depicts a rolling time window, which is updated by calling the `updateload` function.
	• The `$flood_detection_max_sample_size` element defines the maximum number of events to keep during this period.
	
	In the flood rules file, these elements are used to capture the event count in the last *n* seconds before the current time, and to calculate the flood detection rate during this period.
		Change the default values as appropriate for your requirements.

`$flood_detection_startup_time`	This element defines the number of seconds over which the probe runs before event flood detection can begin.	Set a value.

Table 20. flood.config.rules file entries (continued)

Entry	Description	Action

`$anomaly_detection_time_window`
 	
`$anomaly_detection_max_sample_size`
	These elements store values that are used to calculate the rate of receipt of events for detecting an anomalous flow:
	• The `$anomaly_detection_time_window` element defines the maximum time window (in seconds) for which events are kept. This value depicts a rolling time window, which is updated by calling the `updateload` function.
	• The `$anomaly_detection_max_sample_size` element defines the maximum number of events to keep during this period.
	
	In the flood rules file, these elements are used to capture the event count in the last *n* seconds before the current time, and to calculate the event rate during this period.
		Change the default values as appropriate for your requirements.

`$flood_detection_event_rate_flood_threshold`
 	
`$flood_detection_event_rate_normal_threshold`
	These elements store values that are used to specify event rate thresholds for detecting an event flood or a normal event rate.
	
	If the number of events received per second exceeds the value specified for the `$flood_detection_event_rate_flood_threshold` element, event flood detection is triggered.
	
	If the number of events received per second is less than the value specified for the `$flood_detection_event_rate_normal_threshold` element, a normal event rate is assumed.
		Change the default values as appropriate for your requirements.
		
		Ensure that the value of `$flood_detection_event_rate_normal_threshold` is lower than `$flood_detection_event_rate_flood_threshold`.

`$lower_event_rate_threshold_multiplier`
 	
`$upper_event_rate_threshold_multiplier`
	The `$lower_event_rate_threshold_multiplier` element sets the multiplier value that is used to calculate the lower event rate threshold for detecting an anomalous event rate.
	
	The `$upper_event_rate_threshold_multiplier` element sets the multiplier value that is used to calculate the upper event rate threshold for detecting an anomalous event rate.
	
	In the flood rules file, the average event rate is multiplied by these values to set the thresholds for determining unusually low or unusually high event rates.
		Change the default values as appropriate for your requirements.

`$discard_event_during_flood`	This element defines whether an alert is discarded during an event flood. A value of 1 equates to TRUE and a value of 0 equates to FALSE. In the flood rules file, if the `$discard_event_during_flood` value is 1 and the alert is of a lower severity than the value specified for `$forward_event_minimum_severity`, the alert will be discarded.	Change the default value as appropriate for your requirements.

Table 20. flood.config.rules file entries (continued)

Entry	Description	Action
`$divert_event_during_flood`	This element defines whether an alert is diverted to an alternative ObjectServer during an event flood. A value of 1 equates to TRUE and a value of 0 equates to FALSE. In the flood rules file, if the value of `$divert_event_during_flood` is 1 and the alert is of a lower severity than the value specified for `$forward_event_minimum_severity`, the alert will be diverted.	To divert an alert of a particular severity, ensure that the `$divert_event_during_flood` value is set to 1 in the `flood.config.rules` file. Also ensure that the `registertarget` statement with the target of `FloodEventOS` (defined at the top of the file) is uncommented and configured with the appropriate ObjectServer name and table.
`$forward_event_minimum_severity`	This element is set to a value of 4 to indicate that events with a severity of major or critical should be forwarded to the primary ObjectServer during an event flood. In the flood rules file, this element is used in the IF condition that defines whether alert is discarded or diverted during an event flood.	Accept or change the default value as appropriate for your requirements.

Related reference:

"Flood rules file"
Use the `flood.rules` file to calculate event rates for detecting event floods or an anomalous receipt of events, and to specify remedial actions. This file must be used in conjunction with the flood configuration rules file `flood.config.rules`.

Flood rules file

Use the `flood.rules` file to calculate event rates for detecting event floods or an anomalous receipt of events, and to specify remedial actions. This file must be used in conjunction with the flood configuration rules file `flood.config.rules`.

The logic in the `flood.rules` file is described here to help you understand the sample configuration provided.

The first time that the probe processes the rules file, the array (`event_rate_array`) is initialized, and event rate array variables are used to:

- Set the rolling time window and the maximum number of events that can be used for calculating an average load, a flood detection load, and an anomaly detection load. The loads are defined in the format *time_window_in_seconds.max_number_of_samples* by using elements defined in the `flood.config.rules` file.
- Set the event rate mode to *normal*.
- Store the current timestamp as the startup time for the probe.
- Indicate that the average event rate is not yet calculated.

Anomalous event rate calculations

During the first `$average_event_rate_time_window` seconds (default 10 seconds) after the probe starts, an event count is maintained in order to calculate an average event rate for the probe.

At the end of this period, upper and lower event rate thresholds are calculated as percentages of the average event rate. The `$upper_event_rate_threshold_multiplier` and `$lower_event_rate_threshold_multiplier` elements, which are defined in the `flood.config.rules` file, are used to calculate these thresholds. After the average rate is determined, the probe periodically checks the current event rate, and compares it against the upper and lower event rate thresholds as follows:

1. The `updateload` function is used to capture the time window (prior to the current time) and the event count that are used for determining the current event rate for anomalous events. Note that a default time window of one minute, as set by the `$anomaly_detection_time_window` element, is used.
2. The `getload` function is used to calculate the current event rate as events per second.
3. The current event rate is compared to the upper and lower event rate thresholds to determine whether the probe is subject to an anomalous rate of receipt of events.

If an unusually low or unusually high number of events is detected, the `genevent` function is used to generate and send informational alerts to the target ObjectServer that is registered in the `flood.config.rules` file as `DefaultOS`.

As an example, suppose 200 events are received within the average event rate time window, resulting in an average event rate of 20 events per second. Also assume that the `$upper_event_rate_threshold_multiplier` element is set to 5, and the `$lower_event_rate_threshold_multiplier` element is set to 0.1 in the `flood.config.rules` file.

The upper event rate threshold can be calculated as follows:

average event rate * 5 = 100 events per second

The lower event rate threshold can be calculated as follows:

average event rate * 0.1 = 2 events per second

If the current event rate is calculated as 120 events per second, the probe will generate and send an alert to the target `DefaultOS` ObjectServer, with details about the high event rate. If the current event rate is calculated as 1 event per second, the probe will generate and send an alert to the target `DefaultOS` ObjectServer, with details about the low event rate.

Flood detection calculations

When the probe starts, an exclusion period is observed for flood detection. This period is of a fixed duration from the probe startup time, and is set by the `$flood_detection_startup_time` element in the `flood.config.rules` file.

When this period ends, the `updateload` function is used to capture the time window (prior to the current time) and the event count that are used for determining the current event rate for flood detection. The `getload` function is then used to calculate the current event rate as events per second. The current event rate is compared to the event rate thresholds, which are defined in the `flood.config.rules` file, for an event flood and for a normal rate of events. The event mode is then set to either `flood` or `normal`, as appropriate.

If the current event mode is `flood`, the probe determines whether the event flood has just started, is in progress, or has just ended, and takes the appropriate action:

- The `genevent` function is used to generate and send an informational alert to the target ObjectServer that is registered in the `flood.config.rules` file as `DefaultOS`. This informational alert either indicates that an event flood has just started or has just ended, and includes details about the event flood.
- While the event flood is in progress, alerts can be discarded if their severity is below a defined minimum level. The default configuration discards alerts with a severity value that is less than 4 (major).
- While the event flood is in progress, alerts can alternatively be diverted to an ObjectServer when the alert severity is below a defined minimum level. This ObjectServer is registered in the `flood.config.rules` file as the target (`FloodEventOS`) for events during an event flood. The default configuration diverts events with a severity value that is less than 4 (major).

Message logging

Various messages are written to the log file as the flood rules file is processed. Details recorded include:

- The probe startup timestamp
- The average event rate
- The event loads
- Unusually high or low event rates
- Flood detection event rates, flood status, remedial actions, event count, and flood duration

Related reference:

"Flood configuration rules file" on page 71
Use the `flood.config.rules` file to set the configuration variables that are used to detect an event flood or an anomalous event rate. This file must be used in conjunction with the flood rules file `flood.rules`.

Chapter 6, "Common probe properties and command-line options," on page 113
A number of properties and command-line options are common to all probes and TSMs.

"Sending alerts to alternative ObjectServers and tables" on page 47
The `registertarget`, `genevent`, `settarget`, and `setdefaulttarget` functions enable you to send alerts to one or more ObjectServers, and to define the distribution of alerts across the ObjectServers.

Enabling self monitoring of probes

You can configure probes to generate ProbeWatch Heartbeat events as a self-monitoring mechanism to help monitor performance, diagnose performance problems, and highlight possible performance bottlenecks before they begin to affect the system.

A ProbeWatch Heartbeat event is generated by the probe, and is not triggered by an event (or absence of events) from the managed entity. The ProbeWatch Heartbeat event is generated at a configurable interval, which is controlled by the **`ProbeWatchHeartbeatInterval`** property. This interval is set to 60 seconds by default. A ProbeWatch Heartbeat event can either be used as a heartbeat to confirm that the probe is still functioning, or can be used to transport probe statistics:

- The presence of a regularly-occurring ProbeWatch Heartbeat event enables you to assess whether a probe is inactive due to a lack of incoming events from its

source, due to probe failure, or due to a communications failure with the ObjectServer. The value that you specify for the **ProbeWatchHeartbeatInterval** property defines the maximum time over which the probe can remain silent before an indication is required that the probe is still functioning. If no other events have been sent in the previous **ProbeWatchHeartbeatInterval** time window, the probe indicates that it is still active (but just not receiving any events) by sending a ProbeWatch Heartbeat event to the ObjectServer. The Summary field of this ProbeWatch Heartbeat event is populated with the following text: `Heartbeat ...`

- A ProbeWatch Heartbeat event also acts as a carrier for statistical data for probes, such as the processing throughput of probes, and CPU and memory resource utilization. Individual probes can capture usage and resource information, which is then manipulated within the rules file to calculate metrics by using a set of dedicated properties. These metrics can be transferred into the ObjectServer either in one single ProbeWatch Heartbeat event by using nvp_add functions to specify name-value pairs of extended attributes, or within multiple ProbeWatch Heartbeat events that are generated using the genevent function. The generated events are forwarded to the ObjectServer at the interval defined by the **ProbeWatchHeartbeatInterval** property.

The metrics provided in the ProbeWatch Heartbeat events can be analyzed to identify how the different components of the system are running, and to identify potential problems before performance begins to degrade. The data can also be collated for use in reports and charts that can be used to help demonstrate how much of a return on investment is being made.

Related reference:

"ProbeWatch and TSMWatch messages" on page 208
In some situations, a probe or TSM generates events of its own. These events can provide information (such as startup or shutdown messages) or identify problems.

Configuration setup for self monitoring of probes

Probes can be configured to generate statistical data that can be used to assess system performance and to help calculate return on investment.

The following figure shows the configuration setup for probe self monitoring.

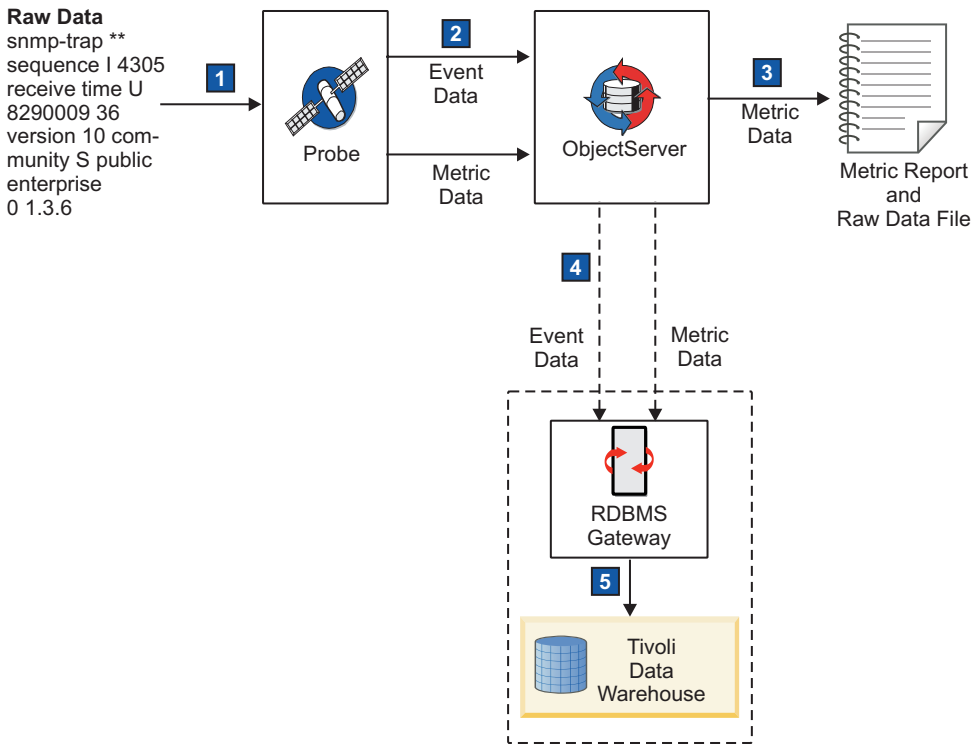

Figure 3. Configuration and data flow for probe self monitoring

The configuration flow is as follows:

1 Usage and resource information is captured together with raw data that is sent to the probe.

2 The probe processes the usage and resource information in order to calculate performance metrics, and to generate ProbeWatch Heartbeat events that are populated with these metrics. The probe also processes the raw data in order to generate generic (standard) events.

Both sets of events are forwarded to the ObjectServer.

3 ObjectServer automations are used to produce a basic textual report that summarizes the statistical information generated by the probe .

4 and 5

The generic event data and ProbeWatch Heartbeat events can optionally be exported from the ObjectServer into Tivoli Data Warehouse by using a relational database management system (RDBMS) gateway. These metrics can then be used for subsequent reporting in Tivoli Common Reporting.

Note: These steps (4 and 5) are outside the scope of the probe self-monitoring functionality provided by Tivoli Netcool/OMNIbus. Integration with IBM Tivoli Monitoring (which provides Tivoli Data Warehouse) and additional configuration will be required for this additional reporting.

Tivoli Netcool/OMNIbus configuration files for the self monitoring of probes

When you install Tivoli Netcool/OMNIbus, a number of configuration files are provided for configuring probes to collect and process statistical data for self monitoring. Samples of these configuration files are available in the `$NCHOME/omnibus/extensions/roi` directory.

Details of the configuration files are as follows:

- `probestats.sql` file: This file provides a set of automations to capture the incoming statistical data collected for a probe, and to log the data to a file. Tables are also created in the ObjectServer to store the probe metrics and to record the last reporting period for the data. Note that the probe metrics are stored in the specially-created master.probestats table, rather than the alerts.status table.

  The log file that is created is similar to the profiling log, and includes:

  – Individual metrics for each connected probe; for example, the number of events processed, generated, and discarded since the last reporting period

  – A set of collated metrics; for example, the total number of alerts.details and alerts.journal inserts since the last reporting period

  You can review this SQL file to familiarize yourself with the potential changes that will be applied to the ObjectServer.
- `probewatch.include` file: This customized rules file is provided for use with probes, and must be embedded within the main rules file for the probe. The `probewatch.include` file expands on the original default ProbeWatch-specific rules. This file contains new CASE statements for two additional ProbeWatch messages and for the ProbeWatch Heartbeat events, which act as a carrier for statistical data.

  The `probewatch.include` rules file is generic to all probes. You can customize and share this file between multiple (or all) probes to centralize the administration of ProbeWatch Heartbeat events.
- `Omnibus_TDW_Reports_ROI.zip` file: This archive file contains a set of sample reports that require user customization, and integration with Tivoli Data Warehouse and Tivoli Common Reporting. Working knowledge of these components is required to support this configuration.

A number of statistical properties are also added to the configuration for probes. These properties are used to collect usage and resource information that is specific to each probe. The statistical properties are different from the standard probe properties because they cannot be set to a meaningful value in the properties file, and they cannot be run as command-line options. These property names are all prefixed with **`OplStats`**, and are displayed in the output obtained when the probe is run with the `-dumpprops` command-line option.

The statistical properties are as follows:

Table 21. Statistical properties for probes

Property	Description
`OplStatsCPUTimeSec`	The CPU time consumed by the probe in seconds. Example: If 6.002345 seconds CPU time has been consumed by the probe, **`OplStatsCPUTimeSec`** = 6

Table 21. Statistical properties for probes (continued)

Property	Description
OplStatsCPUTimeUSec	The subsecond component of CPU time consumed by the probe, in millionths of a second. Example: If 6.002345 seconds CPU time has been consumed by the probe, **OplStatsCPUTimeUSec** = 2345
OplStatsRulesFileTimeSec	The time spent processing rules in seconds. Example: If 4.372700 seconds is spent processing rules, **OplStatsRulesFileTimeSec** = 4
OplStatsRulesFileTimeUSec	The subsecond component of time spent processing rules, in millionths of a second. Example: If 4.372700 seconds is spent processing rules, **OplStatsRulesFileTimeUSec** = 372700
OplStatsProbeStartTime	The time (in UNIX epoch time) at which the probe was started.
OplStatsMemoryInUse	The memory footprint (in KB) of the probe.
OplStatsNumberEvents	The number of events (including ProbeWatch events) that the probe has received from its event source since the probe started.
OplStatsNumberEventsDiscarded	The number of events that are discarded after rules processing.
OplStatsNumberEventsGenerated	The number of events that are generated using the genevent function in the rules file.

Configuring probes for self monitoring

As a self-monitoring mechanism, you can configure a probe to collect statistical data about the amount of memory used for various processing operations, and the number of events received, discarded, and generated.

About this task

To configure a probe to collect and process statistical data:

Procedure

1. Go to the `$NCHOME/omnibus/extensions/roi` directory.
2. Copy the `probestats.sql` file to the `$NCHOME/omnibus/etc` directory, or another preferred location. Apply the ProbeWatch Heartbeat customization to the ObjectServer schema by running the following command from the SQL interactive interface:

   UNIX Linux `$NCHOME/omnibus/bin/nco_sql -user` *`username`* `-password` *`password`* `-server` *`servername`* `< directory_path/probestats.sql`

   Windows `"%NCHOME%\omnibus\bin\isql" -U` *`username`* `-P` *`password`* `-S` *`servername`* `-i` *`directory_path`*`\probestats.sql`

   In these commands, *username* is a valid user name, *password* is the corresponding password, *servername* is the name of the ObjectServer, and *directory_path* is the fully-qualified directory path to the `.sql` file.

   The `probestats.sql` file adds a set of tables and triggers to the ObjectServer.

3. Copy the $NCHOME/omnibus/extensions/roi/probewatch.include file to a preferred local or remote directory where the main rules file or any secondary rules files for the probe is stored. This file is designed to replace the logic in the ProbeWatch section of your primary rules file, which is typically coded as follows:

```
if( match( @Manager, "ProbeWatch" ) )
{
        switch(@Summary)
        {
        case "Running ...":
                @Severity = 1
                @AlertGroup = "probestat"
                @Type = 2
        case "Going Down ...":
                @Severity = 5
                @AlertGroup = "probestat"
                @Type = 1
        default:
                @Severity = 1
        }
        @AlertKey = @Agent
        @Summary = @Agent + " probe on " + @Node + ": " + @Summary
}
else
{
        ...probe specific rules...
}
```

   The code shown in bold text needs to be replaced with an include statement that enables you to embed the contents of the probewatch.include file, as instructed in step 5 on page 83.

4. Remove the default read-only permissions from your copy of the probewatch.include file and review the file to familiarize yourself with its contents. Then edit the file as follows:
   - Update any of the elements at the top of the file to define how a ProbeWatch Heartbeat event should be processed. Use the number sign (#) to comment out any elements that you do not require. The processing logic for these elements is coded within the case "Heartbeat ..." statement in the ProbeWatch section of the file.

Table 22. Elements for ProbeWatch Heartbeat events

Element	Action
$OplHeartbeat_discard	Set this value to 1 if you want to discard the ProbeWatch Heartbeat event. Set this value to 0 if you want to forward the ProbeWatch Heartbeat event to the ObjectServer.
$OplHeartbeat_populate_master_probestats	Set this value to 1 to enable a new probe metrics event to be generated by using the genevent function, which is defined within the case "Heartbeat ..." statement. The event data consists a set of OplStats probe metrics, which are forwarded to the master.probestats table that was created when you ran the probestats.sql script. Set this value to 0 if you do not want to generate this event for insertion into the master.probestats table.

Table 22. Elements for ProbeWatch Heartbeat events (continued)

Element	Action
`$OplHeartbeat_write_to_probe_log`	Set this value to 1 if you want to record the OplStats metrics in the probe log file. Details are logged at the INFO level. The metric details that are logged are defined in the `case "Heartbeat ..."` statement. Set this value to 0 if you do not want to record the metrics in the log file.
`$OplHeartbeat_generate_threshold_events`	Set this value to 1 if you want to generate threshold events that indicate when a particular probe metric violates a defined threshold. By default, no code is provided for threshold events within this rules file because individual preferences can vary widely. If you require threshold events, you must first decide which thresholds you want to monitor. Then, within the `case "Heartbeat ..."` statement, provide the code for generating threshold events.

- In addition to the standard CASE statements, the file includes the following two CASE statements, which contain the logic for two new ProbeWatch events that provide feedback when a probe re-reads its rules files on receipt of a SIGHUP signal. The first CASE statement applies when the re-read was successful:

```
case "Rules file reread upon SIGHUP successful ...":
        @Severity = 1
        @AlertGroup = "rules"
        @Type = 2
```

  The second CASE statement applies when the re-read was unsuccessful. This section of code includes two elements ($msg and $file), where $msg is the error message as reported in the probe log file, and $file is the name of the file where the error exists.

```
case "Rules file reread upon SIGHUP failed ...":
        @Severity = 4
        @AlertGroup = "rules"
        @Type = 1
        if( exists( $msg ) )
        {
                @Summary = @Summary + "("+$msg+")"
        }
        if( exists( $file ) )
        {
                @Summary = @Summary + " in file "+$file
        }
```

  If you do not require these, use the `discard` function to prevent them from being sent to the ObjectServer.
- The final CASE statement (`case "Heartbeat ..."`) contains a set of conditional statements for calculating the probe metrics and processing the data. IF statements are provided with the logic to discard events and to write the probe metrics to a log file. Some user input is also required:

Table 23. case "Heartbeat ..." sections that require user input

Locate the section of code that begins with the following lines:
`if( int( $OplHeartbeat_populate_master_probestats ) == 1 )`
`{`
`log( DEBUG, "HEARTBEAT - SENDING PROBESTATS TO MASTER.PROBESTATS" )`
`...`
This section of code contains a genevent statement with a `DefaultOS` placeholder that identifies a target, registered ObjectServer. This target must be defined in a registertarget statement in the main rules file. Replace this placeholder with the target ObjectServer to which you want to send events.

Locate the section of code that begins with the following lines:
`if( int( $OplHeartbeat_generate_threshold_events ) == 1 ) {`
`#`
`# Area to generate user defined threshold events using genevent`
`...`
If you set the `$OplHeartbeat_generate_threshold_events` element to 1 at the top of the file, you must enter the code for the type of threshold events that you want to monitor.
You can ignore this section if you do not require threshold events.

- If you have modified the ProbeWatch section of your main rules file (typically `$NCHOME/omnibus/probes/`*`arch`*`/`*`probename`*`.rules`), you must make the same modifications to the `probewatch.include` file.
- If the main rules file includes additional ProbeWatch sections that contain code for different ProbeWatch messages that are not covered in the `probewatch.include` file, copy this additional code into the `probewatch.include` file.

Tip: After making all the changes to the `probewatch.include` file, run the Probe Rules Syntax Checker (**nco_p_syntax**) to test the syntax of the rules file.

5. Embed the updated `probewatch.include` file in your main probe rules file by using an `include` statement. Ensure that the path in the `include` statement points to the location where the updated `probewatch.include` file is stored.

```
if( match( @Manager, "ProbeWatch" ) )
{
        include "directory_path/probewatch.include"
}
else
{
        ...probe specific rules...
}
```

6. Specify the interval, in seconds at which probe heartbeat messages are generated, by setting **ProbeWatchHeartbeatInterval** in the probe properties file.
   - Set a positive number to generate the events
   - Set 0 (zero) or a negative number for no events
7. Ensure that the stats_triggers trigger group is enabled. The triggers that are added by the `probestats.sql` file are assigned to this trigger group, which must be enabled for the triggers to run. You can enable the trigger group by using Netcool/OMNIbus Administrator or the ALTER TRIGGER GROUP command, as described in the *IBM Tivoli Netcool/OMNIbus Administration Guide*.
8. Enable the probe_statistics_cleanup trigger, which by default is set to delete probe statistics that are over an hour old. You can change this default period to increase the length of time for which statistics are stored.
9. Start the probe.

The probe metrics that are collected are recorded in the log file `$NCHOME/omnibus/log/server_name_probestats.log`, where *server_name* is the ObjectServer name.

Related concepts:

Chapter 4, “Running probes,” on page 85
When running a probe, you can specify properties in a properties file or options at the command line to configure settings for the probe.

“Properties file” on page 5
Probe properties define the environment in which the probe runs.

“Rules file” on page 7
The rules file defines how the probe processes event data to create a meaningful alert. For each alert, the rules file also creates an identifier that uniquely identifies the problem source.

Related tasks:

“Testing rules files” on page 58
You can test the syntax of a rules file by using the Probe Rules Syntax Checker, **`nco_p_syntax`**. This is more efficient than running the probe to test that the syntax of the rules file is correct.

Related reference:

“Sending alerts to alternative ObjectServers and tables” on page 47
The `registertarget`, `genevent`, `settarget`, and `setdefaulttarget` functions enable you to send alerts to one or more ObjectServers, and to define the distribution of alerts across the ObjectServers.

“Embedding multiple rules files in a rules file” on page 30
You can include a number of secondary rules files in your main rules file by using the `include` statement.

Chapter 6, “Common probe properties and command-line options,” on page 113
A number of properties and command-line options are common to all probes and TSMs.

Chapter 4. Running probes

When running a probe, you can specify properties in a properties file or options at the command line to configure settings for the probe.

A probe has default values for each property. In an unedited properties file, all properties are listed with their default values, commented out with a hash symbol (#) at the beginning of the line.

You can edit the properties file before running the probe, or while the probe is running. If you edit the properties file while the probe is running, the changes that you make take effect the next time you start the probe. You can edit probe property values using a text editor. To override a default value, you must change the setting in the properties file and then remove the hash symbol.

If you change a property setting on the command line when starting a probe, this overrides both the default value and the setting in the properties file. To simplify the command that you type to run the probe, add as many properties as possible to the properties file instead of using the command-line options.

When running a probe, you must also set up your rules file to define how the probe should process event data. You can edit the rules file before running the probe, or while the probe is running. If you edit the rules file while the probe is running, you must force the probe to re-read the rules file, for the changes to take effect. You can edit the rules file using a text editor.

Tip: Always read the publication that is specific to the probe you are running for additional configuration information.

Related concepts:

"Properties file" on page 5
Probe properties define the environment in which the probe runs.

"Rules file" on page 7
The rules file defines how the probe processes event data to create a meaningful alert. For each alert, the rules file also creates an identifier that uniquely identifies the problem source.

Related tasks:

"Rereading the rules file" on page 60
Because probes read the rules file only on startup, you must force the probe to reread the rules whenever you make changes to it. The probe processes the reread request only on receipt of a new event. If the probe is idle or is already processing an event, it will not reread the rules file until a new event is received.

Use of OMNIHOME and NCHOME environment variables

Tivoli Netcool/OMNIbus V7.0 (and earlier) uses the OMNIHOME environment variable in many configuration files. To use these files on Tivoli Netcool/OMNIbus V7.1 (and later), replace occurrences of the OMNIHOME environment variable with NCHOME/omnibus.

On UNIX and Linux operating systems, replace $OMNIHOME with $NCHOME/omnibus.

On Windows operating systems, replace %OMNIHOME% with %NCHOME%\omnibus.

Running probes on UNIX

On UNIX, you can run probes from the command line, or under process control.

Before you begin

After you install a probe, configure the properties and rules files to fit your environment. For example, if you are using a log file probe such as the HTTP Common Log Format Probe, set the **LogFile** property, so that the probe can connect to the event source.

About this task

Important: Use process control to manage probes. For further information about setting up a probe to run under process control, see the *IBM Tivoli Netcool/OMNIbus Administration Guide*.

Procedure

To run a probe, enter the following command:
$OMNIHOME/probes/nco_p_*probename* [-*option* [*value*] ...]
In this command, the *probename* is the abbreviated name of the probe that you want to run. The *-option* variable is a command-line option, and *value* is the value that you are setting the option to. Not every option requires you to specify a value. For example, to run the Sybase Probe in raw capture mode, enter:
$NCHOME/omnibus/probes/nco_p_sybase -raw.
The following command-line options are useful:

- Specify the -name command-line option to determines the name used for the probe files described in the following table:

Table 24. Names of probe files

Type of file	Path and file name
Properties file	$NCHOME/omnibus/probes/*arch*/nco_p_*probename*.props
Rules file	$NCHOME/omnibus/probes/*arch*/nco_p_*probename*.rules
Rules cache file	When the **CacheRules** property is set to 1, the rules file is cached to the file specified by the **CacheRulesFile** property. The default location is $NCHOME/omnibus/var/*probename*.rulescache.
Store-and-forward file	$NCHOME/omnibus/var/*probename*.store.*server*
Message Log file	$NCHOME/omnibus/log/*probename*.log

In these paths, *arch* represents the name of the operating system on which the probe is installed; for example, solaris2 when running on a Solaris system.

- Specify the `-propsfile` command-line option to have the value of the option override the name setting for the properties file.
- If you need to reduce the number of individual connections to the ObjectServer, you can connect multiple probes through a proxy server. Use the `-server` command-line option and specify the name of the proxy server. For more information about the proxy server, see the *IBM Tivoli Netcool/OMNIbus Administration Guide*.

Related reference:

Chapter 6, "Common probe properties and command-line options," on page 113
A number of properties and command-line options are common to all probes and TSMs.

Running probes as SUID root

You can run probes as SUID (SETUID) root to use root privileges with a nonroot probe user account. This configuration is normally only required by the SNMP Probe (**nco_p_mttrapd**) when it must bind to a host port below 1024. You might also want to run the Ping Probe (**nco_p_ping**) as SUID root sometimes.

About this task

The following procedure shows you how to configure SUID root for the SNMP Probe. The probe can be run as SUID root without compromising security because it drops its root privileges after opening the SNMP session and before the Netcool/OMNIbus probe library starts.

Procedure

1. As the root user, run the following command from the `$NCHOME/omnibus/probes/`*arch* directory to make root the owner of the probe binary:

   ```
   chown root nco_p_mttrapd
   ```

   On 64-bit systems, run the `chown` command from the `$NCHOME/omnibus/platform/`*arch*`/probes64` directory.
2. As the root user, run the following command from the same directory to enable the probe binary to be run as SUID root:

   ```
   chmod +s nco_p_mttrapd
   ```

Results

You can now run the probe from the `$NCHOME/omnibus/probes` directory as SUID root.

For more information about the SNMP Probe, see the probe reference guide.

Running probes on Windows

On Windows, you can run probes as console applications, as Windows services, or under process control.

About this task

Probes are installed as console applications by default.

For further information about setting up a probe to run under process control, see the *IBM Tivoli Netcool/OMNIbus Administration Guide*.

Related reference:

Chapter 6, “Common probe properties and command-line options,” on page 113
A number of properties and command-line options are common to all probes and TSMs.

Running a probe as a console application

Run a probe as a console application from the command line.

About this task

To run a probe as a console application, enter the following command from the probe directory:

```
nco_p_probename [ -option [ value ] ... ]
```

In this command, *probename* is the abbreviated name of the probe that you want to run. The *-option* variable is a command-line option, and *value* is the value that you are setting the option to. Not every option requires you to specify a value.

Additional command-line options are available for the Windows version of each probe. To display these, enter the following command:

```
nco_p_probename -?
```

The Windows-specific command-line options are described in the following table.

Table 25. Windows-specific probe command-line options

Command-line option	Description
`-install`	This option installs the probe as a Windows service.
`-noauto`	This option is used with the `-install` option. It disables automatic startup for the probe running as a service. If this option is used, the probe is not started automatically when the machine boots.
`-remove`	This option removes a probe that is installed as a service. It is the opposite of the `-install` command.
`-group` *string*	This option is used with the `-depend` command-line option. You can group all the probes together under the same group name. You can then force that group to be dependent on another service.
`-depend` *srv @grp ...*	This option specifies other services or groups that the probe is dependent on. If you use this option, the probe will not start until the services (*srv*) and groups (*@grp*) specified with this option have been run.
`-cmdLine` "*-option value...*"	This option specifies one or more command-line options to be set each time the probe service is restarted.

Related tasks:

"Running a probe as a service"
To run a probe as a service, use the Windows `/INSTALL` command-line option when running the probe with the `nco_p_probename` command, where *probename* uniquely identifies the probe.

Related reference:

Chapter 6, "Common probe properties and command-line options," on page 113
A number of properties and command-line options are common to all probes and TSMs.

Running a probe as a service

To run a probe as a service, use the Windows `/INSTALL` command-line option when running the probe with the `nco_p_probename` command, where *probename* uniquely identifies the probe.

About this task

After setting up the service, you can configure how the probe starts by defining the Windows services settings as follows:

Procedure

1. Click **Start** > **Control Panel**. The Control Panel opens.
2. Double-click the **Admin Tools** icon, then double-click the **Services** icon. The Services window opens.

   The Services window lists all of the Windows services currently installed on your machine. All Tivoli Netcool/OMNIbus service names start with `NCO`.
3. Use the Services window to start and stop Windows services. Indicate whether the service is started automatically when the machine is booted by clicking the **Startup** button.

   Note: If the ObjectServer and the probe are started as services, the probe may start first. The probe will not be able to connect to the ObjectServer until the ObjectServer is running.

Results

Related tasks:

"Running a probe as a console application" on page 88
Run a probe as a console application from the command line.

Chapter 5. Remotely administering probes

You can use an HTTP interface that runs over an HTTP or HTTPS server contained in the standard probe C library (libOpl) to remotely monitor and manage probes. This functionality can be used against all probes that are compatible with Tivoli Netcool/OMNIbus V7.4. A common URI is provided that contains resources for performing administration tasks.

About this task

Utilities are provided for sending HTTP requests to the probe. Probes can receive HTTP requests over HTTP ports and HTTPS ports. Both ports are disabled by default. Additional configuration is required for HTTPS.

Probes can register their own URIs to expose entities specific to a particular probe. These URIs can be used to trigger actions against the device or system that the probe is connected to or communicating with. For more information, see the documentation for the individual probe.

Enabling remote administration of probes

The HTTP interface to the probe is disabled by default. You can enable the HTTP interface by setting two properties. You can specify values for additional properties to configure the HTTP interface, for example, the number of worker threads and file serving.

About this task

Procedure

To enable and configure remote adminstration:

1. To enable remote administration with no authentication and HTTP transport, set the properties as described in the following table:

Property	Instructions
NHttpd.EnableHTTP	Set this property to TRUE.
NHttpd.ListeningPort	Set this property to the number of any free port.

   The HTTP port is enabled and set to listen for HTTP connections on the specified port number. The port listens on all available interfaces on the host.

2. To further configure the HTTP interface, set the following properties as required:

Property	Instructions
NHttpd.NumWorkThreads	Increase the value to increase the number of worker threads that handle the incoming HTTP requests. The default is 5.
NHttpd.ExpireTimeout	Set the maximum time, in seconds an HTTP/1.1 connection is left in an idle state before it is dropped. The default is 15.

Property	Instructions
NHttpd.AccessLog	Set this property to the path to the access log that is created by the probe. The default is `access.log`.
NHttpd.EnableFileServing	Set this property to `TRUE` to enable file serving by the probe. Set this property together with the **NHttpd.DocumentRoot** property, to enable the probe to act as a simple HTTP server, serving files from the local file system. The default is `FALSE`.
NHttpd.DocumentRoot	Set this property to the document root for HTTP requests. Set this property together with the **NHttpd.EnableFileServing** property, to enable the probe to act as a simple HTTP server, serving files from the local file system. The default is `./`.
NHttpd.AuthenticationDomain	Set this property to the authentication domain that is presented when authentication details are requested over the HTTP or HTTPS connection. The default is `omnibus`.

What to do next

Set up the authentication between the remote system and the HTTP interface and, if required, set up an SSL connection. After you have performed these tasks, restart the probe so that all the changes to the property values can take effect.

Related reference:

Chapter 6, "Common probe properties and command-line options," on page 113
A number of properties and command-line options are common to all probes and TSMs.

Configuring authentication between remote systems and probes

You can configure the remote administration functionality to authenticate the user credentials for every connection to the HTTP interface. Authentication restricts the connections to a valid combination of user name and password. By default, the user credentials are not authenticated, so any user credentials are permitted.

Before you begin

Ensure that the password associated with the user name is hashed with AES-128. Use the **nco_crypt** command-line utility, with the `-aes` option, to hash the password. For example:

```
nco_crypt -aes password
```

About this task

Authentication through the HTTP interface is controlled by the **Nhttpd.BasicAuth** probe property. The default value of this property is "", which accepts any user credential as valid.

Only a single combination of a user name and password is permitted. This authentication does not interface with other methods of authentication, such as secure mode authentication, or Pluggable Authentication Modules (PAM)

The combination of user name and password is sent to the HTTP interface in near-plaintext. To avoid plaintext, configure the HTTP interface for SSL connections.

Procedure

To enable and configure authentication:

In the probe properties file, set the **Nhttpd.BasicAuth** property to a string in the format "*username*:*password*", where *username* is any set of characters that does not include a colon (:) and *password* is the AES-128 hash of the password generated by the **nco_crypt** utility.
For example:

- The following value is a combination of the user name `auth` and a hash of the password `netcool`:

  ```
  "auth:tOufpMhIN7R3L1Np89XRvA=="
  ```

If you specify a blank string ("") for the property, all authentication attempts are permitted, regardless of the supplied credentials.

What to do next

To avoid sending plaintext passwords, configure an SSL connection between the remote system and the probe. To increase the security of the user-password combination, change the file system permissions of the probe properties file. After you have performed these tasks, restart the probe so that all the changes to the property values can take effect.

Related tasks:

"Configuring SSL connections between remote systems and probes" on page 94
To secure communications and HTTP traffic to and from remote probes, configure an SSL connection. This prevents communications from being sent in plaintext.

Related reference:

Chapter 6, "Common probe properties and command-line options," on page 113
A number of properties and command-line options are common to all probes and TSMs.

Configuring SSL connections between remote systems and probes

To secure communications and HTTP traffic to and from remote probes, configure an SSL connection. This prevents communications from being sent in plaintext.

Before you begin

For the SSL connection, ensure that authentication is enabled. Also ensure that the probe is configured to listen on an HTTPS port. For example, create a key database on the host on which the probe is running and then create a self-signed (CA) certificate. Then, create a certificate request from probe host and sign with the CA certificate label. Receive the signed certificate into the key database on probe host, so that the key database contains a certificate for the probe host. Then, extract the signer certificate. Add the signer certificate to the key database on the host from which you want to connect to the HTTP interface, and issue the requests.

Procedure

To enable and configure SSL connections, set the following properties in the probe rules file:

Property	Instructions
NHttpd.SSLEnable	Set this property to TRUE.
NHttpd.SSLListeningPort	Set this property to the port number on which the probe needs to listen for SSL connections.
NHttpd.SSLCertificate	Set this property to the label of the certificate contained in the $NCHOME/etc/security/keys/omni.kdb key database file. This value is used as the certificate for the HTTPS server.
NHttpd.SSLCertificatePwd	Set this property to the password for the $NCHOME/etc/security/keys/omni.kdb key database file.

What to do next

Restart the probe so that all the changes to the property values can take effect.

Related tasks:

"Configuring authentication between remote systems and probes" on page 92
You can configure the remote administration functionality to authenticate the user credentials for every connection to the HTTP interface. Authentication restricts the connections to a valid combination of user name and password. By default, the user credentials are not authenticated, so any user credentials are permitted.

"Sending remote requests to probes (nco_http)" on page 95
Use the **nco_http** utility to connect to the HTTP or HTTPS interface of a probe and issue requests.

Related reference:

Chapter 6, "Common probe properties and command-line options," on page 113
A number of properties and command-line options are common to all probes and TSMs.

Sending remote requests to probes (nco_http)

Use the **nco_http** utility to connect to the HTTP or HTTPS interface of a probe and issue requests.

Before you begin

Ensure that authentication and, if required, an SSL connection is configured.

About this task

To start the **nco_http** utility, use the following command:

UNIX `$OMNIHOME/bin/nco_http command_line_options`

Windows `%OMNIHOME%\bin\nco_http.exe command_line_options`

You can use the `nco_http.props` properties file to store several of the command line option values. You can store frequently used options, such as the user name and password, instead of entering them each time you run the utility.

Properties and command-line options for the **nco_http** utility are described in the following table.

Table 26. nco_http properties and command-line options

Property	Command-line option	Description
ConfigCryptoAlg *string*	N/A	Specifies the cryptographic algorithm to use for decrypting string values (including passwords) that were encrypted with the **nco_aes_crypt** utility and then stored in the properties file. Set the *string* value as follows: • When in FIPS 140–2 mode, use `AES_FIPS`. • When in non-FIPS 140–2 mode, you can use either `AES_FIPS` or `AES`. Use `AES` only if you need to maintain compatibility with passwords that were encrypted by using the tools provided in versions earlier than Tivoli Netcool/OMNIbus V7.2.1. The value that you specify must be identical to that used when you ran **nco_aes_crypt** with the `-c` setting, to encrypt the string values. Use this property in conjunction with the **ConfigKeyFile** property. The default is `AES`.

Table 26. nco_http properties and command-line options (continued)

Property	Command-line option	Description
ConfigKeyFile *string*	N/A	Specifies the path and name of the key file that contains the key used to decrypt encrypted string values (including passwords) in the properties file. The key is used at run time to decrypt string values that were encrypted with the **nco_aes_crypt** utility. The key file that you specify must be identical to the file used to encrypt the string values when you ran **nco_aes_crypt** with the `-k` setting. Use this property in conjunction with the **ConfigCryptoAlg** property. The default is "".
Data *string*	`-data` *string*	Specifies the HTTP data for the request. The default is "".
DataType *string*	`-datatype` *string*	Specifies the HTTP data mime type for the request. The default is "".
Header *string*	`-header` *string*	Specifies extra HTTP header data for the request. The default is "".
N/A	`-help`	Displays help about the command-line options.
N/A	`-messagelevel`	Specifies the logging level for messages. The available logging levels are `FATAL`, `ERROR`, `WARN`, `INFO`, and `DEBUG`. The default level is `INFO`.
MessageLog *string*	`-messagelog` *string*	Specifies the name and location of the log file. The default level is `stderr` (standard error).
Method *string*	`-method` *string*	Specifies the HTTP method for the request. The available methods are `GET`, `POST`, and `PATCH`. The default method is `GET`.
NHttpd.SSL CertificatePwd *string*	N/A	Specifies the password required to access the SSL certificate file. The default is "".
NHttpd.SSLEnable *boolean*	N/A	Enables the use of SSL support. The default is `FALSE`.
Password *string*	`-password` *string*	Specifies the HTTP password for the request. The default is "".

Table 26. nco_http properties and command-line options (continued)

Property	Command-line option	Description
N/A	`-propsfile`	Specifies the name and location of the utility's properties file. The default is: `$OMNIHOME/etc/nco_http.props`
N/A	`-timeout`	Specifies the timeout period for the HTTP request. The default is 60 seconds.
URI *string*	`-uri` *string*	Specifies the HTTP URI for the request. The default is "".
Username *string*	`-username` *string*	Specifies the HTTP user name for the request. The default is "".
N/A	`-version`	Displays version information for the utility. The default is "".

Example

The following example returns the current version information of a probe, the value of the probe reloadrulesflag field, and the values of the probe properties. The request returns encrypted properties in their encrypted form.

```
$OMNIHOME/bin/nco_http -uri http://servername.ibm.com:port_number/probe/common
```

The information returned looks like this:

```
2012-06-22T13:56:31: Information: I-UNK-104-002: {
 "version": {
  "ProbeVersion": "Netcool/OMNIbus Event Simulation Probe probe -
  Version 7.4.0 64-bit\n(C) Copyright IBM Corp. 1994, 2012\n",
  "APIVersion": "Netcool/OMNIbus Probe API Library Version 7.4.0 64-bit",
  "ReleaseID": "5.2012.0620",
  "APIReleaseID": "5.2012.0620",
  "SoftwareCompileInfo": "Wed Jun 20 20:17:13 BST 2012 on servername.ibm.com
  (Linux 2.6.18-274.17.1.el5 #1 SMP Wed Jan 4 22:45:44 EST 2012)"
 },
 "reloadrulesflag": 0,
 "properties": {
  "NHttpd.SSLCertificate": "",
  "RawCaptureFileAppend": 0,
  "StoreAndForward": 1,
  "KeepLastBrokenSAF": 0,
  "MessageLevel": "warn",
  "ConfigCryptoAlg": "AES",
  "StoreSAFRejects": 0,
  "OplStatsNumberEventsGenerated": 0,
  "Buffering": 0,
  "RawCapture": 0,
  "OplDetailsTableName": "alerts.details",
  "SSLServerCommonName": "",
  "LookupTableMode": 3,
  "NHttpd.EnableHTTP": true,
  "NHttpd.AccessLog": "access.log",
  "OplStatsNumberEventsDiscarded": 0,
```

```
"PollServer": 0,
"ProbeWatchHeartbeatInterval": 0,
"RetryConnectionTimeOut": 30,
"OplStatsRulesFileTimeSec": 0,
"MessageLog": "/export/views/dev/omnibus74/omnibus/log/simnet.log",
"NHttpd.AuthenticationDomain": "omnibus",
"NHttpd.SSLListeningPort": 0,
"SAFFileName": "/export/views/dev/omnibus74/omnibus/var/simnet.store",
"LogFile": "/export/views/dev/omnibus74/omnibus/probes/linux2x86/simnet.def",
"MaxLogFileSize": 1048576,
"MsgDailyLog": 0,
"BeatInterval": 2,
"Server": "NCOMS",
"CacheRulesFile": "/export/views/dev/omnibus74/omnibus/var/simnet.rulescache",
"Props.CheckNames": true,
"BeatThreshold": 1,
"CacheRules": 0,
"OplStatsMemoryInUse": 359952,
"BufferSize": 10,
"Help": 0,
"LogFilePoolSize": 10,
"PidFile": "/export/views/dev/omnibus74/omnibus/var/simnet",
"Manager": "Omnibus",
"NHttpd.ExpireTimeout": 15,
"NHttpd.SSLCertificatePwd": "",
"Peerport": 9999,
"RulesFile": "/export/views/dev/omnibus74/omnibus/probes/linux2x86/simnet.rules",
"OldTimeStamp": "FALSE",
"SingleThreadedComms": 0,
"OplStatusTableName": "alerts.status",
"ServerBackup": "",
"OplStatsProbeStartTime": 1340369540,
"TimeBetweenEvents": 1000,
"LogFileUsePool": 0,
"MaxRawFileSize": -1,
"OplPaName": "",
"OplStatsCPUTimeSec": 0,
"NetworkTimeout": 0,
"PeerHost": "localhost",
"NHttpd.NumWorkThreads": 5,
"NHttpd.DocumentRoot": "./",
"LogFileUseStdErr": 0,
"RegexpLibrary": "TRE",
"AuthPassword": "",
"NHttpd.EnableFileServing": false,
"OplStatsNumberEvents": 251,
"AuthUserName": "",
"SecureLogin": 0,
"Version": 0,
"Mode": "standard",
"RetryConnectionCount": 15,
"MsgTimeLog": "0000",
"OplStatsCPUTimeUSec": 50000,
"NHttpd.SSLEnable": false,
"OplDumpProps": 0,
"PropsFile": "/export/views/dev/omnibus74/omnibus/probes/linux2x86/simnet.props",
"OplStatsRulesFileTimeUSec": 7675,
"MaxSAFFileSize": 1048576,
"ConfigKeyFile": "",
"RawCaptureFile": "/export/views/dev/omnibus74/omnibus/var/simnet.cap",
"Name": "simnet",
"OplPaID": 0,
"RollSAFInterval": 90,
"NHttpd.ListeningPort": 4444,
"SAFPoolSize": 3,
"NHttpd.BasicAuth": "",
```

```
  "OplPacketSize": 512,
  "AutoSAF": 0
 }
}
```

The following example sets the **MessageLevel** property of the probe to debug:

```
$OMNIHOME/bin/nco_http -uri http://servername.ibm.com:4444/probe/common
-datatype application/json -data '{ "properties": { "MessageLevel": "debug" } }'
-method patch
```

Related tasks:

"Configuring SSL connections between remote systems and probes" on page 94
To secure communications and HTTP traffic to and from remote probes, configure an SSL connection. This prevents communications from being sent in plaintext.

Related reference:

"About the common URI" on page 103
Use the common URI provided in the standard probe C library (libOpl) to execute standard HTTP commands against all probes.

Reloading rules files (nco_probereloadrules)

You can use the **nco_probereloadrules** utility to remotely reload a probe rules file without restarting the probe.

Before you begin

Ensure that authentication and, if required, an SSL connection is configured. The probe that you want to communicate with must also have its HTTP or HTTPS interface enabled.

About this task

To start the **nco_probereloadrules** utility, use the following command:

UNIX `$OMNIHOME/bin/nco_probereloadrules command_line_options`

Windows `%OMNIHOME%\bin\nco_probereloadrules.cmd command_line_options`

The **nco_probereloadrules** utility uses the **nco_http** utility to reload the rules file over a HTTP or HTTPS connection. It can also make use of the properties file `nco_http.props`. You can store frequently used options, such as the user name and password, in `nco_http.props` instead of entering them each time that you run the utility.

Command-line options for the **nco_probereloadrules** utility are described in the following table.

Table 27. nco_probereloadrules command-line options

Command-line option	Description
`-help`	Displays help about the command-line options.
`-host`	Specifies the host name where the probe is installed.
`-messagelevel`	Specifies the logging level for messages. The available logging levels are `FATAL`, `ERROR`, `WARN`, `INFO`, and `DEBUG`. The default level is `INFO`.
`-password`	Specifies the HTTP password for the update.

Table 27. nco_probereloadrules command-line options (continued)

Command-line option	Description
`-port`	Specifies the port number on which to communicate with the probe.
`-ssl`	Specifies the use of a HTTPS connection.
`-timeout`	Specifies the timeout period for the HTTP response.
`-username`	Specifies the HTTP user name for the request.

Example

The following example reloads the rules file for a probe whose host is `servername` and where the port is `2020`.

```
$OMNIHOME/bin/nco_probereloadrules -host servername -port 2020
```

Related tasks:

"Rereading the rules file" on page 60
Because probes read the rules file only on startup, you must force the probe to reread the rules whenever you make changes to it. The probe processes the reread request only on receipt of a new event. If the probe is idle or is already processing an event, it will not reread the rules file until a new event is received.

Sending property updates to probes (nco_setprobeprop)

You can use the **nco_setprobeprop** utility to update the value of a probe property. The **nco_setprobeprop** utility uses the **nco_http** utility to make the property update over a HTTP or HTTPS connection. **nco_setprobeprop** is also used by the flood control triggers to put probes into flood control mode.

Before you begin

Ensure that authentication and, if required, an SSL connection is configured. The probe that you want to communicate with must also have its HTTP or HTTPS interface enabled.

About this task

To start the **nco_setprobeprop** utility, use the following command:

UNIX
```
$OMNIHOME/bin/nco_setprobeprop command_line_options
```

Windows
```
%OMNIHOME%\bin\nco_setprobeprop.cmd command_line_options
```

If you specify a new value for an existing probe property, the utility will update the property value. If you do not specify a value for a property, the new value of the property will be an empty string.

You can also specify a property name that does not exist and the utility will create the property and set it to the value that you specify for it. In this case, you must have altered the probe's rules file to handle the property (for example, use *%propertyname* to get the value of the property). Properties created by the utility are transient and are lost when the probe shuts down.

Command-line options for the **nco_setprobeprop** utility are described in the following table. Because **nco_setprobeprop** uses the **nco_http** utility to communicate with the probe, it can make use of the properties file nco_http.props. You can store frequently used options, such as the user name and password, in nco_http.props instead of entering them each time that you run the utility.

Table 28. nco_setprobeprop command-line options

Command-line option	Description
-help	Displays help about the command-line options.
-host	Specifies the host name where the probe is installed.
-messagelevel	Specifies the logging level for messages. The available logging levels are FATAL, ERROR, WARN, INFO, and DEBUG. The default level is INFO.
-name	Specifies the name of the property to update or create.
-password	Specifies the HTTP password for the update.
-port	Specifies the port number on which to communicate with the probe.
-ssl	Specifies the use of a HTTPS connection.
-timeout	Specifies the timeout period for the HTTP response.
-username	Specifies the HTTP user name for the request.
-value	Specifies the new value of the property. If you do not specify a value, the new value of the property will be an empty string.

Example

The following example sets the value of the **FloodControl** property to flood. The probe host is servername and the port is 2000.

```
$OMNIHOME/bin/nco_setprobeprop -name FloodControl -value flood -host servername
 -port 2000
```

Generating events with probes (nco_probeeventfactory)

You can use the **nco_probeeventfactory** utility to remotely generate an event with a probe. The utility causes the probe to process the rules file using the name-value pairs that you provide instead of the values supplied by the usual event source. This is useful for testing changes to the rules file. You could also amend the rules file so that the name-value pairs that you supply cause internal variables to change, without the need for a "real" event from an event source.

Before you begin

Ensure that authentication and, if required, an SSL connection is configured. The probe that you want to communicate with must also have its HTTP or HTTPS interface enabled.

About this task

To start the **nco_probeeventfactory** utility, use the following command:

UNIX `$OMNIHOME/bin/nco_probeeventfactory` *command_line_options* [*name=value*]...

Windows `%OMNIHOME%\bin\nco_probeeventfactory.cmd` *command_line_options* [*name=value*]...

You can specify event elements as name-value pairs. You must enclose any element names or values that contain spaces within double quotation marks ("), for example:

```
Summary="This is the summary."
```

You can specify as many name-value pairs as your command-line character limit allows.

The **nco_probeeventfactory** utility uses the **nco_http** utility to generate the event over a HTTP or HTTPS connection. It can also make use of the properties file `nco_http.props`. You can store frequently used options, such as the user name and password, in `nco_http.props` instead of entering them each time that you run the utility.

Command-line options for the **nco_probeeventfactory** utility are described in the following table.

Table 29. nco_probeeventfactory command-line options

Command-line option	Description
`-help`	Displays help about the command-line options.
`-host`	Specifies the host name where the probe is installed.
`-messagelevel`	Specifies the logging level for messages. The available logging levels are `FATAL`, `ERROR`, `WARN`, `INFO`, and `DEBUG`. The default level is `INFO`.
`-password`	Specifies the HTTP password for the update.
`-port`	Specifies the port number on which to communicate with the probe.
`-ssl`	Specifies the use of a HTTPS connection.
`-timeout`	Specifies the timeout period for the HTTP response.
`-username`	Specifies the HTTP user name for the request.

Example

The following example generates an event with a probe whose host is `servername` and where the port is `2020`. The name-value pairs specify that the alert group is set to `Netcool`, the host name of the device is set to `testserver.ibm.com`, and the severity of the alert is set to `1`.

```
$OMNIHOME/bin/nco_probeeventfactory -host servername -port 2020 AlertGroup=Netcool
Host=testserver.ibm.com Severity=1
```

About the common URI

Use the common URI provided in the standard probe C library (libOpl) to execute standard HTTP commands against all probes.

You can use the URI to perform the following actions against probes:

- Report the version and ID of the probe, and information about the current state of the probe, including its properties
- Signal the probe to reload its rules file
- List the properties of the probe
- Get or set a property
- Create synthetic events that are processed by the rules file.

Probes can register their own URIs to expose entities specific to a particular probe. These URIs can be used to trigger actions against the device or system that the probe is connected to or communicating with.

The following table describes the features of this URI.

Feature	Description
Path	`/probe/common`
Methods	GET, POST, PATCH
Accepted MIME types	`application/json`, where applicable
Content-type	`application/json`, where applicable
Caching	A GET request sets the Cache-Control HTTP header field to `no-cache`, to indicate the volatile nature of the data returned by a GET request of this URI.
HTTP return codes	A list of HTTP return codes and their meaning (in relation to the action) is provided. This list is not exhaustive and any HTTP return code might be returned. Build your applications or scripts to expect any possible return codes.

For more information about JavaScript Object Notification (JSON), see http://www.json.org/.

Get the current state of a probe

This request accesses the current version information of a probe, the values of the probe properties and the value of the probe reloadrulesflag field. The request returns encrypted properties in their encrypted form.

When designing an application or script to change values, ensure that it obtains a recent set of values via a GET request before the PATCH request is sent.

The following table describes the features of this request.

Feature	Description
URI	`/probe/common`
Method	GET

Feature	Description
Content-type	`application/json`
Query parameters	search For filtering the properties returned by a GET request.
Caching	The response to the request is marked `no-cache` due to the volatility of the data contained in the response. HTTP ETags are not supported.

Sample response

The following example shows an sample response to this request. The list of returned properties is abridged.

```
{
 "version":{
  "ProbeVersion": "Netcool/OMNIbus Event Simulation Probe probe -
  Version 7.4\n(C) Copyright IBM Corp. 1994, 2007\n",
  "APIVersion": "Netcool/OMNIbus Probe API Library Version 7.4",
  "ReleaseID": "3.0.4082",
  "APIReleaseID": "5.2012.0410",
  "SoftwareCompileInfo":"Tue Jan 29 13:40:19 GMT 2008 on sol8-build2.hursley.ibm.com
  (SunOS 5.8 Generic_117350-33)"
 },
 "reloadrulesflag": 0,
 "properties":{
  "NHttpd.SSLCertificate": "",
  "RawCaptureFileAppend": 0,
  "StoreAndForward": 1,

...

  "OplPacketSize": 512,
  "AutoSAF": 0
 }
}
```

HTTP response codes

The following table shows the possible response codes that might be returned by this request. This table is not exhaustive. Clients must be prepared for all HTTP status codes.

Response code	Explanation
`200`	The request was successful.

Reload the rules file

This request accesses the reloadrulesflag attribute of the probe, which is contained in the standard probe C library (libOpl). By setting the value of the attribute, the request triggers the probe to reload its rules file before the next event is processed. A SIGHUP signal on UNIX triggers the same action.

The reloadrulesflag attribute can be set to 1, which triggers the probe to reload its rules file.

The request is actioned immediately before the next event is processed. This means that a delay can occur between the request being made and being actioned, for example, if the probe is inactive.

The following table describes the features of this request.

Feature	Description
URI	`probe/common`
Method	PATCH
Accepted MIME types	`application/json`
Query parameters	None

Sample request

The following example shows a sample request to a probe to reload its rules file.

```
{ "reloadrulesflag" : 1 }
```

HTTP response codes

The following table shows the possible response codes that might be returned by this request. This table is not exhaustive. Clients must be prepared for all HTTP status codes.

Response code	Explanation
`200 OK`	The request was successful.
`400 Bad Request`	For example, if a PATCH request was made with invalid payload data.
`415 Unsupported Media Type`	Data was provided as input in a format other than `application/json`.

Related tasks:

"Rereading the rules file" on page 60
Because probes read the rules file only on startup, you must force the probe to reread the rules whenever you make changes to it. The probe processes the reread request only on receipt of a new event. If the probe is idle or is already processing an event, it will not reread the rules file until a new event is received.

Related reference:

"Set PATCH or POST requests as blocking or nonblocking" on page 112
Requests that change values or trigger actions by using PATCH or POST can be set to be blocking or nonblocking.

List the probe properties

This request accesses the properties of the probe. You can specify a search query variable to restrict the properties that are returned.

Restriction: The request does not return properties that are encrypted in the properties file.

The following table describes the features of this request.

Feature	Description
URI	`/probe/common`
Method	GET
Content-type	`application/json`
Search parameter	`search` The search is a case-insensitive substring search. An empty search string is equivalent to no search string and returns all properties. The search query variable does not prevent the version information and the value of the reloadrulesflag attribute from being returned.

Sample

The following example shows a sample response to this request, when the search query variable "buff" is specified, that is: `GET /probe/common?search=buff`. In this example, the version information that would typically be returned is omitted.

```
{
 "version": {
  ...
  },
  "reloadrulesflag": 0,
  "properties": {
   "Buffering": 0,
   "BufferSize": 10
  }
}
```

HTTP response codes

The following table shows the possible response codes that might be returned by this request. This table is not exhaustive. Clients must be prepared for all HTTP status codes.

Response code	Explanation
`200 OK`	The request was successful

Create a synthetic event

This request creates a synthetic event that can trigger the execution of the probe rules file, or a fragment of the rules file.

The following table describes the features of this request.

Feature	Description
URI	`/probe/common`
Method	POST
Accepted MIME types	`application/json`

Examples

The following sample request creates an event to go through the rules file, with the following tokens set (in rules file syntax):

- $token1="value1"
- $token2="value2"
- $token3="value3"

```
{ "eventfactory" :
 [
  { "token1" : "value1", "token2" : "value2", "token3" : "value3" }
 ]
}
```

The following sample request creates two events to go through the rules file, with the following tokens set (in rules file syntax):

- For the first event:
  - $token1="value1"
  - $token2="value2"
  - $token3="value3"
- For the second event:
  - $al="v1"
  - $a2="v2"

```
{ "eventfactory" :
 [
  { "token1" : "value1", "token2" : "value2", "token3" : "value3" },
  { "a1" : "v1", "a2" : "v2" }
 ]
}
```

HTTP response codes

The following table shows the possible response codes that might be returned by this request. This table is not exhaustive. Clients must be prepared for all HTTP status codes.

Response code	Explanation
`200 OK`	The request was successful.
`400 Bad Request`	For example, if a PATCH request was made with invalid payload data.

Response code	Explanation
415 Unsupported Media Type	Data was provided as input in a format other than application/json.
500 Internal Server Error	An internal error prevented the action from completing. To troubleshoot the request, perform a GET request to check whether any of the required properties were changed. If they were not, resubmit the request.

Related reference:

"Set PATCH or POST requests as blocking or nonblocking" on page 112
Requests that change values or trigger actions by using PATCH or POST can be set to be blocking or nonblocking.

Set a probe property

This request changes the values of probe properties. It can also create new properties. All new properties created in this way require a string value.

Important: Although this request can change or set the values of all probe properties, the change might not change the behavior of the probe. Most probe properties are read only when the probe starts, and are not subsequently reread. You might need to restart the probe so that the change to the property value changes the behavior of the probe. If you are working with a failover pair, ensure that changes you make to the master probe are also propagated to the slave.

The following table describes the features of this request.

Feature	Description
URI	probe/common
Method	PATCH
Accepted MIME types	application/json
Query parameters	None

Samples

The following example shows a sample request, which causes the **RawCapture** property to be set to 1.

```
{
 "properties": {
  "RawCapture": 1
 }
}
```

The following example shows an invalid request, because the **RawCapture** property requires an integer, not a string. The request will cause a 400 Bad Request response.

```
{
 "properties": {
  "RawCapture": "yes"
 }
}
```

The following example shows a sample request that creates two new properties: **MyProperty**, which has the value busy, and **AnotherProp**, which has the value no.

```
{
 "properties": {
  "MyProperty": "busy",
  "AnotherProp": "no"
 }
}
```

HTTP response codes

The following table shows the possible response codes that might be returned by this request. This table is not exhaustive.

Response code	Explanation
200 OK	The request was successful and the property values were changed.
400 Bad Request	This response code might be for one of the following reasons: • Invalid JSON data was sent. • An invalid data type was provided for an existing property. • An attempt was made to overwrite the value of an encrypted property. • An attempt was made to create new property that is not a string.
500 Internal Server Error	An internal error prevented the action from completing. To troubleshoot the request, perform a GET request to check whether any of the required properties were changed. If they were not, resubmit the request.

Related reference:

Chapter 6, “Common probe properties and command-line options,” on page 113
A number of properties and command-line options are common to all probes and TSMs.

“Set PATCH or POST requests as blocking or nonblocking” on page 112
Requests that change values or trigger actions by using PATCH or POST can be set to be blocking or nonblocking.

Acknowledge event and event_payload

You can use the "options" section of the JSON payload of the /probe/common URI to request the generation of an acknowledgement event in response to an HTTP request. The acknowledgement event contains details of the HTTP request and its outcome and is sent through the probe rules file to the ObjectServer. You can add payload data to the acknowledgement event; this data can be information that is specific to the event or the ObjectServer.

You can use this data to correlate the acknowledgement event with specific events or ObjectServers. Additionally, if you add payload data to the acknowledgement event, you can then write triggers that catch the results of the HTTP requests and update the corresponding events. These triggers can be fired from right-click tools or from automations.

For an acknowledgement event, set the "event_ack" object of the "options" section to true. This setting instructs the standard probe C library (lipOpl) to send an acknowledgement event to the default ObjectServer, via the rules file. For more information, see "Example 1: Acknowledgement event."

For an acknowledgement event with a payload, et the "event_ack" object of the "options" section to true. Additionally, in the "event_ack_payload" object, add a set of name-value pairs of payload data, which is added to the acknowledgement event. The "event_ack" object must be a flat list and cannot contain any child JSON objects, arrays, or NULL items. All items in the list of name-value pairs must be strings, because these items are mapped to probe rules file elements before the acknowledgment event is processed through the rules file. For more information, see "Example 2: Acknowledgement event with payload" on page 111.

Example 1: Acknowledgement event

The following example shows the "options" section of the JSON payload set to request the generation of an acknowledgement with a PATCH request.

```
{
 "options": { "block_response" : false,
   "event_ack" : true
   },
 "properties": {
  "MyNewProperty": "myvalue"
 }
}
```

The {"event_ack" : true } object instructs libOpl to send an acknowledgement event to the default ObjectServer. For the request in this example, acknowledgement event that enters the probe rules file is equivalent to the following elements:

```
@Name=nameproperty
@Agent=agentproperty
@Node=hostname
@OwnerGID=1
@Severity=1
@Manager="ProbeWatch"
@Summary="HTTP event_ack ..."
@Identifier=@Name+"@"+@Node+": "+@Summary

$req_method="PATCH"
$request_outcome="OK"
$req_payload_type="application/json"
$req_port="6789"
$req_uri="/probe/common"
$req_http_type="https"
$req_hostname="probehost.example.com"
$req_auth_type="none"
$http_return_reason="Accepted"
$req_username="test"
$req_ipaddr="192.168.1.1"
$http_return_code="202"
$req_payload_length="178"
```

In this example, *nameproperty* is the value of the probe **Name** property, *agentproperty* is the value of the probe **Agent** property, and *hostname* is the name of the host computer on which the probe is installed.

Example 2: Acknowledgement event with payload

The following example shows the "event_ack_payload" object of the "options" section of the JSON payload, set with an arbitrary set of name-value pairs of payload data for a PATCH request.

```
{
 "options": {
   "block_response" : false,
   "event_ack" : true,
   "event_ack_payload" : {
      "name1" : "value1",
      "name2" : "value2"
   }
 },
 "properties":{
  "MyNewProperty": "myvalue"
 }
}
```

For the request in this example, acknowledgement event that enters the probe rules file is equivalent to the following elements:

```
@Name=nameproperty
@Agent=agentproperty
@Node=hostname
@OwnerGID=1
@Severity=1
@Manager="ProbeWatch"
@Summary="HTTP event_ack ..."
@Identifier=@Name+"@"+@Node+": "+@Summary

$req_method="PATCH"$request_outcome="OK"
$req_payload_type="application/json"
$req_port="6789"
$req_uri="/probe/common"
$req_http_type="https"
$req_hostname="probehost.example.com"
$name1="value1"
$name2="value2"
$req_auth_type="none"
$http_return_reason="Accepted"
$req_username="test"
$req_ipaddr="192.168.1.1"
$http_return_code="202"
$req_payload_length="178"
```

In this example, *nameproperty* is the value of the probe **Name** property, *agentproperty* is the value of the probe **Agent** property, and *hostname* is the name of the host computer on which the probe is installed.

The acknowledgement event is identical to the event requested in "Example 1: Acknowledgement event" on page 110, except for the payload data, which consists of the "name1" and "name2" elements.

Set PATCH or POST requests as blocking or nonblocking

Requests that change values or trigger actions by using PATCH or POST can be set to be blocking or nonblocking.

In a blocking request, which is the default, the action described by the payload is validated. The, the request is completed with the resultant HTTP response code to indicate success or failure. Success is indicated by a `200 – OK` return code, and failure is indicated by the most appropriate HTTP return codes.

To make a request nonblocking, use the `"options"` section of the JSON payload for an action. In the following sample, which uses the `/probe/common PATCH` example data to create a new property, an `"options"` section is added, which specifies the extra JSON data.

```
{
 "options": {"block_response" : false },
 "properties": {
   "MyNewProperty" : "myvalue"
 }
}
```

After this request is syntactically and semantically validated, the HTTP server responds with a `202 – Accepted` response code. At this point, the action is requested, but not executed.

If an invalid request is made, which includes the nonblocking flag, a failure HTTP response code is returned. The following example would return a `400 – BAD REQUEST` response code, because the new property value is not a string. Each new property that is created must have a string value.

```
{
 "options" :  {
  "block_response" : false },
  "properties": {
   "MyNewProperty": 1
  }
}
```

Nonblocking requests that are accepted (by using `{ "block_response" : false }`) can still fail due to internal unforeseen problems. Because the HTTP connection is effectively closed, the HTTP interface cannot inform the client of any subsequent problems. For this reason, an optional “acknowledgement event” can be generated by the HTTP interface for such requests.

Related reference:

“Reload the rules file” on page 105
This request accesses the reloadrulesflag attribute of the probe, which is contained in the standard probe C library (libOpl). By setting the value of the attribute, the request triggers the probe to reload its rules file before the next event is processed. A SIGHUP signal on UNIX triggers the same action.

“Create a synthetic event” on page 107
This request creates a synthetic event that can trigger the execution of the probe rules file, or a fragment of the rules file.

“Set a probe property” on page 108
This request changes the values of probe properties. It can also create new properties. All new properties created in this way require a string value.

Chapter 6. Common probe properties and command-line options

A number of properties and command-line options are common to all probes and TSMs.

For the properties and command-line options that are specific to a particular probe or TSM, see the individual reference guides for each probe and TSM.

Tip: You can encrypt string values in a properties file by using property value encryption.

On startup, a probe reads the property values specified in its properties file. Some properties are only read at startup and changes made to them while the probe is running have no effect. The values of other properties can be changed while the probe is running. Of these properties, some value changes take effect immediately and some do not take effect until the rules file is reread. When you force a reread of the rules file, the connection to the ObjectServer is dropped and reestablished, causing the probe to recheck the property value.

You can use the following methods to change the value of a property:

- Change the property value in the properties file. You must stop a probe and restart it for such changes to take effect.
- When starting a probe, use the available command-line options to assign temporary values to one or more properties.
- Use the probe HTTP interface (if available) to update a property value on a running probe. Depending on the property, such changes take effect either immediately or after the rules file is reloaded.
- Edit the rules file and use the *%Property* setting to change a property value on a running probe. You must reload the rules file for such changes to take effect.

The following table lists the common properties and command-line options that are available to all probes, and provides their default settings. String values in non-ASCII characters are not supported. The **Update type** column indicates when changes to property values take effect: on startup only, immediately, or on reconnection of the probe to the ObjectServer.

Table 30. Common probe properties and command-line options

Property	Command-line option	Description	Update type
AuthPassword *string*	N/A	Specifies the password associated with the user name that is used to authenticate the probe when it connects to a proxy server or an ObjectServer running in secure mode. The default is ''. When in FIPS 140–2 mode, the password can either be specified in plain text, or can be encrypted with the **nco_aes_crypt** utility. If you are encrypting passwords by using **nco_aes_crypt** in FIPS 140–2 mode, you must specify AES_FIPS as the encryption algorithm. When in non-FIPS 140–2 mode, the password can be encrypted with the **nco_g_crypt** or **nco_aes_crypt** utilities. If you are encrypting passwords by using **nco_aes_crypt** in non-FIPS 140–2 mode, you can specify either AES_FIPS or AES as the encryption algorithm. Use AES only if you need to maintain compatibility with passwords that were encrypted using the tools provided in versions earlier than Tivoli Netcool/OMNIbus V7.2.1.	On reconnection
AuthUserName *string*	N/A	Specifies a user name used to authenticate the probe when it connects to a proxy server or an ObjectServer running in secure mode. The default is ''.	On reconnection
AutoSAF 0 | 1	-autosaf -noautosaf	Specifies whether automatic store-and-forward mode is enabled. In this mode, if the probe starts but is unable to send events to the ObjectServer, the probe goes into store mode instead of terminating. By default, automatic store-and-forward mode is not enabled (0). **Note:** For automatic store-and-forward to work, the probe must previously have been connected at least once to the ObjectServer so that it knows the format in which to store events for that ObjectServer. If the probe is trying to connect to a virtual pair of ObjectServers and both of the ObjectServers are down, the probe checks the **AutoSAF** property setting. If automatic store-and-forward is enabled, the probe begins to store events in the store-and-forward file; otherwise, the probe terminates.	On startup only
BeatInterval *integer*	-beatinterval *integer*	Specifies the heartbeat interval for peer-to-peer failover. The default is 2 seconds.	On startup only
BeatThreshold *integer*	-beatthreshold *integer*	Specifies the extra period that the slave probe in a peer-to-peer failover relationship waits for before switching to active mode. The default is 1 second.	On startup only

Table 30. Common probe properties and command-line options (continued)

Property	Command-line option	Description	Update type
Fix Pack 1 **BufferFlush Interval** *integer*	`-bufferflushinterval` *integer*	Specifies the interval (in seconds) that a probe waits before flushing alerts to the ObjectServer. This property limits the time that alerts wait in the buffer while the buffer is still within the maximum size specified by the **BufferSize** property. To use this property, you must enable the **Buffering** property and set the **BufferSize** property to a value greater than `0`. **Note:** Some probes have a **FlushBufferInterval** property that provides an equivalent function to the **BufferFlushInterval** property. If both properties are set for an individual probe, a warning is written to the log and the **BufferFlushInterval** property is not enabled. The default is `0` seconds.	On startup only

Buffering `0|1`	`-buffer`
	
	`-nobuffer`
		Specifies whether buffering is used when sending alerts to the ObjectServer. By default, buffering is not enabled (`0`).
		Note: All alerts sent to the same table are sent in the order in which they were processed by the probe. If alerts are sent to multiple tables, the order is preserved for each table, but not across tables.
		
		If multithreaded processing is in operation (the default), a separate communication thread is used to send data to each registered target ObjectServer, and a separate text buffer is therefore maintained for each ObjectServer.
			On startup only

BufferSize *integer*	`-buffersize` *integer*	Specifies the number of alerts that the probe caches when the **Buffering** property is set to 1. The default is `10`.	On startup only

CacheRules `0|1`	`-cacherules`
	
	`-dontcacherules`
		Use this property to make the probe cache its rules file each time the probe is restarted or made to reread its rules file. When you start the probe, if it cannot read the rules file, it reads the cache file instead.
		
		By default, rules file caching is disabled, that is, the property is set to `0`.
			On reconnection

CacheRulesFile *string*	`-cacherulesfile` *string*	Use this property to specify the file to which you want the probe to cache its rules file. The default is `$OMNIHOME/var/`*probename*`.rulescache`, where *name* is the abbreviated name that is used to identify the probe, for example `simnet` for the simnet probe. Set this property only if you set the **CacheRules** property to 1 and want to change the location of the cache file from the default.	On reconnection

Table 30. Common probe properties and command-line options (continued)

Property	Command-line option	Description	Update type
ConfigCryptoAlg *string*	N/A	Specifies the cryptographic algorithm to use for decrypting string values (including passwords) that were encrypted with the **nco_aes_crypt** utility and then stored in the properties file. Set the *string* value as follows: • When in FIPS 140–2 mode, use AES_FIPS. • When in non-FIPS 140–2 mode, you can use either AES_FIPS or AES. Use AES only if you need to maintain compatibility with passwords that were encrypted by using the tools provided in versions earlier than Tivoli Netcool/OMNIbus V7.2.1. The value that you specify must be identical to that used when you ran **nco_aes_crypt** with the -c setting, to encrypt the string values. Use this property in conjunction with the **ConfigKeyFile** property.	On startup only
ConfigKeyFile *string*	N/A	Specifies the path and name of the key file that contains the key used to decrypt encrypted string values (including passwords) in the properties file. The key is used at run time to decrypt string values that were encrypted with the **nco_aes_crypt** utility. The key file that you specify must be identical to the file used to encrypt the string values when you ran **nco_aes_crypt** with the -k setting. Use this property in conjunction with the **ConfigCryptoAlg** property.	On startup only
Dumpprops *string*	-dumpprops *string*	Displays a probe's current properties settings. Run the probe with the -dumpprops option, for example: `$OMNIHOME/probes/nco_p_tivoli_eif -dumpprops`	On startup only
N/A	-help	Displays the supported command-line options and exits.	On startup only
KeepLastBrokenSAF 0 | 1	-keeplastbrokensaf -dontkeeplastbrokensaf	Specifies whether to automatically save corrupted store-and-forward records for future diagnosis. If set to 1, corrupted store-and-forward records are automatically saved. The default is 0. Use this property in conjunction with the **StoreSAFRejects** property.	Immediate
LogFilePoolSize *integer*	-logfilepoolsize *integer*	Specifies the number of log files to use if the logging system is writing to a pool of files. This property works only when the **LogFileUsePool** property is set to TRUE. The pool size can range from 2 to 99. The default is 10. **Note:** This option is supported only on Windows operating systems.	On startup only

Table 30. Common probe properties and command-line options (continued)

Property	Command-line option	Description	Update type
LogFileUsePool 0 | 1	-logfileusepool -nologfileusepool	Specifies whether to use a pool of log files for logging messages. If set to 1, the logging system opens the number of files specified for the pool at startup, and keeps them open for the duration of its run. (You define the number of files in the pool by using the **LogFilePoolSize** property.) When a file in the pool reaches its maximum size (as specified by the **MaxLogFileSize** property), the logging system writes to the next file. When all the files in the pool are at maximum size, the logging system truncates the first file in the pool and starts writing to it again. Files in the pool are named using the format *probename*.log_*ID*, where *ID* is a two-digit number starting from 01, to the maximum number specified for the **LogFilePoolSize** property. When the logging system starts to use a file pool, the system writes to the lowest-available file number, regardless of which file it was writing to when it last ran. The default is 0. When set to 0, the default *probename*.log file is renamed *probename*.log_OLD and a new log file is started when the maximum size is reached. If the file cannot be renamed, for example, because of a read lock on the _OLD file, and **LogFileUseStdErr** is set to 0, the logging system automatically starts using a pool of log files. If the file cannot be renamed, and **LogFileUseStdErr** is set to 1, messages are logged to the console if the probe was run from the command line. If the file cannot be renamed, and **LogFileUseStdErr** is set to 1, messages are logged to a file named %NCHOME%\omnibus\log*probename*.err if the probe is running as a Windows service. **Note:** This option is supported only on Windows operating systems.	On startup only
LogFileUse StdErr 0 | 1	-logfileusestderr -nologfileusestderr	Specifies whether to use standard error (stderr) as an output stream for logging messages. The default is 0, which causes the logging system to write to the default log file or to a pool of log files, as set by the **LogFileUsePool** property. If set to 1, messages are logged to the console when the probe is run from the command line. **Note:** The **LogFileUsePool** property setting takes precedence over the **LogFileUseStdErr** setting. This option is supported only on Windows operating systems.	On startup only

Table 30. Common probe properties and command-line options (continued)

Property	Command-line option	Description	Update type
LookupTableMode *integer*	-lookupmode *integer*	Specifies how table lookups are performed. It can be set to 1, 2, or 3. The default is 3. If set to 1, all external lookup tables are assumed to have a single value column. Tabs are not used as column delimiters. If set to 2, all external lookup tables are assumed to have multiple columns. If the number of columns on each line is not the same, an error is generated that includes the file name and the line on which the error occurred. If set to 3, the rules engine attempts to determine the number of columns in the external lookup table. An error is generated for each line that has a different column count from the previous line. The error includes the file name and the line on which the error occurred.	On reconnection
Manager *string*	-manager *string*	Specifies the value of the Manager field for the alert. The default value is determined by the probe.	Immediate
MaxLogFileSize *integer*	-maxlogfilesize *integer*	Specifies the maximum size that the log file can grow to, in Bytes. The default is 1 MB. When the log file reaches the size specified, a second log file is started. When the second file reaches the maximum size, the first file is overwritten with a new log file and the process starts again.	On startup only
MaxRawFileSize *integer*	N/A	Specifies the maximum size of the raw capture file, in KB. The default is unlimited (-1).	Immediate
MaxSAFFileSize *integer*	-maxsaffilesize *integer*	Specifies the maximum size (in Bytes) that the store-and-forward file can grow to when disconnected from the ObjectServer. The default is 1 MB.	Immediate
MessageLevel *string*	-messagelevel *string*	Specifies the message logging level. Possible values are: debug, info, warn, error, and fatal. The default level is warn. Messages that are logged at each level are as follows: fatal: fatal only. error: fatal and error. warn: fatal, error, and warn. info: fatal, error, warn, and info. debug: fatal, error, warn, info, and debug.	On startup only
MessageLog *string*	-messagelog *string*	Specifies where messages are logged. The default is $OMNIHOME/log/*probename*.log. **MessageLog** can also be set to stdout or stderr.	On startup only

Table 30. Common probe properties and command-line options (continued)

Property	Command-line option	Description	Update type
Mode *string*	-master -slave	Specifies the role of the instance of the probe in a peer-to-peer failover relationship. The value of the property can be set to: master: This instance is the master. slave: This instance is the slave. standard: There is no failover relationship. The default is standard.	On startup only
MsgDailyLog 0 | 1	-msgdailylog 0 | 1	Specifies whether daily logging is enabled. By default, the daily backup of log files is not enabled (0). **Note:** Because the time is checked regularly, when **MsgDailyLog** is set there is a slight reduction in performance.	On startup only
MsgTimeLog *string*	-msgtimelog *string*	Specifies the time after which the daily log is created. The default is 0000 (midnight). If **MsgDailyLog** set to 0, this value is ignored.	On startup only
Name *string*	-name *string*	Specifies the name of the probe. This value determines the names of the properties file, rules file, message log file, store-and-forward file, and raw capture file. **Note:** You can specify alternative file names by using the **PropsFile**, **RulesFile**, **MessageLog**, **SAFFileName**, and **RawCaptureFile** properties. If you want to set any of these file names in the properties file, they must be specified after the **Name** property. Otherwise, the **Name** property will override any previous setting of the files names. The value of the **Name** property is included in the primary key of the probe entry in the registry.probes ObjectServer table.	On reconnection
NetworkTimeout *integer*	-networktimeout *integer*	Specifies the length of time (in seconds) that the probe can wait without a response; after this time, the connection to the ObjectServer times out. The maximum value is 2147483, and the default is 0, meaning that no timeout occurs. If a timeout occurs, the probe attempts to connect to the backup ObjectServer, identified by the **ServerBackup** property. If a timeout occurs and no backup ObjectServer is specified, the probe enters store-and-forward mode. In a standard failover setup, the value of the **NetworkTimeout** property must be less than the value of the **PollServer** property. The **NetworkTimeout** setting overrides the operating system-level TCP/IP timeout setting.	On reconnection

Table 30. Common probe properties and command-line options (continued)

Property	Command-line option	Description	Update type
NHttpd.Access Log *string*	-nhttpd_accesslog *string*	Specifies the name of the log file where the server logs all requests that it processes. The default is access.log.	On startup only
NHttpd. AuthenticationDomain *string*	-nhttpd_authdomain *string*	Specifies the authentication domain that is used when requesting authentication details over the HTTP or HTTPS connection. The default is omnibus.	On startup only
Nhttpd.Basic Auth *string*	-nhttpd_basicauth *string*	Specifies the user-password combination that is permitted for connections to the HTTP interface. Ensure that the password associated with the user name is encrypted as an AES-128 hash. Use **nco_crypt** with the -aes option to encrypt the password. Specify the password in the format "*username*:*password*", where *username* is any set of characters that does not include a colon (:) and *password* is the AES-128 hash of the password. The default value of this property is "", which accepts any user credential as valid.	On startup only
NHttpd.Document Root *string*	-nhttpd_docroot *string*	Specifies the document root of the embedded Web service requests. The default is ./.	On startup only
NHttpd.Enable File Serving TRUE | FALSE	-nhttpd_enablefs TRUE | FALSE	Use this property to enable default file serving by the probe. This allows the probe to act as a simple HTTP server that serves files from the local filesystem. The default is FALSE.	On startup only
NHttpd.Enable HTTP TRUE | FALSE	-nhttpd_enablehttp	Use this property to enable the use of the HTTP port. The default is TRUE.	On startup only
NHttpd.Expire Timeout *string*	-nhttpd_expire timeout *string*	Specifies the maximum time, in seconds, that an HTTP/1.1 connection remains idle before it is dropped. The default is 15 seconds.	On startup only

Table 30. Common probe properties and command-line options (continued)

Property	Command-line option	Description	Update type
NHttpd. Listening Hostname *string*	`-nhttpd_hostname` *string*	Specifies the listening host name or IP address that can be used as the `hostname` part of a URI to the probe's HTTP or HTTPS interface. The probe registers the value of this property in the Hostname column of the registry.probes table. You can use this property to specify a different host name to the local default host name. You might need to do this if the host name that is automatically resolved is not a fully qualified domain name (FQDN) or if there is no remote host name resolution for the name. For example, this can happen with machines that are provisioned as virtual machines within a private virtual sub-net, or with machines that have multiple network cards, or with machines that are configured without a local domain setting. The default is the host name of the local computer. You can check the host name using the operating system **hostname** command.	On startup only
NHttpd. Listening Port *integer*	`-nhttpd_port` *integer*	Specifies the port on which the probe listens for HTTP requests. The default is `8080`.	On startup only
NHttpd.NumWork Threads *integer*	`-nhttpd_numworkthrs` *integer*	Specifies the maximum number of worker threads that can be used to service incoming HTTP or HTTPS requests. Use this property to specify how many HTTP or HTTPS requests the probe can handle simultaneously. Each request of the probe is handled in a single worker thread, which is returned to the thread pool after the request has been serviced. The default is `5`.	On startup only
NHttpd.SSL Certificate *string*	`-nhttpd_sslcert` *string*	Specifies the name of the Secure Sockets Layer (SSL) certificate of the server. The default is `cacert.pem`.	On startup only
NHttpd.SSL Certificate –Pwd *string*	`-nhttpd_sslcertpwd` *string*	Specifies the password required to access the SSL certificate file. The default is `""`.	On startup only
NHttpd.SSL Enable `TRUE | FALSE`	`-nhttpd_sslenable`	Use this property to enable the use of SSL support. The default is `FALSE`.	On startup only
NHttpd.SSL Listening Port *integer*	`-nhttpd_sslport` *integer*	Specifies the port on which the probe listens for HTTPS requests. The default is `0`.	On startup only

Table 30. Common probe properties and command-line options (continued)

Property	Command-line option	Description	Update type
OldTimeStamp TRUE | FALSE	-oldtimestamp TRUE | FALSE	Specifies the timestamp format to use in the log file. Set the value to TRUE to display the timestamp format that is used in Tivoli Netcool/OMNIbus V7.2.1, or earlier. For example: dd/MM/YYYY hh:mm:ss AM or dd/MM/YYYY hh:mm:ss PM when the locale is set to en_GB on a Solaris 9 computer. Set the value to FALSE to display the ISO 8601 format in the log file. For example: YYYY-MM-DDThh:mm:ss, where T separates the date and time, and hh is in 24-hour clock. The default is FALSE.	On startup only
PeerHost *string*	-peerhost *string*	Specifies the host name of the network element acting as the counterpart to this probe instance in a peer-to-peer failover relationship. The default is localhost.	On startup only
PeerPort *integer*	-peerport *integer*	Specifies the port through which the master and slave communicate in a peer-to-peer failover relationship. The default port is 9999.	On startup only
PollServer *integer*	-pollserver *integer*	If connected to a backup ObjectServer because failover occurred, a probe periodically attempts to reconnect to the primary ObjectServer. This property specifies the frequency in seconds at which the probe polls for the return of the primary ObjectServer. It does this by disconnecting and then reconnecting to the primary ObjectServer if available, or to the secondary ObjectServer if the primary is not available. Polling is the only way that the probe can determine if the primary ObjectServer is available. The default is 0, meaning that no polling occurs. When a probe connects to an ObjectServer, the probe checks the **BackupObjectServer** property setting of the ObjectServer to which it is connecting. Polling occurs only if this property is set to TRUE, indicating a backup ObjectServer. **Note:** A probe can go into store-and-forward mode when the primary ObjectServer becomes unavailable. The first alert is not forwarded to the backup ObjectServer until the second alert opens the connection to the backup. If **PollServer** is set to less than the average time between alerts, the ObjectServer connection is polled before an alert is sent, and the probe does not go into store-and-forward mode. For controlled failback, set **PollServer** to 0 to disable automatic failback of a probe that is connected to a failover pair of ObjectServers.	On reconnection
ProbeWatch Heartbeat Interval *integer*	-probewatchheart beatinterval *integer*	Generates a ProbeWatch Heartbeat event if this property is set to a positive number. The number defines the interval (in seconds) at which the heartbeats are generated. If set to 0 (zero), or a negative number, no ProbeWatch heartbeats are generated.	Immediate
Props.Check Names TRUE | FALSE	N/A	When TRUE, the probe does not run if any specified property is invalid. The default is TRUE.	On startup only

Table 30. Common probe properties and command-line options (continued)

Property	Command-line option	Description	Update type
PropsFile *string*	-propsfile *string*	Specifies the name of the properties file. The default is $OMNIHOME/probes/*arch*/*probename*.props, where *probename* is the name of the probe and *arch* represents the operating system.	On startup only
RawCapture 0 | 1	-raw -noraw	Controls the raw capture mode. Raw capture mode is usually used at the request of IBM Software Support. By default, raw capture mode is disabled (0). **Note:** Raw capture can generate a large amount of data. By default, the raw capture file can grow indefinitely, although you can limit the size using the **MaxRawFileSize** property. Raw capture can also slow probe performance due to the amount of disk activity required for a busy probe.	Immediate
RawCaptureFile *string*	-capturefile *string*	Specifies the name of the raw capture file. The default is $OMNIHOME/var/*probename*.cap, where *probename* is the name of the probe.	Immediate
RawCaptureFile Append 0 | 1	-rawcapappend -norawcapappend	Specifies that new data is appended to the existing raw capture file, instead of overwriting the file. By default, the file is overwritten (0).	Immediate
RegexpLibrary *string*	-regexplib *string*	Defines which regular expression library to use. Possible values are: NETCOOL and TRE. The default value of TRE enables the use of the extended regular expression syntax on single-byte and multi-byte character languages. This setting results in decreased system performance. The NETCOOL value is useful for single-byte character processing and provides optimal system performance.	On startup only
RetryConnection Count *integer*	N/A	Specifies the number of events the probe processes in store-and-forward mode before trying to reconnect to the ObjectServer. The default is 15.	On reconnection
RetryConnection TimeOut *integer*	N/A	Specifies the number of seconds that the probe processes events in store-and-forward mode before trying to reconnect to the ObjectServer. The default is 30.	On reconnection

Table 30. Common probe properties and command-line options (continued)

Property	Command-line option	Description	Update type
RollSAFInterval *integer*	-rollsafinterval *integer*	Used when the probe is connected to an ObjectServer, and circular store and forward is enabled by setting **StoreAndForward** to 2. Specifies the time interval in seconds after which a store-and-forward file is rolled over to the next file in the pool of two files that are used to store a copy of events that are sent to a connected ObjectServer. To minimize event loss during failover and failback, set the time interval to a value that is greater than or equal to the granularity of the ObjectServer. In case of failure, the probe will have a copy of events from the last granularity period, which can be replayed to the backup ObjectServer. The default is 90 seconds, which is 1.5 times greater than the default granularity period of 60 seconds that is set for an ObjectServer.	Immediate
RulesFile *string*	-rulesfile *string*	Specifies the name and location of the rules file. This can be a file name or Web address that specifies a rules file located on a remote server that is accessible using HTTP. The default is $OMNIHOME/probes/*arch*/*probename*.rules, where *probename* is the name of the probe.	On reconnection
SAFFileName *string*	-saffilename *string*	Specifies the name of the store-and-forward file. The default is $OMNIHOME/var/*probename*.store, where *probename* is the name of the probe. A *.servername* extension is automatically appended to the file name, where *servername* is the name of the target ObjectServer. A separate store-and-forward file is created for each registered target ObjectServer.	On startup only
SAFPoolSize *integer*	-safpoolsize *integer*	Used when the probe is not connected to an ObjectServer. Specifies the number of store-and-forward files in a pool of files that can be used to store alerts. The default is 3. Each file rolls over to the next when it reaches the maximum size specified by the **MaxSAFFileSize** property.	Immediate
SecureLogin 0 | 1	-securelogin -nosecurelogin	Specifies whether the probe uses an encrypted secure login to access the host system: • 0: The probe does not use an encrypted secure login. • 1: The probe uses an encrypted secure login. The default is 0. **Note:** Secure login is not available in FIPS 140–2 mode. SSL is more secure than secure login.	On reconnection

Table 30. Common probe properties and command-line options (continued)

Property	Command-line option	Description	Update type
Server *string*	-server *string*	Specifies the name of the primary ObjectServer or the proxy server to which alerts are sent. The default is NCOMS. If you want the probe to operate in circular store-and-forward mode, do not specify a virtual ObjectServer definition as the value of this property.	On reconnection
ServerBackup *string*	N/A	Specifies the name of a backup ObjectServer to which the probe should connect if the primary ObjectServer connection fails. If **NetworkTimeout** is set, use **ServerBackup** to identify a backup ObjectServer. If you want the probe to operate in circular store-and-forward mode, do not specify a virtual ObjectServer definition as the value of this property.	On reconnection
SingleThreaded Comms TRUE | FALSE	-singlethreadedcomms	Specifies whether multithreaded or single-threaded processing is used to process and send alerts to the target ObjectServers. The default is FALSE, which enables multithreaded communication. You can also use the **SingleThreadedComms** property to enforce an order for sending alerts to ObjectServers. With multithreaded processing, alerts are simultaneously sent to the target ObjectServers. In single-threaded mode, the order is defined by the order in which the registertarget statements are listed in the rules file.	On reconnection
SSLServer CommonName *string1,...*	N/A	If the probe is connecting to an ObjectServer using SSL, and the Common Name field of the received certificate does not match the name specified by the **Server** property, use this property to specify a comma-separated list of acceptable SSL Common Names. The default setting is to use the **Server** property.	On reconnection
StoreAndForward *integer*	-saf *integer*	Controls the store and forward operations. Possible values for the property are: • 0: Do not use store and forward. • 1: Use legacy store and forward, which stores alerts in a store-and-forward file only if the alerts cannot be sent to an ObjectServer. • 2: Use circular store and forward, which stores all generated alerts in a rolling pool of store-and-forward files while the probe is connected to an ObjectServer. If the probe is disconnected, the circular store and forward behavior is similar to the legacy store and forward behavior. By default, the legacy store-and-forward mode is enabled (1).	On reconnection

Table 30. Common probe properties and command-line options (continued)

Property	Command-line option	Description	Update type
StoreSAFRejects 0 | 1	-storesafrejects -dontstoresafrejects	Specifies whether the probe should continuously save the individual corrupted store-and-forward records for analysis. If set to 1, corrupted store-and-forward records are continuously saved. The default is 0. Use this property in conjunction with the **KeepLastBrokenSAF** property.	Immediate
N/A	-version	Displays version information and exits.	On startup only

Related concepts:

"Probe property versus probe command-line option usage" on page 6
Each probe property has a corresponding command-line option.

"Probe property types" on page 6
Probe properties can be divided into two categories: common properties and probe-specific properties.

"Store-and-forward mode for probes" on page 11
Probes can continue to run if the target ObjectServer is down. During this period, the probe switches to *store* mode. The probe reverts to *forward* mode when the ObjectServer is functional again.

"Raw capture mode for probes" on page 14
You can use the raw capture mode to save the complete stream of event data acquired by a probe into a file, without any processing by the rules file. This can be useful for auditing, recording, or debugging the operation of a probe.

"Secure mode for probes" on page 15
You can run the ObjectServer in secure mode. When you start the ObjectServer using the `-secure` command-line option, the ObjectServer authenticates probe, gateway, and proxy server connections by requiring a user name and password.

"Peer-to-peer failover mode for probes" on page 16
Two instances of a probe can run simultaneously in a peer-to-peer failover relationship. One instance is designated as the master. The other instance acts as a slave and is on hot standby. If the master instance fails, the slave instance is activated.

Related tasks:

Chapter 5, "Remotely administering probes," on page 91
You can use an HTTP interface that runs over an HTTP or HTTPS server contained in the standard probe C library (libOpl) to remotely monitor and manage probes. This functionality can be used against all probes that are compatible with Tivoli Netcool/OMNIbus V7.4. A common URI is provided that contains resources for performing administration tasks.

"Rereading the rules file" on page 60
Because probes read the rules file only on startup, you must force the probe to reread the rules whenever you make changes to it. The probe processes the reread request only on receipt of a new event. If the probe is idle or is already processing an event, it will not reread the rules file until a new event is received.

Related reference:

"Multithreaded processing of alert data" on page 52
When a probe rules file is processed, multithreaded processing is used by default to apply probe rules to the raw event data that is acquired from the event source, and to send the generated alerts to the registered ObjectServers. Note that this multithreaded processing is different from the multithreaded or single-threaded event capture that is implemented in some classes of probes.

"Lookup table operations" on page 43
Lookup tables provide a way to add extra information in an event. A lookup table consists of a list of keys and values.

Chapter 7. Netcool MIB Manager

Netcool MIB Manager is an IBM® Eclipse-based application that you can use to parse Simple Network Management Protocol (SNMP) management information base (MIB) files, from which you can generate Netcool rules files. It is intended as a replacement for the **mib2rules** utility.

Netcool MIB Manager has the following features:

- It imports SNMP MIB files and resolves MIB dependencies to build an Object Identifier (OID) tree.
- It can export some or all of the imported SNMP objects to Netcool rules files and other file formats.
- It includes the base MIBs and RFC MIBs most commonly required by other MIB files.
- It includes device definitions to enable device-centric grouping of imported MIB modules.
- It can filter and search MIB Modules and the OID tree by object name or OID.
- It can generate SNMP traps to send to the SNMP Probe (`nco_p_mttrapd`) or other SNMP agents.
- It has a command-line interface that you can use to import MIB data and export rules files. You can issue commands manually or call them programmatically.

Starting MIB Manager

You can start MIB Manager from the command line. It is not designed to be run under process control or as a Windows service.

About this task

To start MIB Manager, use the following command:

- On UNIX or Linux: `$NCHOME/omnibus/bin/nco_mibmanager`
- On Windows: `%NCHOME%\omnibus\bin\nco_mibmanager.bat`

MIB Manager also has a command-line interface that you can use manually or call programmatically.

Note:

See the Supported Operating Systems documentation for information about the requirements for running MIB Manager on the AIX and Solaris operating systems.

This requirements information is available in the *Tivoli Netcool/OMNIbus Installation and Deployment Guide*.

On all UNIX and Linux operating systems, ensure that the `$DISPLAY` environment variable is set to a functioning X11 server before starting MIB Manager.

Related reference:

"MIB Manager command-line options" on page 146
MIB Manager has a command-line utility that you can use to import MIB data and export rules files. You can issue commands manually or call them programmatically.

Using Netcool MIB Manager

You can use MIB Manager to parse MIB modules and create Netcool rules files.

MIB Manager includes a set of base MIBs and RFC MIBs, in the `$NCHOME/omnibus/platform/arch/mibmanager/workspace/mibs` directory. These MIBs are supplied to enable you to import additional MIBs. MIB Manager automatically uses these bundled MIBs to resolve dependencies during the import of new MIBs. The bundled MIB modules are visible in the MIB Modules view when you start MIB Manager for the first time.

MIB Manager also includes XML data files that represent an import of the bundled base MIBs. MIB Manager uses these XML files to store the MIB data it imports from the raw text MIB files.

Add your MIBs to a directory that can be accessed from the server from which the MIB Manager is started. All standards-based MIBs must be installed in a separate directory which can be placed in the search path for easy dependency resolution. By default, the basic SNMP MIBs are installed in the `$NCHOME/omnibus/platform/arch/mibmanager/workspace/mibs/base` directory, and all RFC MIBs are installed in the `$NCHOME/omnibus/platform/arch/mibmanager/workspace/mibs/rfc` directory.

The Netcool MIB Manager window

When you start MIB Manager, the following elements are displayed in the initial MIB Manager window: several elements are displayed in the initial MIB Manager window. The following figure shows these elements.

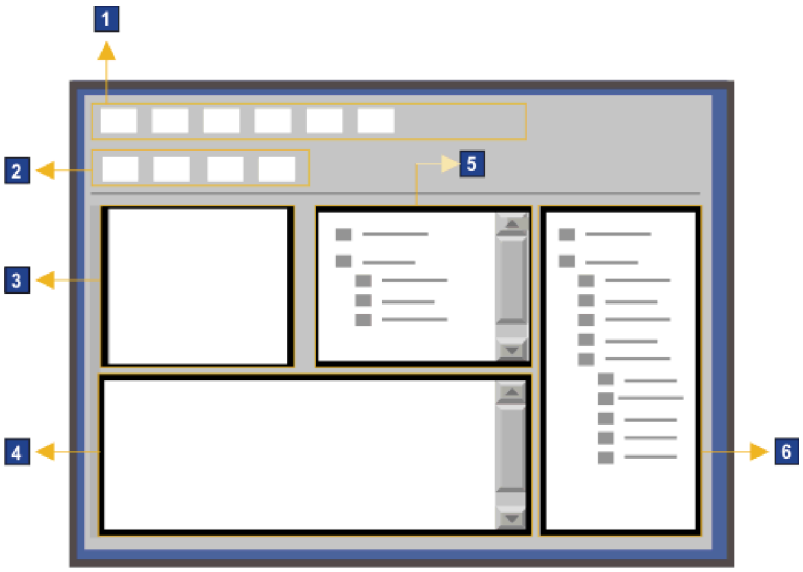

Figure 4. MIB Manager initial window

1 Toolbar
Provides access to the most frequently used MIB Manager functions.

2 Perspectives
You can view MIB data in one of two perspectives, **MIB Manager** or **mib2rules**. The **MIB Manager** perspective is the default perspective. The **mib2rules** perspective focuses on Object Identifiers (OIDs) and is provided for users who are familiar with the **mib2rules** utility.

3 Device view
Use this view to create, update, and delete MIB Manager devices.

4 Details view and Console view
The Details view displays the details of selected MIB elements. The Console view shows all log messages that are generated by MIB Manager while the current instance is running.

5 MIB Modules
Use this view to search for and delete MIB modules, and to search for specific MIB values.

6 OID Tree view
This view gives a representation of all MIB objects that are in the imported MIB modules.

The MIB Modules view

The MIB Modules view provides a representation of all the imported MIB modules and allows you to search for MIB modules, delete MIB modules, and to search for specific MIB values.

You move through the MIB Modules view by expanding and collapsing nodes within the MIB tree. The top level branches of the MIB tree contain the names of the MIB modules, and contained within each MIB module branch are the other elements which comprise the MIB.

You can filter the data in the MIB Modules view by selecting one of the following options from the **View** drop-down list:

- **All**
- **Modules**
- **Enumerations**
- **Textual Conventions**
- **Syntaxes**
- **Imports**
- **Exports**
- **Columns**
- **Descriptions**

To search theMIB Modules view for a specific value, type your text in the **Search** field and either press **Enter** or click (Search).

MIB Manager begins the search from the selected item or, if no item is currently selected, from the first item in the list. To find the next matching item from the list, continue to click (Search).

When an item is located, it is automatically highlighted and displayed in the MIB Modules view.

In the **Search** field, specify a valid regular expressions or enter plain text to represent a complete or partial object name.

To delete one or more MIB modules, right-click the selected MIB module(s), and then from the pop-up menu, select **Delete**. The selected MIB module is deleted from MIB Manager. However, the original MIB source files, supplied for the MIB import, are not be deleted.

Note: You must only delete a MIB module if it contains errors and there is a new replacement version of the MIB module. Deleting MIB modules using this method can result in missing MIB dependencies. Incomplete rules files can also be generated during the export process together with warning or error messages in the log file.

The following table lists the MIB Modules view icons and describes what each one represents.

Table 31. The MIB Modules view icons and the objects they represent

Icon	Representation
	EXPORTS
	EXPORT
	IMPORTS
	IMPORT
	OBJECT IDENTIFIER, OBJECT-IDENTITY
	AGENT-CAPABILITIES
	MODULE-COMPLIANCE
	MODULE, MODULE-IDENTITY
	OBJECT-GROUP
	NOTIFICATION-GROUP
	TRAP-TYPE (v1 trap)
	NOTIFICATION-TYPE (v2 trap)
	Generic OBJECT-TYPE
	COUNTER
	COUNTER32
	COUNTER64

Table 31. The MIB Modules view icons and the objects they represent (continued)

Icon	Representation
	GAUGE
	GAUGE32
	INTEGER
	INTEGER32
	IpAddress
	NetworkAddress
	OID
	OCTET STRING
	SEQUENCE (table)
	SEQUENCE (Entry, table row)
	TIMETICKS
	TEXTUAL-CONVENTION

The OID Tree view

The Object Identifier (OID) Tree view provides a representation of all MIB objects that are located in the imported MIB modules.

You can filter the data in the OID Tree view by selecting one of the following options from the **View** drop-down list:

- **All**
- **Traps/Notifications**
- **Objects**
- **Modules**

To search the OID Tree view for a specific value, type your text in the **Search** field and either press **Enter** or click  (Search).

MIB Manager begins the search from the selected item or, if no item is currently selected, from the first item in the list. To find the next matching item from the list, continue to click (Search).

When an item is located, it is automatically highlighted and displayed in the OID Tree view.

In the **Search** field, specify a valid regular expressions or enter plain text to represent a complete or partial object name, or OID. The number displayed on each branch node is the last digit of the OID for that particular object.

The following table lists the OID Tree view icons and describes what each one represents.

Table 32. The OID Tree tree icons and the associated objects they represent

Icon	Representation
	OBJECT IDENTIFIER, OBJECT-IDENTITY
	AGENT-CAPABILITIES
	MODULE-COMPLIANCE
	MODULE, MODULE-IDENTITY
	OBJECT-GROUP
	NOTIFICATION-GROUP
	TRAP-TYPE (v1 trap)
	NOTIFICATION-TYPE (v2 trap)
	Generic OBJECT-TYPE
	COUNTER
	COUNTER32
	COUNTER64
	GAUGE
	GAUGE32
	INTEGER
	INTEGER32
	IpAddress
	NetworkAddress
	OID
	OCTET STRING

Table 32. The OID Tree tree icons and the associated objects they represent (continued)

Icon	Representation
	SEQUENCE (table)
	SEQUENCE (Entry, table row)
	TIMETICKS
	TEXTUAL-CONVENTION

Importing MIB data

How to import SNMP MIB data into MIB Manager.

Before you begin

In the Preferences window, specify locations for the MIBs that are regularly used by the vendor MIBs. Ensure your MIBs are located in vendor-specific directories, and that any associated equipment numbers are located in subdirectories. Also ensure that these directories do not contain any non-MIB text files. Most MIBs are installed into the `$NCHOME/omnibus/platform/`*`arch`*`/mibmanager/workspace/mibs/base` directory, and all the RFC MIBs are installed in the `$NCHOME/omnibus/platform/`*`arch`*`/mibmanager/workspace/mibs/rfc` directory.

The MIBs that you are importing might be dependant on other MIBs. To ensure that MIB Manager can search for dependant MIBs while performing the import, set the search path to the locations of the dependant MIB files.

Procedure

To import MIB data:

1. Click  **Import** or, from the menu, click **File** > **Import**.
2. In the **Directory** field, specify the location of the MIB modules that you want to import.
3. To make MIB Manager parse all the MIBs found in available subdirectories, select the **Traverse subdirectories** check box.
4. Click **Import**.

   The Import Status window is displayed as the MIB modules are parsed. The left side of this window displays the number of different MIB object types as each one is discovered during the parsing of the imported MIB files. After the MIBs are parsed, MIB Manager reviews each MIB module and parses all the MIB objects that it locates. After all the objects are parsed, MIB Manager reviews each MIB and processes the import statements to locate any referenced MIB modules. If any referenced MIB modules cannot be located in previously imported MIB modules, MIB Manager searches the directories specified in the search path. The right side of the Import Status window displays icons to show the status of any import statements found in the imported MIBs, as shown in the following table:

Icon	Description
?	Indicates that the import statement has not been resolved.
✔	Indicates that the import statement has been resolved.
✖	Indicates that the MIB Module referenced in the import statement could not be resolved.

You can expand the import statement tree to display which MIB modules are using import statements to reference other MIB modules.

Import Complete is displayed at the top of the status frame when all the imported MIB files have been processed. You can view the all the status messages generated during the import by clicking **View Status History** and using the **View** drop-down list to select a filter option.

5. Click **Dismiss** to close the Import Status window.

Results

The MIB Modules view refreshes automatically to show the newly imported MIB module names. The OID Tree view refreshes automatically to show all the newly imported MIB objects To rebuild the tree, MIB Manager searches the parent nodes of the tree (`iso`, `ccitt`, and `joint-iso-ccitt`) recursively for child nodes until it finds a child node that has no associated children. If no errors are detected, a fully populated MIB tree is displayed in the OID Tree view, which contains the parsed objects.

Clicking an object in the OID Tree view automatically displays the object in the MIB Modules view. The Details view displays detailed information about the selected object.

The imported MIB modules are stored in an xml file which represents the structure of that MIB module. The xml files are located in the `workspace/data` directory.

If unresolved import statements are detected, contact the vendor responsible for the missing MIB or MIBs, add the missing file or files to the search path and repeat the import.

Note:

Vendors sometimes specify duplicate object names, or specify an object name that is identical to an object name specified in an RFC.

For example, an object named `system` is defined in module SNMPv2-MI' with OID 1.3.6.1.2.1.1 and another object named `system` is defined in module WINDOWS-NT-PERFORMANCE, with OID 1.3.6.1.4.1.311.1.1.3.1.1.23. Also, an object named `sysDescr` is defined as `system.1` in module SNMPv2-MIB and another object named `sysFileReadOperationsPerSec` is defined as `system.1` in module WINDOWS-NT-PERFORMANCE. In this case, it is unclear which parent object the child objects are associated with.

When an object with a duplicate parent name is located, MIB Manager attempts to find the parent that is defined in the same MIB module. In the previous example, this is WINDOWS-NT-PERFORMANCE. If neither of the parents are defined in the same MIB module, MIB Manager then searches to see if the parent is defined in

other MIB modules on which the object is dependant. MIB Manager also identifies other MIB modules that reference the parent and selects the most commonly referenced parent name. However, this could result in an incorrectly populated MIB tree and incorrectly calculated OIDs. Therefore, each time MIB Manager finds a duplicate object name, it logs a warning and records the actions taken to resolve the duplication in a debug file.

Related reference:

"Setting directory preferences" on page 143
Use the directory preferences to specify the directories used for storing imported MIBs, MIB data, and exported rules files.

"Setting search preferences" on page 145
This section describes how to define the MIB Manager preferences that are used to search for the MIB files.

Exporting MIB data

How to export SNMP MIB data from MIB Manager to a variety of formats.

Procedure

To export MIB data:

1. Click **Export** or, from the menu, click **File** > **Export**.
2. In the **Directory** field, specify the destination directory for the exported files.
3. In the **File Type** field, specify a file type for the exported files.

   You must use the Netcool Knowledge Library (NCKL) whenever possible, because the NCKL rules files are based on the functionality of the device. When using NCKL, you must specify the **Netcool Knowledge Library version 3.x** file type. This enables the generated files to be added to the existing Netcool Include Library installations.

   The following options are available from the **File Type** drop-down list:

File type	Description
CSV	Select this option to specify that all the object data is loaded into CSV files, so that it can be viewed using a spreadsheet application.
CSV Trap Objects	Select this option to specify that all trap data and associated variable bindings object data is loaded into a single CSV file, so that it can be viewed using a spreadsheet application.

File type	Description
HTML with frames	Select this option to create a web page that is suitable for publishing. The selected MIB tree view is displayed on the left side of the generated web page. You can navigate through the MIB tree view by expanding and collapsing nodes within the MIB tree. When an element is selected from the MIB tree, its associated information is displayed on the right side of the web page. This format can often create a large number of export files, each containing a HTML file for each object listed in the MIB tree. The target directory can be zipped up and located under any web server, or the pages can be viewed directly from a disk by opening the `index.html` file created in the export directory. If you want to use individual HTML files with a web browser, export to **HTML with frames** and then use the files in the `data` sub-directory located under the export directory.
HTML without frames	Select this option to create a single file named `oids.html`. This file will contain a list of all objects selected for output, each separated by horizontal rules (using the `<HR>` tag). Each OID is enclosed by an anchor tag so that when it is loaded, the browser automatically scrolls down to any OID that includes a hash symbol (#) in the URL, for example: `file:c:/oids.html#1.3.6.1.4.1.9`. This file can become very large if you are exporting a large number of objects. Therefore, this format should only used for exporting a small number of traps.
Netcool Lookup Table	Select this option to output all specified values in a tabbed list, with the OIDs displayed in the left column and the object names displayed in the right column. This is suitable for inclusion into any rules file.
Netcool Include Library	Select this option to create separate include files for each enterprise, and to generate a single file for the generic traps (that is, for any traps that are not located below the enterprise sub-tree). It is assumed that the include files are inserted into existing `snmp.rules` files, either those previously created by MIB Manager or those created by the Netcool Include Library (NCIL).
Netcool Knowledge Library version 1.1	Select this option to output files from MIB Manager. The files will be included in an existing Netcool Knowledge Library implementation.

File type	Description
Netcool Knowledge Library version 3.x	Select this option to output files from MIB Manager. The files will be included in an existing Netcool Knowledge Library implementation.
Standalone Rules	Select this option to create a single rules file that contains all the selected traps. You can specify this file in the properties file of any Tivoli Netcool/OMNIbus SNMP trap probe.
Individual Text Files	The Individual Text Files format is used for the Netcool Event List right-click menu. When you export to individual text files, each OID is output to a separate text file, for example `1.3.6.1.4.1.9.1.1.txt`. An external tool can then be added to the right-click menu that uses `nco_message` to display all object information for an event that has the OID specified in a field, for example: `cat 1.3.6.1.4.1.9.1.txt |nco_message stdin title "trap info"`.
Tivoli Universal Agent	The Tivoli Universal Agent format exports `.mdl` and `trapcnfg` files based on the selected content. Multiple sub-trees can be selected when this format is selected for exporting files. If multiple sub-trees are selected, `.mdl` and `trapcnfg` files are created for all the objects located below a selected item in the MIB tree. **Note:** The first three letters of the application name, following the //APPL tag in the `.mdl` file, must be unique within an enterprise. MIB Manager tries to adhere to this requirement but it cannot be aware of any other agents that may be running. You may need to modify the `.mdl` file to ensure that the application name is unique in the enterprise.

4. In the **Scope** field, use the drop-down list to specify whether MIB Manager outputs traps only, objects only, or all OIDs.

   The options available depend on the file type that you are exporting.
5. To limit the number of objects included in the generated file, select the **Selected Subtree(s) only** checkbox.
6. Click **Export** to export the files to the destination directory.

Related reference:

"Setting export preferences" on page 143
This section describes how to set your rules file export preferences.

Editing SNMP traps

How to edit SNMP traps.

About this task

In the Details view, you can edit the following fields of an SNMP trap:

- @Severity
- @Type
- @ExpireTime
- CODEBLOCK

Note: SNMP does not provide standard mechanisms for specifying set (@Type) or clear (@Severity) information. You must specify this information manually.

In the Details view, double-click a field value to make it editable. After making a change to a field, click ✔ to save the change or click ✖ to cancel the change.

Procedure

To edit an SNMP trap:

1. In either the MIB Modules view or the OID Tree view, select a trap.

   Detailed information about the trap is displayed in the Details view.
2. To edit the **@Severity** field, double-click the field value and choose a new value from the drop-down list.
3. To edit the **@Type** field, double-click the field value and choose a new value from the drop-down list.
4. To edit the **@ExpireTime** field, double-click the field value and enter a new time period (in seconds, minutes, hours, or days).
5. To edit the **CODEBLOCK** field, double-click the field value (even if the field is empty) and enter a new block of code.

Generating SNMP traps

How to generate SNMP traps. This functionality can also be used for testing rules files generated by MIB Manager.

Procedure

To generate an SNMP trap:

1. Click ! **Trap**.
2. In the **Destination Address** field, specify the host name or IP address of a computer running an SNMP Trapd service.
3. In the **Destination Port** field, specify the UDP port that the SNMP Trapd service is listening on.
4. In the **Community** field, specify a valid string for the Trapd manager.
5. Depending on the type of trap that you are generating, a **Variable Bindings** section might be visible in the Generate an SNMP Trap window. The available fields in the **Variable Bindings** section also depend on the type of trap being generated.

   Mouse over the **Variable Bindings** field names for details of the field value syntax. You can specify multiple values, separated by commas. These values will be randomly chosen to build the trap.

6. In the **Repeat Metric** field, specify the number of traps (from 1 to 1000) that can be generated.

   If you specify a large number, traps are generated in rapid succession and can cause an alarm storm at the Trapd manager. For values greater than one (> 1), and for values that are specified as semicolon separated lists, MIB Manager randomly selects from the specified lists and generates traps with the arbitrary values presented as varbinds.
7. Click **Execute** to generate the trap.

Results

The **Result** field in the Generate an SNMP Trap window will indicate whether the trap was sent successfully.

Using MIB Manager devices

A MIB Manager device is used to group associated MIB Modules and define all the objects required for exporting.

MIB Manager devices are displayed in the Device view. Select a device from the Device tree to display the manufacturer, the model number, and the OS version of the device.

Creating a new device

How to create a new device.

Procedure

To create a new device:

1. In the MIB Modules view, select at least one MIB Module and then click  (Create Device).
2. In the **Manufacturer** field, select a device manufacturer from the drop down list.

   The **Model** and **OS Version** fields are populated with the available options.
3. In the **Model** field, select an option from the drop down list.

   If no models are available, enter a descriptive model name.
4. In the **OS Version** field, select an option from the drop down list.

   If no OS versions are available, enter a descriptive OS version name.
5. In the **Mapped Mib Modules** field, select the MIB modules that you want to associate with the new device.
6. Click **OK** to create the device.

Results

The new device is displayed in the Device view.

Updating a device

How to update a device.

Procedure

To update a device:

1. In the Device view, select a device and expand its subnodes until the OS version is visible.
2. Select the OS version and then click (Edit Device).
3. In the **Manufacturer** field, select a device manufacturer from the drop down list.

   The **Model** and **OS Version** fields are populated with the available options.
4. In the **Model** field, select an option from the drop down list.

   If no models are available, enter a descriptive model name.
5. In the **OS Version** field, select an option from the drop down list.

   If no OS versions are available, enter a descriptive OS version name.
6. In the **Mapped Mib Modules** field, select the MIB modules that you want to associate with the new device.
7. Click **OK** to update the device details.

Results

The updated device details are displayed in the Device view.

Deleting a device

How to delete a device.

Procedure

To delete a device:

In the Device view, select the device and then click (Delete Device).
To delete multiple devices, hold down the Ctrl+Alt keys, select the elements using the left mouse button, and then click (Delete Device).

Results

The device is deleted and removed from the Device view.

Configuring global preferences

You can configure several MIB Manager global preferences from the Preferences window.

To configure the MIB Manager global preferences, click **Preferences** or, from the menu, click **File** > **Preferences**.

In the Preferences window, you can configure directory, export, general, logging, and search preferences.

Setting directory preferences

Use the directory preferences to specify the directories used for storing imported MIBs, MIB data, and exported rules files.

To set the directory preferences, click **Preferences** or, from the menu, click **File** > **Preferences**, and then from the left pane of the Preferences window, click **Directory**.

- In the **Data Directory** field, specify where the imported MIBs will be stored as XML files. The default location is `$NCHOME/omnibus/platform/arch/mibmanager/workspace/data`.
- In the **Export Directory** field, specify where the exported files are generated. A subdirectory based on the type of export is created. The default location is `$NCHOME/omnibus/platform/arch/mibmanager/workspace/export`.
- In the **Import Directory** field, specify where the imported MIB files are located. The default location is `$NCHOME/omnibus/platform/arch/mibmanager/workspace/mibs`.

Related tasks:

"Importing MIB data" on page 135
How to import SNMP MIB data into MIB Manager.

Setting export preferences

This section describes how to set your rules file export preferences.

To set the directory preferences, click **Preferences** or, from the main menu, click **File** > **Preferences**, and then from the left pane of the Preferences window, click **Export**.

- In the **Include path** field, specify the location of the rules that were installed with the Netcool Include Library (NCiL) rules file packages (if they are installed). This option is only required if you are exporting data in the Netcool Include Library file format and when the directory path must point to included rules. As a result, the new rules files do not need to be edited manually. The default value is `$NCHOME/omnibus/probes/arch`.
- In the **Default export file format** field, specify either a Windows/DOS format (CRLF) or Unix format (LF) for the output file. This property allows you to create rules on a Windows machine and copy them to a UNIX machine without running a utility such as `dos2unix`.

  Note: This selection applies to all output files and is not restricted to the rules files. The default value is Unix format.
- In the **Alert types** field, specify the drop-down list options displayed when a trap's `@Type` field is edited in the Detail view.

The following options are only available if the NCKL export format was previously selected:

- If the **Include varbind descriptions** check box is selected, the varbind descriptions are added as comment documentation. The default value is enabled.
- If the **Calculate a better value for @AlertKey** check box is selected, the `@AlertKey` value is generated from a varbind parent object. The default value is enabled.
- If the **Set @Agent once per enterprise** check box is selected, the `@Agent` field is set for each enterprise. The `@Agent` field is based on the names of the MIB modules being processed for the rules file. The default value is enabled.

- If the **Add @Class template** check box is selected, the `@Class` field is set. If the **Add @Class template** check box is selected, the `@Class` field is set to 300. The default value is enabled.

Related tasks:

"Exporting MIB data" on page 137
How to export SNMP MIB data from MIB Manager to a variety of formats.

Setting general preferences

Use the general preferences to specify the MIB import options.

To set the directory preferences, click **Preferences** or, from the main menu, click **File** > **Preferences**, and then from the left pane of the Preferences window, click **General**.

- If the **Traverse subdirectories on import** check box is selected, all subdirectories are searched during the import process to locate all MIB files.
- In the **File splitter maximum thread count** field, specify the number of computational threads used to detect MIBs during the import process. This can improve performance when importing a large number of MIBs. The default value is 5.
- In the **MIB parser maximum thread count** field, specify the number of computational threads used to parse MIB files during the import process. This can improve performance when importing a large number of MIBs The default value is 10.
- To change the highlighter color of any selections made in the MIB Manager window, click  , select a color from the color palette, and then click **OK**.

Setting logging preferences

Use the logging preferences to specify the MIB Manager log file directory, and to set the default message level.

To set the directory preferences, click **Preferences** or, from the main menu, click **File** > **Preferences**, and then from the left pane of the Preferences window, click **Logging**.

- In the **Log directory** field, specify where the application log file is written. The default location is `$NCHOME/omnibus/log`.
- In the **Default Message Level** field, specify the application's default message level. You can also use the **Edit** menu to change the default message logging level. The message level is valid from when the application is first started until when the application is shutdown, unless it is changed using the **Edit** menu.

  Note: Changing the message level in the **Default Message Level** field does not change the current message level of the application. It only affects the message level that was initially set when the application was first started (the default message level). To avoid generating large log files, set the default message level to `warn`.

Setting the logging level

You can change the logging level to provide more detailed log or trace files to help with debugging MIB Manager. The extra information provided by increasing the logging levels can help you when debugging specific problems when parsing MIBs.

To change the logging level, click **Edit** > **Debug** from the main window menu bar. Log files have six message levels that you can cycle through to increase the level of detail that is captured: none, error, warn, info, debug, and verbose.

The following table provides a brief description of each logging level:

Table 33. A list of logging levels with their associated description

Log level	Description
none	Specifies a log level in which no debug messages are output. The performance of the application is improved but no warning or error messages are displayed during parsing.
error	Specifies a log level in which only errors which affect the results are logged. This includes any error identified by MIB Manager which would result in incorrect data being displayed in the main window.
warn	Specifies a log level in which warning and error messages are logged. A warning indicates a condition that could result in erroneous data and therefore must be checked. However, the condition will not cause the data to be corrupted.
info	Specifies a log level in which informational, warning, and critical messages are logged. Informational messages contain detail regarding overall task progress. For example, they provide the complete text for objects as they are being parsed.
debug	Specifies a log level in which all messages are logged. **Note:** Only run MIB Manager in debug mode when you are attempting to resolve and error condition. The volume of data produced while parsing only a moderate number of MIBs will quickly fill your file system.
verbose	Specifies a log level in which all information messages are logged. A verbose message includes the state of variables, arrays, and hashes as they change value and state.

Setting search preferences

This section describes how to define the MIB Manager preferences that are used to search for the MIB files.

To set the search preferences, click **Preferences** or, from the main menu, click **File** > **Preferences**, and then from the left pane of the Preferences window, click **Search**.

In the **MIB Search Path** field, set the search path to where the MIB files are located. The directories must be separated by a semicolon (;). Any subdirectories are searched automatically to locate dependant MIB files.

The default path includes the location of the base MIB files and the RFC MIB files that are packaged with MIB Manager. You can specify any new directories at the

end of the default file path. This ensures the base directory is initially searched for dependencies containing the most commonly used MIBs, and then the `rfc` directory is searched for the less commonly used MIBs.

Related tasks:

"Importing MIB data" on page 135
How to import SNMP MIB data into MIB Manager.

MIB Manager command-line options

Fix Pack 1

MIB Manager has a command-line utility that you can use to import MIB data and export rules files. You can issue commands manually or call them programmatically.

To start the command-line utility, use the following command:

- UNIX Linux `$NCHOME/omnibus/bin/nco_mibmanager_batch [options]`
- Windows `%NCHOME%\omnibus\bin\nco_mibmanager_batch [options]`: The output from MIB Manager is displayed in a new window.

The following table describes the available command-line options. The `-exportmibs`, `-exportobjects`, and `-exportoids` options can parse regular expressions that conform with the Java pattern engine.

Windows Use full directory paths when you specify directories. For example:
`nco_mibmanager_batch -importdir "C:\anydir\mibs"`

Table 34. MIB Manager command-line options

Command-line option	Description
`-clean`	Use this option when you first run the MIB Manager command-line utility, if you previously used the MIB Manager graphical utility. If you do not use this option, an error is displayed.
`-exportdir` *string*	Use this option to specify the directory to which generated rules files are exported. Enclose the directory path in double quotation marks ("). For example: `nco_mibmanager_batch -exportdir "/home/user/export_cmd" -filetype csv`

`-exportmibs` *string*	Use this option to specify the name of the MIB to be exported. You can use regular expressions to specify multiple MIBs.
	
	For example, the following command exports objects from a MIB named "RMON-MIB":
	
	`nco_mibmanager_batch -exportmibs RMON-MIB -filetype csv`
	
	The following command exports objects from all MIBs whose names contain "MON":
	
	`nco_mibmanager_batch -exportmibs .*MON.* -filetype csv`

Table 34. MIB Manager command-line options (continued)

Command-line option	Description
`-exportnewdirectory`	Use this option to specify that exported files are written to a new directory, within the existing export directory. The new subdirectory is automatically named with a time stamp. Use this option to ensure that existing rules files are not overwritten. For example: `nco_mibmanager_batch -exportdir "/home/user/ export_cmd" -exportnewdirectory -filetype csv`

`-exportobjects` *string*	Use this option to specify the name of the MIB object to be exported. You can use regular expressions to specify multiple objects.
	
	For example, the following command exports all objects named "coldStart":
	
	`nco_mibmanager_batch -exportobjects coldStart -filetype csv`
	
	The following command exports all objects whose names begin with "cold":
	
	`nco_mibmanager_batch -exportobjects cold.* -filetype csv`

`-exportoids` *string*	Use this option to specify the OID to be exported. You can use regular expressions to specify multiple OIDs.
	
	For example, the following command exports all objects with the OID "1.3.6.1.6.3.1.1.5.1":
	
	`nco_mibmanager_batch -exportoids 1.3.6.1.6.3.1.1.5.1 -filetype csv`
	
	The following command exports all objects whose OIDs begin with "1.3.6.1.6.3.1.1.5":
	
	`nco_mibmanager_batch -exportoids 1.3.6.1.6.3.1.1.5.* -filetype csv`
	
	The following command exports all objects with the OIDs "1.3.6.1.6.3.1.1.5.1" or "1.3.6.1.6.3.1.1.5.2":
	
	`nco_mibmanager_batch -exportoids 1.3.6.1.6.3.1.1.5.1|1.3.6.1.6.3.1.1.5.2 -filetype csv`

`-exportscope`
 	
`ALL|TRAPS|OBJECTS`
	Use this option to specify which type of MIB objects are exported. This option takes the following values:
	• `ALL`: All MIB objects are exported.
	• `TRAPS`: Only traps are exported.
	• `OBJECTS`: Only objects are exported.
	
	For example:
	
	`nco_mibmanager_batch -exportdir "/home/user/ export_cmd" -exportnewdirectory -filetype csv -exportscope ALL`

Table 34. MIB Manager command-line options (continued)

Command-line option	Description

`-filetype`
 	
`csv|`
`csv_trap_objects|`
`html_with_frames|`
`html_without_frames|`
`lookup|ncil|nckl_1_1|`
`nckl_3_0|standalone|`
`text|`
`tivoli_universal_agent`
	Use this option to specify the file format in which the generated files are exported. This option is required for export operations.
	
	The available parameters are as follows:
	• `csv`: All object data is written to CSV files.
	• `csv_trap_objects`: All trap data and associated variable bindings object data is written to a single CSV file.
	• `html_with_frames`: Creates a web page that is suitable for publishing.
	• `html_without_frames`: Creates a single file named `oids.html` that contains a list of all the objects that were selected for output.
	• `lookup`: Outputs all specified values in a tabbed list suitable for inclusion in any rules file.
	• `ncil`: Creates separate include files for each enterprise and generates a single file for generic traps.
	• `nckl_1_1`: Creates files suitable for use with Netcool Knowledge Library V1.1.
	• `nckl_3_0`: Creates files suitable for use with Netcool Knowledge Library V3.x.
	• `standalone`: Creates a single rules file that contains all the selected traps.
	• `text`: Creates a separate text file for each OID.
	• `tivoli_universal_agent`: Creates `.mdl` and `trapcnfg` files.

`-help`	Use this option to display help information about the command-line options.
`-importdir` *string*	Use this option to specify the directory from which MIB files are imported. Enclose the directory path in double quotation marks ("). For example: `nco_mibmanager_batch -importdir "/home/user/mibs"`
`-importdirtraverse`	Use this option to specify that MIB Manager traverses subdirectories when it is searching for MIB files to import. For example: `nco_mibmanager_batch -importdir "/home/user/mibs" -importdirtraverse`

`-messagelevel`
 	
`ERROR|WARN|`
`INFO|DEBUG|VERBOSE|NONE`
	Use this option to specify the level of message logging. This option takes the following values:
	• `ERROR`: Only error messages are logged.
	• `WARN`: Warning and error messages are logged.
	• `INFO`: Information, warning, and critical messages are logged.
	• `DEBUG`: All messages are logged.
	• `VERBOSE`: Verbose messages include the state of variables, arrays, and hashes as they change value and state.
	• `NONE`: No messages are logged.

Table 34. MIB Manager command-line options (continued)

Command-line option	Description
`-messagelog` *string*	Use this option to specify the directory to which the message log file, `mibmanager.log`, is written. Enclose the directory path in double quotation marks ("). The default is `$NCHOME/omnibus/log`.
`-searchpath` *string*	Use this option to specify a semicolon delimited list of directories that MIB Manager can search to resolve MIB dependencies. Enclose the directory path in double quotation marks ("). For example: `nco_mibmanager_batch -importdir "/home/user/mibs" -searchpath "/home/user/mib_dependencies"` The following command searches 3 specific directories for dependencies: `nco_mibmanager_batch -importdir "/home/user/mibs" -searchpath "/home/user/mib_dependencies;/home/user/mib_other;/home/user/mib_test"`

About SNMP

This section provides additional information about the Simple Network Management Protocol (SNMP) architecture, the SNMP management information base (MIB), and some additional MIB concepts.

In an SNMP architecture, a manager component manages an agent. The agent is software that runs on a network device or application, responds to information requests (SETs and GETs), and then generates autonomous notifications called traps. The manager is software that receives the traps and that provides a mechanism to SET or GET SNMP objects from the network device.

To receive autonomous traps, the manager runs an application that listens on the TCP/IP SNMP trap port (port 162). SNMP SETs and GETs use port 161. This application is typically called a Trapd, or trap daemon. A trap daemon is a process that runs in the background and handles a service on a computer. The Netcool Trapd application is called the SNMP Probe (`nco_p_mttrapd`, where `mt` is an abbreviation of multithreaded) and is located in the `$OMNIHOME/probes/` directory. The rules files generated by MIB Manager are designed to be used by the SNMP Probe.

Security

Basic security in SNMP v1 and v2 is provided by using community strings. Community strings are plain text passwords that are sent with all requests. There are separate community strings for read-only access and read-write access. MIB Manager must know the community string defined on a device before it can execute any requests (read-only for a GET and read-write for a SET), and SNMP traps and notifications are sent to MIB Manager with a predefined community string.

All network devices that support SNMP have a mechanism for defining the community string. The standard default read-only password is public and the standard default read-write password is private. If no community string has been set on a device, it will usually be one of these passwords. For security reasons, the default passwords must be changed as soon as possible.

More information

The following books are useful for gaining a good understanding of the SNMP framework and MIBs:

- *Managing Internetworks with SNMP (Third Edition)* by Mark Miller (Wiley, 1999)
- *Understanding SNMP MIBs* by David Perkins and Evan McGinnis (Prentice Hall, 1996)

For a discussion of ASN.1 and the Basic Encoding Rules (BER) that are used for encoding SNMP data into Protocol Data Unit (PDU) packets for transmission on the network, see the following book:

SNMP, SNMPv2, and CMIP: The Practical Guide to Network-Management Standards by William Stallings (Addison-Wesley, 1993).

MIB concepts and design

All SNMP MIB modules that are defined for use by a specific device comprise the MIB for that device. The term MIB is often used to describe a single module definition but this is technically incorrect. In fact, the MIB is the combination of all of the modules used for managing a specific device, whether the device relates to hardware or software. Therefore, the more precise name for each module defined by a vendor, or in an RFC, is SNMP MIB module.

All MIB modules are eventually extensions of the root module. All released MIB modules, from individual vendors, extend from the enterprises object defined in RFC1155-SMI. Therefore, all SNMP agents must support RFC1155, and all MIB modules are extensions of RFC1155.

Structure of Management Information (SMI)

To make the SNMP management information base (MIB) extensible, related items are arranged into MIB modules that form a structured hierarchy. Each MIB module is defined inside the following construct:

```
ModuleName DEFINITIONS ::= BEGIN END
```

The `BEGIN` and `END` tags in the module enable several modules to be defined within a single text file. MIB compilers should be able to handle any number of modules defined in a single file, but should not require it.

There are conventions for every defined object within the module. For example, a module name must begin with an uppercase alphabetic character and contain only letters, numbers, hyphens (-), or underscores (_). An object name must start with a lowercase alphabetic character and must only contain letters, numbers, hyphens, or underscores. Comments in MIB modules are represented by two consecutive hyphens (--) and any text following this symbol, on any line, can be ignored.

The modular, easily extensible design of MIBs makes them able to support any new functionality or device by adding an additional module. When a module is

written as an extension of another module, it will include an IMPORTS section, located below the DEFINITIONS line. The IMPORTS section defines the objects required by modules higher in the MIB hierarchy and the modules in which they in turn are defined.

The following definition is from RFC1157 and indicates several objects which are imported from RFC1155. This section can be viewed as analogous to the include statement in a programming language such as C or Perl, or in a Netcool rules file. Additionally, in order to understand the objects in the current MIB module (RFC1157-SNMP) you must also be aware of the objects in the previous MIB module (RFC1155-SMI).

```
RFC1157-SNMP DEFINITIONS ::= BEGIN IMPORTS ObjectName, ObjectSyntax, NetworkAddress,
IpAddress, TimeTicks FROM RFC1155-SMI;
```

Typographical errors are easily made when specifying imported MIB names. For example, RFC1212 might be referenced as a MIB module instead of the correct name, RFC-1212. If parsing errors are highlighted by MIB Manager, you must check the IMPORTS section to confirm that the MIB modules are correctly named. Some MIB modules also contain an EXPORTS section (which also ends with a semicolon). This section informs the reader that the MIB author expects other MIB modules to use the same specified objects. For our purposes, this section is irrelevant and can be ignored.

Defined data types

SNMP MIB modules are defined in a format known as ASN.13 (Abstract Syntax Notation 1). SNMP, however, only uses a portion of ASN.14. ASN.1 is defined in ITU-T X.208 and in ISO 8824. The portions of ASN.1 that apply to SNMP are defined in RFC1155. RFC1155 defines the following valid SNMP data types:

- Primitive types: INTEGER, OCTET STRING, OBJECT IDENTIFIER, NULL
- Constructor types: SEQUENCE, SEQUENCE OF
- Defined types: NetworkAddress, IpAddress, Counter, Gauge, TimeTicks, Opaque

A defined type is the mechanism used to specify a particular format for primitive or constructor types. MIB authors can define additional types using the TEXTUAL-CONVENTION construct.

DisplayString is a good example of a defined type. In the SNMPv2-SMI-v1 MIB module, the v1 version of DisplayString has the following definition:

```
DisplayString ::= OCTET STRING (0..255)
```

In the SNMPv2-TC MIB module, the v2 version of DisplayString has the following definition:

```
DisplayString ::= TEXTUAL-CONVENTION DISPLAY-HINT "255a" STATUS current DESCRIPTION
"Represents textual information taken from the NVT ASCII character set, as defined
in pages 4, 10-11 of RFC 854. To summarize RFC 854, the NVT ASCII repertoire
specifies: - the use of character codes 0-127 (decimal) - the graphics characters
(32-126) are interpreted as US ASCII - NUL, LF, CR, BEL, BS, HT, VT and FF have the
special meanings specified in RFC 854 - the other 25 codes have no standard
interpretation - the sequence 'CR LF' means newline - the sequence 'CR NUL' means
carriage-return - an 'LF' not preceded by a 'CR' means moving to the same column on
the next line. - the sequence 'CR x' for any x other than LF or NUL is illegal.
(Note that this also means that a string may end with either 'CR LF' or 'CR NUL',
but not with CR.) Any object defined using this syntax may not exceed 255 characters
 in length." SYNTAX OCTET STRING (SIZE (0..255))
```

The example above shows that a `DisplayString` is an OCTET STRING of 0 to 255 characters in length. Note that each OBJECT DESCRIPTOR that corresponds to an object type in an internet-standard MIB must be a unique, mnemonic, printable string.

Defining objects

A common mistake made when writing MIB modules is to create an object name that is not unique. It is claimed that the RFC1155 statement means that only objects within a single MIB module must be unique. As previously discussed, the MIB is the complete set of modules which, when combined, are used to manage a particular device. Therefore, all objects defined in any MIB module must be unique, not only in its own module, but also in any other object name in any imported modules, and any modules that those modules may import. A common mechanism for ensuring that object names are unique is to pre-pend all module names with the company's ticker symbol or abbreviated company name.

When objects are defined they are mapped into a numerical hierarchy which resembles a spanning tree. Each time an object is defined, it is defined as a leaf of a parent object. The following three root objects are defined in the SNMP MIB tree:

- `ccitt` (root node zero)
- `iso` (root node 1)
- `joint-iso-ccitt` (root node 2)

All other nodes in the MIB tree are children of one of these three root nodes. For example, RFC1155-SMI defines the following objects:

```
internet OBJECT IDENTIFIER ::= { iso org(3) dod(6) 1 } directory
OBJECT IDENTIFIER ::=
 { internet 1 } mgmt OBJECT IDENTIFIER ::= { internet 2 }
experimental OBJECT IDENTIFIER
 ::= { internet 3 }
private OBJECT IDENTIFIER ::= { internet 4 } enterprises OBJECT
IDENTIFIER ::= { private 1 }
```

These definitions indicate the object name, the associated object types, each object's parent name (or ordered list of parents), and the leaf number of this child to that parent (or parents). Graphically, these items take on a hierarchical form.

You move through the MIB tree view by expanding and collapsing nodes within the MIB tree. The top level branches of the MIB tree contain the names of the MIB modules, and contained within each MIB module branch are the other elements which comprise the MIB. As additional modules are added to the MIB, additional objects are added to the MIB tree. Each object can be referred to either by its object name or by its object identifier (OID). The most accurate method is to refer to its OID. Its OID is defined as its number, and each of its ancestor's numbers continuing back to the root node, concatenated together with a period (.) separating each. The OID for the enterprises object (node or leaf) is 1.3.6.1.4.1.

Many vendors do not ensure that their object names are universally unique, therefore it is possible for two vendors to have an object sharing the same name. This makes the use of the object name to identify an object a little ambiguous.

MIB object types

This topic describes the object types defined in SNMP v1 and v2.

You can locate the object information described in the following sections by selecting a module in the MIB Modules view and then searching for `ifIndex` in the **Search** field of the OID Tree view. Click on the `ifIndex` object in the OID Tree hierarchy to see object information and textual convention information in the Details view.

OBJECT IDENTIFIER

The `OBJECT IDENTIFIER` is defined by SNMP v1 and is the main building block of the MIB tree. Object identifiers are analogous to a chapter heading in a book - they contain no real data but do give you an idea of what kind of content is relayed by their descendents.

OBJECT TYPE

The `OBJECT-TYPE` is defined by SNMP v1 and is used as a container for storing information about the managed device, or some measured value on the device.

TEXTUAL CONVENTION

The `TEXTUAL-CONVENTION` (TC) is a definition of a type of object but is not an actual object. In the MIB Modules view, you can select **Textual Conventions** from the **View** list to see the parsed textual conventions displayed in the MIB tree. Select a TC name in the MIB tree to display its definition in the Details view.

SNMP v1 TRAP TYPE and SNMP v2 NOTIFICATION TYPE

The SNMP v1 TRAP-TYPE and v2 NOTIFICATION-TYPE are the SNMP mechanism for generating autonomous events to the SNMP manager. SNMP traps in v1 are not defined as objects within the MIB tree. A TRAP-TYPE object does not have a defined parent in the OBJECT IDENTIFIER format. Instead, a trap definition specifies an enterprise for which a trap is defined. The following is a typical TRAP-TYPE object:

```
bgpEstablished TRAP-TYPE ENTERPRISE bgp VARIABLES
{ bgpPeerRemoteAddr, bgpPeerLastError,
 bgpPeerState }
DESCRIPTION "The BGP Established event is generated when the BGP FSM
enters the ESTABLISHED state." ::= 1
```

The `ENTERPRISE` section defines which object is the parent of the trap. However, it is possible for a MIB tree object to be defined with `bgp` as the parent and it is defined as child number 1. In fact, `bgpVersion` is defined as `{ bgp 1}` in the RFC1269-MIB module. Therefore, it is impossible to add a v1 trap to the MIB tree as a leaf using the `ENTERPRISE` as the parent.

SNMP v2 changes the definition for TRAP-TYPE to NOTIFICATION-TYPE and specifies that this new v2 trap be defined like other MIB objects, with a parent and child number making this only a problem for v1 traps. RFC1155 Section 4.1 defines that using zero (0) as a child number is invalid, and reserved for future use. SNMP v2 makes use of that zero by allowing vendors to add their v1 traps to a v2 MIB, by adding a zero to the enterprise name and then adding the trap number after the zero. Therefore, under v2 it is appropriate to define an object identifier with a zero as a child of the enterprise and then add the v1 traps as children of that zero.

This convention has caused another common mistake made by MIB authors. Section 4 of RFC1155 states the following:

"An object type definition consists of five fields: OBJECT: ------- A textual name, termed the OBJECT DESCRIPTOR, for the object type, along with its corresponding OBJECT IDENTIFIER. Syntax: The abstract syntax for the object type. This must resolve to an instance of the ASN.1 type ObjectSyntax (defined below). Definition: A textual description of the semantics of the object type. Implementations should ensure that their instance of the object fulfills this definition since this MIB is intended for use in multi-vendor environments. As such it is vital that objects have consistent meaning across all machines. Access: One of read-only, read-write, write-only, or not-accessible. Status: One of mandatory, optional, or obsolete. Future memos may also specify other fields for the objects which they define."

According to this rule, all objects must have both an object name and an object number. Some vendor's MIB modules, and even some RFCs, defined a NOTIFICATION-TYPE with a parent of zero but without an object name for that zero. In the following example, the object definition is not actually syntactically correct as there is no object name defined for child number zero of the `adslAtucTraps` object. MIB Manager recognizes the preference of some MIB authors to use such methods as a shortcut, and allow the object to be added without an object name. Additionally, to facilitate adding v1 traps to the MIB tree, MIB Manager automatically adds an object zero as a child of the v1 enterprise object (note that a v1 MIB cannot use a zero in its OID), assign that object zero as Traps where is the enterprise name and add the trap below this new object in the MIB tree. For example, using `bgp` would result in the following traps ancestors: `{ bgp bgpTraps(0) 1 }).`

```
adslAtucPerfLofsThreshTrap NOTIFICATION-TYPE OBJECTS { adslAtucPerfCurr15MinLofs,
adslAtucThresh15MinLofs } STATUS current DESCRIPTION "Loss of Framing 15-minute
interval threshold reached." ::= { adslAtucTraps 0 1 }
```

Varbinds

Objects that are transmitted with the v1 trap or v2 notification are known as varbinds. Varbinds contain additional information about the reported event. In a v1 trap, the varbinds are itemized in the VARIABLES section and in a v2 notification the varbinds are listed in the OBJECTS section. They have the same use in all versions of SNMP. The order in which the varbinds appear in the list is important because the PDU (SNMP Packet) encodes the associated values in the same order in which they are listed in the MIB.

For example, in the OBJECTS section the following three varbinds have been specified: `ifIndex`, `ifAdminStatus`, and `ifOperStatus`. Therefore, `ifIndex` is the first varbind to be encoded, `ifAdminStatus` is the second, and `ifOperStatus` is encoded third. Checking the IF-MIB we find that the `ifIndex` object type is defined as `InterfaceIndex`. Since this is not a valid primitive ASN.1 type for SNMP, it must be a textual convention. Searching through the textual conventions, we find that `InterfaceIndex` actually resolves to an Integer32 (32 bit integer). Therefore, when the PDU arrives at MIB Manager, the first varbind will be an integer. To determine what that integer means, MIB Manager must reference the IF-MIB module, look up `ifIndex`, and read the associated object information. Checking the second varbind, we find an enumerated integer type:

```
SYNTAX INTEGER { up(1), -- ready to pass packets down(2),
testing(3) -- in some test mode }
```

When the varbind is decoded from the SNMP packet, its value will be an integer, the value of which must be interpreted based on the items in this enumerated list. When MIB Manager is used to create a rules file, it will create a lookup table to automatically link the enumerated integer with its textual representation. The third varbind is also an enumerated type with the same values. Therefore, if the `ifAdminStatus` is 1 (up) and the `ifOperStatus` is 2 (down), we know why the event was generated and can proceed to attempt to determine the cause of this outage.

Varbinds are presented to the user in a rulesfile as `$1`, `$2`, `$3`, and so on, with each number representing a varbind number. MIB Manager creates elements based on the varbind elements and uses these to set variables in the details table. For example, the elements used in the details table might be $ifIndex = $1, which will be an integer, $ifAdminStatus = $2, which will be something like up (1), and $ifOperStatus = $3, which will be something like down (3). Any changes made to the object settings are automatically set in the rules file, using the conventions set by the Netcool Knowledge Library (NCKL).

Tables

Tables represent the equivalent of a multidimensional array with rows and columns of data. The table object is defined as a SEQUENCE OF an Entry object. The Entry object is then defined as a SEQUENCE of OBJECT-TYPE objects. Occasionally, a vendor designs an unusual system, for example the Cisco 10k router. This device maintains an internal table of alarm conditions and generates a trap or notification when the table changes. You must then issue an SNMP GET request on the contents of the table to determine the current status of the active alarms on the device. This makes obtaining the alarms by the SNMP manager a bit more difficult, but not impossible if the administrator has the tools to comply.

OCTET STRING

An octet is a data construct consisting of eight bits (commonly known as a byte). An OCTET STRING then, is an array of bytes (or a string of bytes). The term OCTET STRING does not imply that all of the bytes in the string are alphanumeric. They can also be binary characters and are used as bitmasks.

Valid MIB object formats

This topic describes the formats of valid MIB objects.

The following sections describe the valid SNMP MIB object formats.

TEXTUAL-CONVENTION

A v2 TEXTUAL-CONVENTION (TC) has the following syntax, where the object name is followed by the `::=` entry and then `TEXTUAL-CONVENTION`. Several sections follow and are appended with the SYNTAX definition.

```
DisplayString ::= TEXTUAL-CONVENTION
DISPLAY-HINT "255a"
STATUS current
DESCRIPTION "Represents textual information taken from the NVT SCII character set,
 as defined in pages 4, 10-11 of RFC 854. To summarize RFC 854,
the NVT ASCII repertoire specifies: - the use of character codes 0-127 (decimal) -
the graphics characters (32-126) are interpreted as US ASCII - NUL, LF, CR,BEL, BS,
HT, VT and FF have the special meanings specified in RFC 854 -
the other 25 codes have no standard interpretation -
the sequence 'CR LF' means newline - the sequence 'CR NUL' means carriage-return -
an 'LF' not preceded by a 'CR' means moving to the same column on the next line. -
```

```
the sequence 'CR x' for any x other than LF or NUL is illegal.
(Note that this also means that a string may end with either 'CR LF' or 'CR NUL',
but notwith CR.)
Any object defined using this syntax may not exceed 255 characters in length."
SYNTAX OCTET STRING (SIZE (0..255))
```

A v1 TC consists of the object name followed by the `::=` entry, and is then appended with a valid SYNTAX definition. For example:

```
DisplayString ::= OCTET STRING
```

Both these objects exist outside the MIB tree and therefore are not objects in the MIB. They represent a format for which a syntax can be defined. They have no OID. You can view them in the MIB Modules view, in a separate filter, by clicking the **View** drop-down list, and selecting **Textual-Conventions**.

TRAP-TYPE objects

The TRAP-TYPE object is valid for v1 MIBs. These objects were not originally defined to fit cleanly into the MIB tree. They do not have an OID but instead have an enterprise ID and a trap number, for example:

```
newRoot TRAP-TYPE
ENTERPRISE dot1dBridge
DESCRIPTION "The newRoot trap indicates that the sending agent has become the new
root
of the Spanning Tree; the trap is sent by a bridge soon after its election as
the new
root, e.g., upon expiration of the Topology Change Timer immediately subsequent
to its
election."
::= 1
```

A v1 trap begins with an object name followed by the keyword `TRAP-TYPE`. This is followed by a number of sections and ends with the `::=` entry and a number. Curly brackets ({}) are never used before, or after, a number.

MACRO objects

A MACRO object defines the format of other MIB objects. MACRO definitions always begin with the object type, followed by the `MACRO` keyword, and then the `::=` entry. The remainder of the macro definition is enclosed in `BEGIN` and `END` tags, for example:

```
OBJECT-TYPE MACRO ::=
BEGIN
TYPE NOTATION ::= "SYNTAX"
type (TYPE ObjectSyntax) "ACCESS" Access "STATUS" Status VALUE NOTATION ::= value
(VALUE ObjectName) Access ::= "read-only" | "read-write" | "write-only" |
"not-accessible"
Status ::= "mandatory" | "optional" | "obsolete"
END
```

Other objects

All other objects must adhere to the following format:

```
snmpInPkts OBJECT-TYPE SYNTAX Counter ACCESS read-only STATUS mandatory DESCRIPTION
 "The total number of Messages delivered to the SNMP entity from the transport
service." ::= { snmp 1 }
```

These objects begin with an object name, which must begin with a lowercase alphabetic character. This is followed by a keyword that indicates the object type.

Any additional sections follow the keyword, and the `::=` entry, and a list of ancestors in curly brackets ({}) completes the format. The ancestors inside the curly brackets ({}) can have one of two formats. In the first (shown above) the format is an object name followed by a number. The object name is the name of this object's immediate parent and the number is the leaf number of this object to the parent.

The second valid format for an ancestor list is a list of all ancestors back to a known object, for example:

```
internet OBJECT IDENTIFIER ::= { iso org(3) dod(6) 1 }
```

In this example, the list starts at a known object (`iso` being the root of the tree) and continues to define object names and leaf numbers for each successive generation, `org(3)` and `dod(6)`, until the final single integer (`1`) which indicates the object number. Note that spaces between the object names and the object numbers are not allowed.

Chapter 8. About gateways

Tivoli Netcool/OMNIbus gateways enable you to exchange alerts between ObjectServers and complementary third-party applications, such as databases and helpdesk or Customer Relationship Management (CRM) systems.

You can use gateways to replicate alerts or to maintain a backup ObjectServer. Application gateways enable you to integrate different business functions. For example, you can configure a gateway to send alert information to a helpdesk system. You can also use a gateway to archive alerts to a database.

The following figure shows an example gateway architecture.

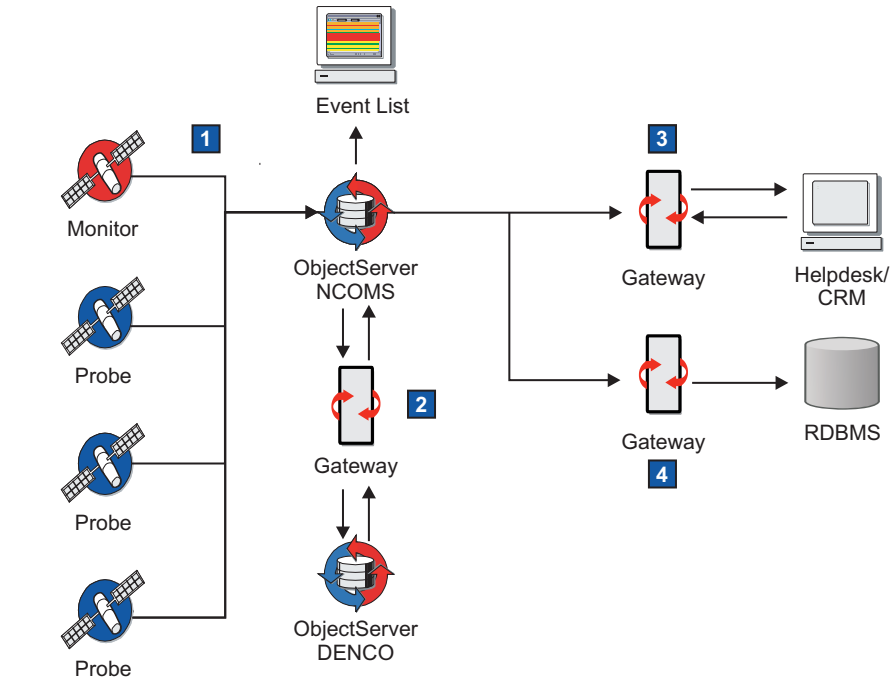

Figure 5. Gateways in the Tivoli Netcool/OMNIbus architecture

The preceding figure illustrates how to use gateways for a variety of purposes:

1. Probes send alerts to the local ObjectServer.
2. The ObjectServer Gateway replicates alerts between ObjectServers in a failover configuration.
3. The Helpdesk gateway integrates the Network Operations Center (NOC) and the helpdesk by converting trouble tickets to alerts, and alerts to trouble tickets.
4. The RDBMS gateway stores critical alerts in a relational database management system (RDBMS) so that you can analyze network performance.

After a gateway is correctly installed and configured, the transfer of alerts is transparent to operators. For example, alerts are forwarded from an ObjectServer to a database automatically without user intervention.

Note: The information in this publication is generic to all gateways. For gateway-specific information, see the individual gateway publications in the IBM Tivoli Network Management Information Center at:

http://publib.boulder.ibm.com/infocenter/tivihelp/v8r1/index.jsp

Related concepts:

"Types of gateways"
There are two main types of gateways: unidirectional gateways and bidirectional gateways.

Types of gateways

There are two main types of gateways: unidirectional gateways and bidirectional gateways.

Unidirectional gateways allow alerts to flow in only one direction. Changes made in the source ObjectServer are replicated in the destination ObjectServer or application, but changes made in the destination ObjectServer or application are not replicated in the source ObjectServer. Unidirectional gateways can be considered as *archiving* tools.

Bidirectional gateways allow alerts to flow from the source ObjectServer to the target ObjectServer or application, and also allow feedback to the source ObjectServer. In a bidirectional gateway configuration, changes made to the contents of a source ObjectServer are replicated in a destination ObjectServer or application, and the destination ObjectServer or application replicates its alerts in the source ObjectServer. Bidirectional gateways can be considered as *synchronization* tools.

Gateways can send alerts to a variety of targets:

- Another ObjectServer
- A database
- A helpdesk application
- Other applications or devices

ObjectServer gateways are used to exchange alerts between ObjectServers. This is useful when you want to create a distributed installation, or when you want to install a backup ObjectServer.

Database gateways are used to store alerts from an ObjectServer. This is useful when you want to keep a historical record of the alerts forwarded to the ObjectServer.

Helpdesk gateways are used to integrate Tivoli Netcool/OMNIbus with a range of helpdesk systems. This is useful when you want to correlate the trouble tickets raised by your customers with the networks and systems you are using to provide their services.

Other gateways are specialized applications that forward ObjectServer alerts to other applications or devices (for example, a flat file or socket).

Note: Only gateways that send alerts to certain targets can be bidirectional.

Related concepts:

Chapter 8, "About gateways," on page 159
Tivoli Netcool/OMNIbus gateways enable you to exchange alerts between ObjectServers and complementary third-party applications, such as databases and helpdesk or Customer Relationship Management (CRM) systems.

Unidirectional ObjectServer gateways

A unidirectional ObjectServer gateway allows alerts to flow from a source ObjectServer to a destination ObjectServer. Changes made in the source ObjectServer are replicated in the destination ObjectServer, but changes made in the destination ObjectServer are not replicated in the source ObjectServer.

For more information about unidirectional ObjectServer Gateways, see the *IBM Tivoli Netcool/OMNIbus ObjectServer Gateway Reference Guide*.

Related concepts:

"Bidirectional ObjectServer gateways"
A bidirectional ObjectServer gateway allows alerts to flow from a source ObjectServer to a destination ObjectServer. Changes made to the contents of a source ObjectServer are replicated in a destination ObjectServer, and the destination ObjectServer replicates its alerts in the source ObjectServer. This enables you, for example, to maintain a system with two ObjectServers configured as a failover pair.

Bidirectional ObjectServer gateways

A bidirectional ObjectServer gateway allows alerts to flow from a source ObjectServer to a destination ObjectServer. Changes made to the contents of a source ObjectServer are replicated in a destination ObjectServer, and the destination ObjectServer replicates its alerts in the source ObjectServer. This enables you, for example, to maintain a system with two ObjectServers configured as a failover pair.

For more information about bidirectional ObjectServer Gateways, see the *IBM Tivoli Netcool/OMNIbus ObjectServer Gateway Reference Guide*.

Related concepts:

"Unidirectional ObjectServer gateways"
A unidirectional ObjectServer gateway allows alerts to flow from a source ObjectServer to a destination ObjectServer. Changes made in the source ObjectServer are replicated in the destination ObjectServer, but changes made in the destination ObjectServer are not replicated in the source ObjectServer.

Database, helpdesk, and other gateways

Most database, helpdesk, and other gateways use a standard architecture, but each gateway has its own binary file, with additional modules to handle the communication with the target applications, devices, or files.

For information about specific gateways and their architectures, see the individual gateway publications.

Gateway components

Gateways have *reader* and *writer* components. Readers extract alerts from the ObjectServer. Writers forward alerts to another ObjectServer or to other applications.

There is only one type of reader, but there are various types of writers depending on the destination application.

Routes specify the destination to which a reader forwards alerts. One reader can have multiple routes to different writers, and one writer can have multiple routes from different readers.

Gateway *filters* and *mappings* configure alert flow. Filters define the types of alerts that can be passed through a gateway. Mappings define the format of these alerts.

Readers, writers, routes, filters, and mappings are defined in the gateway configuration file.

Unidirectional gateways

A unidirectional database, helpdesk, or other gateway allows alerts to flow from a source ObjectServer to a destination application. Changes made in the source ObjectServer are replicated in the destination application, but changes made in the destination application are not replicated in the source ObjectServer.

A simple example of a unidirectional gateway is the Flat File Gateway, which reads alerts from an ObjectServer and writes them to a flat file. This example architecture is shown in the following figure.

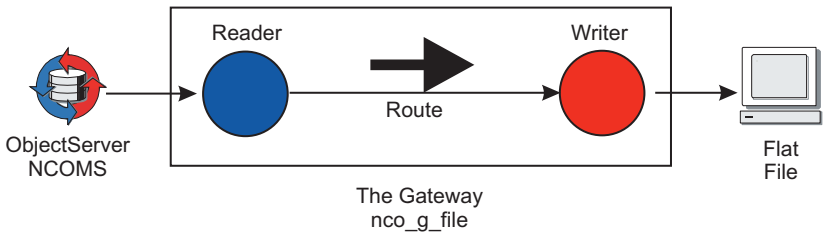

Figure 6. Example Flat File Gateway architecture

Bidirectional gateways

In a bidirectional database, helpdesk, or other gateway configuration, changes made to the alerts in a source ObjectServer are replicated in a destination application, and the destination application replicates changes to its alerts in the source ObjectServer.

This enables you, for example, to raise trouble tickets in a helpdesk system for certain alerts. Changes made to the tickets in the helpdesk system can then be sent back to the ObjectServer.

Bidirectional gateways have a similar configuration to unidirectional gateways, with an additional `COUNTERPART` attribute for the writers. The `COUNTERPART` attribute defines a link between a gateway writer and reader.

The following figure shows an example bidirectional gateway configuration.

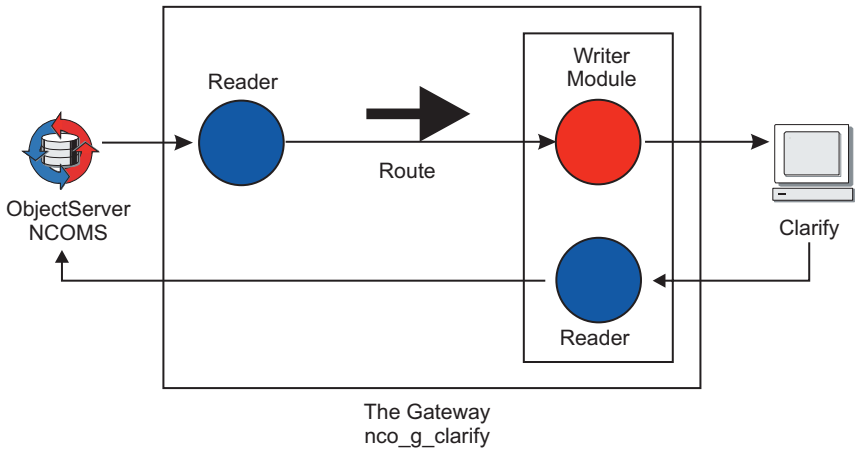

Figure 7. Bidirectional Clarify Gateway

Reader component

A reader extracts alerts from an ObjectServer. There is only one type of reader: the ObjectServer reader.

When the reader starts, the gateway attempts to open a connection to the source ObjectServer. If the gateway succeeds in opening the connection, it immediately starts to read alerts from the ObjectServer.

Writer modules

Writer modules manage communications between gateways and third-party applications, and format the alert correctly for entry into the application.

The writer module generates log files that can help debug the gateway.

Communication between the writer module and the third-party application uses helper applications, which interact directly with the application through its APIs or other interfaces. These processes are transparent to the user (though they are visible using the **ps** command or similar utility).

The writer module uses a reference number cache to track the alerts and their associated reference number in the target application. For each alert, the cache stores the following:

- The serial number of the alert
- A reference number from the target application, such as a helpdesk ticket number

When a ticket is raised in response to an alert, the writer module enters the reference number in the cache and returns it to the ObjectServer where the alert is updated to include the reference number.

The following figure shows a simplified example of the writer module architecture.

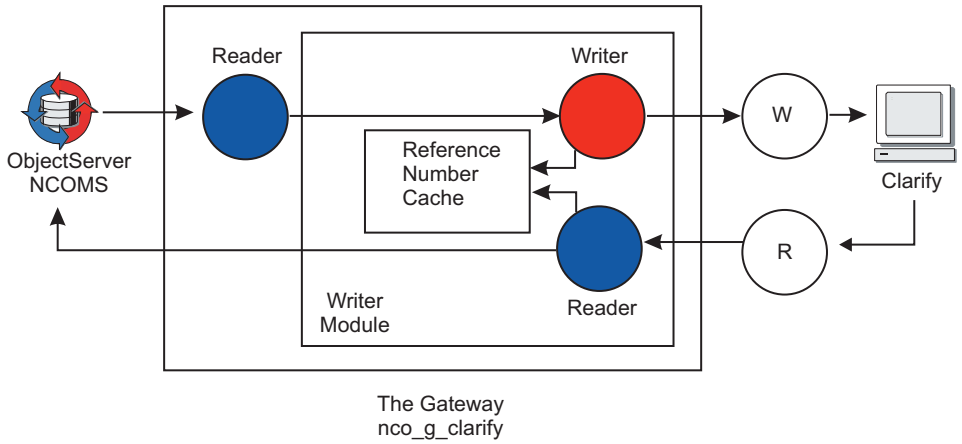

Figure 8. Reader/writer module architecture

Routes

Routes create the link between the source reader and the destination writer.

Any alerts received by the source ObjectServer are read by the reader, passed through the route to the writer, and written into the destination ObjectServer or application.

Alert updates from the helpdesk

When a helpdesk operator makes additional changes to a ticket, these are forwarded to the gateway which runs the corresponding action `.sql` file to update the alert in the ObjectServer.

Typically the following action `.sql` files are provided:

- `open.sql`
- `update.sql`
- `journal.sql`
- `close.sql`

For detailed information on configuring alert updates from the helpdesk, see the individual gateway publications.

Store-and-forward mode for gateways

If there is a problem with the gateway target, the ObjectServer and database writers can continue to run using store-and-forward mode.

When the writer detects that the target ObjectServer or database is not present or is not functioning (usually because the writer is unable to write an alert), it switches into *store* mode. In this mode, the writer stores everything it would normally send to the database in a file named:

`$OMNIHOME/var/`*`writername`*`.`*`destserver`*`.store`

In this file name, *writername* is the name of the writer and *destserver* is the name of the server to which the gateway is attempting to send alerts.

When the gateway detects that the destination server is back on line, it switches into *forward* mode and sends the alert information held in the `.store` file to the destination server. After all of the alerts in the `.store` file have been forwarded, the writer returns to normal operation.

Store-and-forward mode only works when a connection to the ObjectServer or database destination has been established, used, and then lost. If the destination server is not running when the gateway starts, store-and-forward mode is not triggered and the gateway terminates.

If the gateway connects to the destination ObjectServer and a store-and-forward file already exists, the gateway replays the contents of the store-and-forward file before it sends new alerts.

Store-and-forward mode is configured using the attributes `STORE_AND_FORWARD` and `STORE_FILE`.

Note: See the individual gateway publications to determine whether an individual gateway supports store-and-forward mode. Store and forward does not work with bidirectional gateway configurations, with the exception of the bidirectional ObjectServer Gateway.

Secure mode for gateways

When you start the ObjectServer in secure mode, by using the `-secure` command-line option, it authenticates probe, gateway, and proxy server connections by requiring a user name and password.

When a connection request is sent, the ObjectServer issues an authentication message. The probe, gateway, or proxy server must respond with the correct user name and password. If the user name and password combination is incorrect, the ObjectServer issues an error message and rejects the connection.

If the ObjectServer is not running in secure mode, probe, gateway, and proxy server connection requests are not authenticated.

Secure authentication is configured differently for gateways that use single configuration files (`.conf` files) and for gateways that use properties files (`.props` files). See the individual gateway reference guides for gateway-specific configuration information.

You can use the **`nco_g_crypt`** utility to encrypt passwords for use in configuration and properties files. If you are running Tivoli Netcool/OMNIbus in FIPS 140-2 mode, use the **`nco_aes_crypt`** utility to encrypt passwords. Encrypted passwords are decoded by the gateway before they are used to log in to the target system. See the individual gateway reference guides for gateway-specific information about encrypting passwords.

For more information about running the ObjectServer and ObjectServer Gateway in secure mode and FIPS 140–2 mode, see the *IBM Tivoli Netcool/OMNIbus Installation and Deployment Guide* and the *IBM Tivoli Netcool/OMNIbus ObjectServer Gateway Reference Guide*.

Gateway writers and failback

Failover occurs when a gateway loses its connection to the primary ObjectServer and connects to a backup ObjectServer. Failback functionality allows the gateway to reconnect to the primary ObjectServer when it becomes active again.

The ObjectServer reader can fail over and fail back between source ObjectServers without shutting down. This ability is not supported by all gateway writers.

If a writer does not support this mode of failback and failover, the writer, on detection of the reader failover or failback, will shut down the gateway and rely on the process agent to restart the gateway.

The following writers support reader failover and failback without shutting:

- ObjectServer writer
- Sybase database writer
- Sybase Reporter writer
- SNMP writer
- Socket writer
- Flat file writer
- Informix® database writer
- Message Bus (XML) writer
- JDBC writer
- Tivoli EIF writer
- TSRM writer
- JDBC database writer

The following writers support failover and failback with shutdown:

- Remedy ARS writer
- Siebel eCommunications writer
- Oracle database writer
- Oracle Reporter writer
- Peregrine writer
- Clarify writer
- HP Service Desk writer
- ODBC database writer
- JDBC database writer

Because bidirectional ObjectServer gateways are used to resynchronize failover pairs, failback is automatically disabled. This is because one half of the gateway can legitimately be connected to a backup server and so should not be forced to keep failing back to the primary ObjectServer. However, if a bidirectional gateway is being used to share data between two separate sites, and each site has a failover pair operating, you can manually enable failback on each server. When enabled, the writer automatically enables failback on its counterpart reader.

Chapter 9. Configuring gateways

Configuration files define the environment in which gateways operate and how they map data between ObjectServer tables and target databases or applications. Most gateways use multiple configuration files. Some gateways are configured using a single configuration file.

Most gateways, including ObjectServer gateways, are configured using a properties file, a map definition file, a startup command file, and a table replication definition file.*gateway_name*.props), in conjunction with a map definition file (*gateway_name*.map), a startup command file (*gateway_name*.startup.cmd), and a table replication definition file (*gateway_name*.tblrep.def).

The following gateways are configured using a single configuration file with a .conf extension:

- Gateway for HP ServiceCenter
- Gateway for Remedy ARS
- Gateway for Siebel

Using multiple configuration files

Most gateways are configured using a properties file, a map definition file, a startup command file, and a table replication definition file.

Before running a gateway, you must edit the various configuration files to suit your Tivoli Netcool/OMNIbus environment.

The properties file is a text file that contains a set of properties and their corresponding values. These properties define the operational environment of the gateway, such as connection details and the location of the other configuration files.

On startup, the gateway looks for the default properties file $OMNIHOME/etc/NCO_GATE.props. See the individual gateway reference documentation for gateway-specific information about configuring the properties file.

The map definition file and table replication definition file define how the gateway transfers data between ObjectServers and target databases or applications. The startup command file allows you to configure data operations to occur automatically when the gateway starts.

Related reference:

"Common gateway properties and command-line options" on page 175
Gateways have a number of common properties and associated command-line options. Properties define settings for generic functions, such as message logging, for inter-process communication (IPC), and for common gateway settings, such as specifying map definition files. You can override the default property values by editing the properties file or by using the command-line options when you start the gateway.

Map definition file

The map definition file defines how the gateway maps alert fields (columns) in an ObjectServer table to fields in a target database or application. The map definition file is referenced by the table replication definition file, by the startup command file, and by the SQL interactive interface.

The default map definition file is $OMNIHOME/gates/*gateway_name*/*gateway_name*.map.

For example, the default Gateway for JDBC map definition file is $OMNIHOME/gates/jdbc/jdbc.map.

The name and location of the file is specified in the gateway properties file by the **Gate.MapFile** property.

The map definition file can contain one or more CREATE MAPPING commands that use the following syntax. Each field mapping is separated by a comma (,).

```
CREATE MAPPING map_name
(
'destination_field' = {'@source_field'|simple_expression|attribute}
 [ON INSERT ONLY]
 [CONVERT TO {INTEGER|STRING|DATE}]
 [NOT NULL '@source_field'],
 ...
);
```

The CREATE MAPPING command is structured as follows:

- *map_name* is the name of the map definition.
- *destination_field* is the name of the destination field in the target database or application.
- *source_field* is the name of an alert field (column) in the ObjectServer table.
- *simple_expression* is a string, an integer, a Boolean data type, or a sequence of strings or integers joined by one or more of the following operators: +, -, *, /. All operators concatenate strings. All operators work from left to right when operating on integers.
- *attribute* is a mapping attribute.

The following table lists the optional CREATE MAPPING command clauses.

Table 35. CREATE MAPPING command clauses

Clause	Description
ON INSERT ONLY	This clause controls how the field is updated during the lifetime of an alert. If you omit this clause, the field is updated whenever the state of an alert changes. If you include this clause, the field is created only once for an alert and is never updated.
CONVERT TO {INTEGER|STRING|DATE}	This clause defines a forced conversion if the data in a source field does not match the data type of the destination field. The available parameters are INTEGER, STRING, or DATE.

Table 35. CREATE MAPPING command clauses (continued)

Clause	Description
NOT NULL '@*source_field*'	This clause provides an alterative *source_field* value to be used when the original source value is zero or is an empty string. The alterative value can be a field (column) or a constant but it cannot be an expression.

Conversion functions

You can use conversions functions in your map definition, to convert source data types to string, integer, or datetime data types.

The conversion functions use the following syntax:

- TO_STRING('@*source_field*')
- TO_INTEGER('@*source_field*')
- TO_TIME('@*source_field*')

Map attributes

You can use mapping attribute names to include additional data in mapping definitions. You can specify two types of attribute: cache value access attributes or dynamic attributes.

The gateway uses cache value attributes to access values that are stored in the cross-reference cache. The following table describes the cache value attributes that can be used in map definitions.

Table 36. Cache value access attributes

Attribute name	Description
STATUS.SERIAL	Cached serial number for the status table row that is associated with the current journal or details table row.
STATUS.SERVER_SERIAL	Cached server serial number for the status table row that is associated with the current journal or details table row.
STATUS.SERVER_NAME	Cached server name for the status table row that is associated with the current journal or details table row.
STATUS.IDENTIFIER	Cached identifier for the status table row that is associated with the current journal or details table row.
JOURNAL.SERIAL	Cached serial number of the journal table row.
DETAILS.IDENTIFIER	Cached identifier of the details table row.

Dynamic attributes enable the gateway to access dynamic values that are automatically generated by the gateway. The following table describes the dynamic attributes that can be used in map definitions.

Table 37. Dynamic attributes

Attribute name	Description
ACTION_CODE	This attribute displays a single character string that specifies the type of operation performed. Valid values are: • I: Insert • U: Update • D: Delete
ACTION_TIME	This attribute displays the time in UTC at which the action occurred.
DELETEDAT	This attribute displays the date on which the row was deleted, if applicable.

Example map file

The following is an abridged example of a map definition file that maps data between two ObjectServers:

```
CREATE MAPPING StatusMap
(
'Identifier' = '@Identifier' ON INSERT ONLY'
'Node' = '@Node' ON INSERT ONLY,
'NodeAlias' = '@NodeAlias' ON INSERT ONLY NOT NULL '@Node',
'Manager' = '@Manager' ON INSERT ONLY,
'Agent' = '@Agent' ON INSERT ONLY,
'AlertGroup' = '@AlertGroup' ON INSERT ONLY,
'AlertKey' = '@AlertKey' ON INSERT ONLY,
'Severity' = '@Severity',
'Summary' = '@Summary',
'StateChange' = '@StateChange'
);

CREATE MAPPING JournalMap
(
'KeyField' = TO_STRING(STATUS.SERIAL) + ":" +
TO_STRING('@UID') + ":" +
TO_STRING('@Chrono')ON INSERT ONLY,
'Serial' = STATUS.SERIAL,
'Chrono' = '@Chrono',
'UID' = TO_INTEGER('@UID'),
'Text1' = '@Text1',
'Text2' = '@Text2'
);

CREATE MAPPING DetailsMap
(
'KeyField' = '@Identifier' + '####' +
TO_STRING('@Sequence') ON INSERT ONLY,
'Identifier' = '@Identifier',
'AttrVal' = '@AttrVal',
'Sequence' = '@Sequence',
'Name' = '@Name',
'Detail' = '@Detail'
);
```

Table replication definition file

The table replication definition file defines how data is replicated between an ObjectServer and a target database or application.

The default table replication definition file is $OMNIHOME/gates/*gateway_name*/*gateway_name*.tblrep.def. The name and location of the file is specified in the gateway properties file. The name of the property differs between gateways.

For example, the Gateway for JDBC table replication definition file is specified by the **Gate.RdrWtr.TblReplicateDefFile** property. The default file is $OMNIHOME/gates/jdbc/jdbc.rdrwtr.tblrep.def.

The table replication definition file can contain one or more REPLICATE commands that use the following syntax:

```
REPLICATE {ALL|INSERTS, UPDATES, DELETES, FT_INSERTS, FT_UPDATES, FT_DELETES}
 FROM TABLE 'source_table'
 USING MAP 'map_name'
 [FILTER WITH 'filter']
 [INTO 'target_table']
 [ORDER BY 'order_string']
 [WITH NORESYNC]
 [RESYNC DELETES FILTER 'filter']
 [SET UPDTOINS CHECK TO {ENABLED|DISABLED|FORCED}]
 [AFTER IDUC DO 'update_command']
 [CACHE FILTER 'condition'];
```

Use the optional clauses in the order in which they are listed in the syntax description. For example, when you use both the FILTER WITH and AFTER IDUC DO clauses, the FILTER WITH clause must precede the AFTER IDUC DO clause. If you use the CACHE FILTER clause, it must be the last clause in the REPLICATE command.

The available clauses for the REPLICATE command are described in the following table.

Table 38. REPLICATE command clauses

Clause	Description
ALL	Use this clause to specify that all inserts, updates, and deletes on the source table are replicated to the target table. You can restrict replication by using one or more of the INSERTS, UPDATES, DELETES, FT_INSERTS, FT_UPDATES, and FT_DELETES clauses instead of the ALL clause. For example, the following command replicates inserts and updates but it does not replicate deletes. By not replicating deletes, you can reduce the load on memory resources and on the target database. REPLICATE INSERTS, UPDATES
FROM TABLE '*source_table*'	Use this clause to specify the data source, where *source_table* is the table to be replicated into the target database.
USING MAP '*map_name*'	Use this clause to specify the mapping to be used, where *map_name* is the map definition that defines the source table.

Table 38. REPLICATE command clauses (continued)

Clause	Description
`FILTER WITH 'filter'`	Optional. Use this clause to filter the database rows that are selected for replication, where *filter* defines the filter that the gateway uses. **Tip:** To replicate events that are not equal to a particular value, place an exclamation mark (!) before the equals sign (=) in the filter clause. For example, the following filter clause replicates all events whose severity is not 5: `FILTER WITH 'Severity !=5'`
`INTO 'target_table'`	Optional. Use this clause to specify the table in the target database to which data is replicated, where *target_table* is the name of the target table. If you omit this clause, the `REPLICATE` command uses the name of the source table as the target table value. Therefore, this option is not required when you are replicating the main `alerts.staus`, `alerts.details`, and `alerts.journal` ObjectServer tables. It is intended to be used when you are replicating other ObjectServer tables to the target database, but this type of data transfer is normally done by using the `TRANSFER` command.
`ORDER BY 'order_string'`	Optional. Use this clause to specify the order in which rows are returned to the gateway from the ObjectServer, where *order_string* is a comma-separated list of column names. For example: `ORDER BY 'Serial DESC, StateChange ACS'` You can use the optional `ASC` and `DESC` clauses to specify that the rows are sorted in ascending (`ASC`) or descending (`DESC`) order. If you do not specify a sort order, the rows are sorted in ascending order.
`WITH NORESYNC`	Optional. Use this clause to specify the tables that you do not want to resynchronize.
`RESYNC DELETES FILTER 'filter'`	Optional. Use this clause to specify which replicated rows to remove before insertion into the target table, where *filter* defines the resynchronization deletion filter that. Use this filter when the rows in the target and source tables are not an exact match.

Table 38. REPLICATE command clauses (continued)

Clause	Description
`SET UPDTOINS CHECK TO {ENABLED|DISABLED|FORCED}`	Optional. Use this clause to configure the update-to-insert behavior. It has the following parameters: • `ENABLED`: This parameter is the default setting. The gateway performs normal update-to-insert conversions. If an update from the source table contains a row that does not exist in the target table, the update is converted to an insert and the row is written to the target table. • `DISABLED`: For each update received from the source table, the gateway sends an update to the target table. If the update contains rows that do not exist in the target table, those rows are dropped. • `FORCED`: The gateway converts all updates from the source table to an insert on the target table. If the row exists in the target table, the row is deduplicated. This behavior is identical to how probes operate.
`AFTER IDUC DO 'update_command'`	Optional. Use this clause to modify rows in the source ObjectServer immediately after they have been fetched by the gateway, where *update_command* is a comma-separated list of column assignments. The update is applied to all rows that are inserted, updated, or deleted.
`CACHE FILTER 'condition'`	Optional. Use this clause to filter the cache entries that are retrieved during a cache refresh, where *condition* defines the SQL condition that the gateway adds to the `SELECT` statement that it uses to retrieve the cache entries. Use this clause to reduce the amount of data that the gateway retrieves during a unidirectional gateway cache refresh.

The following is an example of a table replication definition file:

```
REPLICATE INSERTS, DELETES FROM TABLE 'alerts.status'
 USING MAP 'StatusMap'
 ORDER BY 'Serial' ASC
 FILTER WITH 'Severity !=5'
 SET UPDTOINS CHECK TO FORCED
 AFTER IDUC DO 'Location=\'PASSED BY GW\''
 CACHE FILTER 'ServerName IN (\'NCOMBS_P\',\'NCOMBS_B\')';

REPLICATE ALL FROM TABLE 'alerts.journal'
 USING MAP 'JournalMap';

REPLICATE ALL FROM TABLE 'alerts.details'
 USING MAP 'DetailsMap';
```

Startup command file

The startup command file contains a set of commands that the gateway runs automatically each time it starts. You can use these commands to transfer data from an ObjectServer table to a target database.

The default startup command file is `$OMNIHOME/gates/gateway_name/gateway_name.startup.cmd`. For example, the default Gateway for JDBC startup command file is `$OMNIHOME/gates/jdbc/jdbc.startup.cmd`.

The name and location of the file is specified in the gateway properties file by the **Gate.StartupCmdFile** property.

The following startup commands are available:

- `SET PROPERTY`: Use this command to set the value of a property in the gateway properties file.

Syntax	`SET PROPERTY 'property_name' TO value;`
Example	`SET PROPERTY 'Gate.NGtkDebug' TO FALSE;`

- `GET PROPERTY`: Use this command to return the value of a specified property from the gateway properties file.

Syntax	`GET PROPERTY 'property_name';`
Example	`GET PROPERTY 'Gate.NGtkDebug';`

- `SHOW PROPS`: Use this command to display the current configuration of the gateway by listing all properties and their values.

Syntax	`SHOW PROPS;`
Example	`SHOW PROPS;`

- `GET CONFIG`: Use this command to display the current configuration of the gateway by listing all properties and their values. `GET CONFIG` provides the same function as the `SHOW PROPS` command.

Syntax	`GET CONFIG;`
Example	`GET CONFIG;`

- `SET LOG LEVEL TO`: Use this command to set the level of message logging for the gateway. The available levels are debug, info, warn, error, and fatal. The default logging level is warn.

Syntax	`SET LOG LEVEL TO message_level;`
Example	`SET LOG LEVEL TO debug;`

- `TRANSFER`: Use this command to transfer data between an ObjectServer table and a target database table. You can add several `TRANSFER` commands to the startup command file.

  You can use a filter condition to transfer a partial table. You can also specify a map as part of the transfer operation. This map must be defined in the map definition file (`gateway_name.map`).

Syntax	TRANSFER FROM '*source_table*' TO '*target_table*' [VIA FILTER '*filter*'] [WITH DELETE VIA '*delete_filter*'] [USING TRANSFER_MAP '*map_name*'];
Example	TRANSFER FROM 'master.names' TO 'resync.names' VIA FILTER 'Name != \'nobody\'' DELETE; In this example, all rows that contain the name nobody are deleted from the resync.names table and replaced with corresponding rows in the master.names table. The single quotation marks (') enclosing nobody must be escaped with backslashes (\).

- FAILOVER_SYNC: Use this command to synchronize data between primary and backup ObjectServers. The command specifies which master tables are transferred during the data transfer operation. You can add several FAILOVER_SYNC commands to the startup command file.

Syntax	FAILOVER_SYNC ADD|REMOVE '*table_name*' TO *writer_name* ;
Example	FAILOVER_SYNC ADD 'master.names' TO ObjectServerA; FAILOVER_SYNC ADD 'master.groups'TO ObjectServerA; FAILOVER_SYNC ADD 'master.members' TO ObjectServerA; FAILOVER_SYNC ADD 'master.permissions' TO ObjectServerA; FAILOVER_SYNC ADD 'master.profiles'TO ObjectServerA;

The following is an example of a startup command file:

```
GET CONFIG;
SET LOG LEVEL TO debug;
TRANSFER FROM 'master.names' TO 'resync.names' VIA FILTER
 'Name != \'nobody\'' DELETE;
```

Common gateway properties and command-line options

Gateways have a number of common properties and associated command-line options. Properties define settings for generic functions, such as message logging, for inter-process communication (IPC), and for common gateway settings, such as specifying map definition files. You can override the default property values by editing the properties file or by using the command-line options when you start the gateway.

The following table lists the available common gateway properties and command-line options. Not all gateways use all of the properties that are listed here. See the individual gateway reference documentation for gateway-specific property information.

Table 39. Common gateway properties and command-line options

Property name	Command-line option	Description
ConfigCryptoAlg *string*	-configcryptoalg *string*	Use this property to specify the encryption algorithm that the gateway uses. Use this property with the **ConfigKeyFile** property and the **nco_aes_crypt** utility that is supplied with Netcool/OMNIbus. The default is AES.

Table 39. Common gateway properties and command-line options (continued)

Property name	Command-line option	Description
ConfigKeyFile *string*	-configkeyfile *string*	Use this property to specify the encryption key that is used to encrypt data. Use this property with the **ConfigCryptoAlg** property and the **nco_aes_crypt** utility that is supplied with Netcool/OMNIbus. The default is "".
Connections *integer*	-connections *integer*	Use this property to specify the maximum number of client connections that can be made to the gateway server. The default is 30.
Gate.CacheHashTblSize *integer*	-cachehtblsize *integer*	Use this property to specify the number of elements that the gateway allocates for the hash table cache. The default is 5023.
Gate.MapFile *string*	-mapfile *string*	Use this property to specify the mapping file that the gateway uses. The default is $OMNIHOME/gates/*gateway_name*/*gateway_name*.map.
Gate.NGtkDebug *boolean*	-ngtkdebug *boolean*	Use this property to enable the logging of NGTK library debug messages. The default is TRUE.
Gate.PAAware *integer*	-paaware *integer*	This property indicates whether the gateway is process agent (PA) aware. This property is maintained by the PA server and is included in the properties file for information only. The default is 0 (not PA aware).
Gate.PAAwareName *string*	-paname *string*	This property indicates the name of the process agent (PA) controlling the gateway. This property is maintained by the PA server and is included in the properties file for information only. The default is "".

Table 39. Common gateway properties and command-line options (continued)

Property name	Command-line option	Description
Gate.StartupCmdFile *string*	-startupcmdfile *string*	Use this property to specify the location of the startup command file. The default is $OMNIHOME/gates/*gateway_name*/*gateway_name*.startup.cmd.
Gate.Transfer.FailoverSyncRate *integer*	-fsyncrate *integer*	Use this property to specify the rate (in seconds) of failover synchronization. The default is 60.
Gate.UnixAdminGrp *string*	-unixadmingrp *string*	Use this property to specify the administration group to which the gateway user account belongs when standard UNIX authentication is used. The default is ncoadmin.
Gate.UsePamAuth *boolean*	-usepamauth *boolean*	Use this property to specify whether PAM authentication is used. To run the gateway in FIPS 140-2 mode, set this property to TRUE. The default is FALSE.
Help *boolean*	-help *boolean*	Use this property to specify that the gateway displays help information when it starts and shuts down. The default is FALSE.
Ipc.Timeout *integer*	-ipctimeout *integer*	Use this property to specify the time period (in seconds) that the client waits for the server to respond. If this time is exceeded, an error is logged. The default is 60.
MaxLogFileSize *integer*	-maxlogfilesize *integer*	Use this property to specify the maximum size (in bytes) that the gateway allocates for the log file. The default is 1024.
MessageLevel *string*	-messagelevel *string*	Use this property to specify the reporting level for gateway log file messages. The default is warn.

Table 39. Common gateway properties and command-line options (continued)

Property name	Command-line option	Description
MessageLog *string*	-messagelog *string*	Use this property to specify the location of the gateway message log file. The default is $OMNIHOME/log/NCO_GATE.log.
Name *string*	-name *string*	Use this property to specify the name of the current gateway instance. If you want to run multiple gateways on one computer, you must use a different name for each instance. The default is NCO_GATE.
Props.CheckNames *boolean*	N/A	Use this property to instruct the gateway to shut down if any property in the properties file is set to an invalid value. The default is TRUE.
PropsFile *string*	-propsfile *string*	Use this property to specify the location of the gateway properties file. The default is $OMNIHOME/etc/NCO_GATE.props.
UniqueLog *boolean*	-uniquelog *boolean*	Use this property to specify that log file names are made unique by adding the Process ID (PID) of the gateway to the file name. The default is FALSE.
Version *boolean*	-version *boolean*	Use this property to specify that the gateway displays version information when it starts and shuts down. The default is FALSE.

Related concepts:

"Using multiple configuration files" on page 167
Most gateways are configured using a properties file, a map definition file, a startup command file, and a table replication definition file.

Issuing commands to running gateways

You can use the SQL interactive interface to issue commands to a running gateway.

Note: If you are running Tivoli Netcool/OMNIbus in FIPS 140-2 mode, you cannot use the SQL interactive interface to issue commands to gateways.

You can issue the commands that are supported by the gateway startup command file, such as `SET PROPERTY` and `TRANSFER`.

Use the following command to connect to a gateway:

- On UNIX and Linux: `$OMNIHOME/bin/nco_sql -server server_name -user user_name`
- On Windows: `%OMNIHOME%\bin\isql -S server_name -U user_name`

where *server_name* is the name of the gateway as configured in the Server Editor and *user_name* is a valid user name.

On UNIX and Linux operating systems, the following requirements apply when you are issuing commands to gateways with the SQL interactive interface:

- If you set the **Gate.UsePamAuth** property to `FALSE`, the user that runs the gateway process must have permission to read the user database. Depending on your operating system configuration, the user needs permission to read the following files: `etc/passwd`, `etc/shadow`, and `etc/group`.

  You must also set the **Gate.UnixAdminGrp** property to the UNIX user administration group to which the gateway user account belongs.
- If you set the **Gate.UsePamAuth** property is to `TRUE`, you must configure the gateway PAM service for the `auth` and `account` modules.

  The gateway might also need permission to read databases used by the PAM configuration. For example, if the PAM configuration uses UNIX user authentication, the gateway user might need read access to the `/etc/shadow` directory.

For more information about the SQL interactive interface, see the *IBM Tivoli Netcool/OMNIbus Administration Guide*.

Gateway interactive command-line tool

For the Gateway for Oracle and the Gateway for ODBC, you can use the gateway interactive command-line tool (**nco_g_icmd**) to issue commands to the gateways while they are running. For more information, see the Gateway for Oracle and Gateway for ODBC reference guides.

Using single configuration files

The Gateway for HP ServiceCenter, Gateway for Remedy ARS, and Gateway for Siebel use a single configuration file to define data mappings and to control the reader and writer components.

Before running one of these gateways, you must edit the configuration file to specify the reader, writer, route, mapping, and filter definitions.

The default configuration file is `$OMNIHOME/etc/G_gateway_name.conf`. For example, the Gateway for Siebel uses the following configuration file: `$OMNIHOME/etc/G_SIEBEL.conf`

Related reference:

"Filter commands" on page 194
A number of filter commands are available for gateways.

"Mapping commands" on page 192
A number of mapping commands are available for gateways.

"Reader commands" on page 188
A number of reader commands are available for gateways.

"Route commands" on page 195
A number of route commands are available for gateways.

"Writer commands" on page 189
A number of writer commands are available for gateways.

Reader configuration

A reader extracts alerts from an ObjectServer. Readers are started using the START READER command, which defines the name of the reader and the name of the ObjectServer from which to read.

For example, to start a reader for an NCOMS ObjectServer, add the following command to the configuration file:

```
START READER NCOMS_READ CONNECT TO NCOMS;
```

After this command is issued, the reader starts and the gateway attempts to open a connection to the source ObjectServer. If the gateway succeeds in opening the connection, it immediately starts to read alerts from the ObjectServer. For the reader to forward these alerts to their destination, you must define an associated route and writer.

Related reference:

"Reader commands" on page 188
A number of reader commands are available for gateways.

Writer configuration

Writers send the alerts acquired by a reader to the destination application or ObjectServer. Writers are created using the START WRITER command, which defines the name of the writer and the information that allows it to connect to its destination.

For example, to create the writer for a Flat File Gateway, add the following command to the configuration file:

```
START WRITER FILE_WRITER
(        TYPE = FILE,
        REVISION = 1,
        FILE = '/tmp/omnibus/log/NCOMS_alert.log',
        MAP = FILE_MAP,
        INSERT_HEADER = 'INSERT: ',
        UPDATE_HEADER = 'UPDATE: ',
        DELETE_HEADER = 'DELETE: ',
        START_STRING = '"',
        END_STRING = '"',
        INSERT_TRAILER = '\n',
        UPDATE_TRAILER = '\n',
        DELETE_TRAILER = '\n'
);
```

After the START WRITER command is issued, the gateway attempts to establish the connection to the alert destination (either an application or another ObjectServer). The writer sends alerts received from the source ObjectServer until the STOP WRITER command is issued.

Related reference:

"Writer commands" on page 189
A number of writer commands are available for gateways.

Route configuration

Routes create the link between readers and writers. Routes are created using the ADD ROUTE command. This command defines the name of the route, the source reader, and the destination writer.

For example, to create the route between the NCOMS ObjectServer reader and the writer for a Flat File Gateway, add the following command to the configuration file:

```
ADD ROUTE FROM NCOMS_READ TO FILE_WRITER;
```

After this command is issued, the connection between a reader and writer is established. Any alerts received by the source ObjectServer are read by the reader, passed through the route to the writer, and written into the destination ObjectServer or application.

Related reference:

"Route commands" on page 195
A number of route commands are available for gateways.

Mapping configuration

Mappings define how alerts received from the source ObjectServer should be written to the destination ObjectServer or application. Each writer has a different mapping that is defined using the CREATE MAPPING command.

For example, to create the mapping between the ObjectServer reader and the writer for a Flat File Gateway, add the following command to the configuration file:

```
CREATE MAPPING FILE_MAP
(
 ''= '@Identifier',
 ''= '@Serial',
 ''= '@Node' ,
 ''= '@Manager',
 ''= '@FirstOccurrence' CONVERT TO DATE,
 ''= '@LastOccurrence'  CONVERT TO DATE,
 ''= '@InternalLast'    CONVERT TO DATE,
 ''= '@Tally',
 ''= '@Class',
 ''= '@Grade',
 ''= '@Location',
 ''= '@ServerName',
 ''= '@ServerSerial'
);
```

In this example, the mapping name is FILE_MAP.

Each line between the parentheses defines how the gateway writes alerts into the file. For the Flat File Gateway, the CREATE MAPPING command defines the fields from which data is written into each alert in the output file. The alert fields from the source ObjectServer are represented by the @ symbol.

The following example shows INSERT and UPDATE commands using the FILE_MAP mapping shown in the preceding example:

```
INSERT: "Downlink6LinkMon4Link",127,"sfo4397","Netcool Probe",12/05/03 15:39:23,
12/05/03 15:39:23,12/05/03 15:30:53,1,3300,0,"","NCOMS",127
UPDATE: "muppetMachineMon2Systems",104,"sfo4397","Netcool Probe",12/05/03 12:29:34,
12/05/03 15:40:06,12/05/03 15:31:36,11,3300,0,"","NCOMS",104
UPDATE: "muppetMachineMon4Systems",93,"sfo4397","Netcool Probe",12/05/03 12:29:11,
12/05/03 15:40:35,12/05/03 15:32:05,12,3300,0,"","NCOMS",93
```

Other gateways might require a field in the target to be specified for each source ObjectServer field. For example, in the Gateway for Remedy ARS, source ObjectServer fields are mapped to Remedy ARS fields, which are identified with long integer values rather than field names. In the following example, the ARS field 536870913 maps to the Serial field from the ObjectServer:

```
536870913 = '@Serial' ON INSERT ONLY
```

The ON INSERT ONLY clause controls when the field is updated. Fields with the ON INSERT ONLY clause are forwarded only once, when the alert is created for the first time in the ObjectServer. Fields that do not have the ON INSERT ONLY clause are updated each time the alert changes.

Related reference:

"Mapping commands" on page 192
A number of mapping commands are available for gateways.

Filter configuration

You might not always want to send all of the alerts that are read by a reader to the destination application. Filters define which of the alerts read by the ObjectServer reader should be forwarded to the destination.

For example, you may want to send only alerts that have a severity level of Critical.

You create filters using the CREATE FILTER command and apply them using the START READER command. For example, to create a filter that forwards only critical alerts to the destination application or ObjectServer, add the following command to the configuration file:

```
CREATE FILTER CRITONLY AS 'Severity = 5';
```

This command creates a filter named CRITONLY, which forwards alerts with a severity level of Critical (5) only.

To apply the filter to an ObjectServer reader, add the following command to the configuration file:

```
START READER NCOMS_READ CONNECT TO NCOMS USING FILTER CRITONLY;
```

Note: To perform string comparisons with filters, you must escape the quotes in the CREATE FILTER command with backslashes. For example, to create a filter that forwards only alerts from a node called fred, the CREATE FILTER command is:

```
CREATE FILTER FREDONLY AS 'NODE = \'fred\'';
```

Creating multiple filters and multiple readers

If you need more than one filter for the same ObjectServer, you can create multiple readers for it.

For example, to create a reader that forwards all critical alerts and another that forwards everything else, use the following commands:

```
CREATE FILTER CRITONLY AS 'Severity = 5';
CREATE FILTER NONCRIT AS 'Severity < 5';
START READER CRIT_NCOMS CONNECT TO NCOMS USING FILTER CRITONLY;
START READER NONCRIT_NCOMS CONNECT TO NCOMS USING FILTER NONCRIT;
```

Loading filters created using the Filter Builder

You can load and use filters that are created in the Filter Builder.

For example:

```
LOAD FILTER FROM '/usr/filters/myfilt.elf';
```

This command loads the file `/usr/filters/myfilt.elf` as a filter. This filter name is defined by the Filter Builder **Name** field.

Note: The **Name** field must be alphabetical and must not contain spaces.

Related reference:

"Filter commands" on page 194
A number of filter commands are available for gateways.

"START READER" on page 188
Use the START READER command to start a reader named *reader_name* that connects to an ObjectServer named *server_name*.

Common gateway command-line options

Gateways that use a single configuration file have a number of common command-line options. These gateways do not use properties files.

The common command-line options described here are used by the following gateways:

- Gateway for HP ServiceCenter
- Gateway for Remedy ARS
- Gateway for Siebel

The following table lists the available common command-line options that you can use when starting a gateway. See the individual gateway reference documentation for gateway-specific command-line options.

Table 40. Common gateway command-line options

Command-line option	Description
`-admingroup` *string*	Specifies the name of the UNIX user group that has administrator privileges. Members of this group can log into the gateway. The default group name is `ncoadmin`.

Table 40. Common gateway command-line options (continued)

Command-line option	Description
`-authenticate UNIX | PAM | HPTCB`	Specifies the authentication mode to use to verify user credentials. The options are `UNIX`, `PAM`, and `HPTCB`. The default authentication mode is `UNIX`, which means that the Posix `getpwnam` or `getspnam` function is used to verify user credentials on UNIX operating systems. Depending on system setup, passwords are verified using the `/etc/password` file, the `/etc/shadow` shadow password file, NIS, or NIS+. If `PAM` is specified as the authentication mode, Pluggable Authentication Modules are used to verify user credentials. The service name used by the gateway when the PAM interface is initialized is `netcool`. PAM authentication is available on Linux, Solaris, and HP-UX 11 operating systems only. If `HPTCB` is specified as the authentication mode, this HP-UX password protection system is used. This option is only available on HP trusted (secure) systems.
`-config` *string*	Specifies the name of the configuration file to be read when the gateway starts. The default is `$OMNIHOME/etc/`*gatename*`.conf`.
`-connections`	The number of permitted connections. The default is `30`.
`-debug`	When specified, debug mode is enabled.
`-help`	Displays help information about the command-line options and exits.
`-ipctimeout`	IPC Session timeout. The default is `60` seconds.
`-logfile` *string*	Specifies the name of the log file. If omitted, the default is `$NCHOME/omnibus/log/`*gateway_name*`.log`.
`-logsize` *integer*	Specifies the maximum size of the log file in KB. The minimum is 16 KB. The default is 1 MB.
`-messagelevel`	The level of messages to be logged. The available levels are: debug, info, warn, error, and fatal. The default is `warn`.
`-messagelog`	The path to the message log file. The default is: `$NCHOME/omnibus/log/NCO_GATE.log`
`-name` *string*	Specifies the gateway name. Specify this name following the `-server` command-line option to connect to the gateway using **`nco_sql`**. If omitted, the default is *GATENAME*.
`-notruncate`	Specifies that the log file is not truncated.
`-oldtimestamp`	Specifies the timestamp format to use in the log file.
`-propsfile`	The full path to the gateway properties file.
`-queue` *integer*	Specifies the size of the internal queues. The default is `1024`. Do not modify unless advised by IBM Software Support.
`-stacksize` *integer*	Specifies the size of the internal threads. The default is `256` KB. Do not modify unless advised by IBM Software Support.

Table 40. Common gateway command-line options (continued)

Command-line option	Description
`-uniquelog`	If `-logfile` is not set, this option forces the log file to be uniquely named by appending the process ID of the gateway to the end of the default log file name. If `-logfile` is set, this has no effect.
`-version`	Displays version information and exits.

Configuring running gateways

You can use the SQL interactive interface to change the configuration of a gateway while it is running.

About this task

Note: To connect to a gateway on UNIX using **`nco_sql`**, you must specify the user name and password of a member of the UNIX user group that is allowed to log into a gateway. This user group is specified using the `-admingroup` command-line option. By default, this is the ncoadmin user group. You might need to ask your system administrator to create this group. Also, the user running the gateway must have access to the appropriate file that is used to verify passwords so that the members of ncoadmin can be authenticated when logging into the gateway using **`nco_sql`**.

Use the SQL interactive interface to connect to a gateway as a specific user, as shown in the following table.

Table 41. Connecting to the gateway using the SQL interactive interface

On	Enter the following command
UNIX	`$OMNIHOME/bin/nco_sql -server servername -user username`
Windows	`%OMNIHOME%\bin\redist\isql -S servername -U username`

In these commands, *servername* is the name of the gateway and *username* is a valid user name. If you do not specify a user name, the default is the user running the command.

You are prompted to enter a password. On UNIX, the default is to enter your UNIX password. To authenticate users using other methods, use the `-authenticate` command-line option.

After connecting with a user name and password, a numbered prompt is displayed.

```
1>
```

You can enter commands to configure the gateway dynamically. The following example shows a session in which new routes are added to a gateway with a single `.conf` configuration file:

```
$ nco_sql -server REMEDY
Password:
User 'admin' logged in.
1> ADD ROUTE FROM DENCO_READ TO ARS_WRITER;
```

```
2> ADD ROUTE FROM DENCO_READ TO OS_WRITER;
3> go

1>
```

Note: If you want to disable interactive configuration, add the following line to the end of the gateway configuration file:

```
SET CONNECTIONS FALSE;
```

Loading and saving configurations

You can use the SQL interactive interface to load and save gateway configurations while the gateway is running.

About this task

You can use the saved configuration file for other gateways.

Procedure

1. To stop any running readers and writers:

   ```
   >1 STOP;
   >2 go
   ```

2. To discard the current configuration:

   ```
   >1 DUMP CONFIG;
   >2 go
   ```

   The `DUMP CONFIG` command will not work if any readers or writers are running, or if the configuration has been changed interactively. To determine if the configuration has been changed interactively, use the `SHOW SYSTEM` command. You can use the `FORCE` option to force the current configuration to be discarded.

3. To load a new configuration:

   ```
   >1 LOAD CONFIG FROM 'file_name';
   >2 go
   ```

   where *file_name* is the path to another configuration file.

4. To save the current configuration:

   ```
   >1 SAVE CONFIG TO 'file_name';
   >2 go
   ```

   where *file_name* is name and path for the new configuration file.

Gateway commands

The following topics describe the commands that you can use with gateways that use a single configuration file.

The commands described here are used by the following gateways:

- Gateway for HP ServiceCenter
- Gateway for Remedy ARS
- Gateway for Siebel

The gateway commands can be used in the configuration file (`G_gateway_name.conf`) or with the SQL interactive interface. The following table lists the available commands.

Table 42. Gateway commands

Command Group	Command	Description
Reader commands	START READER	Starts a reader.
	STOP READER	Stops a reader.
	SHOW READERS	Lists all the current readers.
Writer commands	START WRITER	Starts a writer.
	STOP WRITER	Stops a writer.
	SHOW WRITERS	Lists all the current writers.
	SHOW WRITER TYPES	Lists all the types of writers that are supported by a gateway.
	SHOW WRITER ATTRIBUTES	Lists all the attributes of a writer.
Mapping commands	CREATE MAPPING	Creates a mapping file.
	DROP MAPPING	Removes a mapping file.
	SHOW MAPPINGS	Lists all the mappings being used by a gateway.
	SHOW MAPPING ATTRIBUTES	Lists all the attributes of a mapping.
Filter commands	CREATE FILTER	Creates a filter.
	LOAD FILTER	Loads a filter.
	DROP FILTER	Removes a filter.
Route commands	ADD ROUTE	Adds a route between a reader and a writer.
	REMOVE ROUTE	Removes a route.
	SHOW ROUTES	Shows all the routes configured for a gateway.
Configuration commands	LOAD CONFIG	Loads a configuration file.
	SAVE CONFIG	Saves a configuration file.
	DUMP CONFIG	Clears the current configuration.
General commands	SHUTDOWN	Shuts down the gateway.
	SET CONNECTIONS	Enable or disables connections to the gateway from the SQL interactive interface.
	SHOW SYSTEM	Displays information about the gateway settings.
	SET DEBUG MODE	Sets the debugging mode of the gateway.
	TRANSFER	Transfers the contents of one database table to another database table.

Reader commands

A number of reader commands are available for gateways.

Related concepts:

"Reader configuration" on page 180
A reader extracts alerts from an ObjectServer. Readers are started using the START READER command, which defines the name of the reader and the name of the ObjectServer from which to read.

START READER:

Use the START READER command to start a reader named *reader_name* that connects to an ObjectServer named *server_name*.

Syntax

```
START READER reader_name CONNECT TO server_name [ USING FILTER filter_name ]
[ ORDER BY 'column, ... [ ASC | DESC ]' ] [ AFTER IDUC DO 'update_command' ]
[ IDUC = integer ] [ JOURNAL_FLUSH = integer ] [ IDUC_ORDER ];
```

The optional USING FILTER clause, followed by the name of a filter that has been created using the CREATE FILTER command, enables you to restrict the number of rows affected by gateway updates. The filter replaces an SQL WHERE clause, so the gateway only updates the rows selected by the filter.

The optional ORDER BY clause instructs the gateway to display the results in sequential order, depending on the values of one or more column names, in either descending (DESC) or ascending (ASC) order. If the ORDER BY clause is not specified, no ordering is used.

The optional AFTER IDUC clause instructs the gateway to perform the update specified in the *update_command* in the ObjectServer when it places alerts in the writer queue. This is used to provide feedback when alerts pass through a gateway.

Note: The update command that follows an AFTER IDUC DO statement in the START READER command must be a simple UPDATE statement. It must not use conditions (for example, WHERE or HAVING); these are not supported in this context.

The value specified in the optional IDUC clause indicates an IDUC interval for gateways that is more frequent than the value of the **Granularity** property set in the source ObjectServer. This enables gateway updates to be forwarded to the target more rapidly without causing overall system performance to deteriorate.

The value specified in the optional JOURNAL_FLUSH clause indicates a delay in seconds between when the IDUC update occurs in the ObjectServer (every *Granularity* seconds) and when the journal entries are retrieved by the gateway. Normally, only journal entries that have been made in the last *Granularity* seconds are retrieved. When the system is under heavy load, set this clause so journal entries are retrieved for the last *integer* + *Granularity* seconds. This prevents the loss of any journal entries that are created after the IDUC update but before the gateway retrieves the entries. Any duplicate journal entries retrieved are eliminated by deduplication.

The optional IDUC_ORDER clause specifies the order in which the IDUC data is processed. The default processing mode for gateways is to process DELETE statements, followed by UPDATE statements, followed by INSERT statements. Do

not change this clause unless you have been advised to do so by IBM Software Support.

Example

This example uses the Grade field as a state field. Initially, all probes set Grade to 0. The gateway filters any alerts that have a Grade of 1. After the alerts have passed through the gateway, the AFTER IDUC update provides alert state feedback by changing the value of the Grade field to 2.

```
START READER NCOMS_READER CONNECT TO NCOMS USING FILTER CRIT_ONLY
ORDER BY 'SERIAL ASC' AFTER IDUC DO 'update alerts.status set Grade=2';
```

Related concepts:

"Filter configuration" on page 182
You might not always want to send all of the alerts that are read by a reader to the destination application. Filters define which of the alerts read by the ObjectServer reader should be forwarded to the destination.

STOP READER:

Use the STOP READER command to stop the reader named *reader_name*.

Syntax

```
STOP READER reader_name;
```

This command does not stop the reader if the reader is in use with any routes.

Example

```
STOP READER NCOMS_READ;
```

SHOW READERS:

Use the SHOW READERS command to list all the current readers that have been started, and which are running on the gateway.

Syntax

```
SHOW READERS;
```

This command can only be used interactively.

Example

```
SHOW READERS;
```

Writer commands

A number of writer commands are available for gateways.

Related concepts:

"Writer configuration" on page 180
Writers send the alerts acquired by a reader to the destination application or ObjectServer. Writers are created using the START WRITER command, which defines the name of the writer and the information that allows it to connect to its destination.

START WRITER:

Use the START WRITER command to start a writer named *writer_name*.

Syntax

```
START WRITER writer_name
( TYPE=writer_type , REVISION=number
[ , keyword_setting [ , keyword_setting ] ...] );
```

The START WRITER command is followed by a list of comma-separated keyword settings in parentheses. The first setting must be a TYPE setting indicating the *writer_type*. The next setting must be a REVISION setting. This is currently set to 1 for all writers. The remaining keywords and their settings depend on the type of writer.

Example

This example starts the writer for a Socket Writer Gateway.

```
START WRITER SOCKET_WRITER
(
        TYPE = SOCKET,
        REVISION = 1,
        HOST = 'sfo768',
        PORT = 4010,
        MAP = SOCKET_MAP,
        INSERT_HEADER = 'INSERT: ',
        UPDATE_HEADER = 'UPDATE: ',
        DELETE_HEADER = 'DELETE: ',
        START_STRING = '"',
        END_STRING = '"',
        INSERT_TRAILER = '\n',
        UPDATE_TRAILER = '\n',
        DELETE_TRAILER = '\n'
);
```

STOP WRITER:

Use the STOP WRITER command to stop the writer called *writer_name*.

Syntax

```
STOP WRITER writer_name;
```

If any route is using this writer, the writer does not stop.

Example

```
STOP WRITER ARS_WRITER;
```

SHOW WRITERS:

Use the SHOW WRITERS command to list all the current writers in the gateway.

Syntax

```
SHOW WRITERS;
```

This command can only be used interactively.

Example

```
1>SHOW WRITERS;
2>GO
Name        Type Routes Msgq Id Mutex Id Thread
----------- ---- ------ ------- -------- ------
SNMP_WRITER SNMP 1      15      0        0x001b8cd0

1>
```

SHOW WRITER TYPES:

Use the SHOW WRITER TYPES command to list all the currently known types of writers that are supported by the gateway.

Syntax

```
SHOW WRITER TYPES;
```

This command can only be used interactively.

Example

```
1> SHOW WRITER TYPES;
2> GO
Type             Revision      Description
------           -----------    ----------------------
ARS              1              Action Request System V3.0
OBJECT_SERVER    1              Netcool/OMNIbus ObjectServer V7
SYBASE           1              Sybase SQL Server 10.0 RDBMS
SNMP             1              SNMP Trap forwarder
SERVICE_VIEW     1              Service View
```

SHOW WRITER ATTRIBUTES:

Use the SHOW WRITER ATTRIBUTES command to show all the settings (or attributes) of the writer named *writer_name*.

Syntax

```
SHOW WRITER { ATTRIBUTES | ATTR } FOR writer_name;
```

The ATTRIBUTES keyword is interchangeable with the abbreviated ATTR keyword.

This command can only be used interactively.

Example

```
1> SHOW WRITER ATTR FOR SNMP_WRITER;
2> GO
Attribute            Value
-----------          ------------------------------------------------------
MAP                  SNMP_MAP
TYPE                 SNMP
REVISION             1
GATEWAY              penelope

1>
```

Mapping commands

A number of mapping commands are available for gateways.

Related concepts:

"Mapping configuration" on page 181
Mappings define how alerts received from the source ObjectServer should be written to the destination ObjectServer or application. Each writer has a different mapping that is defined using the CREATE MAPPING command.

CREATE MAPPING:

Use the CREATE MAPPING command to create a mapping file named *mapping_name*, for use by a writer.

Syntax

```
CREATE MAPPING mapping_name ( mapping [ , mapping ] );
```

Mapping lines have the following syntax:

```
{ string | integer } = { string | integer | name | real | boolean }
[ ON INSERT ONLY ] [ CONVERT TO { INT | STRING | DATE } ]
```

The first argument is an identifier for the destination field and the second argument is an identifier for the source field (or a preset value).

The right side of the mapping is dependent on the writer with which the mapping is to be used. (For gateway-specific details, see the writer section of the individual gateway publications.)

The optional ON INSERT ONLY clause determines the update behavior of the mapping. Without the ON INSERT ONLY clause, the field is updated every time a change is made to an alert. With the ON INSERT ONLY clause, the field is inserted at creation time (that is, when the alert appears for the first time) but is not updated on subsequent updates of the alert even if the field value is changed.

The optional CONVERT TO type clause allows the mapping to define a forced conversion for situations where a source field may not match the type of the destination field. The type can be INT, STRING, or DATE. This forces the source field to be converted to the specified data type.

Example

```
CREATE MAPPING SYBASE_MAP
(
'Node'='@Node' ON INSERT ONLY,
'Summary'='@Summary' ON INSERT ONLY,
'Severity'='@Severity' );
```

DROP MAPPING:

Use the DROP MAPPING command to remove the mapping named *mapping_name* from the gateway.

Syntax

```
DROP MAPPING mapping_name;
```

This command does not drop the map if it is being used by a writer.

Example

```
DROP MAPPING SYBASE_MAP;
```

SHOW MAPPINGS:

Use the SHOW MAPPINGS command to list all the mappings that are currently created in the gateway.

Syntax

```
SHOW MAPPINGS;
```

This command can only be used interactively.

Example

```
1> SHOW MAPPINGS;
2> GO
 Name                           Writers
 ------------------------------ -----------
 SNMP_MAP                             1
1>
```

SHOW MAPPING ATTRIBUTES:

Use the SHOW MAPPING ATTRIBUTES command to show the mappings (or attributes) of the mapping named *mapping_name*.

Syntax

```
SHOW MAPPING { ATTRIBUTES | ATTR } FOR mapping_name;
```

The ATTRIBUTES keyword is interchangeable with the abbreviated ATTR keyword. This command can only be used interactively.

Example

```
SHOW MAPPING ATTR FOR SYBASE_MAP;
```

Filter commands

A number of filter commands are available for gateways.

Related concepts:

"Filter configuration" on page 182
You might not always want to send all of the alerts that are read by a reader to the destination application. Filters define which of the alerts read by the ObjectServer reader should be forwarded to the destination.

CREATE FILTER:

Use the CREATE FILTER command to create a filter named *filter_name* for use by a reader.

Syntax

```
CREATE FILTER filter_name AS filter_condition;
```

The filter specification *filter_condition* is an SQL condition.

Example

```
CREATE FILTER HIGH_TALLY_LOG AS 'Tally > 100';
CREATE FILTER NCOMS_FILTER AS 'Agent = \'NNM\'';
```

LOAD FILTER:

Use the LOAD FILTER command to load a filter from a file.

Syntax

```
LOAD FILTER FROM 'filename';
```

Include the path to the file. Filter files have a `.elf` file extension.

Example

```
LOAD FILTER FROM '/disk/filters/newfilter.elf';
```

DROP FILTER:

Use the DROP FILTER command to remove the filter named *filter_name* from the gateway.

Syntax

```
DROP FILTER filter_name;
```

The filter is not dropped if it is being used by a reader.

Example

```
DROP FILTER HIGH_TALLY_LOG;
```

Route commands

A number of route commands are available for gateways.

Related concepts:

"Route configuration" on page 181
Routes create the link between readers and writers. Routes are created using the ADD ROUTE command. This command defines the name of the route, the source reader, and the destination writer.

ADD ROUTE:

Use the ADD ROUTE command to add a route between a reader named *reader_name* and a writer named *writer_name*, to allow alerts to pass through the gateway.

Syntax

```
ADD ROUTE FROM reader_name TO writer_name;
```

Example

```
ADD ROUTE FROM NCOMS_READER TO ARS_WRITER;
```

REMOVE ROUTE:

Use the REMOVE ROUTE command to remove an existing route between a reader named *reader_name* and a writer named *writer_name*.

Syntax

```
REMOVE ROUTE FROM reader_name TO writer_name;
```

Example

```
REMOVE ROUTE FROM NCOMS_READER TO ARS_WRITER;
```

SHOW ROUTES:

Use the SHOW ROUTES command to show all currently-configured routes in the gateway.

Syntax

```
SHOW ROUTES;
```

This command can only be used interactively.

Example

```
1> SHOW ROUTES;
2> GO
 Reader                          Writer
 ------------------------------- -------------------------------
 NCOMS_READER                    SNMP_WRITER

1>
```

Configuration commands

A number of configuration commands are available for gateways.

LOAD CONFIG:

Use the LOAD CONFIG command to load a gateway configuration file from a file named *filename*.

Syntax

```
LOAD CONFIG FROM 'filename';
```

Example

```
LOAD CONFIG FROM '/disk/config/gateconf.conf';
```

SAVE CONFIG:

Use the SAVE CONFIG command to save the current configuration of the gateway into a file named in *filename*.

Syntax

```
SAVE CONFIG TO 'filename';
```

Example

```
SAVE CONFIG TO '/disk/config/newgate.conf';
```

DUMP CONFIG:

Use the DUMP CONFIG command to clear the current configuration.

Syntax

```
DUMP CONFIG [ FORCE ];
```

If the gateway is active and forwarding alerts, this command does not clear the configuration unless the optional keyword FORCE is used.

Example

```
DUMP CONFIG;
```

General commands

A number of general commands are available for gateways.

SHUTDOWN:

Use the SHUTDOWN command to instruct the gateway to shut down; all readers and writers are stopped.

Syntax

```
SHUTDOWN [ FORCE ];
```

By default, the gateway is not shut down if interactive changes to the configuration have not been saved.

If the optional FORCE keyword is used, the gateway is shut down, even if the configuration has been changed interactively.

Example

```
SHUTDOWN;
```

SET CONNECTIONS:

Use the SET CONNECTIONS command to enable or disable connections to the gateway using the SQL interactive interface.

Syntax

```
SET CONNECTIONS { TRUE | FALSE | YES | NO };
```

When set to FALSE or NO, it is not possible to connect to the gateway with **nco_sql**. When set to TRUE or YES, it is possible to connect to the gateway with **nco_sql**. This command determines whether interactive reconfiguration is allowed.

Example

```
SET CONNECTIONS TRUE;
```

SHOW SYSTEM:

Use the SHOW SYSTEM command to display information about the current gateway settings.

Syntax

```
SHOW SYSTEM;
```

The parameters returned are shown in the following table.

Table 43. Show system parameters

System Parameter	Description
Version	Version number of the gateway.
Server Type	Type of server. Set to Gateway.
Connections	Status of the SET CONNECTIONS flag.
Debug Mode	Status of the SET DEBUG MODE flag.
Multi User	Gateway multi-user mode. Set to YES.
Configuration Changed	If the configuration has been changed interactively, this is set to YES.

More parameters can be returned when in debug mode. This command can only be used interactively.

Example

```
1> SHOW SYSTEM;
2> GO
System Parameter     Value
---------------      --------------------------------
 Version             7.0
 Server Type         Gateway
 Connections         ENABLED
 Debug Mode          NO
 Multi User          YES
```

SET DEBUG MODE:

Use the SET DEBUG MODE command to set the debugging mode of the gateway.

Syntax

```
SET DEBUG MODE { TRUE | FALSE | YES | NO };
```

When set to TRUE or YES, debugging messages are sent to the log file. The default setting is NO or FALSE. Use this command only under the advice of IBM Software Support.

Example

```
SET DEBUG MODE NO;
```

TRANSFER:

Use the TRANSFER command to transfer the contents of one database table to another database table.

Syntax

```
TRANSFER 'tablename' FROM readername TO writername [ AS 'tableformat' ]
{ DELETE | DELETE condition | DO NOT DELETE }
[ USE TRANSFER_MAP ] [ USING FILTER filter_clause ];
```

You can use this command to transfer tables between Sybase, Oracle, Informix, ODBC, CORBA, and Socket Writer gateways.

The AS *tableformat* clause specifies the format of the destination table if it is different from the source table format.

The DELETE and DO NOT DELETE clauses define how the destination table is processed. By default, the contents of the destination table are deleted before the contents of the source table are transferred. You can optionally specify a condition that determines whether the deletion occurs. If you specify the DO NOT DELETE clause, the contents of the destination table are not deleted before the contents of the source table are transferred.

Note: The DELETE clause does not function with the Socket Writer Gateway and the CORBA gateways.

The USE TRANSFER_MAP clause instructs the gateway to use the mapping definition that is assigned as the map to the writer used in the TRANSFER command. The USE TRANSFER_MAP clause is only available for use with the Oracle Gateway.

An optional filter clause can be applied by specifying USING FILTER followed by the filter. Enter a valid filter.

Example

```
TRANSFER 'alerts.conversions' FROM NCO_READER TO SYBASE_WRITER AS
'alerts.conversions' DELETE;
TRANSFER 'alerts.status' FROM NCOMS_READ TO DENCO_WRITE AS 'ncoms.status'
USING FILTER 'ServerName = \'NCOMS\'' DELETE USE TRANSFER_MAP;
```

Creating conversion tables

You can create conversion tables to enable certain data conversions to take place between fields.

For example, if you are using the Gateway for Remedy ARS, you can create a table within the ObjectServer to insert values for the Severity field found in Remedy ARS.

To do this, you must use ObjectServer SQL commands. You can run ObjectServer SQL commands using the following tools:

- UNIX SQL interactive interface (**nco_sql**)
- Windows SQL interactive interface (**isql**)
- Netcool/OMNIbus Administrator

The following example ObjectServer SQL creates the table `remedy` in an ObjectServer, and inserts six values and corresponding descriptions for the `Severity` field:

```
create database conversions;
use database conversions;
create table conversions.remedy persistent (
 KeyField varchar(255) primary key,
 Colname  varchar(255),
 OSValue  varchar(255),
 Conversion varchar(255)
);
go

insert into conversions.remedy values ('Severity0','Severity','0','Clear');
insert into conversions.remedy values ('Severity1','Severity','1','Indeterminate');
insert into conversions.remedy values ('Severity2','Severity','2','Warning');
insert into conversions.remedy values ('Severity3','Severity','3','Minor');
insert into conversions.remedy values ('Severity4','Severity','4','Major');
insert into conversions.remedy values ('Severity5','Severity','5','Critical');
go
```

Chapter 10. Running gateways

Before you can run a gateway, you must create an entry for it in the Server Editor. You must also configure the gateway to suit your operational environment.

For information about adding an entry to the Server Editor, see the *IBM Tivoli Netcool/OMNIbus Installation and Deployment Guide*.

Related concepts:

Chapter 9, "Configuring gateways," on page 167
Configuration files define the environment in which gateways operate and how they map data between ObjectServer tables and target databases or applications. Most gateways use multiple configuration files. Some gateways are configured using a single configuration file.

Use of OMNIHOME and NCHOME environment variables

Tivoli Netcool/OMNIbus V7.0 (and earlier) uses the `OMNIHOME` environment variable in many configuration files. To use these files on Tivoli Netcool/OMNIbus V7.1 (and later), replace occurrences of the `OMNIHOME` environment variable with `NCHOME/omnibus`.

On UNIX and Linux operating systems, replace `$OMNIHOME` with `$NCHOME/omnibus`.

On Windows operating systems, replace `%OMNIHOME%` with `%NCHOME%\omnibus`.

Running gateways

On UNIX and Linux operating systems, you start gateways from the command line. On Windows operating systems, you can start gateways from the command line or as Windows services.

To start a gateway that uses the default configuration files, use the following command:

- On UNIX and Linux: `$OMNIHOME/bin/nco_g_`*`gateway_name`*
- On Windows: `%OMNIHOME%\bin\nco_g_`*`gateway_name`*

To start a second instance of a gateway, with a different name, use one of the following commands.

- On UNIX and Linux:
  - Use the following command to start a gateway that uses multiple configuration files:

    ```
    $OMNIHOME/bin/nco_g_gateway_name -name GATE2 -propsfile
    $OMNIHOME/etc/GATE2.props
    ```

  - Use the following command to start a gateway that uses a single configuration file:

    ```
    $OMNIHOME/bin/nco_g_gateway_name -name GATE2 -config
    $OMNIHOME/etc/GATE2.conf
    ```

- On Windows:
  - Use the following command to start a gateway that uses multiple configuration files:

```
%OMNIHOME%\bin\nco_g_gateway_name.exe -name GATE2 -propsfile
%OMNIHOME%\etc\GATE2.props
```

– Use the following command to start a gateway that uses a single configuration file:

```
%OMNIHOME%\bin\nco_g_gateway_name.exe -name GATE2 -config
%OMNIHOME%\etc\GATE2.conf
```

On UNIX and Linux operating systems, you must run gateways under process control.

For information about using process control, see the *IBM Tivoli Netcool/OMNIbus Administration Guide*.

Running gateways as Windows services

To run the gateway as a Windows service, use the following steps:

1. To run the gateway on the same host as the ObjectServer, use the following command to register it as a service:

   ```
   %OMNIHOME%\bin\nco_g_gateway_name.exe -install -depend NCOObjectServer
   ```

2. To run the gateway on a different host to the ObjectServer, use the following command to register it as a service:

   ```
   %OMNIHOME%\bin\nco_g_gateway_name.exe -install
   ```

3. Start the gateway using the Microsoft Services Management Console.

Troubleshooting gateway problems

When troubleshooting gateway problems, start by checking the gateway log file.

The default gateway log file is `$OMNIHOME/log/NCO_gateway_name.log`.

You might receive an error message such as the following:

```
error in srv_select () - file descriptor x is no longer active!
```

This type of error message indicates that the gateway has aborted because one of the reader or writer modules failed. In this case, check the following log files:

- `NCO_gateway_name_XRWY_WRITE.log`
- `NCO_gateway_name_XRWY_READ.log`

where *X* is the name of the gateway and *Y* is the version of the gateway.

Appendix A. Probe error messages and troubleshooting techniques

A number of error messages are common to all probes. This includes ProbeWatch and TSMWatch messages. Troubleshooting information is also available for probes.

See the individual probe publications for information about probe-specific messages.

Generic error messages

Probes can generate the following types of messages: fatal, error, warning, information, and debug.

Fatal-level messages

The probe automatically terminates when a fatal message is issued.

Table 44. Fatal level probe messages

Message	Description	Action
`Connection to ObjectServer marked DEAD - aborting...`	The connection to the ObjectServer ceased (and store and forward is not enabled in the probe).	Check that the ObjectServer is available.

`Failed to access OMNIHOME directory: "directory name"`
 	
`Failed to set interfaces file location`
	The probe was unable to locate the interfaces file.	Check that the OMNIHOME environment variable is set to the correct destination.

`Failed to connect - aborting`	The ObjectServer is not available.	Check that the ObjectServer is running, that the interfaces file on the system where the probe is installed has an entry for the ObjectServer, and that there is no networking problem between the two systems.

`Failed to create property`
 	
`Failed to define argument`
 	
`Failed to initialise`
 	
`Failed to set property`
 	
`Failed to process arguments`
 	
`Session create failed - aborting`
	Internal errors.	See your support contract for information about contacting IBM Software Support.

`Failed to read rules - aborting`	A property or command-line option is pointing to a non-existent rules file.	Check that the command-line option or properties file refers to the correct rules file.

`Field "field name" not found in status table`
 	
`No matching field found for "field name"`
	The rules file being used refers to a field of the format @fieldname which does not exist in the status table.	Check the rules file and correct the problem.

Table 44. Fatal level probe messages (continued)

Message	Description	Action
`Unknown data type returned from ObjectServer`	The ObjectServer has returned unknown data.	See your support contract for information about contacting IBM Software Support.

Error-level messages

The probe is likely to terminate when an error message is issued.

Table 45. Error level probe messages

Message	Description	Action

`Can't set generic property "property name" via command line`
 	
`Property "property name" for option "option name" does not exist`
	An option in the probe is not mapped correctly to a property.	Check the properties file for the named property and see the probe publication for supported properties.

`Could not send alert`	The probe was unable to send an alert (usually an internal alert) to the ObjectServer.	Check that the ObjectServer is available.
`Could not set "fieldname" field`	The probe was unable to set a field value. This may be because the ObjectServer tables have been modified so that default fields are no longer present.	Check if the ObjectServer tables have been modified.

`CreateAndSet failed`
 	
`CreateAndSet failed for attr: "element name"`
	The probe is unable to create an element.	See your support contract for information about contacting IBM Software Support.

`Error Setting SIGINT Handler`
 	
`Error Setting SIGQUIT Handler`
 	
`Error Setting SIGTERM Handler`
	The probe was unable to set up a signal handler for either an `INT` , `QUIT`, or `TERM`.	See your support contract for information about contacting the IBM Software Support.

`Failed to open file: "file name"`	A file referred to in the rules file (for example, with the `table` function) does not exist.	Check the rules file and ensure the file is available.
`Failed to open message log: "file name"`	The probe is unable to open the specified log file.	Check the command line or properties file and correct the problem.
`Failed to open Properties file: "properties file name"`	The probe is unable to open the properties file.	Check the properties file or command line to ensure the properties file is in the specified location.

`Failed to open Rules file: "rules file name"`
 	
`The rules file for the probe is not available or incorrectly specified.`
	The probe is unable to open the rules file.	Check the properties file or command line to ensure the rules file is in the specified location.

Table 45. Error level probe messages (continued)

Message	Description	Action

`No extraction data for "regexp" - missing ()'s?`
 	
`Regexp doesn't match for "string"`
	A regular expression being used in the `extract` function may be missing parentheses.
	
	The string data that is being used to extract may not match the regular expression.
	
	The `extract` function is unable to extract data.
		Check the rules file and correct the problem.

`Option "option name" used without argument`	The option used expects a value which has not been supplied.	Check the probe publication and the contents of the command line.

`OS Error: "error message"`
 	
`Procedure "procedure name": "error message"`
 	
`Server "server name": "error message"`
	There is an error in the Sybase connection. There should be a subsequent message from the probe which details the effect of this error.	See your support contract for information about contacting IBM Software Support.

`Properties file: "error description" at line "line no"`	There is an error in the format of the properties file.	Check the properties file at the specified line number and correct the problem.
`PropGetValue failed`	A required property has not been set.	Check the properties file.
`Regular Expression Error: "regexp"`	A regular expression is incorrectly formed in the rules file.	Check the rules file for the regular expression and correct the problem.

`Results processing failed`
 	
`Unexpected return from results processing`
 	
`Unexpected value during results processing`
	There is a problem with the ObjectServer.	See your support contract for information about contacting IBM Software Support.

`Rules file: "error description" at line "line no"`	There is an error in the rules file format or syntax.	Check the rules file at the specified line number and correct the problem.
`SendAlert failed`	The probe was unable to send an alert to the ObjectServer.	Check that the ObjectServer is available.
`SessionProcess failed`	The probe was unable to process the alert against the rules file.	See your support contract for information about contacting IBM Software Support.
`Unknown message level "message level string" - using WARNING level`	The properties file or command line specified a message level which is not supported.	Check the properties file or command line and use a supported message level (`debug`, `info`, `warning`, `error`, `fatal`).
`Unknown option: "option name"`	An option has been used on the command line to start the probe which is not supported by the probe.	Check the probe documentation and the contents of the command line.
`Unknown property "property name" - ignored`	A property specified in the properties file does not exist in the probe.	Check the properties file for the named property and see the probe publication for supported properties.

Warning-level messages

These messages are issued as warnings but should not cause the probe to terminate.

Table 46. Warning level probe messages

Message	Description	Action

`Failed to install Client Message Callback`
 	
`Failed to install Server Message Callback`
 	
`Failed to retrieve connection status - attempting to continue`
 	
`Results processing failed`
	There is a problem with the ObjectServer.	The probe will try to continue.

`Failed to set SYBASE in environment`	The probe was unable to override the SYBASE environment variable.	Check that the SYBASE environment variable is correctly set.
`New value for field "field name" truncated to "number" characters`	A string being copied into an alert field has had to be truncated to fit the field.	Check the rules file.
`Type mismatch for property "property name" - new value ignored`	A property has been set with the wrong data type.	Check the properties file or command line to ensure that the property is correctly set.

Information-level messages

This message is for information purposes.

Table 47. Information level probe messages

Message	Description	Action
`Using stderr for logging`	The probe was unable to open a log file.	No action required. The probe is writing messages to stderr.

Debug-level messages

Debug level messages provide information about the internal functions of the probe. These messages are aimed at probe developers but are listed here for completeness.

Table 48. Debug level probe messages

Message	Description	Action
`A value for "string" doesn't exist in lookup table "table name"`	A value requested from a lookup table is not available.	No action required. The function in the rules file returns an empty string.

Table 48. Debug level probe messages (continued)

Message	Description	Action

`Attempted to duplicate NULL string`
 	
`Attempted to free NULL pointer`
 	
`Attempted to realloc NULL pointer`
 	
`Failed to allocate memory (Requested size was "number" bytes)`
 	
`Failed to duplicate string`
 	
`Failed to reallocate memory block at address "hex address" (Requested size was "number" bytes)`
	An error or problem has occurred in the memory allocation or string handling components of the probe library.	No action required. The library handles the problem.

`Failed to allocate command structure`
 	
`Failed to allocate context structure`
 	
`Failed to bind column`
 	
`Failed to connect`
 	
`Failed to describe column`
 	
`Failed to fetch number of columns`
 	
`Failed to initialise Sybase internals: "number"`
 	
`Failed to send command`
 	
`Failed to set appname`
 	
`Failed to set command query`
 	
`Failed to set hostname`
 	
`Failed to set password`
 	
`Failed to set username`
 	
`Got a row fail - continuing`
 	
`No columns in result set`
	A problem or error has occurred at the Sybase or ObjectServer connection level.	N/A

`Failed to flush alerts before EXIT`
 	
`Problem during disconnect before EXIT`
 	
`Problem during session destruction before EXIT`
 	
`Problem during shutdown before EXIT`
	A problem has occurred during probe shutdown.	N/A

`New value for field "field name" is "value"`	A field value has been set.	N/A

Table 48. Debug level probe messages (continued)

Message	Description	Action
`OplInitialise() called more than once`	Multiple calls have been made to the OplInitialise C probe API function, which can only be called once.	N/A

ProbeWatch and TSMWatch messages

In some situations, a probe or TSM generates events of its own. These events can provide information (such as startup or shutdown messages) or identify problems.

A number of elements are common to all ProbeWatch and TSMWatch messages.

ProbeWatch and TSMWatch messages are processed in the rules file and converted into alerts like other events. The following table shows the elements that are common to ProbeWatch and TSMWatch events.

Table 49. Common ProbeWatch and TSMWatch elements

Element name	Description
Summary	Summary string, described in the following tables.
Node	Name of the node on which the probe or TSM is running.
Agent	Name of the probe or TSM.
Manager	ProbeWatch or TSMWatch.

The following table describes summary strings that are common to all probes and TSMs.

Table 50. Common ProbeWatch and TSMWatch summary strings

ProbeWatch/TSMWatch message	Description	Cause
`Going down ...`	The probe or TSM is shutting down.	The probe or TSM is running a shutdown routine.
`Running ...`	The probe or TSM has started running.	The probe or TSM has just been started.
`Unable to get events ...`	The probe or TSM encountered a problem while listening for events.	There was a problem initializing the connection or there was a license or connection failure after some events were received. See your support contract for information about contacting IBM Software Support.
`Rules file reread upon SIGHUP successful ...`	The probe successfully re-read its rules file on receipt of a SIGHUP signal.	The probe received a SIGHUP signal.
`Rules file reread upon SIGHUP failed ...`	The probe could not re-read its rules file on receipt of a SIGHUP signal.	The probe received a SIGHUP signal.
`Heartbeat ...`	Heartbeat event	Not applicable

See the individual probe publications for additional summary strings for each probe.

TSMWatch messages are in the same format as ProbeWatch messages. The following table describes summary strings that are common to all TSMs.

Table 51. Common TSMWatch summary strings

TSMWatch message	Description	Action

`Connection Attempted ...`
`Connection Succeeded ...`
`Connection Failed ...`
`Connection Timed out ...`
`Connection Lost ...`
	Messages relating to the establishment of a TCP/IP connection.	N/A

`Disconnection Attempted ...`
`Disconnection Succeeded ...`
`Disconnection Failed ...`
	Messages relating to relinquishing a TCP/IP connection.	N/A

`Wakeup Attempted ...`
`Wakeup Succeeded ...`
`Wakeup Failed ...`
	Messages relating to wake up functionality.	N/A

`Login Attempted ...`
`Login Succeeded ...`
`Login Timed out ...`
`Login Failed ...`
	Messages relating to host login.	N/A

`Logout Attempted ...`
`Logout Succeeded ...`
`Logout Timed out ...`
`Logout Failed ...`
	Messages relating to host logout.	N/A

`Heartbeat Sent ...`
`Heartbeat Received ...`
`Heartbeat Timed out ...`
	Messages relating to sending and receiving heartbeat messages to and from the host.	N/A

`Resynchronisation Attempted ...`
`Resynchronisation Succeeded ...`
`Resynchronisation Failed ...`
	Messages relating to synchronizing current alerts between the switch and Tivoli Netcool/OMNIbus.	N/A

Troubleshooting probes

This topic describes some of the common problems experienced by Tivoli Netcool/OMNIbus users and explains possible causes and solutions.

This troubleshooting information is divided into two areas:

- Common problem causes
- What to do if

Table 52. Troubleshooting probes

Area	Description
Common problem causes	This information contains a list of common problem causes. If you are unsure what your problem is, you should start by reading this part and following the instructions. If you cannot solve your problem by following the instructions in this part, move on to the "What to do if" information.
What to do if	This information describes common symptoms caused by probe problems and step-by-step instructions to help you locate and solve the problem. If none of the headings match the symptoms of your problem, read through the lists of instructions and make sure that you have tried all of the most likely solutions listed there.

Common problem causes

The most common causes of probe problems are:

- Incorrectly set OMNIHOME environment variable
- Errors in the rules file, particularly in `extract` statements
- Configuration errors in the properties file

For information about setting the OMNIHOME environment variable, see the *IBM Tivoli Netcool/OMNIbus Installation and Deployment Guide*.

Check that all of the properties are set correctly in the probe properties file. For example, check that the **`Server`** property contains the correct ObjectServer or proxy server name and that the **`RulesFile`** property contains the correct rules file name.

If you cannot solve the problem, read through the next section and make sure that you have tried all of the most likely solutions listed there.

What to do if

The headings in this topic describe the most common symptoms of probe problems. Find the heading that most closely describes your problem and follow the instructions until you have located the cause and solved the problem.

If none of the headings match the symptoms of your problem, read through the lists of instructions and make sure that you have tried all of the most likely solutions listed there. If you have tried all of the suggested problem solutions and your probe still does not work, contact IBM Software Support.

The probe does not start

If the probe does not start:

1. Run the probe in debug mode.
2. Check that the ObjectServer is running by trying to connect using the **nco_sql** utility.

   If you can connect successfully, the ObjectServer is running. If the ObjectServer is not running, this is likely to be the cause of the problem.
3. Check that there are no other probes running with the same configuration using the following command:

   ```
   ps -ef | grep nco_p
   ```

   A list of probe processes is displayed. Check that none of the processes correspond to the same type of probe. You cannot run two identical probe configurations because this duplicates all of the events forwarded to the ObjectServer.
4. Check that you are using the correct probe for the current version of the target software.
5. Check that there are no syntax errors in the rules file.
6. Check that your system has not run out of system resources and can launch more processes. You can do this using `df -k` or `top`. See the `df` and `top` man pages for more information about using these commands.
7. Check to see if the `$OMNIHOME/var/`*probename*`.saf` store-and-forward file exists. If it exists, check that it has not become too large. If your disk is full, the probes and ObjectServers are not able to work properly.

   Attention: Store and forward is not designed to handle very large numbers of events. Left unattended, a store-and-forward file will continue to grow until it runs out of disk space.
8. Check that the store-and-forward file has not been corrupted. If the store-and-forward file has been corrupted there should be an error message in the log file (`$OMNIHOME/log/`*probename*`.log`). If the file is corrupted, delete it and restart the probe.
9. Check that the probe binary you are trying to run is the correct one for the current architecture by entering:

   ```
   $OMNIHOME/bin/arch/probename -version
   ```

   Check that the probe version matches your system architecture.

   If you are running the probe on a remote host:
10. Check that the probe host can connect to the ObjectServer host using the `ping` command. Try to ping the ObjectServer host machine using the hostname and the IP address. See the `ping` man page for more information about how to do this.

    If you cannot connect to the ObjectServer host using the `ping` command, there is a problem with the connection between your probe host and your ObjectServer host.
11. Check that the ObjectServer has been configured correctly in the Server Editor (**nco_xigen**) and that the interfaces information has been distributed to the ObjectServer and probe hosts.
12. Check to see if there is a firewall between the probe host and the ObjectServer host. If there is, make sure that the firewall allows traffic between the probe and the ObjectServer.

Related tasks:

"Testing rules files" on page 58
You can test the syntax of a rules file by using the Probe Rules Syntax Checker, **nco_p_syntax**. This is more efficient than running the probe to test that the syntax of the rules file is correct.

"Debugging rules files" on page 59
When you change the rules file, add new rules, or create lookup tables, it is useful to test the probe by running it in debug mode. Debug mode shows how an event is being parsed by the probe and can uncover any problems with the rules file.

Related reference:

Chapter 6, "Common probe properties and command-line options," on page 113
A number of properties and command-line options are common to all probes and TSMs.

The probe is not sending alerts to the ObjectServer

If the probe is not sending alerts to the ObjectServer:

1. Check that the probe is running by entering:

   ```
   ps -ef | grep nco_p
   ```

   A list of probe processes is displayed. If the probe is not running, start the probe from the command line.
2. Check that there are no other probes running with the same configuration by entering:

   ```
   ps -ef | grep nco_p
   ```

   A list of probe processes is displayed. Check that none of the processes correspond to the same type of probe. You cannot run two identical probe configurations because this duplicates all of the events forwarded to the ObjectServer.
3. Read the probe properties file and check that all of the properties are set correctly. For example, check that the **Server** property contains the correct ObjectServer name and that the **RulesFile** property contains the correct rules file name.
4. Check that the probe event source has events to send to the ObjectServer.
5. Check that the ObjectServer you are logged in to is the same ObjectServer that the probe is forwarding events to.
6. Check that the event source you are trying to probe is working correctly. See the documentation supplied with your element manager for more information about how to do this.
7. Check that you are using the correct probe.
8. Check that the probe is not running in store-and-forward mode. To do this, check the `$OMNIHOME/var/probename.saf` and `$OMNIHOME/var/probename.reco` files to see if they are growing. If they are, disable store-and-forward mode.
9. Check that your system has not run out of system resources and can launch more processes. You can do this using `df -k` or `top`. See the `df` and `top` man pages for more information about using this command.
10. Check for any `discard` functions in the probe rules file. The `discard` function must be in a conditional statement; otherwise, all events are discarded.

    If you are running the probe on a remote host:
11. Check that the probe host can connect to the ObjectServer host using the `ping` command. Try to ping the ObjectServer host machine using the hostname and the IP address. See the `ping` man page for more information about how to do this.

If you cannot connect to the ObjectServer host using the `ping` command, there is a problem with the connection between your probe host and your ObjectServer host.

12. Check that the ObjectServer has been configured correctly through the Server Editor (**nco_xigen**) and that the interfaces information has been distributed to the ObjectServer and probe hosts.
13. Check to see if there is a firewall between the probe host and the ObjectServer host. If there is, make sure that the firewall allows traffic between the probe and the ObjectServer.

Related concepts:

"Store-and-forward mode for probes" on page 11
Probes can continue to run if the target ObjectServer is down. During this period, the probe switches to *store* mode. The probe reverts to *forward* mode when the ObjectServer is functional again.

The probe is losing events

If not all of the events are being forwarded to the ObjectServer:

1. Run the probe in debug mode.
2. Check that the event source you are trying to probe is working correctly. See the documentation supplied with your element manager for more information about how to do this.
3. Check that the probe event source has events to send to the ObjectServer.
4. Check that all of the properties in the properties file are set correctly. For example, check that the **Server** property contains the correct ObjectServer name and that the **RulesFile** property contains the correct rules file name.
5. Check for any `discard` functions in the probe rules file. The `discard` function discards events based on specified conditions.

Related tasks:

"Debugging rules files" on page 59
When you change the rules file, add new rules, or create lookup tables, it is useful to test the probe by running it in debug mode. Debug mode shows how an event is being parsed by the probe and can uncover any problems with the rules file.

The probe is consuming too much CPU time

If the probe is consuming too much CPU time:

1. Run the probe in debug mode.
2. Check that the probe can connect to the event source.
3. Check to see if the `$OMNIHOME/var/`*probename*`.saf` store-and-forward file exists. If it exists, check that it has not become too large. If your disk is full, the probes and ObjectServer will not be able to work properly.

   Attention: Store and forward is not designed to handle very large numbers of alerts. Left unattended, a store-and-forward file will continue to grow until it runs out of disk space.
4. Check that the store-and-forward file has not been corrupted. If the store-and-forward file has been corrupted there should be an error message in the probe log file (`$OMNIHOME/log/`*probename*`.log`). If the store-and-forward file is corrupted, delete it and restart the probe.

The event list is not being populated properly

If the probe is detecting events and forwarding them to the ObjectServer but the event list fields are not being populated correctly:

1. Run the probe in debug mode.
2. Check that fields which are not being populated properly are being correctly mapped to elements in the rules file.
3. Check that it is not a GUI problem by querying the alerts.status table using ObjectServer SQL.

Appendix B. Common gateway error messages

A number of error messages are common to all gateways. The *gateway_name* in each error message refers to the individual gateway name and indicates which gateway generated the error.

Table 53. Common gateway error messages

Error	Description	Action
Gateway_name Writer: HashAlloc failure in *_gateway_name* CacheAdd(). *Gateway_name* Writer: MemStrDup() failure in *_gateway_name* CacheAdd().	The gateway failed to allocate memory.	Try to free more memory.
Gateway_name Writer: Failed to allocate memory. *Gateway_name* Writer *writer_name*: Memory allocation failed. *Gateway_name* Writer: Memory allocation failure. *Gateway_name* Writer: Memory allocation error. *Gateway_name* Writer: Memory reallocation error. Failed to allocate memory in writer *writer_name*.	The gateway failed to allocate memory.	Try to free more memory.
Gateway_name Writer *writer_name*: Could not create serial cache - memory problems. *Gateway_name* Writer *writer_name*: Failed to allocate memory for a GPCModule handle.	The gateway failed to allocate memory.	Try to free more memory.
Gateway_name Writer: Failed to lock connection mutex.	The writer failed to lock the ObjectServer feedback connection in order to access the connection and feed back problem ticket data for the associated alert. This lock failure may be due to insufficient resources or as a result of the underlying threading system preventing a deadlock between multiple threads that are contending for the resource.	Refer to your support contract for information about contacting the helpdesk.
Gateway_name Writer: Failed to re-acquire alert details from OS.	This error message comes from the gateway cache reclamation subsystem. This message indicates that the gateway failed to re-acquire the trouble ticket number and reclaim its internal cache entry from the ObjectServer.	Refer to your support contract for information about contacting the helpdesk.

Table 53. Common gateway error messages (continued)

Error	Description	Action
Gateway_name Writer: Invalid datatype for problem number feedback field.	The data type is invalid.	Refer to the *IBM Tivoli Netcool/OMNIbus Administration Guide* for information about data types.
Gateway_name Writer: Serial x already in serial Cache. Cannot add.	The gateway tried to add a serial number that already exists.	Refer to your support contract for information about contacting the helpdesk.
Gateway_name Writer: Serial x not found in serial cache. Cannot Delete.	The gateway could not delete this alert because it has already been deleted in Tivoli Netcool/OMNIbus.	You do not need to do anything.
Gateway_name Writer *writer_name*: Failed to construct path to *gateway_name* Read/Write Module.	The gateway could not locate the reader or writer module application.	Check that the module is installed in the correct location.
Gateway_name Writer *writer_name*: Failed to construct the argument list for *gateway_name* Module.	Failed to construct the argument list for gateway module.	Check that the arguments in the configuration file are set correctly.
Gateway_name Writer *writer_name*: GPCModule creation failed.	Failed to create the GPCModule due to insufficient memory.	Try to free more memory.
Gateway_name Writer *writer_name*: Failed to start the OS-*gateway_name* Writer. *Gateway_name* Writer *writer_name*: Failed to start the *gateway_name*-OS Reader.	Failed to start the ObjectServer gateway reader or writer module.	Check that the module is installed in the correct location and that the file permissions are set correctly.
Gateway_name Writer *writer_name*: Failed to shutdown *gateway_name* Writer.	Failed to stop gateway writer module.	Check the writer log file for more information.
Gateway_name Writer *writer_name*: Failed to construct path to *gateway_name* Read/Write Module.	Failed to construct the path to the gateway reader or writer module application.	Check that the module is installed in the correct location and that the file permissions are set correctly.
Gateway_name Writer *writer_name*: Failed to find the *gateway_name* Read/Write Module [x].	Cannot find the module binary.	Check that the module is installed in the correct location and that the file permissions are set correctly.
Gateway_name Writer *writer_name*: Incorrect permissions on the *gateway_name* module binary [x].	The module's file permissions are set incorrectly.	Check that the module is installed in the correct location and that the file permissions are set correctly.
Gateway_name Writer *writer_name*: Failed to create the Serial Cache Mutex.	The gateway writer failed to create the necessary data protection structure for the internal serial number cache due to insufficient resources. This is generally due to insufficient memory.	Try to free more memory.
Gateway_name Writer *writer_name*: Failed to create the Conn Mutex.	The gateway writer failed to create the necessary data protection structure for the ObjectServer connection due to insufficient resources.	Try to free more memory.
Gateway_name Writer *writer_name*: Failed to start the *gateway_name*-to-OS service thread.	The gateway failed to spawn the service thread.	Check that the gateway can access the ObjectServer.

Table 53. Common gateway error messages (continued)

Error	Description	Action
Gateway_name Writer *writer_name*: Failed to send a shutdown request to the *gateway_name* Writer.	The gateway did not shut down cleanly.	Check the writer log file for more information.
Failed to install SIGCHLD handler. Failed to install SIGPIPE handler.	The gateway failed during handler installation.	Refer to your support contract for information about contacting the helpdesk.
No <mapname> attribute for *gateway_name* writer *writer_name*.	The gateway could not find the map name.	Check the configuration file.
<mapname> attribute is not a name for *gateway_name* writer *writer_name*.	Incorrect writer name given.	Check the configuration file.
A MAP called <map> does not exist for *gateway_name* writer *writer_name*.	The gateway could not find the specified map.	Check the configuration file.
MAP <map> is invalid for *gateway_name* writer *writer_name*.	The given map is not valid.	Check the configuration file.
Map <map> is not the journal map and cannot contain the <journal map name> map item in *gateway_name* Writer *writer_name*.	If this map is not the journal map, then the JOURNAL_MAP_NAME attribute is set incorrectly.	Check the JOURNAL_MAP_NAME attribute in the gateway configuration file.
Gateway_name Writer: Failed to send *gateway_name* Event to the *gateway_name* Writer module.	The gateway failed to send a given event.	Check the log files for more information.
Gateway_name Writer: Failed to wait for return from the *gateway_name* Writer module.	There was an error in retrieving the success statement.	Check the log files for more information.
Gateway_name Writer: Failed to read the status return message from the *gateway_name* Writer module.	The gateway failed to retrieve the status of a module.	Check the log files for more information.
Gateway_name Writer: Failed to send event to *gateway_name*.	The module failed to send the event to gateway.	Check the log files for more information.
Gateway_name Writer: *gateway_name* Writer Module experienced Fatal Error.	There was a fatal error.	Check the log files for more information.
Gateway_name Writer: Failed to send event to *gateway_name*. Unknown type.	The gateway received unexpected type.	Refer to your support contract for information about contacting the helpdesk.
Gateway_name Writer: Failed to build serial index.	The gateway failed to build indexes.	Check that the Serial column exists in the ObjectServer alerts.status table.
Incorrect data type for the Serial column.	The gateway did not receive the correct data type.	Check that the data type for the Serial column in the ObjectServer alerts.status table is an integer.
Gateway_name Writer: Failed to build server serial index.	The gateway failed to get the server serial index.	Check that the ServerSerial column exists in the ObjectServer alerts.status table.
Incorrect data type for the Server Serial column.	The gateway did not receive the correct data type.	Check that the data type for the ServerSerial column in the ObjectServer alerts.status table is an integer.

Table 53. Common gateway error messages (continued)

Error	Description	Action
Gateway_name Writer: Failed to build server name index.	The gateway failed to get the server name index.	Check that the ServerName column exists in the ObjectServer alerts.status table.
Incorrect data type for the Server Name column.	The gateway did not receive the correct data type.	Check that the data type for the ServerName column in the ObjectServer alerts.status table is a string.
Gateway_name Writer: Failed to find field <fieldnumber> in *gateway_name* Event.	The gateway could not find the field number it was looking for.	Refer to your support contract for information about contacting the helpdesk.
Gateway_name Writer: Invalid field name for expansion on action SQL [<field>].	The gateway received an invalid field name.	Refer to the *IBM Tivoli Netcool/OMNIbus Administration Guide* for information about ObjectServer SQL.
Gateway_name Writer: Unenclosed field expansion request in action SQL [<sql action>].	The gateway did not find an enclosing bracket.	Check the action.sql file.
Gateway_name Writer: Failed to turn counter-part notification back-on. Fatal error in *gateway_name*-to-OS Feedback. *Gateway_name* Writer: Failed to turn counter-part notification off. *Gateway_name*-to-OS Feedback failed.	The gateway failed to send a notify command.	This is an internal error. Refer to your support contract for information about contacting the helpdesk.
Gateway_name Writer: Failed to send SQL command to ObjectServer. *Gateway_name*-to-OS Feedback failed.	The gateway failed to send the SQL command to the ObjectServer.	Check the ObjectServer log file.
Failed to find the column <column_name> in map <map_name>.	The gateway failed to find the given column.	Check that the given column name is entered correctly in the configuration file and that it is shown in the ObjectServer alerts.status table.
Gateway_name Writer: Failed to lock the cache mutex.	The writer failed to lock the ObjectServer feedback connection in order to access the connection and feed back problem ticket data changes for the associated alert.	This lock failure may be due to insufficient resources or as a result of the underlying threading system preventing a deadlock between multiple threads that are contending for the resource.
Failed to find cached problem ticket for serial <serial number> using map <map name>.	The gateway failed to find the specified cache problem ticket number.	Check that the specified ticket was originally created by the gateway.
Gateway_name Writer: Failed to unlock the cache mutex.	After access to the cache, an attempt to unlock the data structures protection lock failed. This message indicates that the gateway is in a position which will lead to a deadlock situation.	Refer to your support contract for information about contacting the helpdesk.
Gateway_name Writer: Cache add error.	The gateway could not add the serial to the serial cache due to insufficient resources.	Try to free more memory.

Table 53. Common gateway error messages (continued)

Error	Description	Action
Gateway_name Writer *writer_name*: Failed to create *gateway_name* Event for journal update.	The gateway failed to create the journal event update.	Check the writer log file.
Gateway_name Writer *writer_name*: Failed to send journal update event to *gateway_name*.	The gateway failed to send journal event update.	Check the writer log file.
<attribute name> attribute is not a string for *gateway_name* writer *writer_name*.	An attribute in the writer was of an incorrect data type.	Check the writer definition in the configuration file.
No <attribute name> attribute for *gateway_name* writer *writer_name* given.	The gateway failed to find the attribute.	Add the attribute to the writer definition in the configuration file.
Gateway_name Writer *writer_name*: Failed to find the <counterpart attribute> attribute for the writer. This is necessary due to bi-directional nature.	An attempt to find the necessary counterpart attribute failed.	Check the configuration file.
Gateway_name Writer *writer_name*: Is not a name for an Object Server reader.	The gateway found an incorrect data type.	Check the configuration file.
Gateway_name Writer *writer_name*: Reader <reader> was not found for counter part.	The reader was not found.	Check the counterpart configuration in the configuration file.
Gateway_name Writer *writer_name*: Failed to send SKIP Command.	This command failed to disable IDUC on a bidirectional connection.	Refer to your support contract for information about contacting the helpdesk.
Connection to feedback server failed.	The gateway failed to make a connection.	Check the ObjectServer log file.
Failed to set the death call on the feedback connection.	The gateway failed to set the necessary property.	This is an internal error. Refer to your support contract for information about contacting the helpdesk.
Writer counterpart error.	The gateway failed to find the counterpart attribute for gateway writer.	Check the counterpart configuration in the configuration file.
Gateway_name Writer: Failed to stat() the action SQL file "*filename*".	The gateway failed to stat the file in order to determine its size.	Check the file access permissions for the specified action file.
Gateway_name Writer: Empty action SQL file "*filename*".	File "*filename*" is empty.	Check the action SQL file.
Gateway_name Writer: Failed to open the action SQL file "*filename*".	The gateway failed to open the file.	Check the file permissions.
Gateway_name Writer: Failed to read the action SQL file "*filename*".	The gateway failed to read the file.	Check the file permissions.
Gateway_name Writer: No Action SQL find in file "*filename*".	There is no action SQL in the file.	Check the file.
Gateway_name Writer *writer_name*: Failed to read the conversions table.	The gateway failed to read the conversions table.	Check the file permissions.

Table 53. Common gateway error messages (continued)

Error	Description	Action
Gateway_name Writer: Failed to find PM %s in cache for return PMO event.	The gateway has received a Problem Management Open return event from gateway for the problem ticket. When an attempt was made to look up the problem ticket number in the writer's cache, in order to determine the serial number of the ticket's associated alert, no record could be reclaimed or found.	Refer to your support contract for information about contacting the helpdesk.
Gateway_name Writer: Open Feedback Failed.	The gateway failed to construct the open action SQL statement or send the SQL action command to the server.	Check the ObjectServer SQL file.
Gateway_name Writer: No Update action SQL for *gateway_name* Update event.	There is no update action SQL statement.	Check the configuration file.
Gateway_name Writer: Failed to find PM %s in cache for return PMU event.	The gateway has received a Problem Management Update return event from gateway for the problem ticket. When an attempt was made to look up the problem ticket number in the writer's cache in order to determine the serial number of the ticket's associated alert, no record could be reclaimed or found.	Refer to your support contract for information about contacting the helpdesk.
Gateway_name Writer: Update Feedback Failed.	The gateway failed to construct the open action SQL statement or send the SQL action command to the server.	Check the ObjectServer log file.
Gateway_name Writer: Failed to find PM %s in cache for return PMJ event.	The gateway has received a Problem Management Journal return event from gateway for the problem ticket. When an attempt was made to look up the problem ticket number in the writer's cache in order to determine the serial number of the ticket's associated alert, no record could be reclaimed or found.	Refer to your support contract for information about contacting the helpdesk.
Gateway_name Writer: Journal Feedback Failed.	The gateway failed to construct the open action SQL statement or send the SQL action command to the server.	Check the ObjectServer log file.
Gateway_name Writer: Failed to find PM %s in cache for return PMC event.	The gateway has received a Problem Management Close return event from gateway for the problem ticket. When an attempt was made to look up the problem ticket number in the writer's cache in order to determine the serial number of the ticket's associated alert, no record could be reclaimed or found.	Refer to your support contract for information about contacting the helpdesk.

Table 53. Common gateway error messages (continued)

Error	Description	Action
Gateway_name Writer: Close Feedback Failed.	The gateway failed to construct the open action SQL statement or send the SQL action command to the server.	Check the ObjectServer log file.
Received error code <code> from Reader/Writer Module - [<message>].	The gateway received an error message.	Check the module log files.
Gateway_name Writer: Failed to read *gateway_name* event from *gateway_name* Reader Module.	The gateway failed to read the event sent by the gateway reader module.	Check the reader log files.
Gateway_name Writer: Received event of type <event type> which was unexpected.	The gateway received an unknown event type.	Refer to your support contract for information about contacting the helpdesk.
Gateway_name Writer: Received invalid known message from Reader/Writer Module for this system.	The gateway received an invalid known message.	Refer to your support contract for information about contacting the helpdesk.
Gateway_name Writer: Received unknown message from Reader/Writer Module.	The gateway received an invalid unknown message.	Refer to your support contract for information about contacting the helpdesk.
Gateway_name Writer: Failed to block on data feed from *gateway_name* Reader Module.	The gateway failed to block due to a shutdown request. This message is displayed when the gateway is shutting down.	Refer to your support contract for information about contacting the helpdesk.
Gateway_name Writer: Fatal thread termination. Stopping gateway.	A thread exited unexpectedly.	Check the gateway log files.
<attribute name> attribute is not a string for *gateway_name* writer *writer_name* - IGNORED	An attribute name is not recognized. The gateway will ignore it.	Check the gateway log files.
<attribute name> attribute must be set to TRUE or FALSE for writer *writer_name*.	An attribute name has not been set to TRUE or FALSE.	Check the gateway configuration file.
Gateway_name Writer *writer_name*: Failed to shutdown *gateway_name* Reader/Writer Modules.	The gateway failed to shut down the reader and writer modules.	Check the module log file.
Gateway_name Writer *writer_name*: Failed to disconnect feedback connection.	The disconnect of feedback channel failed.	Check the ObjectServer log file.
Failed to create *gateway_name* event structure for a problem management open event in writer *writer_name*.	The gateway writer failed to allocate a gateway event structure for a problem management open event due to insufficient memory resources.	Try to free more memory.
Gateway_name Writer: FEEDBACK FAILED!!	The gateway failed to store the problem number.	Check the ObjectServer log file.
Failed to create journal for *gateway_name* writer *writer_name* (from INSERT)	The gateway failed to create journal.	Check the writer log file.

Table 53. Common gateway error messages (continued)

Error	Description	Action
`Failed to create gateway_name event structure for a problem management update event in writer writer_name.`	The gateway writer failed to allocate a gateway event structure for a problem management update event due to insufficient memory resources.	Try to free more memory.
`Gateway_name Writer writer_name: Failed to delete problem ticket from cache for serial <serial number>.`	The gateway failed to delete serial number from cache.	This is an internal error. You can ignore it.
`Failed to create gateway_name event structure for a PMC event in writer writer_name.`	The gateway writer failed to allocate a gateway event structure for a Problem Management Close event due to insufficient memory resources.	Try to free more memory.

Appendix C. Regular expressions

Tivoli Netcool/OMNIbus supports the use of regular expressions in search queries that you perform on ObjectServer data. Regular expressions are sequences of *atoms* that are made up of normal characters and metacharacters.

An atom is a single character or a pattern of one or more characters in parentheses. Normal characters include uppercase and lowercase letters, and numbers. Metacharacters are non-alphabetic characters that possess special meanings in regular expressions.

Two types of regular expression libraries are available for use with the ObjectServer:

- NETCOOL: This library is useful for single-byte character processing.
- TRE: This library enables use of the POSIX 1003.2 extended regular expression syntax, and provides support for both single-byte and multi-byte character languages. When the UTF-8 encoding is enabled on Windows, only the characters within Unicode plane 0, the Basic Multilingual Plane (BMP), are supported in regular expression patterns. Any character outside of the BMP, which is found in the pattern, will result in an error. The matching strings for the regular expression pattern can contain any UTF-8 character.

  Note: Use of the TRE library can lead to a marked decrease in system performance. Optimal system performance is achieved with the NETCOOL library.

You can use the ObjectServer property **`RegexpLibrary`** to specify which library should be used for regular expression matching. The NETCOOL regular expression library is enabled by default.

NETCOOL regular expression library

If your system supports single-byte character languages, you can use the NETCOOL regular expression library to run search queries on your data. The NETCOOL library provides better system performance than the TRE regular expression library.

Note: When a regular expression is written in SQL, the SQL parser first processes string literals before passing them to the regular expression library. The SQL parser processes backslash sequences, so single backslashes will not subsequently be seen by the regular expression library. Therefore, when you write a regular expression in SQL, use double backslashes to escape reserved characters.

For example, to escape the parentheses in the string `'1_(22)'`, use this expression: `'1_\\(22\\)'`.

The NETCOOL regular expression library supports the use of normal characters and metacharacters. The following table describes the set of metacharacters supported by the NETCOOL regular expression library.

Table 54. Metacharacters

Metacharacter	Description	Examples
*	Matches zero or more instances of the preceding atom. Matches as many instances as possible.	goo* matches my godness, my goodness, and my gooodness, but not my gdness.
+	Matches one or more instances of the preceding atom. Matches as many instances as possible.	goo+ matches my goodness and my gooodness, but not my godness.
?	Matches zero or more instances of the preceding atom.	goo? matches my godness, my goodness, and my gooodness, but not my gdness. colou?r matches color and colour. end-?user matches enduser and end-user.
$	Matches the end of the string.	end$ matches the end, but not the ending.
^	Matches the beginning of the string.	^severity matches severity level 5, but not The severity is 5.
.	Matches any single character.	b.at matches baat, bBat, and b4at, but not bat or bB4at.
[abcd]	Matches any character in the square brackets.	[nN][oO] matches no, nO, No, and NO. gr[ae]y matches both spellings of the word 'grey'; that is, gray and grey.
[a-d]	Matches any character in the range of characters separated by a hyphen (-).	[0-9] matches any decimal digit. [ab3-5] matches a, b, 3, 4, and 5. ^[A-Za-z]+$ matches any string that contains only upper or lowercase characters.
[^abcd] [^a-d]	Matches any character except those in the square brackets or in the range of characters separated by a hyphen (-).	[^0-9] matches any string that does not contain any numeric characters.
()	Indicates that the characters within the parentheses should be treated as a character pattern.	A(boo)+Z matches AbooZ, AboobooZ, and AbooboobooZ, but not AboZ or AboooZ. Jan(uary)? matches Jan and January.
|	Matches one of the atoms on either side of the pipe character.	A(B|C)D matches ABD and ACD, but not AD, ABCD, ABBD, or ACCD. (AB | CD) matches AB and CD, but not ABD and ACD.
\	Indicates that the metacharacter following should be treated as a regular character. The metacharacters listed in this table require a backslash escape character as a prefix to switch off their special meaning.	* matches the * character. \\ matches the \ character. \. matches the . character. \[[0-9]*\] matches an opening square bracket, followed by any digits or spaces, followed by a closed bracket.

Related concepts:

"TRE regular expression library"
Use the TRE regular expression library to run search queries on both single-byte and multi-byte character languages.

TRE regular expression library

Use the TRE regular expression library to run search queries on both single-byte and multi-byte character languages.

The TRE regular expression library supports usage of the POSIX 1003.2 extended regular expression syntax in the form of:

- Metacharacters
- Minimal or non-greedy quantifiers
- Bracket expressions
- Constructs for multicultural support
- Backslash sequences

Restriction: A marked decrease in system performance might be observed when using this library.

Related reference:

"NETCOOL regular expression library" on page 223
If your system supports single-byte character languages, you can use the NETCOOL regular expression library to run search queries on your data. The NETCOOL library provides better system performance than the TRE regular expression library.

Metacharacters

Metacharacters are non-alphabetic characters that possess special meanings in regular expressions.

The set of metacharacters that can be used in extended regular expression syntax is as follows:

```
* + ? $ ^ . () | \ {} [
```

The following table describes all of these metacharacters except the square bracket [metacharacter. You can use the [metacharacter to construct bracket expressions.

Table 55. Metacharacters

Metacharacter	Description	Examples
*	Matches zero or more instances of the preceding atom. Matches as many instances as possible.	goo* matches my godness, my goodness, and my gooodness, but not my gdness.
+	Matches one or more instances of the preceding atom. Matches as many instances as possible.	goo+ matches my goodness and my gooodness, but not my godness.

Table 55. Metacharacters (continued)

Metacharacter	Description	Examples
?	Matches zero or more instances of the preceding atom.	goo? matches my godness, my goodness, and my gooodness, but not my gdness. colou?r matches color and colour. end-?user matches enduser and end-user.
$	Matches the end of the string.	end$ matches the end, but not the ending.
^	Matches the beginning of the string. The ^ metacharacter can also be used in bracket expressions.	^severity matches severity level 5, but not The severity is 5.
.	Matches any single character.	b.at matches baat, bBat, and b4at, but not bat or bB4at.
()	Indicates that the characters within the parentheses should be treated as a character pattern.	A(boo)+Z matches AbooZ, AboobooZ, and AbooboobooZ, but not AboZ or AboooZ. Jan(uary)? matches Jan and January.
|	Matches one of the atoms on either side of the pipe character.	A(B|C)D matches ABD and ACD, but not AD, ABCD, ABBD, or ACCD. (AB | CD) matches AB and CD, but not ABD and ACD.
\	Indicates that the metacharacter following should be treated as a regular character. The metacharacters listed in this section require a backslash escape character as a prefix to switch off their special meaning. The \ metacharacter can also be used to construct backslash sequences.	* matches the * character. \\ matches the \ character. \. matches the . character.
{m , n}	Matches from m to n instances of the preceding atom, where m is the minimum and n is the maximum. Matches as many instances as possible. **Note:** m and n are unsigned decimal integers between 0 and 255.	f{1,2}ord matches ford and fford. N/{1,3}A matches N/A, N//A, and N///A, but not NA or N////A.
{m ,}	Matches m or more instances of the preceding atom.	Z{2,} matches two or more repititions of Z.
{m}	Matches exactly m instances of the preceding atom.	a{3} matches aaa. 1{2} matches 11.

Related reference:

"Bracket expressions" on page 228
Bracket expressions can be used to match a single character or collating element.

"Backslash sequences" on page 230
When constructing regular expressions, the backslash character can be used in a variety of ways.

Minimal or non-greedy quantifiers

Regular expressions are generally considered *greedy* because an expression with repetitions will attempt to match as many characters as possible. The asterisk (*), plus (+), question mark (?), and curly braces ({}) metacharacters exhibit 'repetitious' behavior, and attempt to match as many instances as possible.

To make a subexpression match as few characters as possible, a question mark (?) can be appended to these metacharacters to make them *minimal* or *non-greedy*. The following table describes the non-greedy quantifiers.

Table 56. Minimal/non-greedy quantifiers

Quantifier	Description	Examples
`*?`	Matches zero or more instances of the preceding atom. Matches as few instances as possible.	Given an input string of `Netcool Tool Library`: • The first group in `^(.*l).*$` matches `Netcool Tool .` • The first group in `^(.*?l).*$` matches `Netcool`.
`+?`	Matches one or more instances of the preceding atom. Matches as few instances as possible.	Given an input string of `little`: • `.*?l` matches `l`. • `^.+l` matches `littl`.
`??`	Matches zero or one instance of the preceding atom. Matches as few instances as possible.	`.??b` matches `ab` in `abc`, and `b` in `bbb`. `.?b` matches `ab` in `abc`, and `bb` in `bbb`.
`{m , n} ?`	Matches from `m` to `n` instances of the preceding atom, where `m` is the minimum and `n` is the maximum. Matches as few instances as possible. **Note:** `m` and `n` are unsigned decimal integers between `0` and `255`.	Given an input string of `Netcool Tool Cool Fool Library`: • `^((.*?ool)*).*$` matches `Netcool Tool Cool Fool`. • `^((.*?ool)+).*$` matches `Netcool Tool Cool Fool`. • `^((.*?ool)+?).*$` matches `Netcool`. • `^((.*?ool){2,5}).*$` matches `Netcool Tool Cool Fool`. • `^((.*?ool){2,5}?).*$` matches `Netcool Tool`. • `^((.*?ool){2,5}) [FL].*$` matches `Netcool Tool Cool Fool`. • `^((.*?ool){2,5}?) [FL].*$` matches `Netcool Tool Cool`.

Table 56. Minimal/non-greedy quantifiers (continued)

Quantifier	Description	Examples
`{m ,} ?`	Matches `m` or more instances of the preceding atom. Matches as few instances as possible.	Given an input string of `Netcool Tool Cool Fool Library`: • `^((.*?ool){2,}).*$` matches `Netcool Tool Cool Fool`. • `^((.*?ool){2,}?).*$` matches `Netcool Tool`. • `^((.*?ool){2,}) [FL].*$` matches `Netcool Tool Cool Fool`. • `^((.*?ool){2,}?) [FL].*$` matches `Netcool Tool Cool`.

Bracket expressions

Bracket expressions can be used to match a single character or collating element.

The following table describes how to use bracket expressions.

Table 57. Bracket expressions

Expression	Description	Examples
`[abcd]`	Matches any character in the square brackets.	`[nN][oO]` matches `no`, `nO`, `No`, and `NO`. `gr[ae]y` matches both spellings of the word 'grey'; that is, `gray` and `grey`.
`[a-d]`	Matches any character in the range of characters separated by a hyphen (`-`).	`[0-9]` matches any decimal digit. `[ab3-5]` matches `a`, `b`, `3`, `4`, and `5`. `[0-9]{4}` matches any four-digit string. `^[A-Za-z]+$` matches any string that contains only upper or lowercase characters. `\[[0-9 ]*\]` matches an opening square bracket, followed by any digits or spaces, followed by a closed bracket.

`[^abcd]`
 	
`[^a-d]`
	Matches any character except those in the square brackets or in the range of characters separated by a hyphen (`-`).	`[^0-9]` matches any string that does not contain any numeric characters.

`[.ab.]`	Matches a multi-character collating element.	`[.ch.]` matches the multi-character collating sequence `ch` (if the current language supports that collating sequence).
`[=a=]`	Matches all collating elements with the same primary sort order as that element, including the element itself.	`[=e=]` matches `e` and all the variants of `e` in the current locale.

Note the following points:

- The caret character (^) only has a special meaning when included as the first character after the open bracket ([). Otherwise, it is treated as a normal character.

- The hyphen character (-) is treated as a normal character only under either of the following conditions:
  - The hyphen character is the first or last character within the square brackets, for example, `[ab-]` or `[-xy]`.
  - The hyphen character is the only (both first and last) character; that is, `[-]`.
- To match a closing square bracket within a bracketed expression, the closing bracket must be the first character within the enclosing brackets; for example, `[][xy]` matches `]`, `[`, `x`, and `y`.
- Other metacharacters are treated as normal characters within square brackets, and do not need to be escaped; for example, `[ca$]` will match `c`, `a`, or `$`.

Related reference:

"Metacharacters" on page 225
Metacharacters are non-alphabetic characters that possess special meanings in regular expressions.

Constructs for multicultural support

The sort order of characters (and any of their variants) is locale-dependent, so different regular expressions are generally required to match characters of the same class, in different locales. To facilitate multicultural support, a set of predefined names enclosed in `[:` and `:]` can be used to represent characters of the same class.

The set of valid names depends on the value of the LC_CTYPE environment variable of the current locale, but the names shown in the following table are valid in all locales.

Table 58. Multicultural constructs

Construct	Description
`[:alnum:]`	Matches any alphanumeric character.
`[:alpha:]`	Matches any alphabetic character.
`[:blank:]`	Matches any blank character - that is, space and TAB.
`[:cntrl:]`	Matches any control characters; these are non-printable.
`[:digit:]`	Matches any decimal digits.
`[:graph:]`	Matches any printable character except space.
`[:lower:]`	Matches any lowercase alphabetic character.
`[:print:]`	Matches any printable character including space.
`[:punct:]`	Matches any printable character that is not a space or alphanumeric; that is, punctuation.
`[:space:]`	Matches any whitespace character.
`[:upper:]`	Matches any uppercase alphabetic character.
`[:xdigit:]`	Matches any hexadecimal digit.

Example: Multicultural constructs

`[[:lower:]AB]` matches the lowercase letters and uppercase `A` and `B`.

`[[:space:][:alpha:]]` matches any character that is either whitespace or aphabetic.

`[[:alpha:]]` matches to `[A-Za-z]` in the English locale (`en`), but would include accented or additional letters in another locale.

Backslash sequences

When constructing regular expressions, the backslash character can be used in a variety of ways.

The backslash character (\) can be used to:

- Turn off the special meaning of metacharacters so they can be treated as normal characters.
- Include non-printable characters in a regular expression.
- Give special meaning to some normal characters.
- Specify backreferences. *Backreferences* are used to specify that an earlier matching subexpression is matched again later.

Note: The backslash character cannot be the last character in a regular expression.

When a regular expression is written in SQL, the SQL parser first processes string literals before passing them to the regular expression library. The SQL parser processes backslash sequences, so single backslashes will not subsequently be seen by the regular expression library. Therefore, when you write a regular expression in SQL, use double backslashes to escape reserved characters.

For example, to escape the parentheses in the string `'1_(22)'`, use this expression: `'1_\\(22\\)'`.

The following table describes how to specify backslash sequences for non-printable characters and backreferences. This table also shows how to use backslash sequences to apply special meaning to some normal characters.

Table 59. Backslash sequences

Backslash sequence	Description
`\a`	Matches the bell character (ASCII code 7).
`\e`	Matches the escape character (ASCII code 27).
`\f`	Matches the form-feed character (ASCII code 12).
`\n`	Matches the new-line or line-feed character (ASCII code 10).
`\r`	Matches the carriage return character (ASCII code 13).
`\t`	Matches the horizontal tab character (ASCII code 9).
`\v`	Matches the vertical tab character.
`\<`	Matches the beginning of a word, or the beginning of an identifier, defined as the boundary between non-alphanumerics and alphanumerics (including underscore). This matches no characters, only the context.
`\>`	Matches the end of a word or identifier.
`\b`	Matches a word boundary; that is, matches the empty string at the beginning or end of an alphanumeric sequence. Enables a 'whole words only' search.
`\B`	Matches a non-word boundary; that is, matches the empty string not at the beginning or end of a word.
`\d`	Matches any decimal digit. Equivalent to `[0-9]` and `[[:digit:]]`.

Table 59. Backslash sequences (continued)

Backslash sequence	Description
\D	Matches any non-digit character. Equivalent to [^0-9] or [^[:digit:]].
\s	Matches any whitespace character. Equivalent to [\t\n\r\f\v] or [[:space:]].
\S	Matches any non-whitespace character. Equivalent to [^ \t\n\r\f\v] or [^[:space:]].
\w	Matches a word character; that is, any alphanumeric character or underscore. Equivalent to [a-zA-Z0-9_] or [[:alnum:]_].
\W	Matches any non-alphanumeric character. Equivalent to [^a-zA-Z0-9_] or [^[:alnum:]_].
\[1-9]	A backslash followed by a a single non-zero decimal digit n is termed a *backreference*. Matches the same set of characters matched by the nth parenthesized subexpression.

Example backslash constructs

\bcat\b matches cat but not cats or bobcat.

\d\s matches a digit followed by a whitespace character.

[\d\s] matches any digit or whitespace character.

.([XY]).([XY]). matches aXbXc and aYbYc, but also aXbYc and aYbXc. However, .([XY]).\1. will only match aXbXc and aYbYc.

Related reference:

"Metacharacters" on page 225
Metacharacters are non-alphabetic characters that possess special meanings in regular expressions.

Appendix D. ObjectServer tables and data types

This appendix contains ObjectServer database table information.

alerts.status table

The alerts.status table contains status information about problems that have been detected by probes.

Note: You can display only columns of type CHAR, VARCHAR, INCR, INTEGER, and TIME in the event list. Do not add columns of any other type to the alerts.status table.

The following table describes the columns in the alerts.status table.

Table 60. Columns in the alerts.status table

Column name	Data type	Mandatory	Description

Identifier	varchar(255)	Yes	Controls ObjectServer deduplication. The Identifier field controls the deduplication feature of the ObjectServer, and also supports compatibility with the GenericClear automation by ensuring resolution events are properly inserted into the ObjectServer and not deduplicated with their respective problem events.
			
			The following identifier correctly identifies repeated events in a typical environment:
			`@Identifier=@Node+" "+@AlertKey+" "+@AlertGroup+" "+@Type+" "+@Agent+" "+@Manager`
			
			Additional information might need to be appended to the Identifier field to ensure correct deduplication and compatibility with the GenericClear automation. For example, if an SNMP specific trap contains a status enumeration value in one of its variable bindings, the specific trap number and the value of the relevant varbind must be appended to the Identifier field as follows:
			`@Identifier=@Node +" "+ @AlertKey+" "+@AlertGroup+" "+@Type+" "+@Agent+" "+@Manager+" "+$specific-trap+" "+$2`

Serial	incr	Yes	The Tivoli Netcool/OMNIbus serial number for the row.
Node	varchar(64)	Yes	Identifies the managed entity from which the alarm originated. This could be a device or host name, service name, or other entity. For IP network devices or hosts, the Node column contains the resolved name of the device or host. In cases where the name cannot be resolved, the Node column must contain the IP address of the device or host. For non-IP network devices or hosts, alarms must contain similar information to the IP device or host. That is, the Node column must contain the name of the device or host which allows direct communication, or can be resolved to allow direct communication, with the device or host.

Table 60. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
NodeAlias	varchar(64)	No	The alias for the node. For network devices or hosts, this should be the logical (layer-3) address of the entity. For IP devices or hosts, this must be the IP address. For non-IP devices or hosts, there are several addressing schemes that could be used. When selecting a value for the NodeAlias field, the value should allow for direct communication with the device or host. For example, a device managed by TL-1. The NodeAlias field may be populated by a lookup table or Netcool/Impact policy, with the IP address and port number of the terminal server through which the TL-1 device can be reached.
Manager	varchar(64)	Yes	The descriptive name of the probe that collected and forwarded the alarm to the ObjectServer. This can also be used to indicate the host on which the probe is running. Ideally this is set in the properties file of the probe, however the rules file should check to ensure it is set correctly, and modify if necessary. For example, the following syntax can be used to define the Manager field: `@Manager="MTTrapd Probe on" + hostname()`
Agent	varchar(64)	No	The descriptive name of the sub-manager that generated the alert. Probes which process SNMP traps must set the Agent field to either the name of the vendor or the standards body which defined the trap, and provide a description of the MIB, or MIB Definition Name, where the trap is defined. It must be presented in the following format: `vendor-MIB description` For example:: `Cisco-Accounting Control, Cisco-Health Monitor, IETFBRIDGEMIB, ATMF-ATM-FORUM-MIB` Optionally, vendor-specific information, such as device model numbers, can be appended to the Agent field for vendor-specifc implementations of standard traps. The Syslog probe should set the Agent field to the name of the vendor which defined the received message, and provide any logical description for the family of messages to which the received message belongs. For example, Cisco defines messages received from IOS-based devices in separate documentation from messages received from the PIX Firewall. The format of the messages is also slightly different. Therefore the Syslog messages received from Cisco will have the Agent field set to either `Cisco-IOS` or `Cisco- PIX Firewall`. The TL-1 TSM should set the Agent field to the name of the vendor which defined the received message, and provide any logical description for the family of messages to which the received message belongs.

Table 60. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
AlertGroup	varchar(255)	No	The descriptive name of the failure type indicated by the alert. For example: `Interface Status` or `CPU Utilization`). The AlertGroup field must contain the same value for related problem and resolution events. For example, SNMP trap 2 (linkDown) and trap 3 (linkUp) must both contain the same AlertGroup value of Link Status. The AlertGroup field for a TL-1 message will be set to the value of the message's alarm type.

Table 60. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description

AlertKey	varchar(255)	Yes	The descriptive key that indicates the managed object instance referenced by the alert. For example, the disk partition indicated by a file system full alert or the switch port indicated by a utilization alert.
			
			Probes that process SNMP traps should set the AlertKey field to one of the following values (in order of preference):
			• The SNMP instance of the managed object which is represented by the alarm. This is normally obtained by extracting the instance from the OID of one of the variable bindings of the trap. Additionally, it might also be contained in a combination of one or more of the trap's variable binding values. For example, the first variable binding of a linkDown trap contains the ifIndex value (interface number) of the interface which failed. The AlertKey can be set with either of the following:
			– `@AlertKey = extract($OID1, "\.([0-9]+)$")`
			– `@AlertKey = $1`
			• A textual description of the instance derived from the trap name or trap description. For example, a device with two power supplies (A and B) might be able to send two separate specific traps, without variable bindings, to indicate the failed status of either power supply. The appropriate power supply instance would need to be derived from the trap definitions of the MIB and then encoded in the rules file:
			`switch($specific-trap)`
			`{`
			`case "1":`
			`@AlertKey = "A"`
			`case "2":`
			`@AlertKey = "B"`
			`default:`
			`}`
			• A mixed combination of variable binding values and information derived from the trap name or trap description. Any instance information that is not available for the previous two values, but that is required to ensure correct deduplication and GenericClear compatibility, is suitable.
			
			The Syslog Probe should set the AlertKey to a textual description of the instance derived from the log message text. Ideally this is a textual name of the same managed entity. For example:
			`Nov 20 13:12:57 device.customer.net`
			`195.180.208.193 19986: 37w0d: %LINK-3-UPDOWN:`
			`Interface FastEthernet0/13, changed state to down`
			
			In this example, the AlertKey would be set to `FastEthernet0/13` using the following syntax:
			`@AlertKey = extract($Details, "Interface`
			`(.*), changed")`
			
			Typically the AlertKey field for a TL-1 message is set to the value of the message's alarm location.

Table 60. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
Severity	integer	Yes	Indicates the alert severity level, which indicates how the perceived capability of the managed object has been affected. The color of the alert in the event list is controlled by the severity value: 0: Clear. The Clear severity level indicates that one or more previously reported alarms has been cleared. The alarms have either been cleared manually by a network operator, or automatically by a process which has determined the fault condition no longer exists. Automatic processes, for example the GenericClear Automation process, typically clear all alarms for a managed object (the AlertKey) that have the same Alarm Type and/or probable cause (the Alert Group). 1: Indeterminate. The Indeterminate severity level indicates that the severity level cannot be determined. Additionally, all problem resolving alarms are initially defined as indeterminate until they have been correlated with problem indicating alarms (for example by the GenericClear Automation), when all correlated alarms are set to Clear. 2: Warning. The Warning severity level indicates the detection of potential or impending service affecting faults. If necessary, a further investigation of the fault should be made to prevent it from becoming more serious. 3: Minor. The Minor severity level indicates the existence of a non-service affecting fault condition. Corrective action should be taken to prevent it from becoming a more serious fault. This severity level may be reported, for example, when the detected alarm condition is not currently degrading the capacity of the managed object. 4: Major. The Major severity level indicates that a service affecting condition has developed and corrective action is urgently required. This severity level may be reported, for example, when there is a severe degradation in the capability of the managed object, and its full capability must be restored. 5: Critical. The Critical severity level indicates that a service affecting condition has occurred, and corrective action is immediately required. This severity level may be reported, for example, when a managed object is out of service, and its capability must be restored.

Table 60. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description

Summary	varchar(255)	Yes	Contains text which describes the alarm condition and the affected managed object instance.
			• You must ensure that the information presented in the Summary field is concise and sufficiently detailed.
			• The Summary field must contain, in parenthesis, a description of the managed object instance provided by the available alarm data. For example, a linkDown trap from a Cisco device will contain the `ifDescr` value in the 2nd variable binding. The text summary of such an event would be similar to:
			`"Link Down ( FastEthernet0/13 )"`
			• For alarms that relate to thresholds containing the compared or threshold values, you should select one of the following formats based on the available data:
			– No values provided:
			`"Link Utilization High ( BRI2/0:1 )"`
			– Compared value name provided:
			`"Link Utilization High: inOctets Exceeded Threshold ( BRI2/0:1)"`
			– Compared value name and value provided:
			`"Link Utilization High: inOctets, 7100, Exceeded Threshold (BRI2/0:1 )"`
			– Threshold name provided:
			`"Link Utilization High: inOctetsMax Exceeded ( BRI2/0:1 )"`
			– Threshold Value provided:
			`"Link Utilization High: inOctetsMax, 7000, Exceeded ( BRI2/0:1)"`
			– Compared value and threshold value provided:
			`"Link Utilization High: 7100 Exceeded 7000 ( BRI2/0:1 )"`
			– Both names and values provided:
			`"Link Utilization High: inOctets, 7100, Exceeded inOctetsMax,7000 ( BRI2/0:1 )"`

StateChange	time	Yes	An automatically-maintained ObjectServer timestamp of the last insert or update of the alert from any source.
FirstOccurrence	time	Yes	The time in seconds (from midnight January 1, 1970) when this alert was created or when polling started at the probe.
LastOccurrence	time	Yes	The time when this alert was last updated at the probe.
InternalLast	time	Yes	The time when this alert was last updated at the ObjectServer.
Poll	integer	No	The frequency of polling for this alert in seconds.

Table 60. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description

Type	integer	No	The type of alarm, where type refers to the problem or resolution state of the Alarm. This field is important for the correct correlation of events by the GenericClear Automation. The following values are valid for the Type field:
			
			0: Type not set
			
			1: Problem
			
			2: Resolution
			
			3: Netcool/Visionary problem
			
			4: Netcool/Visionary resolution
			
			7: Netcool/ISMs new alarm
			
			8: Netcool/ISMs old alarm
			
			11: More Severe
			
			12: Less Severe
			
			13: Information
			
			Some scenarios cannot be categorized as either a Problem or Resolution. For example, events which are increasingly becoming an issue but do not currently represent a failure, and events which are becoming less of an issue but do not currently indicate the failure has been completely resolved. In which case, the Type field must be set to Problem, More Severe or Less Severe to maintain compatibility with the GenericClear Automation.
			
			For example, the following rule file logic is used for handling traps associated with BGP Peer Connection Status:
			`switch ($bgpPeerState)`
			`{`
			`case "1": ### idle`
			`@Severity = 4`
			`@Type = 1`
			`case "2": ### connect`
			`@Severity = 2`
			`@Type = 12`
			`case "3": ### active`
			`@Severity = 2`
			`@Type = 12`
			`case "4": ### opensent`
			`@Severity = 2`
			`@Type = 12`
			`case "5": ### openconfirm`
			`@Severity = 2`
			`@Type = 12`
			`case "6": ### established`
			`@Severity = 1`
			`@Type = 2`
			`default:`
			`@Severity = 2`
			`@Type = 1`
			`}`

Table 60. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
Tally	integer	Yes	Automatically-maintained count of the number of inserts and updates of the alert from any source. This count is affected by deduplication.
Class	integer	Yes	The alert class used to identify the probe or vendor from which the alert was generated. Controls the applicability of context-sensitive event list tools.
Grade	integer	No	Indicates the state of escalation for the alert: `0`: Not Escalated `1`: Escalated
Location	varchar(64)	No	Indicates the physical location of the device, host, or service for which the alert was generated.
OwnerUID	integer	Yes	The user identifier of the user who is assigned to handle this alert. The default is `65534`, which is the identifier for the `nobody` user.
OwnerGID	integer	No	The group identifier of the group that is assigned to handle this alert. The default is `0`, which is the identifier for the `public` group.
Acknowledged	integer	Yes	Indicates whether the alert has been acknowledged: `0`: No `1`: Yes Alerts can be acknowledged manually by a network operator or automatically by a correlation or workflow process.
Flash	integer	No	Enables the option to make the event list flash.
EventId	varchar(255)	No	The event ID (for example, `SNMPTRAP-link down`). Multiple events can have the same event ID. The event ID is populated by the probe rules file and used by IBM Tivoli Network Manager IP Edition.
ExpireTime	integer	Yes	The number of seconds from the time this alert was last received by the ObjectServer (LastOccurence) until it is cleared automatically. Used by the Expire automation.
ProcessReq	integer	No	Indicates whether the alert should be processed by IBM Tivoli Network Manager IP Edition. This is populated by the probe rules file and used by IBM Tivoli Network Manager IP Edition.

Table 60. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
Suppress Escl	integer	Yes	Used to suppress or escalate the alert: 0: Normal 1: Escalated 2: Escalated-Level 2 3: Escalated-Level 3 4: Suppressed 5: Hidden 6: Maintenance The suppression level is manually selected by operators from the event list.
Customer	varchar(64)	No	The name of the customer affected by this alert.
Service	varchar(64)	No	The name of the service affected by this alert.
PhysicalSlot	integer	No	The slot number indicated by the alert.
PhysicalPort	integer	No	The port number indicated by the alert.
Physical Card	varchar(64)	No	The card name or description indicated by the alert.
TaskList	integer	Yes	Indicates whether a user has added the alert to the Task List: 0: No 1: Yes Operators can add alerts to the Task List from the event list.
NmosSerial	varchar(64)	No	The serial number of the event that is suppressing the current event. Populated by IBM Tivoli Network Manager IP Edition.
NmosObj Inst	integer	No	Populated by IBM Tivoli Network Manager IP Edition during alert processing.
NmosCause Type	integer	No	The alert state, populated by IBM Tivoli Network Manager IP Edition as an integer value: • 0: Unknown • 1: Root cause • 2: Symptom
Nmos Domain Name	varchar(64)	No	The name of the IBM Tivoli Network Manager IP Edition domain that is managing the event. By default, this column is populated only for events that are generated by IBM Tivoli Network Manager IP Edition polls. To populate this column for other event sources such as probes, you must modify the rules files.

Table 60. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
Nmos EntityId	integer	No	A unique numerical ID that identifies the IBM Tivoli Network Manager IP Edition topology entity with which the event is associated. This column is similar to the NmosObjInst column, but is more granular. For example, the NmosEntityId value can represent the ID of an interface within a device.
Nmos Managed Status	integer	No	The managed status of the network entity for which the event was raised. Can apply to events from IBM Tivoli Network Manager IP Edition and from any probe. You can use this column to filter out events from interfaces that are not considered relevant.
NmosEvent Map	varchar(64)	No	Contains the required IBM Tivoli Network Manager IP Edition V3.9 or later, eventMap name and optional precedence for the event, which indicates how IBM Tivoli Network Manager IP Edition should process the event. The optional precedence number can be concatenated to the end of the value, following a period (.). If the precedence is not supplied, it is set to 0. The following examples show the configuration for an event map with an explicit event precedence of 900, and another where the precedence defaults to 0: • ItnmLinkdownIfIndex.900 • PrecisionMonitorEvent
LocalNode Alias	varchar(64)	Yes	The alias of the network entity indicated by the alert. For network devices or hosts, this is the logical (layer-3) address of the entity, or another logical address that enables direct communication with the device. For use in managed object instance identification.
LocalPriObj	varchar(255)	No	The primary object referenced by the alert. For use in managed object instance identification.
LocalSecObj	varchar(255)	No	The secondary object referenced by the alert. For use in managed object instance identification.
LocalRoot Obj	varchar(255)	Yes	An object that is equivalent to the primary object referenced in the alarm. For use in managed object instance identification.
Remote Node Alias	varchar(64)	Yes	The network address of the remote network entity. For use in managed object instance identification.
RemotePri Obj	varchar(255)	No	The primary object of a remote network entity referenced by an alarm. For use in managed object instance identification.
RemoteSec Obj	varchar(255)	No	The secondary object of a remote network entity referenced by an alarm. For use in managed object instance identification.
Remote RootObj	varchar(255)	Yes	An object that is equivalent to the remote entity's primary object referenced in the alarm. For use in managed object instance identification.

Table 60. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
X733 EventType	integer	No	Indicates the alert type: 0: Not defined 1: Communications 2: Quality of Service 3: Processing error 4: Equipment 5: Environmental 6: Integrity violation 7: Operational violation 8: Physical violation 9: Security service violation 10: Time domain violation
X733 Probable Cause	integer	No	Indicates the probable cause of the alert.
X733 Specific Prob	varchar(64)	No	Identifies additional information for the probable cause of the alert. Used by probe rules files to specify a set of identifiers for use in managed object instance identification.
X733 CorrNotif	varchar(255)	No	A listing of all notifications with which this notification is correlated.
ServerName	varchar(64)	Yes	The name of the originating ObjectServer. Used by gateways to control propagation of alerts between ObjectServers.
ServerSerial	integer	Yes	The serial number of the alert on the originating ObjectServer (if it did not originate on this ObjectServer). Used by gateways to control the propagation of alerts between ObjectServers.
URL	varchar(1024)	No	Optional URL which provides a link to additional information in the vendor's device or ENMS.

Table 60. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description

Extended Attr	varchar(4096)	No	Holds name-value pairs (of Tivoli Enterprise Console® extended attributes) or any other additional information for which no dedicated column exists in the alerts.status table.
			
			Use this column only through the `nvp_get`, `nvp_set`, and `nvp_exists` SQL functions.
			
			An example of a name-value string is:
			`Region="EMEA";host="sf01392w";`
			`Error="errno=32: ""Broken pipe"""`
			
			In this example, the `Region` attribute has a value of `EMEA`, the `host` attribute has a value of `sf01392w`, and the `Error` attribute has a value of `errno=32: "Broken pipe"`.
			
			Notice that quotation marks are escaped by doubling them, as shown with the `Error` attribute value.
			
			In name-value pairs, the value is always enclosed in quotation marks (" ") and embedded quotation marks are escaped by doubling them. The separator between name-value pairs is a semicolon (;). No whitespace is allowed around the equal sign (=) or semicolon.
			Note: The column can hold only 4096 bytes, so there will be fewer than 4096 characters if multi-byte characters are used.

OldRow	integer	No	Maintains the local state of the row in each ObjectServer during resynchronization in the failover pair. This column must not be added to the gateway mapping files. The value of OldRow is changed to 1 in the destination ObjectServer for the duration of resynchronization if the **`Gate.ResyncType`** property of the gateway is set to `Minimal`. The default is `0`.
ProbeSub SecondId	integer	No	For those alerts that a probe sends within the same one-second interval, and which therefore have the same LastOccurrence value, an incremental value, starting at 1, is added to the ProbeSubSecondId field to differentiate the LastOccurrence time. The default is `0`.
MasterSerial	integer	No	Identifies the master ObjectServer if this alert is being processed in a desktop ObjectServer environment. This column is added when you run the database initialization utility **`nco_dbinit`** with the `-desktopserver` option. **Note:** MasterSerial must be the last column in the **`alerts.status`** table if you are using a desktop ObjectServer environment.
BSM_ Identity	varchar(1024)	No	The unique identifier of the resource from where the event originates, and is used to correlate the event to that resource in IBM Tivoli Business Service Manager (TBSM).

alerts.details table

The alerts.details table contains the detail attributes of the alerts in the system.

The following table describes the columns in the alerts.details table.

Table 61. Columns in the alerts.details table

Column name	Data type	Description
KeyField	varchar(255)	Internal sequencing string for uniqueness. The Keyfield value is composed of an Identifer value plus four # plus a sequence number starting at a count of 1; for example: *Identifier####1* Where *Identifier* is a data type of varchar(255), which is used to relate details to entries in the alerts.status table. If the Identifier value is over a certain length, there is a possibility that the Keyfield value could exceed its defined 255 limit, resulting in truncation of the sequence number. Keyfield values could therefore no longer be unique, and the unintended duplication could cause inserts into the alerts.details table to fail. **Tip:** To prevent an overflow in KeyField (and ensure uniqueness), the length of the Identifier value must be sufficiently less than 255 to allow the four # and a sequence number (of one or more digits) to be appended.
Identifier	varchar(255)	Identifier to relate details to entries in the alerts.status table. The Identifier is used to compute the Keyfield value, and is required to be less than a certain length to ensure that each computed Keyfield value remains unique. For guidelines on the maximum length of the Identifier value, see the tip in the preceding KeyField row.
AttrVal	integer	Boolean; when false (0), just the Detail column is valid. Otherwise, the Name and Detail columns are both valid.
Sequence	integer	Sequence number, used for ordering entries in the event list Event Information window.
Name	varchar(255)	Name of attribute stored in the Detail column.
Detail	varchar(255)	Attribute value.

alerts.journal table

The alerts.journal table provides a history of work performed on alerts.

The following table describes the columns in the alerts.journal table.

Table 62. Columns in the alerts.journal table

Column name	Data type	Description
KeyField	varchar(255)	Primary key for table.
Serial	integer	Serial number of alert that this journal entry is related to.
UID	integer	User identifier of user who made this entry.
Chrono	time	Time and date that this entry was made.
Text1	varchar(255)	First block of text for journal entry.

Table 62. Columns in the alerts.journal table (continued)

Column name	Data type	Description
Text2	varchar(255)	Second block of text for journal entry.
Text3	varchar(255)	Third block of text for journal entry.
Text4	varchar(255)	Fourth block of text for journal entry.
Text5	varchar(255)	Fifth block of text for journal entry.
Text6	varchar(255)	Sixth block of text for journal entry.
Text7	varchar(255)	Seventh block of text for journal entry.
Text8	varchar(255)	Eighth block of text for journal entry.
Text9	varchar(255)	Ninth block of text for journal entry.
Text10	varchar(255)	Tenth block of text for journal entry.
Text11	varchar(255)	Eleventh block of text for journal entry.
Text12	varchar(255)	Twelfth block of text for journal entry.
Text13	varchar(255)	Thirteenth block of text for journal entry.
Text14	varchar(255)	Fourteenth block of text for journal entry.
Text15	varchar(255)	Fifteenth block of text for journal entry.
Text16	varchar(255)	Sixteenth block of text for journal entry.

service.status table

The service.status table is used to control the additional features required to support IBM Tivoli Composite Application Manager for Internet Service Monitoring.

The following table describes the columns in the service.status table.

Table 63. Columns in the service.status table

Column name	Data type	Description
Name	varchar(255)	Name of the service.
CurrentState	integer	Indicates the state of the service: 0: Good 1: Bad 2: Marginal 3: Unknown
StateChange	time	Indicates the last time the service state changed.
LastGoodAt	time	Indicates the last time the service was Good (0).
LastBadAt	time	Indicates the last time the service was Bad (1).
LastMarginalAt	time	Indicates the last time the service was Marginal (2).
LastReportAt	time	Time of the last service status report.

registry.probes table

The registry.probes table is used to track dynamic runtime information about probes. When a probe connects to the ObjectServer, it registers information about itself in the registry.probes table. The probe controls what data is entered into the table.

If you have two or more instances of a probe running on one computer, and each instance has the same name, only one instance will be registered in the registry.probes table. To enable registration of all the instances of a probe running on the same computer, you must use unique values for each probe's **Name** property.

The registry.probes table is a virtual table. Because probes update the table when they connect to the ObjectServer, the data in the table does not need to persist when the ObjectServer shuts down.

The following table describes the columns in the registry.probes table.

Table 64. Columns in the registry.probes table

Column name	Data type	Description
Name	varchar(128)	The value of the **Name** property in the probe's properties file.
Hostname	varchar(64)	The fully qualified domain name (FQDN) of the computer that the probe is running on.
PID	integer	The probe's current process ID (PID).
Status	integer	Indicates the status of the probe: 0: The probe has shut down. 1: The probe is running.
HTTP_port	integer	The port number on which the HTTP interface of the probe is listening. The probe properties **NHttpd.EnableHTTP** and **NHttpd.ListeningPort** must be enabled for this port to be active. When the port is not active, the default value of this field is 0.
HTTPS_port	integer	The port number on which the HTTPS interface of the probe is listening. The probe properties **NHttpd.SSLEnable** and **NHttpd.SSLListeningPort** must be enabled for this port to be active. When the port is not active, the default value of this field is 0.
StartTime	time	The time at which the probe started up. This information enables you to determine whether the probe is starting up or is reconnecting.
ProbeType	varchar(128)	A string representation of the type of probe connecting to the ObjectServer, for example, “simnet” or “tivoli_eif”.

Table 64. Columns in the registry.probes table (continued)

Column name	Data type	Description
ConnectionID	integer	The connection ID assigned to the probe when it connects to the ObjectServer. This corresponds to the connection ID stored in the catalog.connections table. This column is populated by the registry_new_probe ObjectServer trigger. **Note:** When a probe is connected to the ObjectServer through a proxy server, the connection ID of the probe can change over time and it might therefore be registered incorrectly. This is because the proxy server optimizes its ObjectServer connections and dynamically shuffles probe connections around. However, the connection ID stored in the registry.probes table remains the same. It is not updated when a probe is moved to another connection on the same proxy server. A workaround for this problem is to not use a proxy server in multitiered deployments.
LastUpdate	time	The time stamp of the most recent update to the registry.probes table. This column is populated by the registry_new_probe ObjectServer trigger.

ObjectServer data types

Each column value in the ObjectServer has an associated data type. The data type determines how the ObjectServer processes the data in the column.

For example, the plus operator (+) adds integer values or concatenates string values, but does not act on Boolean values. The data types supported by the ObjectServer are listed in the following table:

Table 65. ObjectServer data types

SQL type	Description	Default value	ObjectServer ID for data type
INTEGER	32-bit signed integer.	`0`	0
INCR	32-bit unsigned auto-incrementing integer. Applies to table columns only, and can only be updated by the system.	`0`	5
UNSIGNED	32-bit unsigned integer.	`0`	12
BOOLEAN	`TRUE` or `FALSE`.	`FALSE`	13
REAL	64-bit signed floating point number.	`0.0`	14
TIME	Time, stored as the number of seconds since midnight January 1, 1970. This is the Coordinated Universal Time (UTC) international time standard.	`Thu Jan 1 01:00:00 1970`	1
CHAR(*integer*)	Fixed size character string, *integer* characters long (8192 Bytes is the maximum). The char type is identical in operation to varchar, but performance is better for mass updates that change the length of the string.	`''`	10

Table 65. ObjectServer data types (continued)

SQL type	Description	Default value	ObjectServer ID for data type
VARCHAR(*integer*)	Variable size character string, up to *integer* characters long (8192 Bytes is the maximum). The varchar type uses less storage space than the char type and the performance is better for deduplication, scanning, insert, and delete operations.	''	2
INTEGER64	64 bit signed integer.	0	16
UNSIGNED64	64 bit unsigned integer.	0	17

Note: You can display only columns of type CHAR, VARCHAR, INCR, INTEGER, and TIME in the event list. Do not add columns of any other type to the alerts.status table.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
958/NH04
IBM Centre, St Leonards
601 Pacific Hwy
St Leonards, NSW, 2069
Australia

IBM Corporation
896471/H128B
76 Upper Ground
London SE1 9PZ
United Kingdom

IBM Corporation
JBF1/SOM1
294 Route 100
Somers, NY, 10589-0100
United States of America

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the

names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Portions of this product include software developed by Daniel Veillard.

- `libxml2-2.7.8`

The `libxml2-2.7.8` software is distributed according to the following license agreement:
© Copyright 1998-2003 Daniel Veillard.
All Rights Reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE DANIEL VEILLARD BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
Except as contained in this notice, the name of Daniel Veillard shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization from him.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

AIX, IBM, the IBM logo, ibm.com®, Informix, Netcool, System z, Tivoli, and Tivoli Enterprise Console are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Adobe, Acrobat, Portable Document Format (PDF), PostScript, and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

Printed in the Republic of Ireland

SC14-7530-02