IBM Tivoli Composite Application Manager for Transactions
V7.4.0.1
for AIX, Linux, Solaris, Windows, and z/OS

SDK Guide

SC14-7410-03

Note
Before using this information and the product it supports, read the information in "Notices" on page 91.

This edition applies to V7.4 of IBM Tivoli Composite Application Manager for Transactions (product number 5724-S79) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

About this publication

This guide provides information about instrumenting applications to provide tracking information for Transaction Tracking, and enabling the development of third-party modules for decoding and processing network protocols by Web Response Time.

Intended audience

This guide is for system administrators who enable applications to send events to Transaction Tracking, and those who want to decode multiple protocols with Web Response Time.

Use the information in the IBM Tivoli Composite Application Manager for Transactions *User's Guide* and *Administrator's Guide* together with the *IBM Tivoli Monitoring User's Guide* to monitor and manage the performance of your systems.

Publications

This section lists publications relevant to the use of the IBM Tivoli Composite Application Manager for Transactions. It also describes how to access Tivoli® publications online and how to order Tivoli publications.

Documentation library

The following documents are available in the IBM Tivoli Composite Application Manager for Transactions library:

- *IBM Tivoli Composite Application Manager for Transactions Administrator's Guide*

  This guide provides information about configuring elements of IBM Tivoli Composite Application Manager for Transactions.
- *IBM Tivoli Composite Application Manager for Transactions Installation and Configuration Guide*

  This guide provides information about installing and configuring elements of IBM Tivoli Composite Application Manager for Transactions.
- *IBM Tivoli Composite Application Manager for Transactions Quick Start Guide*

  This guide provides a brief overview of IBM Tivoli Composite Application Manager for Transactions.
- *IBM Tivoli Composite Application Manager for Transactions Troubleshooting Guide*

  This guide provides information about using all elements of IBM Tivoli Composite Application Manager for Transactions.
- *IBM Tivoli Composite Application Manager for Transactions SDK Guide*

  This guide provides information about the Transaction Tracking API.
- *IBM Tivoli Composite Application Manager for Transactions User's Guide*

  This guide provides information about the GUI for all elements of IBM Tivoli Composite Application Manager for Transactions.
- *IBM Tivoli Composite Application Manager for Transactions Installation and Configuration Guide for z/OS*

  This guide provides information about using IBM Tivoli Composite Application Manager for Transactions on z/OS.

Prerequisite publications

To use the information in this guide effectively, you must know about IBM Tivoli Monitoring products that you can obtain from the following documentation:

- *IBM Tivoli Monitoring Administrator's Guide*
- *IBM Tivoli Monitoring Installation and Setup Guide*
- *IBM Tivoli Monitoring User's Guide*

If you do not have IBM Tivoli Monitoring installed already you can do a basic IBM Tivoli Monitoring installation using the IBM Tivoli Monitoring Quick Start Guide as a guide.

See IBM Tivoli Monitoring Information Center for further information.

Accessing terminology online

The IBM® Terminology website consolidates the terminology from IBM product libraries in one convenient location.

You can access the Terminology website at the following web address:

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

Accessing publications online

IBM posts publications for all products, as they become available and whenever they are updated, to IBM Knowledge Center.

Access IBM Knowledge Center (http://www.ibm.com/support/knowledgecenter) using a browser.

Find supporting information on the Application Performance Management community (http://www.ibm.com/developerworks/servicemanagement/apm/index.html) and connect, learn, and share with experts.

Ordering publications

You can order many Tivoli publications online at the following website:

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:

1. Go to http://www.ibm.com/planetwide/.
2. In the alphabetic list, select the letter for your country and then click the name of your country. A list of numbers for your local representatives is displayed.

Accessibility

Accessibility features help users with a physical disability, such as restricted mobility or limited vision, to use software products. 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 most features of the graphical user interface.

For additional information, see Appendix F, "Accessibility," on page 89.

Tivoli technical training

For information about Tivoli technical training, see the following IBM Tivoli Education website:

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

Support information

If you have a problem with your IBM software, you want to resolve it quickly.

Online

Access the Tivoli Software Support site at http://www.ibm.com/software/sysmgmt/products/support/index.html?ibmprd=tivman. Access the IBM Software Support site at http://www.ibm.com/software/support/probsub.html .

IBM Support Assistant

The IBM Support Assistant is a free local software serviceability workbench that helps you resolve questions and problems with IBM software products. The Support Assistant provides quick access to support-related information and serviceability tools for problem determination. The IBM Support Assistant provides the following tools to help you collect the required information:

- Use the IBM Support Assistant Lite program to deploy the IBM Support Assistant data collection tool. This tool collects diagnostic files for your product.

  Tip: When you install the IBM Support Assistant data collection tool on 64-bit systems, use a 32-bit Java Runtime Environment to ensure that data collection functions as expected.
- Use the Log Analyzer tool to combine log files from multiple products in to a single view and simplify searches for information about known problems.

For information about installing the IBM Support Assistant software, see http://www.ibm.com/software/support/isa.

Troubleshooting Guide

For more information about resolving problems, see the *IBM Tivoli Composite Application Manager for Transactions Troubleshooting Guide*.

Conventions used in this guide

This guide uses several conventions for operating system-dependent commands and paths, special terms, actions, and user interface controls.

Typeface conventions

This guide 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

- Words defined in text
- Emphasis of words
- New terms in text (except in a definition list)
- Variables and values you must provide

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 guide uses the UNIX system convention for specifying environment variables and for directory notation.

When using the Windows command line, replace **$***variable* with **%***variable***%** for environment variables. Replace each forward slash (**/**) with a backslash (\) in directory paths. 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.

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

Variables

The following variables are used in this documentation:

$CANDLE_HOME
The default IBM Tivoli Monitoring installation directory. On UNIX systems, the default directory is `/opt/IBM/ITM`.

%CANDLE_HOME%
The default IBM Tivoli Monitoring installation directory. On Windows systems, the default directory is `C:\IBM\ITM`.

$ALLUSERSPROFILE
On UNIX systems, `/usr`

%ALLUSERSPROFILE%
On Windows 7 and 2008, the default directory is `C:\ProgramData`.

Chapter 1. Introduction

ITCAM for Transactions provides a default set of tracking and decoding information. You can use the Transaction Tracking API or the Web Response Time Module API to extend the capability of ITCAM for Transactions.

Transaction Tracking is a solution for tracking transactions across applications and networks. It provides an upgrade path from Response Time Tracking, and consolidates domain-specific tracking technologies. Transaction Tracking tracks applications by accepting information from applications, monitoring software and other sources that specify a point in the life of an application. Each piece of information is an *event*. Transaction Tracking Data collectors such as MQ Tracking automatically send these events to Transaction Tracking.

The Transaction Tracking Application Programming Interface (Transaction Tracking API), provides developers with a means of sending their own events and providing tracking information to Transaction Tracking. In this way, developers can enhance tracking beyond that provided by Transactions Data Collectors.

- Use the Transaction Tracking API to construct events and send those events to ITCAM for Transactions for processing.
- Use the Web Response Time Module API to develop third-party modules for decoding and processing network protocols.

Chapter 2. Transaction Tracking API

Before you start

Transaction Tracking API is supported on a range of operating systems and architectures. It supports a number of programming languages.

Platform support

Follow these steps for information about the supported operating systems and hardware architectures on which you can use Transaction Tracking API:

1. Link to the required ITCAM for Transactions version from ITCAM for Transactions on Documentation Central.
2. In the navigation pane, select **Composite Application Manager for Transactions** > **Prerequisites**.
3. In the System requirements and prerequisites page, select Transaction Tracking from the Supported operating systems list.

Supported programming languages

The Transaction Tracking API supports the following programming languages:

- C
- C++
- Enterprise COBOL (z/OS® only)
- Enterprise PL/I (z/OS only)
- IBM High Level Assembler (HLASM - z/OS only)
- Java™ 1.4 and 1.5

The following program environments are supported on z/OS:

- C, C++ and Java applications running in 64 bit mode on z/OS
- C and C++ XPLINK and non-XPLINK programs on z/OS
- C and C++ programs statically and dynamically linking Transaction Tracking API
- COBOL and PL/I programs statically linking Transaction Tracking API

Preparing your environment

The way in which you prepare your environment is dependent on whether it is a distributed or z/OS environment.

Distributed environments

Depending on the target platform, the Transaction Tracking API Software Development Kit (Transaction Tracking API SDK) is packaged as a single *.zip* or *tar* archive. This archive contains everything required to instrument an application. However, it does not include any other related components, such as the Transaction Collector. The Transaction Tracking API SDK is located in *ITM installation*`/`*platform*`/tu/tusupport`. You may unpack the Transaction Tracking API SDK anywhere on your system.

The files contained in the Transaction Tracking API SDK are:

```
include/
  ttapi.h
lib/
  ttapi4j.jar
  ttapi.lib, ttapi.dll, pthread.dll (Windows)
  libttapi.so (UNIX - suffixes vary by platform)
  kbb.dll on Windows
  libkbb.so on UNIX
```

z/OS environments

For z/OS systems, the Transaction Tracking API SDK is installed as part of the Transactions Base installation. However:

- C programmers on UNIX Systems Services (USS) may wish to copy the SCYTSAMP member CYTAPI to a USS directory of their choice, renaming it to `cytapi.h`. This file holds all C and C++ includes necessary to use the Transaction Tracking API.
- Java programmers must ensure that the Transactions JAR files are in the Java classpath, and external links to the Transactions JNI modules are in the Java libpath. See the *IBM Tivoli Composite Application Manager for Transactions Installation and Configuration Guide* for more information.

Getting started

Use this information to help you create and send events.

Introduction

To send a transactions event, you must complete these steps.

1. Initialize Transaction Tracking API using the `init` function.
2. Construct the event.
3. Send the event using the `track` function.
4. Shut down the Transaction Tracking API using the `shutdown` function.

Initialize Transaction Tracking API

The `init` function only needs to be called once per process. This function sets up the Transaction Tracking API environment, and populates the Configuration Block with information required by all other Transaction Tracking API functions. The Configuration Block must not be changed after `init` has been called. The `init` command must be called before any other Transaction Tracking API function. However, `TT_check_version` should be called before the `init` command, otherwise it has no effect.

Callers of the init function must allocate an area for the Configuration Block, and populate the servername field – the destination where events are to be sent. This must be in the format specified in Appendix A, “Transport address format,” on page 65.

Java callers use the `ServerFactory.getServer` class, however this call performs the same functions as the `init` function. Non-z/OS C and C++ users may choose to use the `check_version` function to check the header versions.

Create the event

An event block must be allocated and completed by the caller. "How to build an event" on page 9 describes the event format in detail, and shows how to code an event block.

Java users construct events by calling the `createEvent` method of a `ttapi4j.Server` object.

Send the event

The `track` function (native) and `ttapi4j.Server.track` method (Java) are used to send the event. Note that calling `track` does not modify the contents of the event constructed. Users calling `track` multiple times do not have to recreate the entire event, but can reuse the existing event – replacing only the individual fields required.

Shut down the Transaction Tracking API

The Transaction Tracking API should be shut down when it is no longer needed to track events.

Java users should invoke the close method of the Server object.

Note: The shutdown function is not required for z/OS.

Program requirements and include files

Before using the Transaction Tracking API, you must first provide standard preamble statements and include files in your code.

C/C++

You must include the Transaction Tracking API include file.

For example, for distributed systems:

```
#include <ttapi.h>
```

For z/OS users:

```
#include <cytapi.h>
```

COBOL

You must copy the CYTABCON constants copybook into the Data-Division of your Working Storage Section. For example:

```
 DATA DIVISION.
       Working-Storage Section.
       COPY CYTABCON.
```

PL/I

You must include the supplied event block structure. For example:

```
%include CYTAPEVT;
```

Compiling, linking, and executing with Transaction Tracking API

References for compiling, linking, and executing with Transaction Tracking API.

Compiling

C/C++

To compile C or C++ programs against the Transaction Tracking API library, add the `include` directory found in the Transaction Tracking API SDK or SCYTSAMP dataset to the compiler's preprocessor include path.

For example, if compiling with Microsoft Visual C 7.1 and the SDK is installed in `C:\TTAPI`:

```
cl /I C:\TTAPI\include <custom-flags> /c <source-filename>
```

Java

To compile a Java program, ensure that the Transactions `cytapi4j.jar` (z/OS) or `ttapi4j.jar` (non-z/OS platforms) JAR file is in the Java classpath. This can be achieved by adding the file's absolute path to the `CLASSPATH` environment variable, or by specifying it on the command line by using the `-classpath` flag.

COBOL and PL/I

To compile COBOL or PL/I programs, ensure the SCYTSAMP library is in the compiler's SYSLIB DD concatenation.

High Level Assembler

To assemble HLASM programs, ensure the SCYTSAMP library is in the assembler's SYSLIB DD concatenation.

Linking on distributed platforms

To link the resultant objects into an application, link against the libttapi library provided with the Transaction Tracking API SDK. For example, if linking with Microsoft Visual C 7.1 and the SDK is installed in `C:\TTAPI`:

```
link /libpath: C:\TTAPI\lib <custom-flags> <object-files> ttapi.lib
```

Binding on z/OS systems

The Transaction Tracking API programs are stored as DLLs in the SCYTLOAD library. When binding programs with Transaction Tracking API on z/OS:

- If a C or C++ program is dynamically calling Transaction Tracking API:
  - If compiling in batch, include the SCYTSAMP member CYTASIDE in the SYSLIN DD.
  - If compiling in UNIX Systems Services, copy the SCYTSAMP member CYTASIDE to an HFS directory, renaming it to cytaside.x. Include cytaside.x when binding your program. For example: `c89 –W'l,dll' pgm1.o cytaside.x`
- Otherwise ensure the SCYTLOAD library is included in the binder search path. For example, adding SCYTLOAD to the binder SYSLIB DD if binding in batch.

Running with Transaction Tracking API

C/C++

If running on distributed (non-z/OS) platforms, ensure the lib directory of the Transaction Tracking API SDK is included in the runtime library search path.

For example, on Linux you must modify the LD_LIBRARY_PATH environment variable to include the directory.

If running on z/OS, ensure the SCYTLOAD library is in the z/OS linklist concatenation, and that external links to the Transactions JNI modules are defined in the Java libpath. See *IBM Tivoli Composite Application Manager for Transactions Installation and Configuration Guide* for more information.

COBOL and PL/I

If running on z/OS, ensure the SCYTLOAD library is in the z/OS linklist concatenation.

Java

If running on distributed (non-z/OS) platforms, ensure the lib directory of the Transaction Tracking API SDK is included in the runtime library search path.

For example, on Linux you must modify the `LD_LIBRARY_PATH` environment variable to include the directory.

If running on z/OS, ensure the SCYTLOAD library is in the z/OS linklist concatenation.

In both cases, ensure the `cytapi4j.jar` (z/OS) or `ttapi4j.jar` (non-z/OS) JAR files are in the Java classpath.

Error handling

Transaction Tracking API functions that fail return an error code. Check the error codes to determine whether the Transaction Tracking API function succeeded.

If a function succeeds it returns TT_SUCCESS (zero). If it fails, it returns a non-zero error code, as described in Appendix B, "Return codes," on page 67. Additionally, Transaction Tracking API provides logging facilities to further isolate the problem. For normal operation, this logging is disabled.

Invalid events

The `track` function validates all events and returns a code, as described in Appendix B, "Return codes," on page 67, which specifies in detail the invalid field or value.

Undefined behavior

There are certain error conditions that the Transaction Tracking API cannot detect. For example:

- Passing one Configuration Block to `init`, and a different block to other functions.

- Modifying the Configuration Block after `init` has been called.
- Incorrect length value specified.
- Configuration Block, event, or name/value pairs not initialized to nulls before use.

In these cases, Transaction Tracking API processing is unpredictable.

Error logging and debugging

In addition to returning error codes from Transaction Tracking API functions, Transaction Tracking API logs error and debug messages at significant points in the process of initializing, shutting down, sending events to a Transaction Collector, and various states in between. In general, this logging will not be of interest - it will usually be turned on to provide support professionals with enough information to help isolate an error or misuse of the API.

On platforms other than z/OS, Transaction Tracking API uses the IBM Tivoli Monitoring standard `RAS1` logging. On z/OS, log information is written to standard output. You can control the amount of logging produced by the `RAS1` logger by configuring the environment variables listed in Table 1.

Table 1. Logging configuration environment variables

Environment variable	Description
`KBB_RAS1=ALL`	Enable logging of **all** messages. Generally this setting provides too much information.
`KBB_RAS1=ERROR`	Enable logging of error messages only. This provides a restricted set of messages. Set additional logging to provide the exact information you require. For example, set the following to see the return code from **`TT_Track`**: `KBB_RAS1=ERROR (COMP:transport ERROR,FLOW,DETAIL) (COMP:ttapi ERROR,FLOW,DETAIL)`
`KBB_RAS1=`	Disable all message logging. This is the default.
`KBB_RAS1_LOG=`	Log to standard output.
`KBB_RAS1_LOG=`*`log file path name`*	Set the log file name and other parameters. See the format example below.
`KBB_VARPREFIX=%`	Set the prefix for variables specified in `KBB_RAS1_LOG`.
`CYTA_LOGGER=ttapi`	Enable Transaction Tracking API logging for Java programs running on UNIX and Linux systems.

`KBB_RAS1_LOG` has the following format:

```
KBB_RAS1_LOG=filename [INVENTORY=inventory_filename]
    [COUNT=count]
    [LIMIT=limit]
    [PRESERVE=preserve]
    [MAXFILES=maxfiles]
```

The settings for `KBB_RAS1_LOG` are:

count Maximum number of log files to create in one invocation of the application.

inventory_filename
A file in which to record the history of log files across invocations of the application.

limit Maximum size per log file.

maxfiles
Maximum number of log files to create in any number of invocations of the application. This value takes effect only when *inventory_filename* is specified.

preserve
Number of log files to preserve when log files wrap over *count*.

Logging to 64-bit Windows systems

To enable logging to 64-bit Windows systems, use the `stdout` logger, which is an alternative to the RAS1 logger. To enable logging, set the following environment variable:

`CYTA_LOGGER=`*`path-to-ttapi.dll`*`:stdout`

For example, `CYTA_LOGGER=C:\IBM\ITM\tmaitm6\tusupport\64\ttapi.dll:stdout`. All logging is sent to the standard output of the process.

To log to a file, also set the following environment variable:

`CYTA_LOGFILE=`*`path-to-output`*

For example, `CYTA_LOGFILE=C:\temp\mylog.log`.

How to build an event

Instrumenting an application requires you to create events that indicate the flow of a transaction.

The Transaction Tracking API comes in multiple forms:

- High Level Language (HLL) package – for C, C++, COBOL and PL/I programs.
  See "C types and structures" on page 23 for information specific to instrumenting C/C++ applications.
- TTAPI4J – wrapper for Java applications.
  See "Java reference" on page 28 for information specific to instrumenting Java applications.
- z/OS macros for HLASM callers.
  See "High Level Assembler Reference" on page 28.

Detailed information on each of these forms is given in the appropriate reference chapter. Complete examples for all languages are provided in the Appendixes.

Transaction Tracking API events contain the major components described in Table 2.

Table 2. Event components

Component	Description
Event type	The type of the event, for example outbound or inbound.
Instance ID	Information specific to the event's enclosing transaction instance.
Horizontal ID	Information used to correlate events, where the events occur in separate processes, potentially on separate machines.

Table 2. Event components (continued)

Component	Description
Vertical ID	Information used to correlate events, where the events occur in the same process.
Horizontal context	Information used to aggregate events across processes.
Vertical context	Information used to aggregate events within a process.
Blocked Status	An attribute that indicates whether or not an event is related to a synchronous interaction in its transaction.

These components are described in detail in the following sections.

Event types

Every event sent by Transaction Tracking API has an associated type that is used in event correlation.

The event types are described in Table 3.

Table 3. Event types

Event type	Description
STARTED	The beginning of a transaction. No events in a transaction may come before STARTED. The STARTED event type is used as the lower bound by the correlation system when searching for related events.
FINISHED	The end of a transaction. No events in a transaction may come after FINISHED. The FINISHED event type is used as the upper bound by the correlation system when searching for related events.
INBOUND	A message has been received. These events are typically used to correlate cross-process interactions.
OUTBOUND	A message has been sent. These events are typically used to correlate cross-process interactions.
HERE	Usually indicates the blocking or unblocking of an asynchronous transaction. May also be used in situations where there is not enough context to determine whether an event is the result of an outgoing or incoming message.
STARTED_INBOUND	Combination of STARTED and INBOUND events used to reduce the number of events produced. STARTED_INBOUND events are used both as a STARTED lower bound, and for INBOUND correlation.
OUTBOUND_FINISHED	Combination of OUTBOUND and FINISHED events used to reduce the number of events produced. OUTBOUND_FINISHED events are used both as a FINISHED upper bound, and for OUTBOUND correlation.
INBOUND_FINISHED	Combination of INBOUND and FINISHED events used to reduce the number of events produced. INBOUND_FINISHED events are used both as a FINISHED upper bound, and for INBOUND correlation.
COMMIT	Reserved for use by the WebSphere MQ data collector.
ROLLBACK	Reserved for use by the WebSphere MQ data collector.

The event is set in the event block `Type` field. For example:

Java

```
event.setType(Event.Type.OUTBOUND);
```

C/C++

```
event.type = TT_OUTBOUND_EVENT;
```

COBOL

```
MOVE CYTA-STARTED TO CYTA-E-TYPE.
```

PL/I

```
cytaetyp = cytaesta;
```

HLASM

```
CYTATRAK STARTED,
```

Event type examples

The Transaction Tracking API can be used to track both synchronous and asynchronous transactions.

Synchronous transactions

The simple example shown in Figure 1 demonstrates event type usage in a client application making a synchronous request to a server application.

The dashed line in Figure 1 illustrates the flow of the transaction from start to finish.

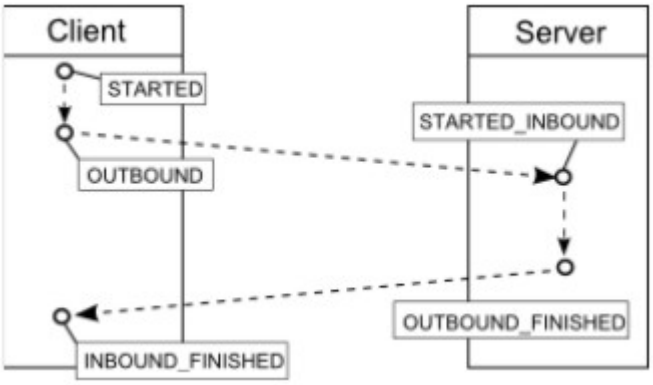


Figure 1. Synchronous transaction

Transactions typically start with a `STARTED` event, unless the transaction is known to have started as the result of an inbound message, in which case a `STARTED_INBOUND` event type is used. In the example, the overall transaction begins with a `STARTED` event, and the subtransaction in the server process begins with a `STARTED_INBOUND`.

Transactions typically terminate with an `INBOUND_FINISHED` or `OUTBOUND_FINISHED` event, because a transaction usually terminates upon receipt of a reply to a prior request. If the transaction terminates because of some other condition, for example if the request from the client to the server times out, indicate this with a `FINISHED` event.

Asynchronous transactions

See "Blocking events" on page 17 for an example of event-type usage in asynchronous transactions.

Linking and stitching

One of the most important elements of the Transaction Tracking API event is the association ID, which is composed of linking and stitching IDs. Linking and stitching IDs are used to determine the relationships between events.

For example, if an outbound and an inbound event are related to a particular interprocess interaction, then they must both contain some identical information so that they can be matched against each other. Each event may have either or both *horizontal* and *vertical* association IDs. Typically, the horizontal ID correlates events across processes, and the vertical ID correlate events within a single process.

Note: The term *technology domain* is introduced in this section. This term refers to a (potential) Transaction Tracking API event source, such as ARM, MQ, ITCAM for SOA, or a custom application. Each domain is expected to provide enough information to correlate Transaction Tracking API events.

Linking and stitching IDs

Linking IDs identify interactions within a single technology domain. For example, where a domain, such as ARM, tracks transactions by passing tokens along, that token, or some part of it, might be used as the linking ID. The linking ID will not match any other domain, but it allows events within the ARM domain to be correlated. You must provide Transaction Tracking API with one and only one linking ID.

Stitching IDs identify interactions between technology domains, both within and across processes. In-process interactions occur only if there are two sources of tracking information within that process. If a process lies at the edge of two technology domains, for example ARM and MQ, then it is possible that the process will produce events for both domains.

Linking and stitching IDs are opaque to the Transaction Tracking API; they carry no special meaning, and have no particular formatting constraints beyond their size limitations. The event correlation system performs a simple byte-array comparison for equality.

Linking and stitching IDs must be globally unique for each interaction, from the first STARTED event to the last FINISHED event.

Tip: You can improve the uniqueness of linking IDs in custom applications by adding a prefix or suffix to all linking IDs generated by your application. In doing this, you will achieve the same effect as setting the caller type to some value unique to your application, provided that the prefix or suffix you choose is not used by another application.

Stitching IDs

While Transaction Tracking API provides an interface for providing arbitrary stitching IDs, they are useless if there is no commonality between each technology domain. Horizontal stitching IDs generally must be provided on a pair-by-pair

basis - that is for each pair of technology domains. The developers instrumenting these domains will have to communicate to determine common stitching IDs. Transaction Tracking API events may contain multiple stitching IDs. For any two events, if a stitching ID of one event is equal to the stitching ID with the same name of the other event, then some interaction is assumed to have occurred between the two events. The Transaction Collector `kto_stitching.xml` file defines how this stitching occurs. See Appendix D, "The kto_stitching file," on page 83 for further information on this file.

Vertical stitching can generally be accomplished by using the thread ID of the thread in which the transaction event occurred. This enables the correlation system to correlate events from one transaction that are interleaved with events from another transaction in another thread. This depends on the structure of the instrumented application, and on a single thread being used to service a transaction.

Link types

All links must have a type (CALL TYPE ID) that is an integer value and can be any number 0 to 255. Events have a default caller type of ANY - this caller type will only be matched against other events with the ANY caller type, and may be used instead of a domain-specific caller type. However, use values from the range 200-255 rather than ANY.

Currently defined values are:

- 0 ANY
- 1 GPS
- 2 ARM
- 3 WSA
- 4 CICS Transaction Gateway
- 5 Websphere MQ
- 6 SOA
- 7 Web Resource
- 8 CICS
- 9 IMS
- 10 WebSphere Message Broker
- 11 DB2
- 12 Reserved for IBM use
- 13 IMS Connect
- 14 Tuxedo
- 15 Optim Performance Manager
- 16 .NET
- 17 Citrix and Terminal servers
- 18 Web Response Monitors
- 19-20 Reserved for IBM use
- 21 SOAP
- 22 IIOP
- 23 Java Message Service
- 24-199 Reserved for IBM use
- 200-255 Available to users

Examples

Vertical linking

This example uses C/C++ to link events using the current thread ID:

```
pthread_t current_thread;
char thread_id[sizeof(pthread_t)];
struct tt_event_t event;
 /* Set the event's vertical link ID to the current thread ID. */
current_thread = pthread_self();
memcpy(thread_id, &current_thread, sizeof(pthread_t));
event.vertical_id.link_id = thread_id;
event.vertical_id.caller_type = TT_ARM_CALLER;
event.vertical_id.link_id_size = sizeof(pthread_t);
```

Horizontal linking

This example uses C/C++ to link events using a token embedded in a message sent by the instrumented application. Because both applications are members of the same technology domain they are capable of communicating this way. That is, the server knows how to decode the message so that it can extract the value of the horizontal link ID:

```
/* Create the token that will be sent with the message */
uint16_t token_size = 0;
char *token = create_token(&token_size);
 /* Set the event's horizontal link ID to the message's token value. */
event.horizontal_id.link_id = token;
event.horizontal_id.caller_type = TT_IMS_CALLER;
event.horizontal_id.link_id_size = token_size;
```

Stitching

Using horizontal and vertical stitching IDs is similar to using horizontal and vertical linking IDs. Below are some simple examples of how to configure stitching IDs.

Java:

```
Event event = server.createEvent();
event.getHorizontalID().getStitchingIDs().put("name", "value");
event.getVerticalID().getStitchingIDs().put("name", "value");
```

C/C++:

```
tt_event_t event;
tt_values_list_t horizontal_stitching_ids;
tt_values_list_t vertical_stitching_ids;
 horizontal_stitching_ids.name  = "name";
horizontal_stitching_ids.value = "value";
horizontal_stitching_ids.size  = sizeof("name") - 1;
horizontal_stitching_ids.next  = 0;
event.horizontal_id.stitch_ids = &horizontal_stitching_ids;
vertical_stitching_ids.name  = "name";
vertical_stitching_ids.value = "value";
vertical_stitching_ids.size  = sizeof("name") - 1;
vertical_stitching_ids.next  = 0;
event.vertical_id.stitch_ids = &vertical_stitching_ids;
```

Transaction Instance IDs

Transaction Tracking API events have an optional instance ID.

The instance ID contains either or both of the following fields:

- Transaction ID
- Transaction Data

The transaction ID exists purely for assisting the event correlation system; it is an identifier common to all (or a subset of) events belonging to an instance of a transaction. Normally, the correlation system will have to iteratively build up a transaction by following each linking and stitching ID. If a transaction ID is specified, the correlation system can request all events with that ID up-front, thus reducing the time to completion.

Events may also contain transaction data. Transaction data is data particular to an instance of a transaction. It is only used for labeling in reports and graphs of transactions. Only data that is particular to a instance of the transaction is included, that is, data that is not aggregated, and is not used for linking and stitching. The data is presented when the transaction instance is visualized. For example, a web application might report the parameters of an HTTP request in instance data, and the server and page part of the request in the aggregated data (context).

Example: Java

```
Event event = server.createEvent();
event.getInstanceID().setTransactionID("SomeUniqueTransaction");
event.getInstanceID().getTransactionData.put("name", "value");
```

Example: C/C++

```
tt_event_t event;
tt_values_list_t transaction_data;

transaction_data.name = "name";
transaction_data.value = "value";
transaction_data.size = sizeof("value") - 1;
transaction_data.next = 0;

event.instance_id.transaction_id = "SomeUniqueTransaction";
event.instance_id.size = sizeof("SomeUniqueTransaction") - 1;
event.instance_id.transaction_data = &transaction_data;
```

Context information

A typical environment monitored by Transaction Tracking produces large amounts of tracking data. For this reason, Transaction Tracking aggregates the tracking data. Timings and other statistics are aggregated by their common contextual information.

Contextual information is also used for labeling nodes in the visualization of a transaction, and for linking to domain-specific IBM Tivoli Monitoring applications.

Contextual information is information related to the circumstances in which the transaction event occurred. For example, the name or address of the host on which the event occurred, or the host that caused the event to occur. Similarly, the name of the application from which the transaction originated or the application that is presently processing the transaction are also contextual information. Such information allows users to aggregate response times between hosts, between

applications, and so on. It is not, however, instance data, that is, it is not specific to one event, but typically specific to a *flow* of events.

Contextual information is stored in two fields: the vertical context, and horizontal context. Vertical context is intended to contain information about the transaction, application or host where the event occurred. Horizontal context is intended to contain information about the message or interaction between two applications or hosts. For example, the vertical context might contain the host name of the machine on which an event occurred, and the horizontal context might contain the type of HTTP request that caused the event to occur.

For a transaction moving through a physical topology, an event's vertical context (for example, the hostname, physical location, application name) is used to label the individual nodes (that is, the hosts) in the graph. An event's horizontal context (for example, the query type, message queue name) is used to label the edges between those nodes.

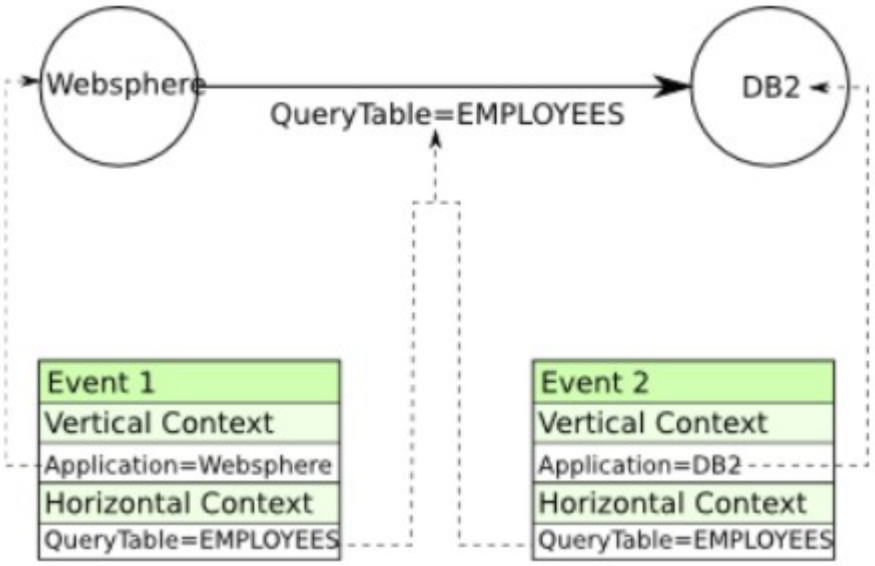


Figure 2. Contextual information in a transaction

Similarly to creating stitching IDs, providing contextual information requires cooperation between the programmers instrumenting the various applications. In particular, the names of the items of information should match where transactions should be grouped by that information. Names are case-sensitive, and aggregation performs a binary equality comparison on them.

Transaction Tracking workspaces also depend on names to provide a further hierarchy of information. The following four Vertical Contexts must be set at least once for every linked set of events. For example, set the Vertical Contexts in the STARTED event.

ServerName

The server name or address of the machine on which the event occurred. For example, `win001`.

For z/OS users, this must be the Sysplex name and the z/OS host name (SMF id), separated by a forward slash (/). For example, `SYSPLEXQ/MVS1`.

ComponentName
The name of the component in which the event occurred, For example CICS®, Websphere Application Server, MQ: `MQ`.

ApplicationName
The name of the application in which the event occurred. For example, the CICS region name or the MQ Queue Manager name: `CICS001`.

TransactionName
A common identifier for a group of transactions. If this is known by all participants in the transaction, you can easily view aggregate information for all events occurring within transactions of that group. For example, `TXN1`.

Note: Values for Vertical Contexts can be updated by subsequent vertically linked events. For example, a value for TransactionName could be set in the STARTED event and then be overwritten in an INBOUND event.

The horizontal context distinguishes between interactions at an aggregate level, which is of particular benefit when a mesh of interactions occurs. Transaction Tracking workspaces do not currently display this information, but the interaction data presented is more accurate when provided. Some suggestions for common contextual information to include in events are:

Resource
For applications that query some resource in an external application, such as a message queue or database table, the name of that resource. For example, the queue name or database table name.

SourceHost
The host name of the source of the interaction.

DestinationHost
The host name of the destination of the interaction.

The contexts that the Transaction Collector should aggregate are specified in the Context Mask file. See Appendix E, "Transaction Collector Context Mask," on page 87 for more information on this file.

Blocking events

Blocking events indicate the start of a transaction that causes an application or process to wait for a response.

Transaction Tracking API allows you to provide enough information when creating an event so that the correlation system can determine whether or not the event is part of a synchronous transaction. This can help the system when it calculates the System Time metric of an interaction. A description of the metrics produced by the correlation system is outside the scope of this document.

Specify events as blocked if and only if they are related to the start of a transaction that will cause the code to wait for a response, which is referred to as a *synchronous transaction*. Note that blocking can still occur in an asynchronous transaction; for example, an asynchronous transaction may synchronize at some point, leading to a blocking event being generated.

Java

```
event.setBlocked(true);
```

C/C++

```
event.blocked = 1;
```

Example: blocking events

This example shows how to instrument a partially asynchronous transaction.

In the transaction example shown in Figure 3, the client application makes a nonblocking request to the server application, and then continues to perform some computation. When it is ready to block while waiting for the response, it sends a `HERE` event with the `blocked` flag set. This indicates that a previously asynchronous transaction has become synchronous. The `HERE` event type indicates that an event occurs after a transaction starts and before the transaction ends, but is not necessarily related to an interaction between applications.

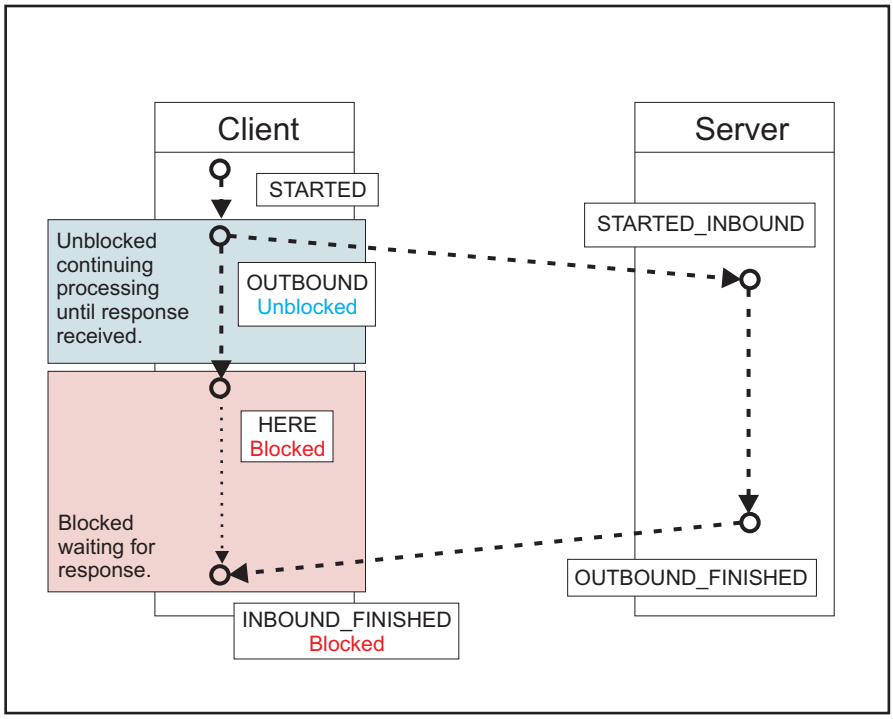


Figure 3. Partially asynchronous transaction

Platform-specific issues

Applications on z/OS systems send event data using EBCDIC with the exception of Java applications.

By default, Transaction Tracking translates all event data from EBCDIC to ASCII. However if some data supplied is binary or ASCII data, this translation must be avoided. The Transaction Tracking API structure includes flags that can be set to stop this translation. See the SCYTSAMP dataset for examples.

High Level Language reference

This section is for COBOL, C, C++ and PL/I High Level Languages. Transaction Tracking API function names all begin with `cyta` for z/OS, and `tt` for non-z/OS platforms (although `cyta` can also be used).

Functions

Descriptions of Transaction Tracking functions callable from High Level Languages.

Function: check_version

Name: `CYTA_check_version` (z/OS and non-z/OS); `TT_check_version` (non-z/OS)

Purpose: Checks that the header being compiled against matches the library being linked against.

Parameters required: None.

Return codes: See Appendix B, "Return codes," on page 67.

C definition: `int CYTA_check_version(void);`

Notes:

- Call `CYTA_check_version` before any other Transaction Tracking API function, including `CYTA_init`.
- Not available on z/OS systems.
- Not required, however it can assist developers in cases where structures defined in an older header do not line up with what is expected by a newer library.

Examples:

C:

```
#include <cytapi.h>
rc = CYTA_check_version();
```

Function: init

Name: `CYTA_init` (z/OS and non-z/OS); `TT_init` (non-z/OS)

Purpose: InitializeTransaction Tracking API for High Level Language Callers.

Parameters required: Configuration Block with valid server as described in Appendix A, "Transport address format," on page 65.

Return codes: See Appendix B, "Return codes," on page 67.

C Definition: `int CYTA_init(cyta_config_t *config);`

Notes:

- After `init` has been called, the Configuration Block must not be modified
- Callers must first initialize the Configuration Block to zeros.
- If the server field is blank, the default server is used. See Appendix A, "Transport address format," on page 65 for further information.

Examples:

C:

```
#include <cytapi.h>
  cyta_config_t  configblk;
 memset(&configblk, 0, sizeof(config));
 configblk.server = "tcp:svr.mycompany.com:5455";
  rc = CYTA_init(&configblk); /* Config token in configblk*/
```

COBOL:

```
DATA DIVISION.
Working-Storage Section.
COPY CYTABCON.
COPY CYTABCFG.
01 SERVER  pic x(8) value 'SSN:CYTZ';
01 RC  pic S(9) comp.
PROCEDURE DIVISION.
SET CYTA-CFG-SERVER TO ADDRESS OF SERVER.
CALL "CYTA_init" USING CYTA-CFG-BLOCK RETURNING RC.
```

PL/I:

```
%include CYTAPEVT;            /* Area to hold event blk  */
%include CYTAPCFG;           /* Area to hold Config Blk */
Dcl server Char(8) Init("SSN:CYTZ");
Dcl stgarea      Area;    /* stgarea is 1000 bytes    */
Allocate cytacfg In(stgarea);
cytacsrv = Addr(server);    /* Set Container subsystem  */
Call CYTA_init(cytacfg);    /* Config token in cytacfg  */
```

Function: shutdown

Name: `CYTA_shutdown` (z/OS and non-z/OS); `TT_shutdown` (non-z/OS)

Purpose: Shut down Transaction Tracking API. For z/OS, this function is not required, and is included for compatibility with other platforms only.

Parameters required: Configuration Block initialized with `CYTA_init`.

Return codes: See Appendix B, "Return codes," on page 67.

C Definition: `int CYTA_shutdown(cyta_config_t *config);`

Example:

C:

```
#include <cytapi.h>
int rc;
cyta_config_t  configblk;
cyta_event_t eventblk;
memset(&configblk, 0, sizeof(configblk));
memset(&eventblk, 0, sizeof(eventblk));
configblk.server = "tcp:svr.mycompany.com:5455";
rc = CYTA_init(&configblk);
/* code here to populate event block     */
rc = CYTA_track(&configblk, &eventblk);
rc = CYTA_shutdown(&configblk);
```

Function: strerror

Name: `CYTA_strerror` (z/OS and non-z/OS); `TT_strerror` (non-z/OS)

Purpose: Return a string describing a return code from a Transaction Tracking API function.

Parameters required: Return code from a Transaction Tracking API function.

Output: String describing the error code.

Return codes: None.

C Definition: `const char* CYTA_strerror(int errno);`

Notes: Only available for C and C++.

Examples:

C:

```
#include <cytapi.h>
int rc;
rc = CYTA_init(&configblk);
if (rc > 0)
    printf("Init error: %s\n", CYTA_strerror(rc));
```

Function: time

Name: CYTA_time (z/OS and non-z/OS); TT_time (non-z/OS)

Purpose: Get time now in seconds and microseconds since 00:00:00, Jan 1, 1970.

Parameters required: tt_time_t structure to receive time.

Output: Current wall clock time returned in the tt_time_t structure.

Return codes: None.

C Definition: cyta_time_t* CYTA_time(cyta_time_t*);

Notes: This function is only required if a timestamp different from the current timestamp is required on an event. If the timestamp on an event sent to track is zero, the current time is automatically inserted.

Examples:

C:

```
#include <cytapi.h>
  cyta_time_t now;
  CYTA_time (&now);
```

COBOL:

```
DATA DIVISION.
Working-Storage Section.
COPY CYTABEVT.
01 RC  pic S(9) comp.
PROCEDURE DIVISION.
CALL "CYTA_time" USING CYTA-E-SECONDS RETURNING RC.
```

PL/I:

```
%include CYTAPEVT;
Dcl stgarea      Area;    /* stgarea is 1000 bytes */
Allocate cytacfg In(stgarea);
Call CYTA_time(cytaesec);
```

Function: token

Name: CYTA_token (z/OS and non-z/OS); TT_token (non-z/OS)

Purpose: Obtain a fullword token unique across the enterprise for High Level Language Callers.

Parameters required: Configuration Block initialized with CYTA_init.

Output: Fullword unique token returned in area supplied.

Return codes: See Appendix B, “Return codes,” on page 67.

C Definition: int CYTA_token(cyta_config_t *config, cyta_int32_t *token);

Examples:

C:

```
#include <cytapi.h>
  int rc;
int token;
 cyta_config_t  configblk;
memset(&configblk, 0, sizeof(config));
 configblk.server = "tcp:svr.mycompany.com:5455";
  rc = CYTA_init(&configblk);
 if (rc == TT_SUCCESS)
 rc = CYTA_token(&configblk, &token);
```

COBOL:

```
DATA DIVISION.
Working-Storage Section.
COPY CYTABCON.
COPY CYTABCFG.
01 SERVER  pic x(8) value 'SSN:CYTZ';
01 TOKEN  pic S(9) comp.
01 RC  pic S(9) comp.
PROCEDURE DIVISION.
SET CYTA-CFG-SERVER TO ADDRESS OF SERVER.
CALL "CYTA_init" USING CYTA-CFG-BLOCK RETURNING RC.
CALL "CYTA_token" USING CYTA-CFG-BLOCK TOKEN RETURNING RC.
```

PL/I:

```
%include CYTAPEVT;
 %include CYTAPCFG;
Dcl server Char(8) Init("SSN:CYTZ");
Dcl token Fixed(32);
Dcl stgarea      Area;    /* stgarea is 1000 bytes */
Allocate cytacfg In(stgarea);
cytacsrv = Addr(server);
Call CYTA_init(cytacfg);
Call CYTA_token(cytacfg,token);
```

Function: track

Name: `CYTA_track` (z/OS and non-z/OS); `TT_track` (non-z/OS)

Purpose: Send completed event for High Level Language Callers.

Parameters required: Configuration Block initialized with `CYTA_init`. Completed event block.

Output: Event sent if valid and Transactions operational.

Return codes: See Appendix B, "Return codes," on page 67.

C Definition: `int CYTA_track(cyta_config_t *config, cyta_event_t *event);`

Notes:

- If the time in the event block is zero, the current time is automatically inserted for the event.
- The contents of the event block are unchanged by `track`. Hence the same event block can be used for multiple `track` calls, with only the changed fields being updated

Examples:

C:

```
#include <cytapi.h>
  int rc;
int token;
 cyta_config_t  configblk;
```

```
 cyta_event_t eventblk;
memset(&configblk, 0, sizeof(configblk));
 memset(&eventblk, 0, sizeof(eventblk));
 configblk.server = "tcp:svr.mycompany.com:5455";
  rc = CYTA_init(&configblk);
 /* code here to populate event block     */
 rc = CYTA_track(&configblk, &eventblk);
```

COBOL:

```
DATA DIVISION.
Working-Storage Section.
COPY CYTABCON.
COPY CYTABCFG.
COPYT CYTABEVT.
01 SERVER  pic x(8) value 'SSN:CYTZ';
01 TOKEN  pic S(9) comp.
01 RC  pic S(9) comp.
PROCEDURE DIVISION.
SET CYTA-CFG-SERVER TO ADDRESS OF SERVER.
CALL "CYTA_init" USING CYTA-CFG-BLOCK RETURNING RC.
* code here to populate event block
CALL "CYTA_track" USING CYTA-CFG-BLOCK CYTA-EVENT RETURNING RC.
```

PL/I:

```
%include CYTAPEVT;
 %include CYTAPCFG;
 Dcl server Char(8) Init("SSN:CYTZ");
 Dcl token Fixed(32);
 Dcl stgarea      Area;    /* stgarea is 1000 bytes */
 Allocate cytacfg In(stgarea);
 cytacsrv = Addr(server);
 Call CYTA_init(cytacfg);
 /* code here to populate event block   */
 Call CYTA_token(cytacfg,token);
```

C types and structures

Transaction Tracking API provides a set of publicly available data types and structures. The ttapi.h (non-z/OS) and SCYTSAMP CYTAPI (z/OS) include files define many C and C++ data structures that can be used throughout your program. For z/OS, all data types begin with `cyta`, for non-z/OS users, they can start with `tt` or `cyta`.

Basic data types

Transaction Tracking API defines common names for various basic data types:

tt_uint64_t
: Unsigned 64-bit integer. (cyta_uint64_t for z/OS)

tt_int64_t
: Signed 32-bit integer. (cyta_int64_t for z/OS)

tt_uint32_t
: Unsigned 32-bit integer. (cyta_uint32_t for z/OS)

tt_int32_t
: Signed 32-bit integer. (cyta_int32_t for z/OS)

tt_uint16_t
: Unsigned 16-bit integer. (cyta_uint16_t for z/OS)

tt_int16_t
: Signed 16-bit integer. (cyta_int16_t for z/OS)

tt_uint8_t
Unsigned 8-bit integer. (cyta_uint8_t for z/OS)

tt_int8_t
Signed 8-bit integer. (cyta_int8_t for z/OS)

tt_byte_t
Indicates that a memory address is considered as opaque, and may contain any 8–bit value. (cyta_byte_t for z/OS)

tt_config_t (cyta_config_t for z/OS)

Defines generic configuration parameters for Transaction Tracking API.

```
typedef struct TT_CONFIG_T
{
    const char* server;
    tt_uint32_t connect_timeout;
    tt_uint32_t connect_retries;
    tt_uint32_t connect_retry_interval;
    void*       handle;
    const char* token_filename;
    tt_uint16_t queue_size;
} tt_config_t;
```

tt_config_t.server
The address of the Transaction Collector to send events to. The address format is defined in Appendix A, “Transport address format,” on page 65. If this field is set to zero, it is replaced with the default server.

tt_config_t.connect_timeout
Length of time (in milliseconds) to wait for a TCP/IP connection to the Transaction Collector before timing out. Default is `30000` (30 seconds).

Note: Non-listening ports behave differently on different platforms:

- On UNIX platforms, if nothing is listening on the specified port the connect function returns immediately with an error, regardless of the timeout specified.
- On Windows, the connection times out.
- All systems time out if something is listening but the system is too busy to accept the connection in time.
- On all systems, if `tt_config_t.connect_timeout = 0`, the default value of `30000` (30 seconds) is used.

tt_config_t.connect_retries
Number of times to retry a failing TCP/IP connection. If set to zero, Transaction Tracking retries indefinitely.

If `connect_retries=0`, then on `TT_shutdown` it behaves as if there are *no* retries.

Upon shutdown Transaction Tracking API tries up to `connect_retries+1` attempts. The connect timeout doesn't have an effect without a listening port, however, the **`connect_retry_interval`** does have an affect.

tt_config_t.connect_retry_interval
Interval to wait before retrying TCP/IP connection. Default is `5000` (5 seconds). If `tt_config_t.connect_retry_interval = 0`, the default value of `5000` (5 seconds) is used.

tt_config_t.handle
Internal use only.

tt_config_t.token_filename
The file name of the token prefix file. This functionality is reserved for future use.

tt_config_t.queue_size
Maximum number of events to queue while waiting for a TCP/IP connection. Default is `1000`. If `tt_config_t.queue_size = 0`, the default value of `1000` is used. If more than 1000 events are received and there is no TCP/IP connection to the Transaction Collector, the oldest event is discarded.

Note: If you specify an incorrect transport address when the transport prefix does not refer to an existing transport, such as `tcp` in `tcp:127.0.0.1:5455`, an error is returned.

tt_time_t (cyta_time_t for z/OS)

Describes a point in time relative to the epoch, which is 00:00:00 UTC, January 1, 1970.Transaction Tracking API replaces instances of `tt_time_t` with both zero seconds and zero microseconds with the current time.

```
typedef struct TT_TIME_T
{
    tt_uint32_t sec;
    tt_uint32_t usec;
} tt_time_t;
```

tt_time_t.sec
Seconds component

tt_time_t.usec
Microseconds component

tt_event_t (cyta_event_t for z/OS)

Decribes an event that has occurred in the instrumented application. For example, the event may describe the beginning or completion of a transaction, or the sending or receipt of a request or response.

```
typedef struct TT_EVENT_T
{
    tt_uint32_t         type;
    tt_time_t           timestamp;
    tt_instance_id_t    instance_id;
    tt_association_id_t horizontal_id;
    tt_association_id_t vertical_id;
    tt_values_list_t*   horizontal_context;
    tt_values_list_t*   vertical_context;
    int                 blocked;
    void*               reserved1;
} tt_event_t;
```

tt_event_t.type
The event type. For example, `TT_STARTED_EVENT`, or `TT_OUTBOUND_EVENT`.

tt_event_t.timestamp
The point in time at which the event occurred.

tt_event_t.instance_id
Transaction ID and instance-specific data.

tt_event_t.horizontal_id
Horizontal linking and stitching IDs.

tt_event_t.vertical_id
Vertical linking and stitching IDs.

tt_event_t.horizontal_context
Horizontal context. A NULL-pointer is interpreted as an empty set.

tt_event_t.vertical_context
Vertical context. A NULL-pointer is interpreted as an empty set.

tt_event_t.blocked
Determines the blocked status of the event. Zero equates to unblocked, while a non-zero value equates to blocked.

tt_event_t.reserved1
Reserved for future use.

tt_association_id_t (cyta_association_t for z/OS)

Defines the information that identifies associations between sets of events, that is, the linking and stitching IDs.

```
typedef struct TT_ASSOCIATION_ID_T
{
    tt_uint32_t       caller_type;
    const tt_byte_t*  link_id;
    tt_uint8_t        link_id_size;
    tt_uint16_t       flags;
    tt_values_list_t* stitch_ids;
} tt_association_id_t;
```

tt_association_id_t.caller_type
Caller type for the link ID. This is used to eliminate collisions in link IDs between different callers of Transaction Tracking API.

tt_association_id_t.link_id
Address of the link ID for this event.

tt_association_id_t.link_id_size
Size of the link ID, in 8-bit bytes.

tt_association_id_t.flags
Flags that affect how the association ID is interpreted by Transaction Tracking API. For z/OS, `CYTA_ASSOCIATION_FLAG_LINK_RAW (1)` specifies that the link_id field is binary or ASCII data, and is not to be translated from EBCDIC to ASCII.

tt_association_id_t.stitch_ids
Stitching IDs. A NULL-pointer is interpreted as an empty set.

tt_instance_id_t (cyta_instance_id_t for z/OS)

Defines the information that identifies transaction-specific data.

```
typedef struct TT_INSTANCE_ID_T
{
    const tt_byte_t*  transaction_id;
    tt_uint16_t       size;
    tt_uint16_t       flags;
    tt_values_list_t* transaction_data;
} tt_instance_id_t;
```

tt_instance_id_t.transaction_id
Address of the transaction ID for the event.

tt_instance_id_t.size
Size of the transaction ID for the event, in 8-bit bytes.

tt_instance_id_t.flags
Flags which affect how the instance ID is interpreted by Transaction Tracking API. For z/OS, `CYTA_INSTANCEID_FLAG_RAW (1)` specifies that the `transaction_id` field is binary or ASCII data, and is not to be translated from EBCDIC to ASCII.

tt_instance_id_t.transaction_data
Transaction instance-specific data. A NULL pointer is interpreted as an empty set.

tt_values_list_t (cyta_values_list_t for z/OS)

Defines a singly-linked list of name-value pairs.

```
typedef struct TT_VALUES_LIST_T
{
    struct TT_VALUES_LIST_T* next;
    const char*              name;
    const tt_byte_t*         value;
    tt_uint16_t              size;
    tt_uint16_t              flags;
} tt_values_list_t;
```

tt_values_list_t.next
Pointer to the next item in the list. The last item in the list must have next set to zero.

tt_values_list_t.name
Name portion of the name/value pair. This is expected to be a null-terminated UTF-8 string of size less than or equal to 256 characters (including the null character).

tt_values_list_t.value
Value portion of the name-value pair. This is treated as a binary string. The value does not need to be null-terminated.

tt_values_list_t.size
Size of the value, in 8-bit bytes.

tt_values_list_t.flags
Flags which affect how the name/value pair is interpreted by Transaction Tracking API. Valid flags are:

- `CYTA_VALUELIST_FLAG_NAME_RAW (1)`– for z/OS only. Specifies that the name field is binary or ASCII data, and is not to be translated from EBCDIC to ASCII.
- `CYTA_VALUELIST_FLAG_VALUE_RAW (2)`– for z/OS only. Specifies that the value field is binary or ASCII data, and is not to be translated from EBCDIC to ASCII.
- `CYTA_VALUELIST_FLAG_VALUE_RAW (3)`– for z/OS only. Specifies that the name or value field is binary or ASCII data, and is not to be translated from EBCDIC to ASCII.

Java reference

Reference information for the Transaction Tracking API Java wrapper, TTAPI4J, is available separately as Java API documentation (Javadoc).

See the Transaction Tracking API SDK and Javadoc for TTAPI4J that are installed in `tusupport/ttapi/doc/ttapi4j` as part of the Transaction Collector installation for further information.

High Level Assembler Reference

This section is for High Level Assembler (HLASM) on z/OS systems.

HLASM Macro: CYTADFV

Macro to create Name/Value pairs to specify the minimal Vertical Context for an event.

Purpose

Prepare minimum required linked Vertical Context Name/Value pair entries for an event.

Input registers

No requirements.

Output registers

- R0 - 3 used as work register
- R4 - 13 unchanged
- R14 - 15 used as work register

Syntax

Syntax	Description
name	*name*: symbol. Begin name in column 1
`CYTADFV`	One or more blanks must follow `CYTADFV`
`APPL=`*`appl`*	*appl* – Application Name. Constant string in single quotation marks or pointer to string (register R2 – R12 in brackets or Rx address). Default: Jobname (if batch) or Address Space name (otherwise). Optional.
`APPLLEN=`*`length`*	*length* – Length of Application Name. Decimal constant in single quotation marks, register (R2-R12 in brackets) or halfword label. Optional if APPL is a label or constant string.
`XLATEA=`*`yes/no`*	If NO, no translation from EBCDIC to ASCII is performed for the application name when sending event to Transaction Collector. Otherwise the application name is translated from EBCDIC to ASCII. Set this value to YES (default) if the application name is an EBCDIC string, NO otherwise. Optional.

Syntax	Description
COMPONENT=*comp*	*comp* – Component Name. Constant string in single quotation marks or pointer to string (register R2 – R12 in brackets or Rx address). Default: BATCH (if batch) or STC (otherwise). Optional.
COMPONENTL=*length*	*length* – Length of Component name. Decimal constant in single quotation marks, register (R2-R12 in brackets) or halfword label. Optional if COMPONENT is a label or constant string
XLATEC=*yes/no*	If NO, no translation from EBCDIC to ASCII is performed for the component name when sending event to Transaction Collector. Otherwise the component name is translated from EBCDIC to ASCII. Set this value to YES (default) if the component name is an EBCDIC string, NO otherwise. Optional.
HOST=*host*	*host* – Host Name. Constant string in single quotation marks or pointer to string (register R2 – R12 in brackets or Rx address). Default: Sysplex name and z/OS SMFID (separated by a '/' – for example SYSPLEX1/MVS1). Optional.
HOSTL=*length*	*length* – Length of Host Name. Decimal constant in single quotation marks, register (R2-R12 in brackets) or halfword label. Optional if HOST is a label or constant string.
XLATEH=*yes/no*	If NO, no translation from EBCDIC to ASCII is performed for the host name when sending event to Transaction Collector. Otherwise the host name is translated from EBCDIC to ASCII. Set this value to YES (default) if the host name is an EBCDIC string, NO otherwise. Optional
TXN=*txn*	*txn* – Transaction Name. Constant string in single quotation marks or pointer to string (register R2 – R12 in brackets or Rx address). Default: (unknown). Optional
TXNL=*length*	*length* – Length of Transaction Name. Decimal constant in single quotation marks, register (R2-R12 in brackets) or halfword label. Optional if TXNL is a label or constant string.
XLATET=*yes/no*	If NO, no translation from EBCDIC to ASCII is performed for the transaction name when sending event to Transaction Collector. Otherwise the transaction name is translated from EBCDIC to ASCII. Set this value to YES (default) if the transaction name is an EBCDIC string, NO otherwise. Optional.

Syntax	Description
CHAINTO=*ptr*	*ptr* – Pointer to an existing Vertical Context list from which the Name/Value pair entries are chained. If there are no pair entries, pointer is zero. Pointer can point to any existing Name/Value pair in an existing Vertical Context list. If set, these Name/Value pairs will be added to the end of the list. Default 0. Optional.
MF=*form*	*form* one of S (inline), L (list form), or (E, list addr) (execute form). Default S. Optional.

Return codes

- none

Notes®

- This macro creates four chained Name/Value pairs that provide the minimum Vertical Context required for an event.
- If using execute form, ensure that list address is the list form of the CYTADFV macro, NOT the list form of the CYTANV macro.

Sample

```
    LA    R2,VCONTV
    LH    R3,=AL2(L'VCONTV)
NVNAMC3 CYTANV NAME='Division',Value='Payroll'
    CYTADFV TXN='Txn1',CHAINTO=NVNAMC3,MF=(E,NVNAMC4)
    BR  R14

NVNAMC4 CYTADFV MF=L
```

HLASM Macro: CYTAINIT

Macro to call the CYTA_init function that initializes Transaction Tracking API for HLASM callers.

Purpose

Initialize Transaction Tracking API for HLASM callers, and return a Configuration Token for use by other Transaction Tracking macros.

Input registers

No requirements.

Output registers

- R0 - 3 used as work register
- R1 holds Configuration token if R15 = 0, zero otherwise
- R2 - 13 unchanged
- R14 used as work register
- R15 return code

Syntax

Syntax	Description
name	*name*: symbol. Begin name in column 1
CYTAINIT	One or more blanks must follow CYTAINIT
SUB=*subsystem*	*subsystem* – Container subsystem – constant in single quotation marks (4 chars), register in brackets, or Rx address
MF=*form*	*form* one of S (inline), L (list form) or (E, list addr) (execute form). Default S. Optional.

Return codes

- As for CYTA_init function, see Appendix B, "Return codes," on page 67.

Sample

```
   CYTAINIT SUB='CYTZ'
   CYTAINIT SUB=#SUB1
   LA    R5,#SUB1
   CYTAINIT SUB=(R5),MF=(E,INITL)
INITL CYTAINIT MF=L
#SUB1 DC   C'CYTZ'
```

HLASM Macro: CYTANV

Macro to create a Name/Value pair entry.

Purpose

Prepare a Name/Value pair.

Input registers

No requirements.

Output registers

- R0 - 1 used as work register
- R2 - 13 unchanged
- R14 - 15 used as work register

Syntax

Syntax	Description
name	*name*: symbol. Begin name in column 1.
CYTANV	One or more blanks must follow CYTANV.
NAME=*name*	*name* – Name/Value pair name. Constant string in single quotation marks or pointer to **null terminated** string (register R2 – R12 in brackets or Rx address). Required.
VALUE=*value*	*value* – Name/Value pair value. Constant string in single quotation marks or pointer to string (register R2 – R12 in brackets or Rx address – value string does not have to be null terminated). Required.

Syntax	Description
LEN=*length*	*length* – Length of value. Decimal constant in single quotation marks, register (R2-R12 in brackets) or halfword label. Optional if VALUE is a label or constant string.
XLATEN=*yes/no*	If NO, no translation from EBCDIC to ASCII is performed for the name when sending events to Transaction Collector. Otherwise the name is translated from EBCDIC to ASCII. Set this value to YES (default) if the name is an EBCDIC string, NO otherwise. Optional.
XLATEV=*yes/no*	If NO, no translation from EBCDIC to ASCII is performed for the value when sending events to Transaction Collector. Otherwise the value is translated from EBCDIC to ASCII. Set this value to YES (default) if the value of the Name/Value pair is an EBCDIC string, NO otherwise. Optional.
CHAINTO=*ptr*	*ptr* – Pointer to an existing Name/Value pair list hat this Name/Value pair entry is to be chained off, or zero if none. Pointer can point to any existing Name/Value pair in an existing Name/Value pair list. If set, this Name/Value pair will be added to the end of the list. Default 0. Optional.
MF=*form*	*form* one of S (inline), L (list form) or (E, list addr) (execute form). Default S. Optional.

Return codes

- none

Notes

- Label is required for inline form of CYTANV.

Sample

```
NVNAMC3 CYTANV NAME='Transaction',VALUE=TXNVAL,     X
          LEN=TXNVALL
    LA    R2,VCONTV
    LA    R3,L'VCONTV
    CYTANV NAME=VCONTN,VALUE=(R2),LEN=(R3),     X
     CHAINTO=NVNAMC3,MF=(E,NVNAMC4)
    BR  R14

NVNAMC4  CYTANV MF=L
VCONTN   DC  'Process'        Name
         DC  X'00'            MUST be null terminated
VCONTV   DC  'ATM'            Value (not null terminated)
TXNVAL   DC  'Txn1'           Value (not null terminated)
TXNVALL  DC  AL2(L'TXNVAL)    Length of value
```

HLASM Macro: CYTATOK

Macro to call the CYTA_token function.

Purpose

Obtain fullword token unique across enterprise for HLASM callers.

Input registers

No requirements.

Output registers

- R0 used as work register
- R1 holds unique token if R15 = 0, zero otherwise
- R2 - 13 unchanged
- R14 used as work register
- R15 return code

Syntax

Syntax	Description
name	*name*: symbol. Begin name in column 1.
`CYTATOK`	One or more blanks must follow `CYTATOK`
`TOKEN=`*`token`*	*token* – Fullword Configuration token from `CYTAINIT` macro. Register in brackets, or Rx address. Required.
`MF=`*`form`*	*form* one of S (inline), L (list form) or (E, list addr) (execute form). Default S. Optional.

Return codes

- As for `CYTA_token` function. See Appendix B, "Return codes," on page 67.

Sample

```
    CYTAINIT SUB='CYTZ'
    ST  R1,ETOKEN
    LR  R5,R1
    CYTATOK TOKEN=ETOKEN
    CYTATOK TOKEN=(R5),MF=(E,TOKL)
TOKL   CYTATOK MF=L
ETOKEN  DS  F
```

HLASM Macro: CYTATRAK

Macro to call the CYTA_track function.

Purpose

Send completed event for HLASM callers.

Input registers

No requirements.

Output registers

- R0, R1 used as work register
- R2 - 13 unchanged
- R14 used as work register
- R15 return code

Syntax

Syntax	Description
name	*name*: symbol. Begin name in column 1.
`CYTATRAK`	One or more blanks must follow `CYTATRAK`
type	*type* – Type of event. Must be one of: STARTED, HERE, INBOUND, OUTBOUND, FINISHED, STARTED_IN, OUTBOUND_FIN, INBOUND_FIN. Required.
`TIME=`*`time`*	*time* – STCK value for timestamp of event. If not specified, current time is used. Register R2 – R12 in brackets, or Rx address. Optional.
`TXN=`*`transaction`*	*transaction* – Transaction Identifier. Constant string in single quotation marks or pointer to string (register R2 – R12 in brackets or Rx address). Optional.
`TXNLEN=`*`length`*	*length* – Length of Transaction Identifier. Decimal constant in single quotation marks, register (R2-R12 in brackets) or halfword label. Optional if TXN is a label or constant string.
`XLATET=`*`yes/no`*	If NO, no translation from EBCDIC to ASCII is performed for the Transaction Identifier when sending events to Transaction Collector. Otherwise the Transaction Identifier is translated from EBCDIC to ASCII. Set this value to YES (default) if the Transaction Identifier is an EBCDIC string, NO otherwise. Optional.
`TXNTXT=`*`list`*	*list* – Pointer to Transaction Context List – a list of Name/Value pointers. Pointer (register R2 – R12 in brackets or Rx address) or zero if none. Default 0. Optional.
`HTYPE=`*`type`*	*type* – Horizontal Caller type. Must be a valid Caller Type or a number between 0 and 255. Default is ANY. Optional.
`HLINK=`*`linkid`*	*linkid* – Horizontal Link. Constant string in single quotation marks or pointer to string (register R2 – R12 in brackets or Rx address). Optional.
`HLINKL=`*`length`*	*length* – Length of Horizontal Link. Decimal constant in single quotation marks, register (R2-R12 in brackets) or halfword label. Optional if HLINK is a label or constant string.

Syntax	Description
XLATEH=*yes/no*	If NO, no translation from EBCDIC to ASCII is performed for Horizontal Link when sending event to Transaction Collector. Otherwise Horizontal Link is translated from EBCDIC to ASCII. Set this value to YES (default) if Horizontal Link is an EBCDIC string, NO otherwise. Optional.
HCTXT=*list*	*list* – Pointer to Horizontal Context List – a list of Name/Value pointers. Pointer (register R2 – R12 in brackets or Rx address) or zero if none. Default 0. Optional
HSTITCH=*list*	*list* – Pointer to Horizontal Stitch List – a list of Name/Value pointers. Pointer (register R2 – R12 in brackets or Rx address) or zero if none. Default 0. Optional.
VTYPE=*type*	*type* – Vertical Caller type. Must be a valid Caller Type or a number between 0 and 255. Default is ANY. Optional.
VLINK=*linkid*	*linkid* – Vertical Link. Constant string in single quotation marks or pointer to string (register R2 – R12 in brackets or Rx address). Optional.
VLINKL=*length*	*length* – Length of Vertical Link. Decimal constant in single quotation marks, register (R2-R12 in brackets) or halfword label. Optional if VLINK is a label or constant string.
XLATEV=*yes/no*	If NO, no translation from EBCDIC to ASCII is performed for Vertical Link when sending events to the Transaction Collector. Otherwise Vertical Link is translated from EBCDIC to ASCII. Set this value to YES (default) if Vertical Link is an EBCDIC string, NO otherwise. Optional.
VCTXT=*list*	*list* – Pointer to Vertical Context List – a list of Name/Value pointers. Pointer (register R2 – R12 in brackets or Rx address) or zero if none. Default 0. Optional.
VSTITCH=*list*	*list* – Pointer to Vertical Stitch List – a list of Name/Value pointers. Pointer (register R2 – R12 in brackets or Rx address) or zero if none. Default 0. Optional.
TOKEN=*token*	*token* – Fullword Configuration token from CYTAINIT macro. Register in brackets, or Rx address. Required.
MF=*form*	*form* one of S (inline), L (list form) or (E, *list addr*) (execute form). Default S. Optional.

Return codes

- As for CYTA_track function. See Appendix B, “Return codes,” on page 67.

Notes

- If using execute form, event block is NOT cleared before event is sent. Any previous event values will not be reset.
- If execute form of CYTATRAK is used with no parameters except TOKEN, then the list address must point to a fully formed event block.
- Valid Caller Types are:
  - ANY – Type 0 – no specified caller type
  - GPS – Type 1 - GPS
  - ARM – Type 2 - ARM
  - WSA – Type 3 – WSA
  - CTG – Type 4 – CICS Transaction Gateway
  - MQ – Type 5 – Websphere MQ
  - SOA – Type 6 - SOA
  - WR – Type 7 – Web Resources
  - CICS – Type 8 - CICS
  - IMS – Type 9 – IMS
  - WMB – Type 10 – WebSphere Message Broker
  - DB2 – Type 11 – DB2
  - IMS Connect – Type 13 – IMS Connect
  - Tuxedo – Type 14 – Tuxedo
  - Optim Performance Manager – Type 15 – Optim Performance Manager
  - .Net – Type 16 – .NET
  - CAT – Type 17 – Citrix and Terminal servers
  - WRM – Type 18 – Web Response Monitors

  Caller Types can be any of these strings, or any number between 0 and 255. Caller Types 0-199 are reserved by IBM. Caller Types 200-255 are available to users.

Sample

```
CYTAINIT SUB='CYTZ'
ST R1,ETOKEN
LA       R5,VCONTXT
CYTATRAK TYPE=STARTED,VCTXT=VCONTXT,VLINK='LINK1', X
         HLINK=HLINK1,HLINKL=HLINK1L,TOKEN=ETOKEN
CYTATRAK TYPE=HERE,VCTXT=(R5),MF=(E,TRAKL)
* Code to fully populate event EVENT1 here
CYTATRAK TOKEN=ETOKEN,MF=(E,EVENT1)
BR R14
TRAKL CYTATRAK MF=L
EVENT1   CYTAEVNT MF=L
ETOKEN  DS F
#HLINK1  DC    C'HLinkID'
#HLINK1L DC    AL2(L'#HLINK1)
```

.NET bindings for Transaction Tracking API

Transaction Tracking supports .NET bindings on Windows 2003 and 2008 32-bit and 64-bit systems.

.NET bindings are included in the Transaction Tracking API. It consists of a single .NET Assembly DLL file, *IBM.Tivoli.TTAPI.dll*, and is packaged inside the Windows Transaction Tracking API SDK. The SDK also contains a sample application and API documentation.

Windows 2003 and 2008 32-bit and 64-bit system use different versions of .NET bindings. Windows 32-bit uses .NET 1.1, and Windows 64-bit uses .NET 2.0.

Chapter 3. Generic TCP Decoder

The Generic TCP Decoder enables you to develop modules for decoding and processing extra network protocols.

The Generic TCP Decoder consists of the following components:

- Web Response Time Module API framework
- Generic TCP Module

Generic TCP Decoder file location

You can find files for the Generic TCP Decoder in the following locations:

- On Linux and UNIX systems, `$CANDLE_HOME/tmaitm6/wrm/$plat/modules`
- On Windows 32-bit systems, `%CANDLE_HOME%\TMAITM6\wrm\Analyzer\modules`
- On Windows 64-bit systems, `%CANDLE_HOME%\TMAITM6\wrm\Analyzer\amd64\modules`

Web Response Time Module API

The Web Response Time Module API is an Application Programming Interface for developing third-party modules for decoding and processing network protocols using Web Response Time.

Overview

The API consists of two major components:

- Module management (definition, initialization, termination)
- Processing (decoding, information extraction)

Module management

Modules can be defined, initialized and terminated.

Defining a module

Modules are defined by two required elements:

- XML file specifying some information regarding the module, such as its name, what data it requires as input, and what it generates as output
- DLL shared object implementing the `wrt_module_t` API

The XML file schema describes:

- Name of the module (for example, the protocol name)
- Filters (for example, the TCP port number for traffic of interest to the module)
- Required input (context and metrics)
- Context and metrics generated by the module during processing

The `wrt_module_t` API is a C structure containing:

- Module version
- Module name (matching the name specified in the XML file)

- Function pointers for initializing and terminating an instance of the module, and for processing data

The C header provided defines a macro, `WRT_MODULE_DEFINE`, that should be used to define a module. For example:

```
WRT_MODULE_DEFINE(foo) = {
    WRT_MODULE_VERSION,
    "foo"
    &foo_init_function,
    &foo_term_function,
    &foo_process_function
};
```

The `WRT_MODULE_DEFINE` macro creates a globally visible symbol that defines the information described above, that is the module's name, API version, and entry points to the module. Use the `WRT_MODULE_VERSION` macro as the first argument, which is always the most current API version defined by the C header.

Module initialization

Web Response Time is multi-threaded, and has the capability to process multiple protocol sessions in parallel. To optimize performance, it is often preferable to have independent instances of a module for each thread, as they process independent protocol sessions.

The module's initialization function (as specified in the `wrt_module_t`) is invoked once for each thread performing data processing. The function, provided by the module implementor, accepts a `wrt_module_api_t*`, `wrt_module_config_t*`, and a (output parameter) `wrt_module_instance_t*`.

The `wrt_module_api_t` structure contains function pointers for callbacks to the module container. The `wrt_module_config_t` structure corresponds to and is pre-populated with the module's XML descriptor file contents.

If the module wishes to store any instance-specific data, it may store it in the `wrt_module_instance_t` pointer. This data is passed on to the other module functions (`terminate`, `process`).

Aside from general resource initialization, one significant activity for initialization functions is locating the unique numeric IDs of context and metric items. In order to minimize performance overhead in the API, context and metrics are referred to by a numeric ID. The XML file names each input and output context and metric; the `wrt_module_config_t` applies a unique numeric ID to each name.

If the initialization function returns a successful status, the module instance is provided with data for processing according to the filters specified in the module's XML descriptor file. If the function returns an error code, there is no processing, and no corresponding termination call; initialization calls that fail must ensure that they release all acquired resources before returning.

An example initialization function is listed below:

```
typedef struct foo_instance {
    wrt_context_id_t baz_id;
} foo_instance_t;

wrt_api_status_t foo_init_function(wrt_module_api_t *api,
                                   const wrt_module_config_t *config,
                                   wrt_module_instance_t *instance) {
```

```
    wrt_context_descriptor_t *d;
    foo_instance_t *foo_instance = malloc(sizeof(foo_instance_t));
    if (!foo_instance)
        return WRT_API_STATUS_NOMEM;
    memset(foo_instance, 0, sizeof(foo_instance_t));

    /* Look for the output context with the name "baz", and store its
     * numeric ID. */
    d = config->output_context;
    for (; d && !foo_instance->baz_id; d = d->next) {
        if (strcmp(d->name, "baz") == 0)
            foo_instance->baz_id = d->id;
    }
    if (!foo_instance->baz_id) {
        free(foo_instance);
        return WRT_API_STATUS_BADCFG;
    }

    *instance = foo_instance;
    return WRT_API_STATUS_OK;
}
```

Module termination

For each successful invocation of the initialization function, there is a matching call to the termination function when the module is unloaded (which is currently only at process termination time). The termination function is provided with the `wrt_module_instance_t` that may have been set by the initialization function; this may be used to manage resource lifetime.

An example initialization function is shown below. This function complements the `foo_init_function` listed in the previous section.

```
wrt_api_status_t foo_term_function(wrt_module_instance_t instance) {
    foo_instance_t *foo_instance = (foo_instance_t*)instance;
    /* release resources stored in foo_instance */
    free(foo_instance);
    return WRT_API_STATUS_OK;
}
```

Data processing

The purpose of a module is to extract some information from its input, and generate some contextual information and metrics as output.

Limitations

As this is a sample module, some protocols may have only limited implementation. For example, for LDAP, only search type requests are extracted. You can expand the information that can be decoded for a protocol using the “Generic TCP Module” on page 45.

Event-driven processing

Processing is triggered by a change in the input (for example, additional payload), which leads to an invocation of the module’s `process` function.

Note: The Web Response Time Module API is designed to support chaining of modules. This feature is not currently implemented, however it is referred to in this guide. In the current implementation, each user module is provided with data from a TCP or IPv4 segment reassembler, which delivers data to modules based on

request/reply state changes. Data from the user modules is not processed by any further modules; the data is sent directly to the Web Response Time agent for filtering, aggregation, and reporting.

Each invocation of the module's `process` function is provided with three parameters: the `wrt_module_instance_t` initialized by `init`, a `wrt_api_session_t` handle, and a `wrt_api_data_t` handle. The session handle is common for each call to `process` for the same network session (for example, TCP session); this handle can be used to maintain state between calls to `process`. The data handle provides access to the currently available request/reply data, context, and metrics.

TCP-based protocols are usually stateful, which means some state must be stored between calls to the module's `process` function to decode them. Even non-stateful protocols may require some state passing, as the processing may be provided with partial data that must be either processed immediately, or buffered by the module. Both scenarios are described in the following sections.

Storing state

As mentioned above, the `wrt_api_session_t` handle may be used to maintain state between calls to `process`. This can be done by using the `wrt_module_api_t` `set_userdata`, and `get_userdata` functions.

For example, to store some data in the session, use the following code:

```
void my_destructor(wrt_api_session_t session, void *data) {
    free(data);
}
...
void *userdata = malloc(sizeof(long)); /* any data that fits in void* */
wrt_api_status_t status = api->set_userdata(session, userdata, &my_destructor);
```

If the call to `set_userdata` succeeds (that is, it returns zero), retrieve the value later with `get_userdata`. When the session terminates, the destructor (if specified) will be invoked with the `session` and the `userdata` as arguments.

To retrieve the `userdata`, call the API as follows:

```
void *userdata;
wrt_api_status_t status = api->get_userdata(session, &userdata);
```

If no data was previously set, `get_userdata` returns `WRT_API_STATUS_NODATA`; otherwise it copies the value into the provided pointer, and then returns `WRT_API_STATUS_OK` (zero). For a new session, `userdata` is always unset. A common pattern for initializing state for session decoding is to first call `get_userdata`, check if `WRT_API_STATUS_NODATA` was returned, and if so create a new state object and call `set_userdata`.

```
struct my_session_state *state = NULL;
wrt_api_status_t status = api->get_userdata(session, (void**)&state);
if (status == WRT_API_STATUS_NODATA)
{
    state = malloc(sizeof(struct my_session_state));
    /* init state */
    status = api->set_userdata(session, state, &destroy_state);
    if (status != WRT_API_STATUS_OK)
    {
        /* catastrophic failure: could not set state. */
    }
}
```

Buffering data

In order to minimize resource requirements, the module container does not retain payload data after it has provided it to a module. If a module is presented with partial data, and the module cannot process the data until it is received in its entirety, the module must perform its own buffering.

To buffer data, use the session state and userdata mechanism described above. For example, you could store a state structure which contains the amount of data buffered so far, and a pointer to a heap-allocated copy of the data. The API flow in process is similar to the following:

1. Obtain or initialise the session state, using the pattern described above
2. Obtain the request/reply payload data, accumulating it into any previously buffered payload data.
3. Process the currently buffered data, and retain only the unprocessed data.

Contextual information and metrics

When a module processes some data, it may choose to send it along to the next module in the processing chain, typically with some additional information that it has extracted from the input. The data that is sent is transferred through a `wrt_api_data_t` handle.

A `wrt_api_data_t` has associated context information (e.g. the source and destination IP addresses, source and destination TCP ports), and some metrics (e.g. the request/reply response time, request timestamp, reply timestamp). Each module in a processing chain may add to or modify the values in the set, but never remove information. Thus, all input context and metrics are implicitly output; only their values may be modified, and additional context and metrics may be added.

To set context and metrics, a module requires a unique numeric ID for each context and metric item as described in "Module initialization". These IDs are provided to the module via a `wrt_module_config_t` structure, and the module supplies them to calls to the `get/set_metric` and `get/set_context` API functions.

Context example

A call to get or set context is shown below. In "Module initialization" on page 40 a context ID was extracted for the context item **baz**.

```
wrt_context_id_t baz_id; /* Assigned in foo_init_function. */
wrt_context_type_t ctx_type;
const void *ctx_value;
size_t ctx_size;
wrt_api_status_t status;

status = api->get_context(data, baz_id, &type, &ctx_value, &ctx_size);
/* Do something with the value, then update it. */
status = api->set_context(data, baz_id, type, ctx_value, ctx_size);
```

There are various in-built context items, depending on the underlying protocol. The numeric IDs for these context items can be obtained in the same way as described previously. For request/reply TCP or IPv4, the keys of the context items are:

Context key	Description	Type
tcp.srcport	Source TCP port	uint16
tcp.dstport	Destination TCP port	uint16
ipv4.srcaddr	Source IPv4 address	ipv4
ipv4.dstaddr	Destination IPv4 address	ipv4
ipv4.origsrcaddr	Original source IPv4 address	ipv4
ipv4.origdstaddr	Original destination IPv4 address	ipv4

Note: The value of ipv4.srcaddr may be updated to represent a source address other than the actual address, for example, for HTTP, report X-Forwarded-For. The value of ipv4.origsrcaddr should always be the actual source IPv4 address (for example, of the proxy server).

Metrics example

Metrics are handled similarly to context. See below for an example of using get_metric and set_metric:

```
wrt_metric_id_t server_time_id; /* Assigned in foo_init_function. */
wrt_metric_type_t type;
wrt_metric_value_t value;
wrt_api_status_t status;

status = api->get_metric(data, server_time_id, &type, &value);
/* wrt_metric_value_t is a union of basic integer types.
 * If you don't know the type of the metric ahead of time,
 * check the "type" variable updated by get_metric, and switch
 * on the result. For brevity, we assume a specific type here. */

/* The Server Time metric is an unsigned 64-bit integer. */
value.u64 += 42;
status = api->set_metric(data, server_time_id, type, value);
```

As with context, there are various in-built metrics, depending on the underlying protocol. For request/reply TCP or IPv4, the metrics are:

- tcp.response_time.total
- tcp.response_time.server
- tcp.response_time.network
- tcp.response_time.load
- tcp.response_time.resolve
- tcp.response_time.client_render

These metrics all have a type of uint64.

See the Enhanced network timing calculations for Web Response Time metrics in the *Administrator's Guide* for definitions of these metrics.

Trace logging

To enable you to debug a module, the API provides two functions for logging: init_log and log_message.

`init_log` is an optional function for registering a logging handle with a specified filename. The filename is used only for identifying log messages. A typical call to `init_log` looks like:

```
wrt_api_log_handle_t log_handle;
wrt_api_status_t status = api->init_log(__FILE__, &log_handle);
```

`log_message` formats and logs a message to the module container's log, optionally specifying a log handle initialized with `init_log`. The format is the same as in the C89/C99 `vprintf` function. Calls to `log_message` specify a log level, which is interpreted by the container to determine whether or not to log the message. A typical call to `log_message` looks like:

```
api->log_message(log_handle, WRT_API_LOG_ERROR,
                 __func__, __LINE__,
                 "send_data failed with status code %d",
                 (int)status);
```

The log handle parameter may optionally be NULL, in which case the log message is associated with a filename of the module container's choosing, instead of a filename specified by the module.

Generic TCP Module

The Generic TCP Module builds on the Web Response Time Module API framework to provide a simple configuration file approach to protocol decoding. The configuration file format is an extension of the normal Web Response Time Module API XML configuration file.

Basic Generic TCP Module configuration

Because theGeneric TCP Module file is an extension to the Web Response Time Module API configuration file, configuration directives used to define filtering and context generation are processed as normal by the framework. Refer to the Generic TCP Module in the name directive:

```
<module>
 <name>generic</name>
 ...
</module>
```

Filters and context output

Define filters and context output as you would for the Web Response Time Module API XML configuration file:

```
<filter>
        <port>21</port>
    </filter>
    <input>
    </input>
    <output>
       <context>
           <item>
               <name>tcp.protocol</name>
                <type>string</type>
            </item>
            <item>
                <name>ftp.client.name</name>
                <type>string</type>
            </item>
           <item>
                <name>ftp.command</name>
```

```
            <type>string</type>
        </item>
        <item>
            <name>ftp.responsecode</name>
            <type>string</type>
        </item>
    </context>
</output>
```

In this example, the decoder requests TCP data from port 21 (FTP), and produces four items of context (`tcp.protocol`, `ftp.client.name`, `ftp.command` and `ftp.responsecode`).

Protocol definitions and actions

Next in the XML configuration is the definition of instructions for the generic decoder. The decoder requires two configuration items: the protocol definition, and a set of actions. These are defined in the `<config>` section of the XML configuration file.

```
<config>
 <section name="rules">

NonCRLFChar = VCHAR / SP
CRLFTerminatedString = *NonCRLFChar CRLF

 ; Basic request flow
FTP_Session = &lt;ENTRYPOINT> FTP_Banner *FTP_Transaction &lt;END_OF_STREAM>
; The main loop

FTP_Banner = FTP_Responseline

; A transaction is a request followed by a response
FTP_Transaction = FTP_Requestline FTP_Responseline

; Request lines are read from the request stream
; All requests are single lines terminated with CRLF
FTP_Requestline = &lt;FROM REQUEST_STREAM> CRLFTerminatedString

; Response lines are read from the response stream
; Responses may be single line or multi line responses
FTP_Responseline = &lt;FROM RESPONSE_STREAM> FTP_SingleLineResponse /
FTP_MultiLineResponse

FTP_SingleLineResponse = 3DIGIT SP CRLFTerminatedString
; Single line response is nnn <message>

FTP_MultiLineMiddle = CRLFTerminatedString
FTP_MultiLineResponse = 3DIGIT "-" CRLFTerminatedString
; Multi line response is nnn-<message>
                        *FTP_MultiLineMiddle
; zero or more text lines
                        FTP_SingleLineResponse
; Terminated by a normal nnn <message>

;
; Deeper decoding
;

; FTP_Request extends the FTP_Requestline rule
FTP_Request = <FROM RULE FTP_Requestline> FTP_Commands CRLF
; Strip the CRLF
; By separating Command and Unknown_Command here,
; we can extend FTP_Command is subsequent rules

FTP_Unknown_Command = FTP_Unknown *NonCRLFChar
```

```
; Entire command is unknown but not necessarily

; bad. We just don't know/care about it
FTP_Unknown = 1*ALPHA

FTP_Commands = FTP_Command / FTP_Unknown_Command
FTP_Command = FTP_RETR_Command / FTP_STOR_Command / FTP_USER_Command
 / FTP_LIST_Command
FTP_RETR_Command = "RETR" 1*WSP FTP_Get_Filename
FTP_Get_Filename = *NonCRLFChar
; End will consume to the end of the current input

; In this case, that is the end of FTP_Requestline
FTP_STOR_Command = "STOR" 1*WSP FTP_Put_Filename
FTP_Put_Filename = *NonCRLFChar

FTP_USER_Command = "USER" 1*WSP FTP_Username
FTP_Username = *NonCRLFChar

FTP_LIST_Command = "LIST" 1*WSP FTP_ListParameters
FTP_ListParameters = *NonCRLFChar
; Response Handling
FTP_Reply = &lt;FROM RULE FTP_SingleLineResponse> FTP_ResponseCode SP
CRLFTerminatedString

; How do you like your responses reporteds ?
FTP_ResponseCode = 3DIGIT
; Single context with all codes


 </section>
 <section name="action:FTP_Username">
  session.command = "USER " .. MATCH
  session.ctxname = "ftp.client.name"
  session.ctxval = MATCH
 </section>
 <section name="action:FTP_Unknown">
  session.command = "ftp.command"
  session.ctxname = "ftp.command"
  session.ctxval = MATCH
 </section>
 <section name="action:FTP_Get_Filename">
  session.command = "GET " .. MATCH
  session.ctxname = "ftp.get.filename"
        session.ctxval = MATCH
 </section>
 <section name="action:FTP_Put_Filename">
  session.command = "PUT " .. MATCH
  session.ctxname = "ftp.put.filename"
  session.ctxval = MATCH
 </section>
 <section name="action:FTP_ListParameters">
  session.command = "LIST"
  session.ctxname = "ftp.list.parameters"
  session.ctxval = MATCH
 </section>
 <section name="action:FTP_ResponseCode">
  session.responsecode = MATCH
 </section>
 <section name="action:FTP_Transaction">
  set_context("tcp.protocol", "ftp")
  set_context("ftp.command", session.command);
  set_context(session.ctxname, session.ctxval)
  set_context("ftp.responsecode", session.responsecode)
  send_data()
 </section>
</config>
```

The first `config` section defines the logical structure of the protocol to be decoded. This definition is encoded as a `CDATA` block containing Augmented Backus-Naur Form (ABNF) rules describing the protocol. ABNF (defined in RFC 5234) is a format commonly used to define protocol structure in RFC documents. The generic decoder uses these rule definitions to decode packet data. As ABNF definitions do not normally include directional semantics (for example, server sends X, client sends Y) several directives are provided by the Generic TCP Module to indicate flow information in the rule definitions. These are declared as ABNF prose values (indicated by surrounding brackets, < >). Following the rule definitions, a series of action sections are used to map rule matches to output context values.

Generic TCP Decoder

The Generic TCP Decoder enables you to develop modules for decoding and processing extra network protocols.

The Generic TCP Decoder consists of the following components:

- Web Response Time Module API framework
- Generic TCP Module

Generic TCP Decoder file location

You can find files for the Generic TCP Decoder in the following locations:

- On Linux and UNIX systems, `$CANDLE_HOME/tmaitm6/wrm/$plat/modules`
- On Windows 32-bit systems, `%CANDLE_HOME%\TMAITM6\wrm\Analyzer\modules`
- On Windows 64-bit systems, `%CANDLE_HOME%\TMAITM6\wrm\Analyzer\amd64\modules`

Generic TCP Decoder rules

The decoding engine implemented in the Generic TCP Module takes the provided rules and attempts to match incoming data from the Web Response Time Module API against those rules. When rules are matched successfully, the engine passes the matched data back to the Generic TCP Module, which processes and associates action sections against the matched rule. The actions accumulate the matched data until a completed transaction is observed, and the data can be published to the Web Response Time Module API as context.

ABNF rule definition

The Generic TCP Decoder uses ABNF syntax and defines protocol structure in a normalized way. The decoder supports ABNF syntax as defined in RFC 5234. This section describes where the decoder syntax deviates from the standard.

Rules for the decoder are listed in the `rules` section of the `config` block. Each rule has a name and a series of elements and is terminated by the end of line. RFC 5234 defines a rule as being terminated by an internet standard newline (commonly referred to as CRLF). An internet standard newline is a carriage return (character 0x0d) followed by a line feed (character 0x0a). The Generic TCP Decoder differs from RFC 5234 in that it allows both CRLF and single LF termination of rules. When the next rule line is not a rule definition, the line is interpreted as a continuation of the current rule.

For example:

```
RuleA = DIGIT OCTET
```

is equivalent to:

```
RuleA = DIGIT
    OCTET
```

Comments may be included in the rule syntax. Comments are specified using the semicolon (;) character. Any text between the semicolon and the end of line is interpreted as a comment. Comments do not affect rule interpretation, so the following rule definitions are valid:

```
RuleA = DIGIT OCTET ; A single digit followed by an octet
RuleA = DIGIT  ; A single digit
     OCTET ; Followed by an octet
```

Basic rules

All rules are named. The name for a rule is part of the definition, and the rule is assigned the corresponding syntax

```
rulename = rule-elements
```

Fast path: RFC 5234 defines a `rulename` as starting with an alphabetic character, followed by any combination of other alphabetic characters, digits, and hyphens, and are not case sensitive. The Generic TCP Decoderdiffers in that `rule` names are case sensitive, and may also include underscore (_) characters.

Rule elements may be:

- A terminal value (that is, a specific character, byte or string)
- The name of another rule
- A repetition of `rule-elements`
- A sequence of `rule-elements`
- A choice of alternative `rule-elements`

Terminal values

Terminal values indicate an exact character, byte, or string match. If the input data matches the terminal value element, a match is indicated.

Single byte terminal values are defined using either their numerical value (specified in decimal or hexadecimal format), or as a string of length 1:

```
Digit_Zero = %x30
Digit_Zero = "0"
Digit_Zero = %d48
```

In this example, all three definitions are equivalent. The rule `Digit_Zero` matches the character `0`.

This is different to a byte match:

```
NULL_Byte = %x00
NULL_Byte = %d00
```

which matches the byte `0` (a NULL byte).

Strings of bytes may be defined using concatenation. For numerically defined values (specified in decimal or hexadecimal) the period (.) character is used to represent the concatenation. For literal strings, multiple characters may appear between the double quotation marks (" ").

For example, the following strings are equivalent:

```
hello = "hello"
hello = %d104.101.108.108.111
hello = %x68.65.6c.6c.6f
```

Decimal terminals are defined using the `%d` notation. Hexadecimal terminals are defined using the `%x` notation.

Both decimal and hexadecimal terminals may also define a range of bytes to match. Using the range specifier (`-`), you can specify a low and high byte. If the input byte is inside the range (or equal to either the low or high byte) a match is indicated.

For example:

```
lowercase_letter = %x61-7a
lowercase_letter = %d97-122
```

In this example, both strings match any single lowercase letter (a - z).

Decimal and Hexadecimal values must be in the range of one byte (`0 - 255` or `0x00 – 0xFF`). Enclose literal string terminals in double quotation marks (`" "`).

Restriction: RFC 5234 also defines bit values (`%b` notation). This is not supported.

Repetition rules

Normal repetitions are defined using an asterisk (`*`), with optional minimum and maximum indicators.

The rule element following a repetition definition must be matched a number of times according to these indicators.

The minimum indicator is a decimal number preceding the asterisk, and represents the minimum number of times the repeated element must match before the repetition may indicate a match. The minimum may be omitted to represent a minimum of `0` (an optional repetition).

The maximum indicator is a decimal number following the asterisk, and represents the maximum number of times the repeated element may match before the repetition indicates a match. The maximum indicator is used only (and maximal repetition checks are only made) if the minimum number of repetitions have matched. To represent an exact repetition, both the minimum and maximum indicators may be the same value. In this case, the asterisk may be omitted, and a single decimal number used to indicate the exact repetition.

For example:

```
lowercase_letter = %x61-7a
word = 1*lowercase_letter       ; 1 or more  letters
optional_word = *lowercase_letter   ; any number of letters
short_word = 1*5lowercase_letter   ; 1 to 5 letters
long_word = 6*lowecase_letter    ; 6 or more  letters
optional_short_word = *5lowercase_letter ; 0 to 5 letters
seven_letter_word = 7*7lowercase_letter  ; exactly 7 letters
seven_letter_word = 7lowercase_letter    ; exactly 7 letters
```

The following tables show examples of rule matches with different input.

Table 4. Rule matches with input "1 long day"

Rule	Indicates...	Match data
lowercase_letter	NO_MATCH	N/A
word	NO_MATCH	N/A
optional_word	MATCH	*empty*
short_word	NO_MATCH	N/A
long_word	NO_MATCH	N/A
optional_short_word	MATCH	*empty*
seven_letter_word	NO_MATCH	N/A

Table 5. Rule matches with input "the quick brown fox"

Rule	Indicates...	Match data
lowercase_letter	MATCH	t
word	MATCH	the
optional_word	MATCH	the
short_word	MATCH	the
long_word	NO_MATCH	N/A
optional_short_word	MATCH	the
seven_letter_word	NO_MATCH	N/A

Table 6. Rule matches with input "someone said hello"

Rule	Indicates...	Match data
lowercase_letter	MATCH	s
word	MATCH	someone
optional_word	MATCH	someone
short_word	MATCH	N/A
long_word	NO_MATCH	someone
optional_short_word	MATCH	N/A
seven_letter_word	NO_MATCH	someone

In all of these examples, the termination of the repetition where the maximum indicator is not specified is implied by the next byte of input not matching the repetition rule (in this case, the space between the words is not a lowercase letter). Take care to ensure that the repeated element does not arbitrarily match input data. Generally, the rule after the repetition is not processed until the repetition has returned a match.

The rule engine attempts to look ahead to the next rule to terminate repetitions as soon as possible. After the minimum number of repetitions has been matched, the look-ahead processing attempts to match the look-ahead rule. If it matches, the repetition is complete and a match is indicated. This look-ahead is limited to subsequent rules in the same rule definition.

Sequence rules

Sequences of rules are created by specifying more than one rule element in a single rule.

A rule that is defined using a sequence of elements does not indicate a match unless every rule in the sequence matches in the order specified. As soon as one of the elements of the sequence indicates that the input data does not match, the rule indicates that it does not match and will not process the sequence any further.

Sequences are defined by listing a set of rule elements separated with whitespace, either spaces or tab characters. For example:

```
SP = %x20
lowercase_letter = %x61-7a
word = 1*lowercase_letter
simple_sentence = word SP word SP word
```

The `simple_sentence` rule defines a sequence. For this rule to match, the sequence of elements must all match.

The following tables show the results of sentence matching with similar, but slightly different input.

Table 7. `simple_sentence` rule matches with input `"someone said hello"`

Rule	Sub rule	Sub input	Indicates...	Match data
`simple_sentence`			MATCH	`someone said hello`
	`word`	`someone said hello`	MATCH	`someone`
	`SP`	`said hello`	MATCH	`" "`
	`word`	`said hello`	MATCH	`said`
	`SP`	`hello`	MATCH	`" "`
	`word`	`hello`	MATCH	`hello`

With two spaces between `said` and `hello`:

Table 8. `simple_sentence` rule matches with input `"someone said hello"`

Rule	Sub rule	Sub input	Indicates...	Match data
`simple_sentence`			NO_MATCH	N/A
	`word`	`someone said hello`	MATCH	`someone`
	`SP`	`said hello`	MATCH	`" "`
	`word`	`said hello`	MATCH	`said`
	`SP`	`hello`	MATCH	`" "`
	`word`	`hello`	NO_MATCH	N/A

`simple_sentence` uses only as much input as it needs:

Table 9. `simple_sentence` rule matches with input `"someone said hello again"`

Rule	Sub rule	Sub input	Indicates...	Match data
`simple_sentence`			MATCH	`someone said hello`
	`word`	`someone said hello again`	MATCH	`someone`
	`SP`	`said hello again`	MATCH	`" "`
	`word`	`said hello again`	MATCH	`said`
	`SP`	`hello again`	MATCH	`" "`
	`word`	`hello again`	MATCH	`hello`

Here, the string `" again"` is left as input for the next rule.

Choice rules

Choice rules provide a list of elements to try to match against. If any of these elements matches, a match is indicated for the rule.

Choices are listed in a similar way to sequences, but use the forward slash (/) character to separate choices. For example:

```
lowercase_letter = %x61-7a
uppercase_letter = %41-5a
any_letter = lowercase_letter / uppercase_letter
```

The rule `any_letter` indicates a match if either the `lowercase_letter` rule or the `uppercase_letter` rule indicate a match.

You can provide any number of choice elements, however attempts to match choice elements are only made until a match is returned. When using choices to control logical flow of the rule processing, take care to ensure that the least specific match is placed last in the choice. Consider the following rules:

```
dog = "dog"
cat = "cat"
caterpiller = "caterpiller"
animal = dog / cat / caterpiller
```

Table 10. Input `"caterpillar"`

Rule	Sub rule	Sub input	Indicates...	Match data
`animal`			MATCH	`cat`
	`dog`	`dog`	NO_MATCH	N/A
	`cat`	`caterpillar`	MATCH	`cat`
	`caterpillar`	N/A	NOT_TRIED	N/A

In this example, the `cat` rule was processed before the `caterpillar` rule, and because the input starts with `cat` the rule returns a match. To correct this matching problem, ensure that the `caterpillar` rule is processed before the `cat` rule:

```
animal = caterpiller / dog / cat
```

Table 11. Input "cat"

Rule	Sub rule	Sub input	Indicates...	Match data
animal			MATCH	cat
	caterpillar	cat	NO_MATCH	N/A
	dog	cat	NO_MATCH	N/A
	cat	cat	MATCH	cat

Table 12. Input "caterpillar"

Rule	Sub rule	Sub input	Indicates...	Match data
animal			MATCH	caterpillar
	caterpillar	caterpillar	MATCH	caterpillar
	cat	N/A	NOT_TRIED	N/A
	any_other_animal	N/A	NOT_TRIED	N/A

Meta rules

The rule engine supports meta rules so that it can process request and response data from the Web Response Time Module API. These rules do not consume input data, but are used to identify the origin of the data and to match specific conditions. All meta rules are defined as prose values (values enclosed in angle brackets <>). There are two types of meta rules.

Flow control

Instructs the rule engine of special considerations for a rules data source. The normal source of input data for a rule is from a rules parent. Flow control meta rules provide alternative data sources for a rule. Flow control meta rules must appear as the first element of a rules definition, and a single rule may not use more than one flow control meta rule.

- Supported flow control meta rules are:
  - <ENTRYPOINT>

    Declares the rule that contains it to be the entry point for the rule engine. This is the first rule to be processed when a new protocol session is identified. Entrypoint rules do not have a datasource, and therefore may not reference rule elements that expect to have parent input data. There must be only one rule in the configuration that is defined as the entrypoint.
  - <FROM REQUEST_STREAM>

    Declares that the rule that contains it will read data from the request data stream provided by the Web Response Time Module API. Each time that a rule that has this meta rule indicates a match, the matched data is consumed from the request stream.
  - <FROM RESPONSE_STREAM>

    Declares that the rule that contains it will read data from the response data stream provided by the Web Response Time Module API. Each time that a rule that has this meta rule indicates a match, the matched data is consumed from the response stream.
  - <FROM RULE *rulename*>

    Declares that the rule that contains it will read data from matches indicated by the specified *rulename*. Any time that the specified rule matches, the data

matched is as the input to this rule. Processing of this rule completes before the rule that triggered the match continues.

Matching meta rules

Rules that can indicate matches without input data. Matches are made against specific conditions. Matching meta rules may be used wherever a normal rule element is valid, and are also valid for use in the `entrypoint` rule. Matching meta rules do not require and do not consume input data. Valid matching meta rules are:

- `<END_OF_REQUESTS>`

  Returns a match if the request data stream is complete, and all of the data from the request stream has been consumed. Otherwise it does not return a match.

- `<END_OF_RESPONSES>`

  Returns a match if the response data stream is complete, and all of the data from the request stream has been consumed. Otherwise it does not return a match.

- `<REMAINING BUFFER>`

  Returns a match against any data remaining in the parent buffer. If there is no data remaining in the buffer, the rule returns a match of zero length.

- `<MATCHED` *`rulename`*`>repeatedrule`

  May be used instead of a number for an exact repetition. The repetition length is defined by the value of the last match that occurred for the rule specified as *rulename*. For example:

  ```
  classnamelength = <BIG ENDIAN>2OCTET ; Length is stored as network byte
  order short
  classname = <MATCHED classnamelength>OCTET
  ```

  If the rule specified as *rulename* did not match, the repetition returns a zero length match.

Interpretation meta rules

Interpretation meta rules affect the interpretation of binary data. The interpreted data is applied when the containing rule is used as a length variable for the `<MATCHED>` repetition rule. Interpretation meta rules must appear as the first rule in a rules definition. Supported interpretation rules include the following:

- `<BIG ENDIAN>`

  Data matched by this rule is in big endian byte ordering

- `<NETWORK>`

  Data matched by this rule is in network byte order (synonym for `BIG ENDIAN`)

- `<LITTLE ENDIAN>`

  Data matched by this rule is in little endian byte ordering.

Built in rules

RFC 5234 defines several core rules.

The Generic TCP Module makes the following rules available for use in rule definitions:

```
ALPHA   =  %x41-5A / %x61-7A ; Characters A-Z / a-z
BIT     =  "0" / "1"
CHAR    =  %x01-7F      ; any 7-bit US-ASCII character
                           ;  excluding NULL
CR      =  %x0D       ; carriage return
CRLF    =  CR LF      ; Internet standard newline
CTL     =  %x00-1F / %x7F  ; controls
DIGIT   =  %x30-39     ; Characters 0-9
DQUOTE  =  %x22       ; Double Quote (")
HEXDIG  =  DIGIT / "A" / "B" / "C" / "D" / "E" / "F"
HTAB    =  %x09       ; horizontal tab
LF      =  %x0A       ; linefeed
LWSP    =  *(WSP / CRLF WSP) ; Linear whitespace
OCTET   =  %x00-FF     ; any single byte
SP      =  %x20
VCHAR   =  %x21-7E     ; printable characters
WSP     =  SP / HTAB    ; white space
```

Actions on rule matches:

Each rule may have an accompanying action. An action is a script that executes when the rule is matched, and is defined in the module's configuration XML file.

Defining actions

To define an action, create a configuration section like `<section name="action:$RULE">`, where `$RULE` is the name of a rule. For example:

```
<module>
 <name>generic</name>
 ...
 <config>
  <section name="action:MyRule">
   set_context("ctxname", MATCH)
   send_data()
  </section>
 </config>
</module>
```

Actions are Lua scripts, and have access to the full Lua standard library. You can do additional processing and manipulation on the extracted context before sending transactions. Use the examples in this section for simple scripts, or go to http://www.lua.org/ for the full syntax and standard library documentation.

Action API

An API is defined for actions to call to generate transaction data. The functions are described below:

- `get_context(name)` - gets the value of the context item with the specified name and value. The name must refer to a context item in the config file's output context.
- `set_context(name, value)` - sets the value of the context item with the specified name. The name may refer to a context item in the config file's input or output context.
- `send_data()` - sends a complete transaction datum.

In addition to the functions specified above, there are two built in identifiers provided by the framework:

- `MATCH` - the value of the matched rule that triggered the action
- `session` - a unique object for the TCP session, which may be used to store state

There is an additional function for logging from the match callbacks:

- `log_msg(text)` - output text to the log
- `*Optional*` - the message is logged at the `STATE` log level

Action execution

Actions are executed as rules are matched. There are several important things to consider in your actions:

- A sub-rule may be matched even though its enclosing rule is not. Thus, you should defer any `"send_data"` calls until you know that a complete transaction is matched.
- Calls to `set_context` may be ineffective if the accompanying `send_data` is not called from within the same action. This depends on when packets are observed by the module. Accumulate context in the `"session"` object, and then `set_context` and `send_data` in the rule that marks the end of a transaction.

Example - decoding FTP protocol

The FTP protocol is a text based protocol that uses a similar request and response structure to other internet protocols such as HTTP and SMTP. RFC 959 defines the FTP protocol, and there are numerous extensions to the protocol defined in subsequent RFCs.

Begin to define this protocol by firstly identifying the basic request and response flow. A normal FTP session begins with the client initiating a TCP connection to the FTP server. The FTP server sends the client a connection greeting, usually identifying the server and indicating that the client may proceed to send commands. From then on, the protocol is a series of requests from the client and responses from the server.

Requests to the server are simple strings terminated with a CRLF sequence. Responses from the server may be either single line response, with three digits followed by a space, followed by a text string terminated with a CRLF sequence, or they may be a multiline response, with three digits followed by a dash (`'-'`) followed by one or more text lines terminated with a CRLF sequence, followed by a normal single line response.

Express this as a set of rules as follows:

1. Define an `entrypoint` to define the general flow:

   ```
   FTP_Session = <ENTRYPOINT> FTP_Banner *FTP_Transaction <END_OF_RESPONSES>
   ```

   This rule indicates that it is the `entrypoint` for the engine. The rule logic is a sequence. It starts with an `FTP_Banner`, then has 0 or more `FTP_Transactions` and is completed when there are no more responses.
2. Define the rule elements.
   - The `FTP_Banner` is the same format as the response to a client request, so define this as:

     ```
     FTP_Banner = FTP_ResponseLine
     ```

An FTP Transaction represents a single client request and server response:

```
FTP_Banner = FTP_RequestLine FTP_ResponseLine
```

- Continue defining rules. First, the FTP_RequestLine which is a string terminated by a CRLF. To represent this, define:

  ```
  FTP_RequestLine = *NonCRLFCharacter CRLF
  NonCRLFCharacter = %x00-09 / %x0b-0c / %x0e-ff
  ```

  That is, the FTP_RequestLine is a sequence. The first element of the sequence is a repetition of any number of characters that are not CR or LF. The second element is the CRLF itself. The definition of NonCRLFCharacter may be a little lenient, as the protocol typically uses ASCII characters. The important fact is that it excludes the CR(0x0d) and LF(0x0a) characters.
- Define a rule for CRLF terminated strings as they are detected often:

  ```
  CRLFTerminatedString = *NonCRLFCharacter CRLF
  ```
- FTP_RequestLine also needs a source of data, so tell the engine to pass request data into it:

  ```
  FTP_RequestLine = <FROM REQUEST_STREAM> CRLFTerminatedString
  ```
- FTP Responses are more complicated, they may be either single or multi line responses. These are read from the response stream:

  ```
  FTP_ResponseLine = <FROM RESPONSE_STREAM>
    FTP_SingleLineResponse / FTP_MutliLineResponse
  ```
- Continue defining rules until they are terminal:

  ```
  FTP_SingleLineResponse = 3DIGIT SP CRLFTerminatedString
   FTP_MultiLineResponse = 3DIGIT "-" CRLFTerminatedString
        *CRLFTerminatedString
        FTP_SingleLineResponse
  ```

To ensure that FTP_ResponseLine chooses the right response type, the responses provide enough information to determine which it is. For it to match FTP_SingleLineResponse, it must be 3 DIGITs followed by a space, followed by a CRLF terminated string.

FTP_MultiLineResponse is 3 DIGITs followed by a dash. This rule uses the look ahead functionality of repetitions. In this case data that would match FTP_SingleLineResponse also matches CRLFTerminatedString. The look ahead detects the FTP_SingleLineResponse and completes the repetition so it can be processed.

This small set of rules is sufficient to process the FTP protocol. However, the output is limited to the full requests and responses. Ideally, you want to identify specific transactions and extract context data from them. To achieve this aim, define some more rules that take data from the FTP_RequestLine and FTP_ResponseLine rules.

Rather than creating a separate rule for single and multiline responses, create a single rule that triggers from matches to FTP_SingleLineResponse. Because FTP_MulitLineResponse uses the FTP_SingleLineResponse rule, the new rule is used to match and extract the response code from the server responses.

```
FTP_ResponseCode = <FROM RULE FTP_SingleLineResponse> 3DIGIT
```

This rule needs only match the 3 digits. The match for FTP_ResponseCode contains those 3 digits.

Processing the requests requires some more processing. There are any number of commands that the client could have sent, including commands that are not valid for the FTP protocol. For this example, we will build rules that extract the username that logged into the FTP server, and filenames for any file transfers.

Generally, FTP commands are in the format: command followed by an optional parameter. The command and parameter are separated by a space.

```
FTP_Request = <FROM RULE FTP_RequestLine> FTP_Commands CRLF
FTP_Commands = FTP_USER_Command / FTP_RETR_Command
   / FTP_STOR_Command / FTP_Other_Command
FTP_USER_Command = "USER" SP FTP_Username
FTP_Username = *NonCRLFCharacter
FTP_STOR_Command = "STOR" SP FTP_Put_Filename
FTP_Put_Filename = *NonCRLFCharacter
FTP_RETR_Command = "RETR" SP FTP_Get_Filename
FTP_Get_Filename = *NonCRLFCharacter
FTP_Other_Command = *NonCRLFCharacter
```

These rules process the request line. If the request is a `USER` command, the `FTP_Username` rule matches the parameter. Similar rules are defined for the parameters of `STOR` and `RETR`. These rules are defined separately so that they can have specific actions defined for them, rather than having one action that would require extra logic to determine the command type.

The `FTP_Other_Command` rule matches the entire line for rules that are not of interest. Either publish these, or silently ignore them.

Finally, do not expose user passwords. If you publish `FTP_Other_Command`, user password may become visible. Add the `PASS` command to our list of commands to avoid exposing passwords:

```
FTP_Commands = FTP_USER_Command / FTP_RETR_Command
   / FTP_STOR_Command / FTP_PASS_Command
   / FTP_Other_Command
FTP_PASS_Command = "PASS" SP *NonCRLFCharacter
```

Publishing context for FTP

Define actions to publish the matches to the Web Response Time Module API. For this example, for each ftp transaction, publish the username (if the user has logged in), the request type and the response code the server returned. Also inform the Web Response Time Module API what the protocol is.

1. Define the context to publish:

```
<output>
        <context>
            <item>
                <name>tcp.protocol</name>
                <type>string</type>
            </item>
            <item>
                <name>ftp.client.name</name>
                <type>string</type>
            </item>
            <item>
                <name>ftp.command</name>
                <type>string<type>
            </item>
            <item>
                <name>ftp.put.filename<name>
                <type>string</type>
            </item>
```

```
            <item>
                <name>ftp.get.filename</name>
                <type>string<type>
            </item>
            <item>
                <name>ftp.responsecode</name>
                <type>string</type>
            </item>
        </context>
    </output>
```

2. Associate actions with the FTP rules to populate this context:

```
<section name="action:FTP_Username">
            session.command = "USER " .. MATCH
            session.ctxname = "ftp.client.name"
            session.ctxval = MATCH
        </section>
        <section name="action:FTP_Other_Command">
            session.command = "ftp.command"
            session.ctxname = "ftp.command"
            session.ctxval = MATCH
        </section>
        <section name="action:FTP_Get_Filename">
            session.command = "GET " .. MATCH
            session.ctxname = "ftp.get.filename"
            session.ctxval = MATCH
        </section>
        <section name="action:FTP_Put_Filename">
            session.command = "PUT " .. MATCH
            session.ctxname = "ftp.put.filename"
            session.ctxval = MATCH
        </section>
        <section name="action:FTP_ResponseCode">
            session.responsecode = MATCH
        </section>
        <ection name="action:FTP_Transaction">
            set_context("tcp.protocol", "ftp")
            set_context("ftp.command", session.command);
            set_context(session.ctxname, session.ctxval)
            set_context("ftp.responsecode", session.responsecode)
            send_data()
        </section>
```

For this example, the actions for matches on `FTP_Username`, `FTP_Other_Command`, `FTP_Get_Filename`, `FTP_Put_Filename`, and `FTP_ResponseCode` store the value of the match in the internal session storage.

The definition for `FTP_Transaction` is a `Request` followed by a `Response`, so when the action is triggered for a match on `FTP_Transaction`, there are accumulated matches from one of the request rules and from the response rule. The stored values in the session storage are published through the Web Response Time Module API and the transaction record is sent.

Full FTP configuration

```
<module>
 <name>generic</name>
 <filter>
  <port>21</port>
 </filter>
 <input>
 </input>
 <output>
        <context>
            <item>
```

```
                <name>tcp.protocol</name>
                <type>string</type>
            </item>
            <item>
                <name>ftp.client.name</name>
                <type>string</type>
            </item>
            <item>
                <name>ftp.command</name>
                <type>string<type>
            </item>
            <item>
                <name>ftp.put.filename<name>
                <type>string</type>
            </item>
            <item>
                <name>ftp.get.filename</name>
                <type>string<type>
            </item>
            <item>
                <name>ftp.responsecode</name>
                <type>string</type>
            </item>
        </context>
    </output>
  <config>
  <section name="rules">
   <![CDATA[
NonCRLFCharacter = %x00-09 / %x0b-0c / %x0e-ff
                 ; Matches any byte that isn't CR or LF
CRLFTerminatedString = *NonCRLFCharacter CRLF
                  ; Matches all non CRLF bytes, then CRLF

 ; Basic request flow
 ; This is the main 'loop' for the session
FTP_Session = <ENTRYPOINT> FTP_Banner *FTP_Transaction <END_OF_RESPONSES>

FTP_Banner = FTP_ResponseLine
   ; The FTP Server greeting is really just a server
      ; response without a request

FTP_Transaction = FTP_RequestLine FTP_ResponseLine
; A transaction is a request followed by a response

; Request lines are read from the requestStream
FTP_RequestLine = <FROM REQUEST_STREAM> CRLFTerminatedString

; Response lines are read from the responseStream
; Responses may be single line or multi line responses
FTP_ResponseLine = <FROM RESPONSE_STREAM> FTP_SingleLineResponse /
FTP_MultiLineResponse
FTP_SingleLineResponse = 3DIGIT SP CRLFTerminatedString
     ; Single line response is nnn <message>

FTP_MultiLineResponse = 3DIGIT "-" CRLFTerminatedString
     ; Multi line response is nnn-<message>
                         *CRLFTerminatedString
         ; zero or more text lines
                         FTP_SingleLineResponse
       ; Terminated by a normal nnn <message>

; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;
; Deeper decoding
;
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
; Response Handling
; FTP_ResponseCode processes matches of the FTP_SingleLineResponse
; FTP_SingleLineResponse will match on a single line response, or the last line
; of a multi line response
FTP_ResponseCode = <FROM RULE FTP_SingleLineResponse> 3DIGIT
   ; All we want is the first 3 digits

; Request Handling
; FTP_Request processes matches of the FTP_RequestLine rule
FTP_Request = <FROM RULE FTP_RequestLine> FTP_Commands CRLF
 ; Strip the CRLF

FTP_Commands = FTP_USER_Command
             / FTP_RETR_Command
             / FTP_STOR_Command
             / FTP_PASS_Command
             / FTP_Other_Command

FTP_USER_Command = "USER" SP FTP_Username
    ; We could use 1*WSP to account for multiple spaces
FTP_Username = *NonCRLFCharacter
; We are separating the username parameter from the command so that we can expose
;  it as separate context

FTP_STOR_Command = "STOR" SP FTP_Put_Filename
FTP_Put_Filename = *NonCRLFCharacter

FTP_RETR_Command = "RETR" SP FTP_Get_Filename
FTP_Get_Filename = *NonCRLFCharacter

; FTP_PASS_Command and FTP_Other_Command are identical rules - however
; FTP_PASS_Command is referenced before FTP_Other_Command, so it will
; match first. This will prevent the FTP_Other_Command action handler
; exposing passwords in the context.

FTP_PASS_Command = "PASS" SP *NonCRLFCharacter

FTP_Other_Command = *NonCRLFCharacter

    ]]>
   </section>
   <section name="action:FTP_Username">
            session.command = "USER " .. MATCH
            session.ctxname = "ftp.client.name"
            session.ctxval = MATCH
       </section>
   <section name="action:FTP_Other_Command">
            session.command = "ftp.command"
            session.ctxname = "ftp.command"
            session.ctxval = MATCH
       </section>
   <section name="action:FTP_Get_Filename">
            session.command = "GET " .. MATCH
            session.ctxname = "ftp.get.filename"
            session.ctxval = MATCH
       </section>
   <section name="action:FTP_Put_Filename">
            session.command = "PUT " .. MATCH
            session.ctxname = "ftp.put.filename"
            session.ctxval = MATCH
       </section>
   <section name="action:FTP_ResponseCode">
            session.responsecode = MATCH
       </section
   <section name="action:FTP_Transaction">
            set_context("tcp.protocol", "ftp")
            set_context("ftp.command", session.command);
```

```
            set_context(session.ctxname, session.ctxval)
            set_context("ftp.responsecode", session.responsecode)
            send_data()
        </section>
 </config>
</module>
```

Appendix A. Transport address format

This appendix provides definitions of the available addressing schemes for connecting an instrumented application to a Transaction Collector.

The addressing format that Transaction Tracking API uses is modular - multiple transports may be available depending on which platform the Transaction Tracking API is running. A subset of transports is guaranteed to be available on all platforms. Each transport has a unique string associated with it.

The addresses used by Transaction Tracking API are always prefixed by the unique identifier of the transport to be used, followed by a colon (:). The remainder of the address following the colon, is interpreted in a module-dependant manner as follows:

```
<module>:<address>
```

The following sections describe each module and the formats they define for their addresses.

TCP/IP module (tcp)

The TCP/IP transport supports both IPv4 and IPv6 (where the platform supports IPv6). The module's unique identifier is `tcp`. This format is not supported on z/OS.

Addresses for this module follow a URL-like format:

```
tcp:host:port
```

To allow the use of IPv6 addresses in this format, the host must be enclosed in square brackets (as is done in URLs). For example, to connect to port 5455 on the IPv6 local host, specify the following address:

```
tcp:[::1]:5455
```

The default TCP/IP value is `tcp:127.0.0.1:5455`.

Subsystem module (ssn)

The subsystem format is supported only on z/OS, and specifies the four character subsystem name of the destination Transactions Container.

Addresses for this module are:

```
ssn:subsystem
```

For example, `ssn:SS01`.

The default SSN value is `ssn:CYTZ`. All other fields in the Configuration Block are ignored for Subsystem users.

Appendix B. Return codes

Transaction Tracking API functions return values in a return code – a fullword area.

The return codes are:

- **0** Operation successful
- **10 - 30: z/OS Related Error** (returned from calls to CYTATRAK and CYTABLOK):
  - 10 - Container subsystem not found
  - 11 - Invalid Configuration Token passed
  - 12 - Container subsystem inactive
  - 13 - Dispatcher unavailable
  - 14 - No more storage available
  - 15 - No more ASIDs in ASID pool
  - 16 - No Couriers available
  - 17 - System Error Occurred
  - 18 - Invalid API request
  - 19 - ASID table full, no more ASIDs can be tracked. Only 100 address spaces can be tracked by each Container. Start another Container and configure some of your data collectors to use this Container.
  - 20 - Free chain anchor is null. The CADS is full. The Container may not be able to keep up with the incoming work load. Start another Container and configure some of your data collectors to use this Container.
  - 21 - Not used
  - 22 - Free block is flagged as in use.
  - 23 - Can not find ASID for tracked event
  - 24 - CBH pointer in ASID block is null
  - 25 - Event block has been queued but not yet processed by the Container.
  - 26 - CADS not available
  - 27 - Block has already been queued to the Container.
  - 28 - Block is already free.
  - 29 - QELM header is bad.
  - 30 - Block being freed is on the active queue.
- **100-200 Event Record Invalid:**
  - 100 - Internal Error
  - 101 - Invalid event block address
  - 102 - Invalid event type
  - 103 - Invalid timestamp
  - 104 - Invalid transaction ID length (not positive)
  - 105 - Invalid transaction ID flag
  - 106 - Invalid transaction ID address
  - 110 - Invalid horizontal ID length (not positive)
  - 111 - Invalid horizontal ID flag

- 112 - Invalid horizontal caller type
- 113 - Invalid horizontal ID address
- 120 - Invalid vertical ID length (not positive)
- 121 - Invalid vertical ID flag
- 122 - Invalid vertical caller type
- 123 - Invalid vertical caller address
- 130 - Invalid Name string address in transaction list
- 131 - Invalid Value string address in transaction list
- 132 - Invalid Value string length in transaction list
- 133 - Invalid Name Value flag in transaction list
- 134 - Invalid Name/Value pair address in transaction list
- 140 - Invalid Name string address in horizontal stitch list
- 141 - Invalid Value string address in horizontal stitch list
- 142 - Invalid Value string length in horizontal stitch list
- 143 - Invalid Name Value flag in horizontal stitch list
- 144 - Invalid Name/Value pair address in horizontal stitch list
- 150 - Invalid Name string address in vertical stitch list
- 151 - Invalid Value string address in vertical stitch list
- 152 - Invalid Value string length in vertical stitch list
- 153 - Invalid Name Value flag in vertical stitch list
- 154 - Invalid Name/Value pair address in vertical stitch list
- 160 - Invalid Name string address in horizontal context list
- 161 - Invalid Value string address in horizontal context list
- 162 - Invalid Value string length in horizontal context list
- 163 - Invalid Name Value flag in horizontal context list
- 164 - Invalid Name/Value pair address in horizontal context list
- 170 - Invalid Name string address in vertical context list
- 171 - Invalid Value string address in vertical context list
- 172 - Invalid Value string length in vertical context list
- 173 - Invalid Name Value flag in vertical context list
- 174 - Invalid Name/Value pair address in vertical context list

Appendix C. Samples

Code examples.

z/OS samples provided

On z/OS, examples can also be found in the SCYTSAMP file, as described in Table 13.

Table 13. Samples in the SCYTSAMP library

Language	Member	Description
HLASM	CYTAASM	Sample HLASM program to send events.
	CYTAINIT	Macro to call the CYTA_init function.
	CYTATRAK	Macro to call the CYTA_track function.
	CYTATOK	Macro to call the CYTA_token function.
	CYTAEVNT	Macro to map an event.
	CYTACFG	Macro to map a Configuration Block.
	CYTANVAL	Macro to map a Name/Value pair entry.
	CYTANV	Macro to create a Name/Value pair entry.
	CYTADFV	Macro to create Name/Value pairs to specify the minimal Vertical Context for an event.
COBOL	CYTABCFG	COBOL definitions for a Configuration Block.
	CYTABCON	COBOL constants required to use the Transaction Tracking API.
	CYTABEVT	COBOL definitions for an event.
	CYTABNV	COBOL definitions for a Name/Value pair entry.
	CYTABSMP	Sample COBOL program to send events.
C	CYTACSMP	Sample C program to send events. C header file definitions are in the SCYTH dataset.
Java	CYTAJSMP	Sample Java program to send events.
PL/I	CYTAPCFG	PL/I definitions for a Configuration Block.
	CYTAPEVT	PL/I definitions for an event.
	CYTAPNV	PL/I definitions for a Name/Value pair entry.
	CYTAPSMP	Sample PL/I program to send events.
All	CYTASIDE	Binder Side Deck required by dynamic callers.

C

```
/*================================================================
  Includes
  ================================================================*/
#pragma runopts(POSIX(ON))
#pragma runopts(TRAP(ON,SPIE))
#include <cytapi.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
```

```
/*================================================================
Mainline Code
================================================================ */
int main(int argc, char **argv)
{
    /* ------------------------------------------------------------
       Variables
       ------------------------------------------------------------ */
      int rc;                           /* Return code from functions  */
      int hlink1;                       /* Store numeric Hor Link IDs  */
      /* --- Area for Communications Configuration Block -------------- */
     cyta_config_t configblk;           /* Configuration block         */
      /* --- Area for Event Block ------------------------------------ */
     cyta_event_t eventblk;             /* Event block                 */
      /* --- Area for 'Standard' Vertical Context Name/Value Pairs ---- */
     cyta_values_list_t vert1, vert2, vert3, vert4; /* Vertical Context*/
     /* ------------------------------------------------------------
       Put together 'standard' Vertical Contexts

   These are the minimum Vertical Context needed to display
       event in ITCAM for Txns workspaces. We need:
          HOST - Host Name. Normally Sysplex/SMFID
          COMPONENT - Component - what we are running under -eg.
             BATCH, STC, IMS, CICS, WAS, TSO.
          APPLICATION - Usually Job or Started Task Name
          TRANSACTION - The transaction we are running

       In your program, you will modify the values to suit your
       installation, however the names of the name/value pairs should
       not be changed
       ------------------------------------------------------------ */
    /* --- Hostname ------------------------------------------------ */
     memset(&vert1, 0, sizeof(vert1));
     static char *server = "Sysplex/Host";
     static char *server_lbl = "ServerName";
     vert1.name = server_lbl;
     vert1.value = server;
     vert1.size = strlen(server);
     /* --- ComponentName------------------------------------------- */
     memset(&vert2, 0, sizeof(vert2));
     static char *component = "STC";
     static char *component_lbl = "ComponentName";
     vert1.next = &vert2;
     vert2.name = component_lbl;
     vert2.value = component;
     vert2.size = strlen(component);
     /* --- ApplicationName ----------------------------------------- */
     memset(&vert3, 0, sizeof(vert3));
     static char *application = "Application";
     static char *application_lbl = "ApplicationName";
     vert2.next = &vert3;
     vert3.name = application_lbl;
     vert3.value = application;
     vert3.size = strlen(application);
     /* --- Transaction (no EBCDIC -> ASCII Translation) ------------- */
     memset(&vert4, 0, sizeof(vert4));
     static int transaction = 254;      /* Value is a number: 254      */
     static char *transaction_lbl = "TransactionName";
     vert3.next = &vert4;
     vert4.name = transaction_lbl;
     vert4.value = &transaction;
     vert4.size = sizeof(transaction);
     vert4.flags = CYTA_VALUELIST_FLAG_VALUE_RAW;/* No translation     */
     /* ------------------------------------------------------------
       Get Configuration Token
          The server field specifies where the ITCAM for Transactions
          events are to be sent. It must be of the form:
```

```
            SSN:sub
          Where sub is the 4 character subsystem name used by the ITCAM
          for Transactions Collector Started Task
        ------------------------------------------------------------- */
      memset(&configblk, 0, sizeof(configblk));  /* Zero config block   */
      configblk.server = "SSN:CYTZ";     /* Send events to CYTZ subsys  */
      rc = CYTA_init(&configblk);        /* Get the token               */
      printf("(CYTACSMP) CYTAINIT Return Code=%d\n", rc);
      /* ------------------------------------------------------------
        Send a 'Started' Event
          We don't specify a timestamp, so the time now is automatically
          inserted.
        ------------------------------------------------------------- */
      memset(&eventblk, 0, sizeof(eventblk));  /* Zero event block      */
      eventblk.type = CYTA_STARTED_EVENT;      /* Started Event         */
      eventblk.vertical_context = &vert1;      /* Vertical Context Addr */
      eventblk.vertical_id.link_id = "CYTACSMP";  /* Vertical Link ID   */
      eventblk.vertical_id.link_id_size = 8;   /* Link ID Length        */
      rc = CYTA_track(&configblk, &eventblk);  /* Send the event        */
      printf("(CYTACSMP) STARTED Event Return Code=%d\n", rc);
      /* ------------------------------------------------------------
        Send an Outbound Event
        (we only need to specify changed fields - all other fields remain
        from the Started event)
        ------------------------------------------------------------  */
      eventblk.type = CYTA_OUTBOUND_EVENT;     /* Started Event         */
      eventblk.horizontal_id.link_id = "Hlink Value";/* Horizontal ID   */
      eventblk.horizontal_id.link_id_size = 11;/* Link ID Length        */
      rc = CYTA_track(&configblk, &eventblk);  /* Send the event        */
      printf("(CYTACSMP) OUTBOUND Event Return Code=%d\n", rc);
      /* ------------------------------------------------------------
        Send an Inbound Finished Event
        We change the Horizontal Link to the incoming Link ID - this
         is the ID specified by the application sending the response
         in its OUTBOUND event. In this case, the Link ID is a number,
         so we MUST set the flag so it is NOT translated from EBCDIC to
         ASCII.
        ------------------------------------------------------------  */
      eventblk.type = CYTA_INBOUND_FINISHED_EVENT;   /* Event Type      */
      hlink1 = 56;                       /* Horizontal Link ID=56       */
      eventblk.horizontal_id.link_id = &hlink1 /* Horizontal ID         */
      eventblk.horizontal_id.link_id_size = 4; /* Link ID Length        */
      eventblk.horizontal_id.flags = CYTA_ASSOCIATION_FLAG_LINK_RAW;
                                         /* Do NOT xlate from EBCDIC    */
      rc = CYTA_track(&configblk, &eventblk);  /* Send the event        */
      printf("(CYTACSMP) INBOUND FINISHED Event Return Code=%d\n", rc);
 }                                       /* main                        */
```

COBOL

```
CBL   RENT,PGMNAME(LM),LIB,NODYNAM,NODLL
      * ===============================================================
      *
      * Identification Division
      *
      * ===============================================================
       IDENTIFICATION DIVISION.

       PROGRAM-ID. "CYTABSMP".

      * ===============================================================
      *
      * Environment Division
      *
      * ===============================================================
       ENVIRONMENT DIVISION.
```

```
      * ============================================================
      *
      * Data Division
      *
      * ============================================================
       DATA DIVISION.
       Working-Storage Section.
      * ------------------------------------------------------------
      * Constants Needed to Use the ITCAM for Txns API
      * ------------------------------------------------------------
       COPY CYTABCON.

      * ------------------------------------------------------------
      * Area to hold our event block
      * ------------------------------------------------------------
       COPY CYTABEVT.

      * ------------------------------------------------------------
      * Area to hold our Configuration Block
      * ------------------------------------------------------------
       COPY CYTABCFG.

      * ------------------------------------------------------------
      * Area for 'Standard' 4 Vertical Context Name/Value pairs.
      *    These are the minimum Vertical Context needed to display
      *    event in ITCAM for Txns workspaces
      *       HOST - Host Name. Normally Sysplex/SMFID
      *       COMPONENT - Component - what we are running under -eg.
      *          BATCH, STC, IMS, CICS, WAS, TSO.
      *       APPLICATION - Usually Job or Started Task Name
      *       TRANSACTION - The transaction we are running
      *
      *    Each has three variables:
      *       xxx-LBL - the label of the Name/Value Pair, ending in
      *                 nulls.
      *       xxx     - area to hold the actual value
      *       xxx-NV  - area to hold the name/value pair
      *
      *    In your program, you will modify the values to suit your
      *    installation, however the labels should not be changed
      * ------------------------------------------------------------
       01  HOST-LBL               pic x(11) value z"ServerName".
       01  HOST                  pic x(12) value "Sysplex/Host".
       01  HOST-NV                pic x(16).

       01  COMPONENT-LBL          pic x(14) value z"ComponentName".
       01  COMPONENT              pic x(3)  value "STC".
       01  COMPONENT-NV           pic x(16).

       01  APPLICATION-LBL        pic x(16) value z"ApplicationName".
       01  APPLICATION            pic x(11) value "Application".

01  APPLICATION-NV         pic x(16).

       01  TRANSACTION-LBL        pic x(16) value z"TransactionName".
       01  TRANSACTION            pic s9(9) binary value 254.
       01  TRANSACTION-NV         pic x(16).

      * ------------------------------------------------------------
      * Outbound and Inbound Horizontal Link IDs
      *    For your organisation, specify unique values here. But
      *    for this example, constant values will be used
      *    HLINK-OUT - Outgoing Horizontal Link (string)
      *    HLINK-IN  - Incoming Horizontal Link (number)
      * ------------------------------------------------------------
       01  HLINK-OUT                   pic x(11) value 'HLINK VALUE'.
       01  HLINK-IN                    pic s9(9) binary value 56.
```

```
* -------------------------------------------------------------
* Vertical Link ID
*    This value should be unique for every work unit. We will
*    use a constant in this example.
* -------------------------------------------------------------
 01  VLINK                      pic x(8) value 'CYTABSMP'.

* -------------------------------------------------------------
* String Specifying Destination for Events
*    This must be of the form SSN:sub - sub is the ITCAM for
*    Transactions Container subsystem.
* -------------------------------------------------------------
 01  SERVER                     pic x(8) value 'SSN:CYTZ'.

* -------------------------------------------------------------
* Definition used to insert a one byte length field
*    LINK-LEN - Halfword Link Name length
*    LINK-LEN-BYTE - One byte Link Name length
* -------------------------------------------------------------
 01  LINK-LEN                   pic s9(3) binary.
 01  LINK-LEN-STR               redefines LINK-LEN.
     02   filler                pic x(1).
     02   LINK-LEN-BYTE         pic x(1).

* -------------------------------------------------------------
* Fullword to hold return code from CYTA_track
* -------------------------------------------------------------
 01  RC                         pic S9(9) comp.

 Linkage Section.
* -------------------------------------------------------------
* Map Name/Value Pair Entry
* -------------------------------------------------------------
 COPY CYTABNV.

* =============================================================
*
* Procedure Division
*
* =============================================================
 PROCEDURE DIVISION.
     DISPLAY "(CYTABSMP) Entry".

* -------------------------------------------------------------
* Initialize Our Event Block
* -------------------------------------------------------------
     INITIALIZE CYTA-EVENT REPLACING ALPHANUMERIC BY x"00".

* -------------------------------------------------------------
* Setup a Name/Value Pair for Host
*    1. Address the Name/Value pair
*    2. Initialise the Name/Value pair to nulls
*    3. Set the Name pointer
*    4. Set the Value pointer
*    5. Set the Value length
* -------------------------------------------------------------
     SET ADDRESS OF CYTA-NV-LIST TO ADDRESS OF HOST-NV.
     INITIALIZE CYTA-NV-LIST REPLACING ALPHANUMERIC BY x"00".
     SET CYTA-NV-NAME-POINTER TO ADDRESS OF HOST-LBL.
     SET CYTA-NV-VALUE-POINTER TO ADDRESS OF HOST.
     MOVE LENGTH OF HOST TO CYTA-NV-VALUE-LENGTH.

* -------------------------------------------------------------
* Setup a Name/Value Pair for Component
*    1. Host Name/Value pair chains to Component Name/Value pair
*    2. (steps as for Host Name/Value Pair)
```

```
*    7. Stop EBCDIC->ASCII transaction of Department ID (as it
*       is a number, not a string)
* -------------------------------------------------------------
     SET CYTA-NV-NEXT-POINTER TO ADDRESS OF COMPONENT-NV.
     SET ADDRESS OF CYTA-NV-LIST TO ADDRESS OF COMPONENT-NV.
     INITIALIZE CYTA-NV-LIST REPLACING ALPHANUMERIC BY x"00".
     SET CYTA-NV-NAME-POINTER TO ADDRESS OF COMPONENT-LBL.
     SET CYTA-NV-VALUE-POINTER TO ADDRESS OF COMPONENT.
     MOVE LENGTH OF COMPONENT TO CYTA-NV-VALUE-LENGTH.

* -------------------------------------------------------------
* Setup a Name/Value Pair for Application
* -------------------------------------------------------------
     SET CYTA-NV-NEXT-POINTER TO ADDRESS OF APPLICATION-NV.
     SET ADDRESS OF CYTA-NV-LIST TO ADDRESS OF APPLICATION-NV.
     INITIALIZE CYTA-NV-LIST REPLACING ALPHANUMERIC BY x"00".
     SET CYTA-NV-NAME-POINTER TO ADDRESS OF APPLICATION-LBL.
     SET CYTA-NV-VALUE-POINTER TO ADDRESS OF APPLICATION.
     MOVE LENGTH OF APPLICATION TO CYTA-NV-VALUE-LENGTH.

* -------------------------------------------------------------
* Setup a Name/Value Pair for Transaction
*    Note that the Transaction Value is a number, so we
*    set the flags so that NO EBCDIC to ASCII translation
*    will be performed.
* -------------------------------------------------------------
     SET CYTA-NV-NEXT-POINTER TO ADDRESS OF TRANSACTION-NV.
     SET ADDRESS OF CYTA-NV-LIST TO ADDRESS OF TRANSACTION-NV.
     INITIALIZE CYTA-NV-LIST REPLACING ALPHANUMERIC BY x"00".
     SET CYTA-NV-NAME-POINTER TO ADDRESS OF TRANSACTION-LBL.
     SET CYTA-NV-VALUE-POINTER TO ADDRESS OF TRANSACTION.
     MOVE LENGTH OF TRANSACTION TO CYTA-NV-VALUE-LENGTH.
     MOVE CYTA-DONT-TR-VALUE-FROM-EBCDIC TO CYTA-NV-FLAGS.

* -------------------------------------------------------------
* Call CYTA_init to get Configuration Token
*    1. Specify server string - if this is omitted, the
*       default is: SSN:CYTZ
*    2. Call CYTA_init
* -------------------------------------------------------------
     SET CYTA-CFG-SERVER TO ADDRESS OF SERVER.

     CALL "CYTA_init" USING CYTA-CFG-BLOCK RETURNING RC.
     DISPLAY "(CYTABSMP) CYTA_init Return Code=" RC.

* -------------------------------------------------------------
* Send a STARTED event
*    1. Move in event type
*    2. Move in Vertical link ID
*    3. Move in Vertical link length
*    4. (Don't specify time, so it is automatically inserted)
*    5. Specify Vertical Context that we've built
*    6. Call the API (statically linked).
* -------------------------------------------------------------
     MOVE CYTA-STARTED TO CYTA-E-TYPE.
     SET CYTA-E-VERT-LINK-ID TO ADDRESS OF VLINK.
     MOVE LENGTH OF VLINK TO LINK-LEN.
     MOVE LINK-LEN-BYTE TO CYTA-E-VERT-LINK-LENGTH.
     SET CYTA-E-VERT-CONTEXT-LIST TO ADDRESS OF HOST-NV.

     CALL "CYTA_track" USING CYTA-CFG-BLOCK CYTA-EVENT
        RETURNING RC.
     DISPLAY "(CYTABSMP) STARTED Event Return Code=" RC.

* -------------------------------------------------------------
* Send an OUTBOUND event
*    1. Move in event type
```

```
      *    2. (No need to change Vertical Link or Context)
      *    3. Specify Horizontal Link (program at other end will
      *       need to specify this on it's INBOUND event).
      *    4. Specify Horizontal Link length
      *    5. Call the API
      * -------------------------------------------------------------
           MOVE CYTA-OUTBOUND TO CYTA-E-TYPE.
           SET CYTA-E-HORZ-LINK-ID TO ADDRESS OF HLINK-OUT.
           MOVE LENGTH OF HLINK-OUT TO LINK-LEN.
           MOVE LINK-LEN-BYTE TO CYTA-E-HORZ-LINK-LENGTH.

           CALL "CYTA_track" USING CYTA-CFG-BLOCK CYTA-EVENT
              RETURNING RC.
           DISPLAY "(CYTABSMP) OUTBOUND Event Return Code=" RC.

      * -------------------------------------------------------------
      * Send an INBOUND FINISHED event
      *    1. Move in event type
      *    2. (No need to change Vertical Link or Context)
      *    3. Specify Horizontal Link (program at other end will
      *       need to specify this on it's OUTBOUND event).
      *    4. Specify Horizontal Link length
      *    5. As Horizontal Link is a number, stop conversion from
      *       EBCDIC
      *    6. Call the API
      * -------------------------------------------------------------
           MOVE CYTA-INBOUND-FINISHED TO CYTA-E-TYPE.
           SET CYTA-E-HORZ-LINK-ID TO ADDRESS OF HLINK-IN.
           MOVE LENGTH OF HLINK-IN TO LINK-LEN.
           MOVE LINK-LEN-BYTE TO CYTA-E-HORZ-LINK-LENGTH.
           MOVE CYTA-DONT-TR-FROM-EBCDIC TO CYTA-E-HORZ-LINK-FLAGS.

           CALL "CYTA_track" USING CYTA-CFG-BLOCK CYTA-EVENT
              RETURNING RC.
           DISPLAY "(CYTABSMP) INBOUND FINISHED Event Return Code=" RC.

      * -------------------------------------------------------------
      * And we're done
      * -------------------------------------------------------------

           GOBACK.
```

Java

```
import ttapi4j.ServerFactory;
import ttapi4j.Server;
import ttapi4j.Event;
import ttapi4j.InstanceID;

public class CYTAJSMP
{
    public static void main(String[] args) throws Exception
    {
        System.out.println("(CYTAJSMP) Entry");

        /* --- Get Configuration Token - Sending to Subsys CYTZ  ---- */
        Server s = ServerFactory.getServer("ssn:CYTZ");

        /* --- Create Started Event  ------------------------------- */
        Event e = s.createEvent();
        e.setType(Event.Type.STARTED);
        e.getVerticalID().setLinkID("CYTAJSMP");
        e.getVerticalContext().put("ServerName","Sysplex/Host");
        e.getVerticalContext().put("ComponentName", "USS Shell");
        e.getVerticalContext().put("ApplicationName", "Application");
        e.getVerticalContext().put("TransactionName", "CYTAJSMP");
        s.track(e);
```

```
        /* --- Create Outbound Event  ------------------------------ */
        e.getHorizontalID().setLinkID("Hlink Value");
        e.setType(Event.Type.OUTBOUND);
        s.track(e);

        /* --- Create Inbound Finished Event  ---------------------- */
        e.setType(Event.Type.INBOUND_FINISHED);
        s.track(e);

        /* --- Cleanup and Exit ------------------------------------ */
        s.close();
        System.out.println("(CYTAJSMP) Exit");
    }
}
```

PL/I

```
%PROCESS Limits(Extname(15)) Source Arch(1);
%PROCESS Default(Linkage(Optlink) Nullsys);
%PROCESS Margins(2,72) Rules(IBM) System(MVS);
 CYTAPSMP: Procedure Options(main);

 /*====================================================================

  Storage Definitions

  ====================================================================*/
 /* ------------------------------------------------------------------
    Include Structures for ITCAM for Transactions
    ------------------------------------------------------------------ */
 %include CYTAPEVT;
 %include CYTAPCFG;
 %include CYTAPNV;

 /* ------------------------------------------------------------------
    Declarations for 'Standard' 4 Vertical Context Name/Value pairs.
       These are the minimum Vertical Context needed to display
       an event in ITCAM for Txns workspaces:
          Host        - Host Name. Normally Sysplex/SMFID
          Component   - Component - what we are running under, like:
             BATCH, STC, IMS, CICS, WAS, or TSO.
          Application - Usually Job or Started Task Name
          Transaction - The transaction we are running

       We're defining three variables each:
          xxxlbl  - the label of the Name/Value Pair, ending in
                    nulls.
          xxxnam  - area to hold the actual value
          xxxaddr - address of name/value pair

       In your program, you will modify the values to suit your
       installation, however the labels should not be changed
    ------------------------------------------------------------------ */
 Dcl hostaddr    Pointer;
 Dcl hostlbl     Char(11) Varyingz Init('ServerName');
 Dcl hostnam     Char(12) Init('Sysplex/Host');

 Dcl compaddr    Pointer;
 Dcl complbl     Char(14) Varyingz Init('ComponentName');
 Dcl compnam     Char(3)  Init('STC');

 Dcl appladdr    Pointer;
 Dcl applbl      Char(16) Varyingz Init('ApplicationName');
 Dcl applnam     Char(11) Init('Application');

 Dcl tranaddr    Pointer;
```

```
Dcl tranlbl     Char(16) Varyingz Init('TransactionName');
Dcl trannam     Fixed(32) Binary Unsigned Init(254);

/* ------------------------------------------------------------------
   Outbound and Inbound Horizontal Link IDs
      For your organisation, specify unique values here. But
      for this example, constant values will be used
      hlinkout  - Outgoing Horizontal Link (string)
      hlinkin   - Incoming Horizontal Link (number)
   ------------------------------------------------------------------ */
Dcl hlinkout    Char(11) Init("Hlink Value");
Dcl hlinkin     Fixed(32) Binary Unsigned Init(56);

/* ------------------------------------------------------------------
   Vertical Link ID
      This value should be unique for every work unit. We will
      use a constant in this example.
   ------------------------------------------------------------------ */
Dcl vlink       Char(8) Init("CYTAPSMP");

/* ------------------------------------------------------------------
   Communications Server
      This string specifies where the ITCAM for Transactions Events
      are to be sent. It must be of the form:
         SSN:sub
      Where sub is the 4 character subsystem name used by the ITCAM
      for Transactions Collector Started Task
   ------------------------------------------------------------------ */
Dcl server      Char(8) Init("SSN:CYTZ");

/* ------------------------------------------------------------------
   Area that will hold Event Block and all Name/Value Pairs
   ------------------------------------------------------------------ */
Dcl stgarea     Area;                /* stgarea is 1000 bytes long  */

/*====================================================================

 Main Program

 ====================================================================*/

/* ------------------------------------------------------------------
   Setup Name/Value Pair for Host
   ------------------------------------------------------------------ */
Allocate cytanval In(stgarea);        /* Allocate storage           */
hostaddr = cytanvalp;                 /* Save the address           */
cytannam = Addr(hostlbl);             /* Name                       */
cytanvl  = Addr(hostnam);             /* Value                      */
cytanvll = Length(hostnam);           /* Value Length               */

/* ------------------------------------------------------------------
   Setup Name/Value Pair for Component
   ------------------------------------------------------------------ */
Allocate cytanval In(stgarea);        /* Allocate storage           */
compaddr = cytanvalp;                 /* Save the address           */
cytannam = Addr(complbl);             /* Name                       */
cytanvl  = Addr(compnam);             /* Value                      */
cytanvll = Length(compnam);           /* Value Length               */
hostaddr->cytannxt = compaddr;        /* Chain off Host Pair        */

/* ------------------------------------------------------------------
   Setup Name/Value Pair for Application
   ------------------------------------------------------------------ */
Allocate cytanval In(stgarea);        /* Allocate storage           */
appladdr = cytanvalp;                 /* Save the address           */
cytannam = Addr(appllbl);             /* Name                       */
cytanvl  = Addr(applnam);             /* Value                      */
```

```
cytanvll = Length(applnam);           /* Value Length               */
compaddr->cytannxt = appladdr;        /* Chain off Host Pair        */

/* ------------------------------------------------------------------
   Setup Name/Value Pair for Transaction
   NB: Because Transaction is a number, we set the flag so that this
      value is NOT translated from EBCDIC to ASCII.
   ------------------------------------------------------------------ */
Allocate cytanval In(stgarea);        /* Allocate storage           */
tranaddr = cytanvalp;                 /* Save the address           */
cytannam = Addr(tranlbl);             /* Name                       */
cytanvl  = Addr(trannam);             /* Value                      */
cytannvx = '1'B;                      /* Do NOT xlate Value         */
cytanvll = 4;                         /* Value Length               */
appladdr->cytannxt = tranaddr;        /* Chain off Host Pair        */

/* ------------------------------------------------------------------
   Get our Configuration Token
   ------------------------------------------------------------------ */
Allocate cytacfg In(stgarea);         /* Allocate stg for cfg block  */
cytacsrv = Addr(server);              /* Server                     */
Call CYTA_init(cytacfg);
Display ('(CYTAPSMP) CYTAINIT Return Code=' || PLIRETV());

/* ------------------------------------------------------------------
   Send a Started Event
   ------------------------------------------------------------------ */
Allocate cytaevnt In(stgarea);        /* Allocate storage for event  */
cytaetyp = cytaesta;                  /* Started Event Type         */
cytaevli = Addr(vlink);               /* Vertical Link ID           */
cytaevll = Length(vlink);             /* Vertical Link Length       */
cytaevcn = hostaddr;                  /* Vertical Context Start     */
Call CYTA_track(cytacfg, cytaevnt);
Display ('(CYTAPSMP) STARTED Event Return Code=' || PLIRETV());

/* ------------------------------------------------------------------
   Send an Outbound Event
   (we only need to specify changed fields - all other fields remain
   from the Started event)
   ------------------------------------------------------------------ */
cytaetyp = cytaeob;                   /* Oubound Event Type         */
cytaehli = Addr(hlinkout);            /* Horizontal Link ID         */
cytaehll = Length(hlinkout);          /* Horizontal Link Length     */
Call CYTA_track(cytacfg, cytaevnt);
Display ('(CYTAPSMP) OUTBOUND Event Return Code=' || PLIRETV());

/* ------------------------------------------------------------------
   Send an Inbound Finished Event
   ------------------------------------------------------------------ */
cytaetyp = cytaeibf;                  /* Oubound Event Type         */
cytaehli = Addr(hlinkin);             /* Horizontal Link ID         */
cytaehll = 4;                         /* Horizontal Link Length     */
cytaehnx = '1'b;                      /* Link ID is a number, so do  */
                                      /* NOT translate from EBCDIC.  */
Call CYTA_track(cytacfg, cytaevnt);
Display ('(CYTAPSMP) INBOUND FINISHED Event Return Code=' ||
        PLIRETV());

/* ------------------------------------------------------------------
   Free up our storage
   ------------------------------------------------------------------ */
stgarea = Empty();                    /* Free all storage           */

End CYTAPSMP;
```

HLASM

```
*====================================================================*
* Main Program
*====================================================================*
CYTAASM  RSECT
CYTAASM  AMODE 31
CYTAASM  RMODE ANY
         BAKR  R14,0
         LR    R12,R15
         USING CYTAASM,R12

*--------------------------------------------------------------------*
* Setup workarea
*--------------------------------------------------------------------*
         STORAGE OBTAIN,LENGTH=@WORKL,ADDR=(R1)
         ST    R1,8(R13)              New savearea ptr back to caller
         ST    R13,4(R1)              Old save area ptr in new
         LR    R13,R1
         USING WORK,R13
         MVC   SAVEAREA+4,=C'F1SA'    Show we are using linkage stack

*--------------------------------------------------------------------*
* Setup Vertical Context
*    Your event needs some basic values here, so we use the CYTADFV
*    to fill them in. We chain these values of our own IDNum entry.
*    Note that IDNum is a number, so we don't translate it from EBCDIC
*    to ASCII before sending it down.
*--------------------------------------------------------------------*
         LA    R1,#IDNUM
         CYTANV NAME='IDNum',VALUE=(R1),LEN=#IDNUML,XLATEV=NO,         x
               MF=(E,NAMVALV1)
         CYTADFV TXN='CYTAASM',CHAINTO=NAMVALV1,MF=(E,NAMVALVC)

*--------------------------------------------------------------------*
* Get a Configuration Token - we will use the subsystem CYTZ.         *
*--------------------------------------------------------------------*
         CYTAINIT SUB='CYTZ'          Get Token
         LTR   R15,R15                If successful
         BNZ   LEAVEX
         ST    R1,TTOKEN                 Save it

*--------------------------------------------------------------------*
* Always start with a STARTED Event
* The Vertical Link should be the same for all events in this work
* unit until a FINISHED event is sent.
* Set the entire event block to zeroes before filling it in.
*--------------------------------------------------------------------*
         XC    EVENTBLK(@EVENTBLK),EVENTBLK
         CYTATRAK STARTED,                                           X
               VCTXT=NAMVALV1,                                       X
               TOKEN=TTOKEN,                                         X
               VLINK='CYTAASM',                                      X
               MF=(E,EVENTBLK)
         LTR   R15,R15                If not successful
         BNZ   LEAVEX                 Exit

*--------------------------------------------------------------------*
* Send an Outbound Event - this indicates that we've sent something
* to someone else.
* Remember we need a Horizontal Link and/or a Horizontal Stitch
* here - and the program on the other end also needs to specify the
* same link/stitch on their Inbound event.
* Remember also that this event remembers all the values from the
* previous CYTATRAK call. So we only need to specify the Event Type,
* and the values we are overriding (in this case, HLINK).
*--------------------------------------------------------------------*
         CYTATRAK OUTBOUND,HLINK='CYTAOUT',TOKEN=TTOKEN,MF=(E,EVENTBLK)
```

```
         LTR   R15,R15                If not successful
         BNZ   LEAVEX                 Exit

*--------------------------------------------------------------------*
* (often a program would wait here for something to come back)
*--------------------------------------------------------------------*

*--------------------------------------------------------------------*
* Send an Inbound Finished Event - this is two events (Inbound and
* Finished) rolled into one.
* We also need a Horizontal Link and/or a Horizontal Stitch here, but
* not the one we sent - the one that the program at the other end has
* specified.
* Remember that we also finish with a FINISHED event.
*--------------------------------------------------------------------*
         CYTATRAK INBOUND_FIN,HLINK='CYTABACK',TOKEN=TTOKEN,           X
               MF=(E,EVENTBLK)
         LTR   R15,R15                If not successful
         BNZ   LEAVEX                 Exit

*--------------------------------------------------------------------*
* Return to Caller
*--------------------------------------------------------------------*
LEAVE    DS    0H
         LR    R4,R15                 Save return code
         STORAGE RELEASE,LENGTH=@WORKL,ADDR=(R13) Release workarea
         LR    R15,R4
         PR                           Return to caller

*--------------------------------------------------------------------*
* Error Routine - Error occurred. Exit with return code
*--------------------------------------------------------------------*
LEAVEX   DS    0H
         B     LEAVE

*====================================================================
*
*  PROGRAM CONSTANTS AND LITERALS
*
*====================================================================
#IDNUM   DC    F'45'
#IDNUML  DC    AL2(L'#IDNUM)
         LTORG

*====================================================================
*
*  Mapping Macros and DSECTs
*
*====================================================================
*--------------------------------------------------------------------*
* Workarea
*--------------------------------------------------------------------*
WORK     DSECT
SAVEAREA  DS    18F                   Savearea
STCKTIME  DS    D                     STCK Timestamp

TTOKEN    DS    F                     Fullword for Config Token
NAMVALVC  CYTADFV MF=L                Vertical Ctxt Nam/Val Pairs
NAMVALV1  CYTANV MF=L                 Extra Vert Context Nam/Val pair
EVENTBLK  CYTATRAK MF=L               Event Block
@EVENTBLK EQU  *-EVENTBLK             Length of Event Block

@WORKL    EQU   *-WORK                Length of Workarea

*--------------------------------------------------------------------*
* Register Equates
*--------------------------------------------------------------------*
```

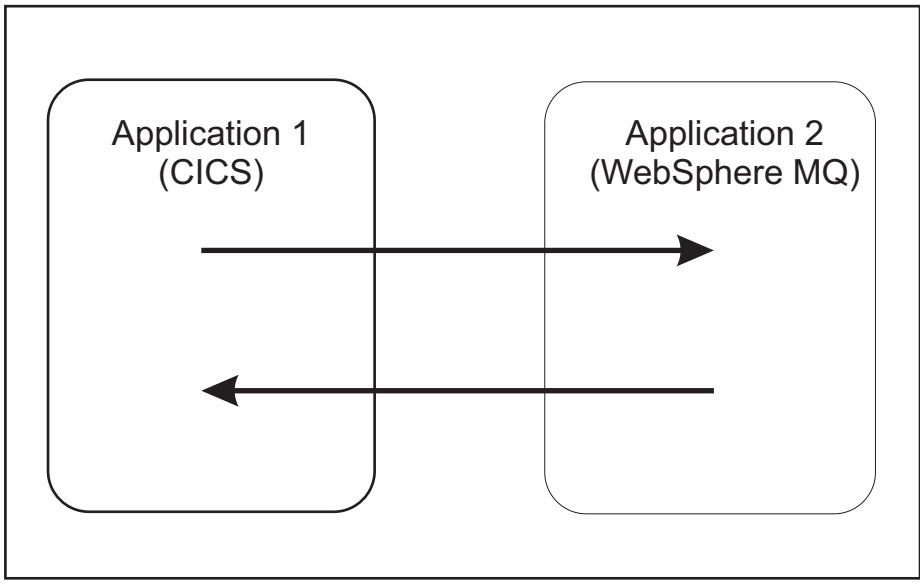
```
R0        EQU   0
R1        EQU   1
R2        EQU   2
R3        EQU   3
R4        EQU   4
R5        EQU   5
R6        EQU   6
R7        EQU   7
R8        EQU   8
R9        EQU   9
R10       EQU   10
R11       EQU   11
R12       EQU   12
R13       EQU   13
R14       EQU   14
R15       EQU   15

          END
```

Appendix D. The kto_stitching file

The `kto_stitching.xml` file defines how horizontal and vertical stitching occurs for events. It consists of definitions for Stitch Pairs.

See the following example of a CICS region and a Websphere MQ application communicating with each other. A monitor in CICS and a monitor in Websphere MQ both send Transactions events.



For Transactions to be able to stitch the CICS and Websphere MQ events together, it must know which Stitching name/value pairs must be matched. For example, CICS and WebSphere® MQ may send a ServerName stitching name/value pair. The Stitch Pair entry in `kto_stitching.xml` tells the Transaction Collector that a CICS event (type 8) ServerName field must match a Websphere MQ event (type 5) ServerName field for two events to be eligible for stitching. The `kto_stitching.xml` file resides in the Transaction Collector directory.

Format

`kto_stitching.xml` is an XML file with the following format:

```
TTEMA Stitching
  Stitch Criteria
   StitchPair
    StitchName
  Stitch Priorities
```

TTEMA Stitching defines the beginning of Transactions stitching definitions. It has no fields.

Stitch Criteria defines the beginning of a list of Stitch Pairs. It has no fields.

StitchPair defines global parameters for the Stitch Pair entry. It has the following fields:

- **Name** – name of the Stitch Pair – can be any string.
- **horizontal** – `true` if this pattern is for matching horizontal stitches. Default: `false`.
- **vertical** – `true` if this pattern is for matching vertical stitches. Default: `false`.
- **reflective** – `true` if this pattern is to be applied for messages sent both ways. If false, then this Stitch Pair will only apply to events sent from the first caller to the second caller. Default: `false`.

StitchNameList defines the beginning of a list of stitching pairs to be matched. It has the following fields:

- **caller** – caller type for the following pairs. Must be a valid link type, that is, a number between 0 and 255. See the Building and Event section for a list of link types.

StitchName defines a pair to be matched. It has the following fields:

- **name** – the name of a stitching name/value pair. Remember that this is case sensitive.

Example

The following is an example of a `kto_stitching.xml` file:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<TTEMAStitching>
        <StitchCriteria>
                <StitchPair name="MQCICS" horizontal="true" reflective="true">
                        <StitchNameList caller="5">
                                <StitchName name="ServerName" />
                                <StitchName name="QMgrName" />
                                <StitchName name="ObjName" />
                                <StitchName name="MsgId" />
                                <StitchName name="CorrelId" />
                                <StitchName name="PutDate" />
                                <StitchName name="PutTime" />
                        </StitchNameList>
                        <StitchNameList caller="8">
                                <StitchName name="ServerName" />
                                <StitchName name="qmgr" />
                                <StitchName name="rslvdQueue" />
                                <StitchName name="msgId" />
                                <StitchName name="corrId" />
                                <StitchName name="putdate" />
                                <StitchName name="puttime" />
                        </StitchNameList>
                </StitchPair>
        </StitchCriteria>
</TTEMAStitching>
```

This example file defines how MQ Tracking and CICS Tracking events will be stitched. In this example, *all* the fields in Table 14 must match fully for a stitch to occur.

Table 14. Field matching

MQ Tracking	CICS Tracking
ServerName	ServerName
QMgrName	qmgr
ObjName	rslvdQueue
MsgId	msgId
CorrelId	corrId
PutDate	putdate

Table 14. Field matching (continued)

MQ Tracking	CICS Tracking
PutTime	puttime

Event flows in both directions (MQ to CICS and CICS to MQ) are covered by this rule.

Appendix E. Transaction Collector Context Mask

The Transaction Collector performs aggregation of events based on context.

For example, all events must specify the ServerName in the vertical context. The Transaction Collector aggregates all events by ServerName, and this is displayed in the Transactions workspaces.

The contexts that the collectors aggregate are specified in a Context Mask file. This file is a text file that resides in the Transaction Collector directory. The format of this file is as follows:

```
# Comments begin in column 1. Do not use blank spaces.
# Don't leave blank lines
#
# Use the Compare statement to tell the Transactions Collector to # aggregate.
# On the line following, specify the name of the context to aggregate on.
# Finish the statement with a * in column 1.
# For example:
new mask
compare
ApplicationName
*
# The above statement tells Transactions Collector to
# aggregate on ApplicationName context values. You
# cannot use wildcards – specify the complete context
# name.
#
# You can also use the ignore statement. This tells the
# Transactions Collector not to aggregate. For example:
ignore
UserID
*
# The above statement tells the Transactions Collector NOT
# to aggregate on UserID.
# This is the default – all context values are ignored
# unless specified in a compare statement.
```

Specify the name of the Context Mask file during installation of the Transaction Collector, and if reconfigured, in the **Transaction Collector Configuration** dialog box.

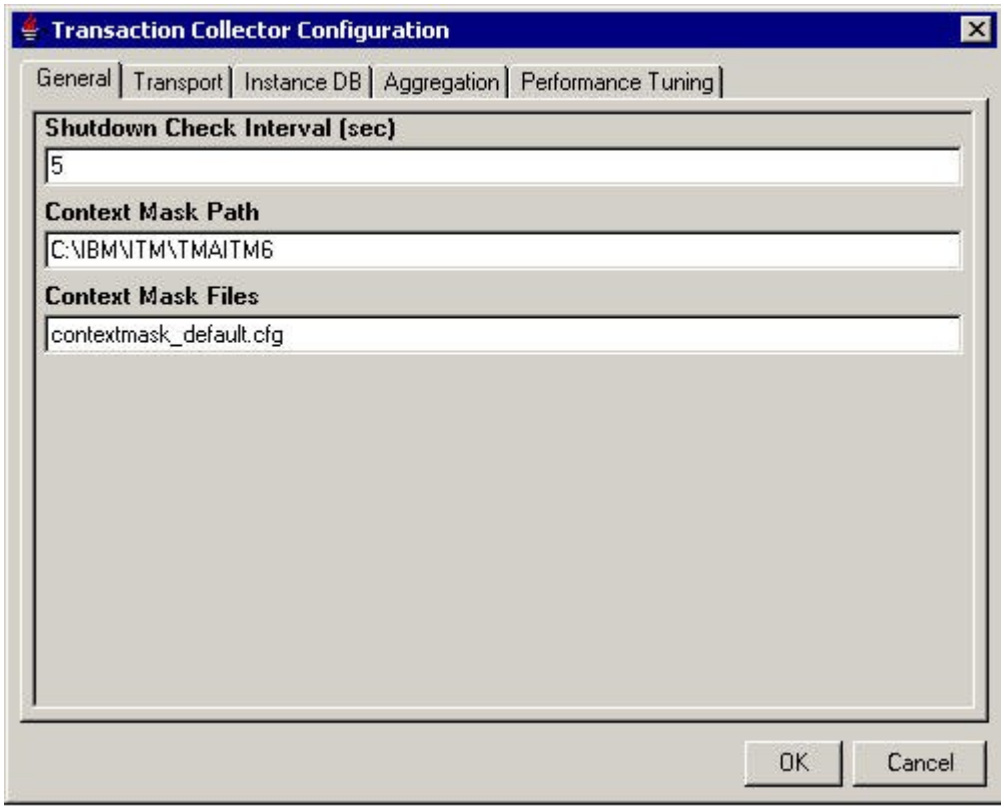


Figure 4. Transaction Collector Configuration dialog box

In Figure 4, the Context Mask file name and location is `C:\IBM\ITM\TMAITM6\contextmask_default.cfg`. This is the default Context Mask supplied with Transaction Tracking. More than one file can be specified in the **Context Mask Files** field: separate each file name with a semi-colon (;).

Appendix F. Accessibility

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

The major accessibility features in this product enable users to do the following:

- Use assistive technologies, such as screen-reader software and digital speech synthesizer, to hear what is displayed on the screen. Consult the product documentation of the assistive technology for details on using those technologies with this product.
- Operate specific or equivalent features using only the keyboard.
- Magnify what is displayed on the screen.

In addition, the product documentation was modified to include the following features to aid accessibility:

- All documentation is available in both HTML and convertible PDF formats to give the maximum opportunity for users to apply screen-reader software.
- All images in the documentation are provided with alternative text so that users with vision impairments can understand the contents of the images.

Navigating the interface using the keyboard

Standard shortcut and accelerator keys are used by the product and are documented by the operating system. See the documentation provided by your operating system for more information.

Magnifying what is displayed on the screen

You can enlarge information on the product windows using facilities provided by the operating systems on which the product is run. For example, in a Microsoft Windows environment, you can lower the resolution of the screen to enlarge the font sizes of the text on the screen. See the documentation provided by your operating system for more information.

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 websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites 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 want 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
2Z4A/101
11400 Burnet Road
Austin, TX 78758 U.S.A.

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 information 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.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a trademark of Linus Torvalds 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.

Privacy policy considerations

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, See IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Glossary

agent Software installed to monitor systems. The agent collects data about an operating system, a subsystem, or an application.

agent group
A group of management agents that run the same policy or policies. Each management agent is associated with one or more listening and playback components.

agentless
A method a data collection where data is collected from traffic on networks monitored by Web Response Time rather than a domain-specific agent or Data Collector plug-in.

aggregate
(1) An average of all response times detected by the monitoring software over a specific time period. (2) In Transaction Tracking, a node in a transaction topology.

aggregate record
A summary of instance data from all transactions that match a defined pattern.

aggregate topology
A transaction topology that displays all known and implied transactions which may not all be related. See also instance topology.

Aggregation agent
An agent that stores the tracking data from more than one Data Collector plug-in and other monitors and computes aggregates for use by the Transaction Reporter. The Transaction Collector and Web Response Time agent are examples of a Aggregation agent.

aggregation period
The time period, measured in minutes, over which monitoring occurs.

alert A message or other indication that signals an event or an impending event.

application
One or more computer programs or software components that provide a function in direct support of a specific business process or processes.

application pattern
A rule that determines what transactions to monitor and how to group them.

arithmetic expression
A statement that contains values joined together by one or more arithmetic operators and that is processed as a single numeric value. See also arithmetic operator.

arithmetic operator
A symbol, such as + or -, that represents a fundamental mathematical operation. See also arithmetic expression.

ARM-instrumented application
An application in which ARM calls are added to the source code to enable the performance of the application to be monitored by management systems.

attribute
The application properties that are measured and reported on, such as the amount of memory used or a message ID. See also attribute groups.

attribute group
A set of related attributes that can be combined in a data view or a situation.

availability
The successful execution of a monitored transaction over a specified period of time.

client A software program or computer that requests services from a server.

client pattern
A method to define which clients to monitor, and how to group them for reporting.

client time
The time it takes to process and display a web page in a browser.

condition
A test of a situation or state that must be in place for a specific action to occur.

configuration
The manner in which the hardware and software of an information processing system are organized and interconnected.

context
The means used to group tracking data as part of a transaction flow.

Data Collector plug-in
The monitoring component that records the transaction data.

data interval
A time period in minutes for the summary data record. See also summary data.

data source
An application, server, transaction, or other process from which raw data is gathered.

domain
A part of a network that is administered as a unit with a common protocol.

down time
See mean time to recovery.

edge
In transaction monitoring, the point at which a transaction first comes in contact with the monitoring instrumentation.

event
An occurrence of significance to a task or system. Events can include completion or failure of an operation, a user action, or the change in state of a process. See also situation.

failure
An individual instance of a transaction that did not complete correctly. See also incident.

firewall
A network configuration, typically both hardware and software, that prevents unauthorized traffic into and out of a secure network.

horizontal
Pertaining to data that is tracked between applications in a domain. See also vertical.

horizontal context
A method of identifying a transaction flow within a transaction which is used to group interactions based on the application supplying the tracking data.

host
A computer that is connected to a network and that provides an access point to that network. The host can be a client, a server, or both a client and a server simultaneously.

hot spot
A graphical device used in topologies to highlight the part of an end-to-end transaction that has crossed specified thresholds and has a significant transaction time deviation.

incident
A failure or set of consecutive failures over a period of time without any successful transactions. An incident concerns a period of time when the service was unavailable, down, or not functioning as expected.

instance
A single transaction or subtransaction.

implied node
A node that is assumed to exist and is therefore drawn in the Transaction Tracking topology. An implied node is created when an aggregate collected in an earlier aggregation period is not collected for the current aggregation period.

instance algorithm
A process used by the Transaction Reporter to track composite applications with multiple instances.

instance topology
A transaction topology that displays a specific instance of a single transaction. See also aggregate topology.

interval
The number of seconds that have elapsed between one sample and the next.

linking
In Transaction Tracking, the process of tracking transactions within the same domain or from data collector plugins of the same type.

load time
The time elapsed between the user's request and completion of the web page download.

managed system
A system that is being controlled by a given system management application.

Management Information Base
(1) In the Simple Network Management

Protocol (SNMP), a database of objects that can be queried or set by a network management system. (2) A definition for management information that specifies the information available from a host or gateway and the operations allowed.

mean time between failures
The average time in seconds between the recovery of one incident and the occurrence of the next one.

mean time to recovery
The average number of seconds between an incident and service recovery.

metric A measurement type. Each resource that can be monitored for performance, availability, reliability, and other attributes has one or more metrics about which data can be collected. Sample metrics include the amount of RAM on a PC, the number of help desk calls made by a customer, and the mean time to failure for a hardware device.

metrics aggregation
A process used by the Transaction Collector to summarize tracking data using vertical linking and stitching to associate items for a particular transaction instance. Metrics aggregation ensures that all appropriate tracking data is aggregated.

MIB See Management Information Base.

monitor
An entity that performs measurements to collect data pertaining to the performance, availability, reliability, or other attributes of applications or the systems on which the applications rely. These measurements can be compared to predefined thresholds. If a threshold is exceeded, administrators can be notified, or predefined automated responses can be performed.

monitoring agent
See agent.

monitoring schedule
A schedule that determines on which days and at what times the monitors collect data.

MTBF See mean time between failures.

MTTR
See mean time to recovery.

network time
Time spent transmitting all required data through the network.

node A point in a transaction topology that represents an application, component, or server whose transaction interactions are tracked and aggregated by Transaction Tracking.

over time interval
The number of minutes the software aggregates data before writing out a data point.

parameter
A value or reference passed to a function, command, or program that serves as input or controls actions. The value is supplied by a user or by another program or process.

pattern
A process used to group data into manageable pieces.

platform
The combination of an operating system and hardware that makes up the operating environment in which a program runs.

predefined workspace
A workspace that is included in the software which is optimized to show specific aspects of the collected data, such as agentless data.

probe A monitor that tests a transaction and then detects and reports any errors that were generated during that test.

profile element
An element or monitoring task belonging to a user profile. The profile element defines what is to be monitored and when.

pseudo node
A node that represents an untracked part of a transaction where information about a remote node is provided by a Data Collector plug-in, but that remote node is not itself tracked.

query In a Tivoli environment, a combination of statements that are used to search the configuration repository for systems that meet certain criteria.

regular expression
A set of characters, meta characters, and operators that define a string or group of strings in a search pattern.

reporting rule
A rule that the software uses for naming the collected data that is displayed in the workspaces.

request
See transaction.

response time
The elapsed time between entering an inquiry or request and receiving a response.

round-trip response time
The time it takes to complete the entire page request. Round-trip time includes server time, client, network, and data transfer time.

robotic script
A recording of a typical customer transaction that collects performance data which helps determine whether a transaction is performing as expected and exposes problem areas of the web and application environment.

SAF See Store and Forward.

sample
The data that the product collects for the server.

schedule
A planned process that determines how frequently a situation runs with user-defined start times, stop times, and parameters.

SDK Software Development Kit.

server A software program or a computer that provides services to other software programs or other computers.

server time
The time it takes for a web server to receive a requested transaction, process it, and respond to it.

service
A set of business processes (such as web transactions) that represent business-critical functions that are made available over the internet.

service level agreement
A contract between a customer and a service provider that specifies the expectations for the level of service with respect to availability, performance, and other measurable objectives.

service level classification
A rule that is used by a monitor to evaluate how well a monitored service is performing. The results form the basis for service level agreements (SLAs).

service recovery
The time it takes for the service to recover from being in a failed state.

situation
A set of conditions that, when met, create an event.

SLA See service level agreement.

status The state of a transaction at a particular point in time, such as whether it failed, was successful, or slow.

stitching
The process of tracking transactions between domains or from different types of data collector plugins.

store and forward
The temporary storing of packets, messages, or frames in a data network before they are retransmitted toward their destination.

subtransaction
An individual step (such as a single page request or logging on to a web application) in the overall recorded transaction.

summary data
Details about the response times and volume history, as well as total times and counts of successful transactions for the whole application.

summary interval
The number of hours that data is stored on the agent for display in the Tivoli Data Warehouse workspaces.

summary status
An amount of time in which to collect data on the Tivoli Enterprise Management Agent.

threshold
A customizable value for defining the

acceptable tolerance limits (maximum, minimum, or reference limit) for a transaction, application resource, or system resource. When the measured value of the resource is greater than the maximum value, less than the minimum value, or equal to the reference value, an exception or event is raised.

tracking data
Information emitted by composite applications when a transaction instance occurs.

transaction
An exchange between two programs that carries out an action or produces a result. An example is the entry of a customer's deposit and the update of the customer's balance.

transaction definition
A set of filters and maintenance schedules created in the Application Management Configuration Editor which are applied to the collected data and determine how that data is processed and displayed.

transaction flow
The common path through a composite application taken by similar transaction instances.

transaction interaction
See transaction.

transaction pattern
The pattern for specifying the name of specific transactions to monitor. Patterns define groupings of transactions that map to business applications and business transactions.

trend A series of related measurements that indicates a defined direction or a predictable future result.

uptime
See Mean Time Between Failure.

user profile
For Internet Service Monitoring, an entity such as a department or customer for whom services are being performed.

vertical
Pertaining to data that is tracked within the same application and domain. See also horizontal.

vertical context
The method used to distinguish one transaction flow from another within an application or group of applications. The vertical context enables Transaction Tracking to group individual transactions as part of a flow, label a node in a topology map, and link to an IBM® Tivoli Monitoring application.

view A logical table that is based on data stored in an underlying set of tables. The data returned by a view is determined by a SELECT statement that is run on the underlying tables.

workspace
In Tivoli management applications, the working area of the user interface, excluding the Navigator pane, that displays one or more views pertaining to a particular activity. Predefined workspaces are provided with each Tivoli application, and systems administrators can create customized workspaces.

Index

Special characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

S

T

V

W

Z

Printed in USA

SC14-7410-03