

Netcool/OMNIbus
Version 7 Release 3

Probe and Gateway Guide



Netcool/OMNIbus
Version 7 Release 3

Probe and Gateway Guide



Note

Before using this information and the product it supports, read the information in “Notices” on page 171.

This edition applies to version 7, release 3, modification 1 of IBM Tivoli Netcool/OMNIBus (product number 5724-S44) and to all subsequent releases and modifications until otherwise indicated in new editions.

© **Copyright IBM Corporation 1994, 2011.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this publication	v
Intended audience	v
What this publication contains	v
Publications	vi
Accessibility	viii
Tivoli technical training	viii
Support information	viii
Conventions used in this publication	viii

Chapter 1. About probes 1

Types of probes	2
Device probes	2
Log file probes	2
Database probes	3
API probes	3
CORBA probes	3
Miscellaneous probes	4
Probe components	4
Executable file	4
Properties file	5
Rules file	6
Naming conventions for probe component files	7
Probe architecture	8
How unique identifiers are constructed for events	9
Modes of operation of probes	10
Store-and-forward mode for probes	10
Raw capture mode for probes	13
Secure mode for probes	13
Peer-to-peer failover mode for probes	14

Chapter 2. Probe rules file syntax . . . 17

Elements, fields, properties, and arrays in rules files	17
Assigning values to ObjectServer fields	17
Assigning temporary elements in rules files	18
Assigning property values to fields	18
Assigning values to properties	19
Using arrays	19
Rules file development guidelines	20
Control statements in rules files	21
FOREACH statement	21
IF statement	26
SWITCH statement	26
BREAK statement	27
Embedding multiple rules files in a rules file	27
Rules file functions and operators	28
Math and string operators	31
Bit manipulation operators	31
Comparison operators	32
Logical operators	32
Existence function	33
Elements and event functions	33
String functions	33
Math functions	36
Date and time functions	37
Host and process utility functions	38

Lookup table operations	39
Update on deduplication function	41
Details function	41
Message logging functions	42
Sending alerts to alternative ObjectServers and tables	43
Search and replace function	47
Service function	49
Monitoring probe loads	50
Reserved words in the probe rules language	51
Testing rules files	53
Debugging rules files	53
Rules file examples	54

Chapter 3. Probe rules file customizations 57

Detecting event floods and anomalous event rates	57
Configuring probes to detect event floods and anomalous event rates	58
Flood configuration rules file	59
Flood rules file	62
Enabling self monitoring of probes	64
Configuration setup for self monitoring of probes	65
Tivoli Netcool/OMNIbus configuration files for the self monitoring of probes	66
Configuring probes for self monitoring	68

Chapter 4. Running probes 73

Running probes on UNIX	73
Running probes on Windows	74
Running a probe as a console application	74
Running a probe as a service	75
Use of OMNIHOME and NCHOME environment variables for probes	76

Chapter 5. Common probe properties and command-line options 77

Chapter 6. About gateways 89

Types of gateways	90
ObjectServer gateways	91
Unidirectional ObjectServer Gateway	91
Bidirectional ObjectServer Gateway	91
ObjectServer Gateway writers and failback (alert replication between sites)	92
Database, helpdesk, and other gateways	93
Gateway components	93
Unidirectional gateways	93
Bidirectional gateways	94
Modes of operation of gateways	96
Store-and-forward mode for gateways	96
Secure mode for gateways	96
Gateway configuration	98
Gateway configuration file	98

Reader configuration	99
Writer configuration	99
Route configuration	100
Mapping configuration	100
Filter configuration	101
Gateway debugging	102
Gateway writers and failback	103
Creating conversion tables	103

Chapter 7. Running gateways 105

Running gateways on UNIX	105
Running gateways on Windows	105
Running a gateway as a console application	105
Running a gateway as a service	106
Configuring gateways interactively	106
Saving configurations interactively	107
Dumping and loading gateway configurations interactively	107
Use of OMNIHOME and NCHOME environment variables for gateways	108

Chapter 8. Gateway commands and command-line options 109

Common gateway command-line options	109
Reader commands	111
START READER	111
STOP READER	112
SHOW READERS	112
Writer commands	112
START WRITER	113
STOP WRITER	113
SHOW WRITERS	113
SHOW WRITER TYPES	114
SHOW WRITER ATTRIBUTES	114
Mapping commands	115
CREATE MAPPING	115
DROP MAPPING	115
SHOW MAPPINGS	116
SHOW MAPPING ATTRIBUTES	116
Filter commands	116
CREATE FILTER	116
LOAD FILTER	117
DROP FILTER	117
Route commands	117
ADD ROUTE	117
REMOVE ROUTE	117
SHOW ROUTES	118
Configuration commands	118
LOAD CONFIG	118

SAVE CONFIG	118
DUMP CONFIG	118
General commands	119
SHUTDOWN	119
SET CONNECTIONS	119
SHOW SYSTEM	119
SET DEBUG MODE	120
TRANSFER	120

Appendix A. Probe error messages and troubleshooting techniques . . . 123

Generic error messages	123
Fatal-level messages	123
Error-level messages	124
Warning-level messages	126
Information-level messages	126
Debug-level messages	126
ProbeWatch and TSMWatch messages	128
Troubleshooting probes	130
Common problem causes	130
What to do if	130

Appendix B. Common gateway error messages 135

Appendix C. Regular expressions . . . 143

NETCOOL regular expression library	143
TRE regular expression library	145
Metacharacters	145
Minimal or non-greedy quantifiers	147
Bracket expressions	148
Constructs for multicultural support	149
Backslash sequences	149

Appendix D. ObjectServer tables and data types 153

alerts.status table	153
alerts.details table	166
alerts.journal table	167
service.status table	167
ObjectServer data types	168

Notices 171

Trademarks	173
----------------------	-----

Index 175

About this publication

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

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

Intended audience

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

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

What this publication contains

This publication contains the following sections:

- Chapter 1, “About probes,” on page 1
Provides information about probes, their architecture, components, and modes of operation.
- Chapter 2, “Probe rules file syntax,” on page 17
Describes the syntax of the rules file that defines how the probe must process event data to create a meaningful Tivoli Netcool/OMNIBus alert.
- Chapter 3, “Probe rules file customizations,” on page 57
Describes the customizations that can be applied to probe rules files to extend the functionality of probes.
- Chapter 4, “Running probes,” on page 73
Describes how to run probes.
- Chapter 5, “Common probe properties and command-line options,” on page 77
Describes the properties and command-line options that are common to all probes and TSMs.
- Chapter 6, “About gateways,” on page 89
Provides information about gateways, their modes of operation, and gateway components.
- Chapter 7, “Running gateways,” on page 105
Describes how to run gateways.
- Chapter 8, “Gateway commands and command-line options,” on page 109
Describes the gateway commands and command-line options that are common to all gateways.
- Appendix A, “Probe error messages and troubleshooting techniques,” on page 123
Provides information on probe error messages and troubleshooting.

- Appendix B, “Common gateway error messages,” on page 135
Provides information on gateway error messages.
- Appendix C, “Regular expressions,” on page 143
Provides reference information on regular expressions.
- Appendix D, “ObjectServer tables and data types,” on page 153
Provides reference information on relevant ObjectServer tables.

Publications

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

Your Tivoli Netcool/OMNIBus library

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

- *IBM Tivoli Netcool/OMNIBus Installation and Deployment Guide*, SC14-7604
Includes installation and upgrade procedures for Tivoli Netcool/OMNIBus, and describes how to configure security and component communications. The publication also includes examples of Tivoli Netcool/OMNIBus architectures and describes how to implement them.
- *IBM Tivoli Netcool/OMNIBus Administration Guide*, SC14-7605
Describes how to perform administrative tasks using the Tivoli Netcool/OMNIBus Administrator GUI, command-line tools, and process control. The publication also contains descriptions and examples of ObjectServer SQL syntax and automations.
- *IBM Tivoli Netcool/OMNIBus Web GUI Administration and User's Guide*, SC14-7606
Describes how to perform administrative and event visualization tasks using the Tivoli Netcool/OMNIBus Web GUI.
- *IBM Tivoli Netcool/OMNIBus User's Guide*, SC14-7607
Provides an overview of the desktop tools and describes the operator tasks related to event management using these tools.
- *IBM Tivoli Netcool/OMNIBus Probe and Gateway Guide*, SC14-7608
Contains introductory and reference information about probes and gateways, including probe rules file syntax and gateway commands.
- *IBM Tivoli Monitoring for Tivoli Netcool/OMNIBus Agent User's Guide*, SC14-7610
Describes how to install the health monitoring agent for Tivoli Netcool/OMNIBus and contains reference information about the agent.
- *IBM Tivoli Netcool/OMNIBus Event Integration Facility Reference*, SC14-7611
Describes how to develop event adapters that are tailored to your network environment and the specific needs of your enterprise. This publication also describes how to filter events at the source.
- *IBM Tivoli Netcool/OMNIBus Error Messages Guide*, SC14-7612
Describes system messages in Tivoli Netcool/OMNIBus and how to respond to those messages.
- *IBM Tivoli Netcool/OMNIBus Web GUI Administration API (WAAPI) User's Guide*, SC22-5403-00
Shows how to administer the Tivoli Netcool/OMNIBus Web GUI using the XML application programming interface named WAAPI.

Accessing terminology online

The *Tivoli Software Glossary* includes definitions for many of the technical terms related to Tivoli software. The *Tivoli Software Glossary* is available at the following Tivoli software library Web site:

<http://publib.boulder.ibm.com/tividd/glossary/tivoliglossarymst.htm>

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

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

Accessing publications online

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

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

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

Ordering publications

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

<http://www.elink.ibm.link.ibm.com/publications/servlet/pbi.wss>

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

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

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

1. Go to the following Web site:
<http://www.elink.ibm.link.ibm.com/publications/servlet/pbi.wss>
2. Select your country from the list and click **Go**. The Welcome to the IBM Publications Center page is displayed for your country.
3. On the left side of the page, click **About this site** to see an information page that includes the telephone number of your local representative.

Accessibility

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

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

Tivoli technical training

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

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

Support information

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

Online

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

IBM Support Assistant

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

Conventions used in this publication

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

Typeface conventions

This publication uses the following typeface conventions:

Bold

- Lowercase commands and mixed case commands that are otherwise difficult to distinguish from surrounding text
- Interface controls (check boxes, push buttons, radio buttons, spin buttons, fields, folders, icons, list boxes, items inside list boxes, multicolumn lists, containers, menu choices, menu names, tabs, property sheets), labels (such as **Tip:** and **Operating system considerations:**)
- Keywords and parameters in text

Italic

- Citations (examples: titles of publications, diskettes, and CDs)
- Words defined in text (example: a nonswitched line is called a *point-to-point* line)

- Emphasis of words and letters (words as words example: "Use the word *that* to introduce a restrictive clause."; letters as letters example: "The LUN address must start with the letter *L*.")
- New terms in text (except in a definition list): a *view* is a frame in a workspace that contains data
- Variables and values you must provide: ... where *myname* represents....

Monospace

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

Operating system-dependent variables and paths

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

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

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

Operating system-specific directory names

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

Table 1. Directory names for the *arch* variable

Directory name represented by <i>arch</i>	Operating system
<i>aix5</i>	AIX® systems
<i>hpux11</i>	HP-UX PA-RISC-based systems
<i>hpux11hpie</i>	HP-UX Integrity-based systems
<i>linux2x86</i>	Red Hat Linux and SUSE systems
<i>linux2s390</i>	Linux for System z®
<i>solaris2</i>	Solaris systems
<i>win32</i>	Windows systems

Chapter 1. About probes

Probes connect to an event source, detect and acquire event data, and forward the data to the ObjectServer as alerts. Probes use the logic specified in a rules file to manipulate the event elements before converting them into fields of an alert in the ObjectServer alerts.status table.

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

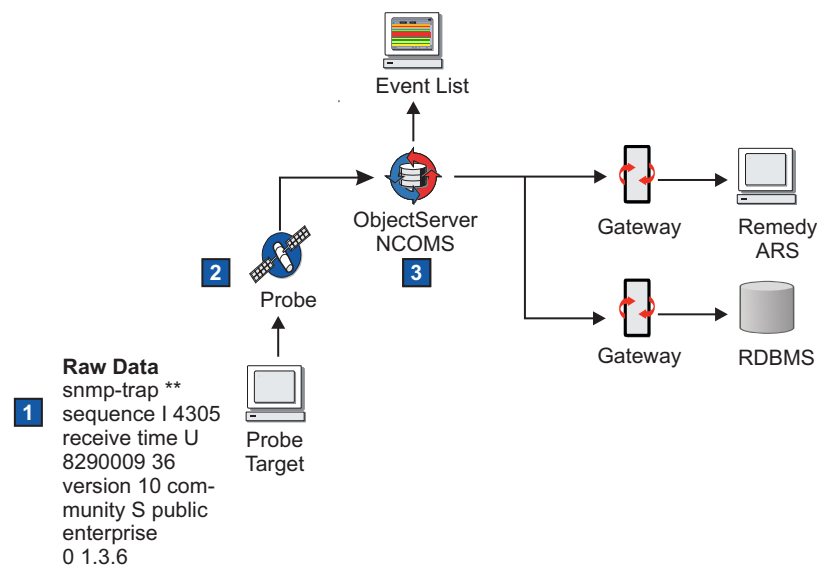


Figure 1. Event processing in Tivoli Netcool/OMNIBus

The flow of event data is as follows:

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

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

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

Related concepts

"Types of probes" on page 2

Types of probes

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

The types of probes are:

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

The probe type is determined by the method in which the probe detects events. For example, the Probe for Agile ATM Switch Management detects events produced by a device (an ATM switch), but it acquires events from a log file, not directly from the switch. Therefore, this probe is classed as a log file probe and not a device probe. Likewise, the Probe for Oracle obtains event data from a database table, and is therefore classed as a database probe.

Device probes

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

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

After connecting to the target machine, the probe detects events and forwards them to the ObjectServer. Some device probes are passive, and wait to detect an event before forwarding it to the ObjectServer; for example, the Probe for Marconi ServiceOn EMOS. Other device probes are more active, and issue commands to the target device in order to acquire events; for example, the TSM for Ericsson AXE10.

Log file probes

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

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

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

Database probes

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

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

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

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

Database probes treat each row of the source table as a single entity. Even if only one field of a row in the source table changes, all of the fields of that row are forwarded to the logging table, and from there to the ObjectServer. If a row in the source table is deleted, the probe forwards the contents of the row before it was deleted. If a row in the source table is inserted or updated, the probe forwards the contents of the row after the insert or update action.

API probes

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

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

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

CORBA probes

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

Equipment vendors publish the details of their specific CORBA interface as Interface Definition Language (IDL) files. These IDL files are used to create the CORBA client and server applications. A specific probe is required for each specific CORBA interface. CORBA probes use the Borland VisiBroker Object Request Broker (ORB) to communicate with other vendor ORBs. You must obtain this ORB from IBM Software Support. Most CORBA probes are written using Java, and require specific Java components to be installed to run the probe, as described in the individual publications for these probes. Probes written in Java use the following additional processes:

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

For example, the Probe for Marconi MV38/PSB manages the alarm lifecycle by collecting events from the Marconi ServiceOn Optical Network Management System. To do this, the probe connects to the Practical Service and Business (PSB) CORBA interface using the CORBA Naming Service running on the PSB host.

Miscellaneous probes

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

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

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

Probe components

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

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

Executable file

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

Probe executable files are stored in the directory `$OMNIHOME/probes/arch`, where *arch* represents the operating system. For example, the executable file for the Ping Probe that runs on HP-UX 11.00 is:

```
$OMNIHOME/probes/hpux11/nco_p_ping
```

To start a probe on UNIX with the appropriate configuration information, run the wrapper script in the directory `$OMNIHOME/probes`. For example, to start the Ping Probe, enter:

```
$OMNIHOME/probes/nco_p_ping
```

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

Related concepts

“Properties file”

“Rules file” on page 6

Properties file

Probe properties define the environment in which the probe runs.

For example, the **Server** property specifies the ObjectServer to which the probe forwards alerts. Probe properties are stored in a properties file in the directory `$OMNIHOME/probes/arch`, where *arch* represents the operating system directory. Properties files are identified by the `.props` file extension.

For example, the properties file for the Ping Probe that runs on HP-UX 11.00 is:

```
$OMNIHOME/probes/hpux11/ping.props
```

Properties files are formed of name-value pairs separated by a colon. For example:

```
Server : "NCOMS"
```

In this name-value pair, `Server` is the name of the property and `NCOMS` is the value to which the property is set. String values must be enclosed in quotation marks; other values do not require quotation marks.

Related concepts

“Executable file” on page 4

Chapter 4, “Running probes,” on page 73

Probe property types

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

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

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

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

Related reference

Chapter 5, “Common probe properties and command-line options,” on page 77

Probe property versus probe command-line option usage

Each probe property has a corresponding command-line option.

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

```
Server : "NCOMS"
```

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

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

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

Related reference

Chapter 5, "Common probe properties and command-line options," on page 77

Rules file

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

When the probe acquires raw data from the event source, it breaks it down into tokens. Each token represents a piece of event data. The probe then parses these tokens into elements. Elements are identified within the rules file by the `$` symbol. For example, `$Node` is an element containing the node name of the event source.

The probe processes the elements according to the rules in the rules file to assign values to the fields which are available in the ObjectServer. When they have been processed, the field values contain the event details in the form used by the ObjectServer. At this stage, the `@Identifier` field is also assigned a unique value. The probe then forwards the field values to the ObjectServer as a Tivoli Netcool/OMNIBus alert. Field values are identified in the rules file by the `@` symbol. For example, `@Node` can be a field value containing the node name of the event source.

Duplicate alerts (those with the same identifier) are correlated so that they are displayed in the event list only once.

Local rules files are stored in the directory `$OMNIHOME/probes/arch`, and are identified by the `.rules` file extension. For example, the rules file for the Ping Probe that runs on HP-UX 11.00 is:

```
$OMNIHOME/probes/hpux11/ping.rules
```

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

Related concepts

“Executable file” on page 4

“Probe architecture” on page 8

Chapter 4, “Running probes,” on page 73

Related reference

Chapter 2, “Probe rules file syntax,” on page 17

Re-reading the rules file

Whenever you update the rules file, the probe must be forced to re-read the rules file in order for the changes to take effect.

To force the probe to re-read the rules file, issue the following command on the probe process ID (PID):

```
kill -HUP pid
```

where *pid* is the PID.

See the **ps** and **kill** man pages for more information.

This method is preferable to restarting the probe, because the probe will not lose events. If the updated rules file contains syntax errors or references fields that do not exist, when the probe is sent a HUP signal, it sends an error message to the log file and continues to use the previous version of the rules file.

Tip: For CORBA probes, issue the command `kill -HUP` on the **nco_p_nonnative** process.

Related tasks

“Debugging rules files” on page 53

“Defining lookup tables in the rules file” on page 39

Naming conventions for probe component files

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

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

Table 2. Naming conventions for probe file names

Probe file type	File name and location
Executable file	<code>\$OMNIHOME/probes/arch/nco_p_probename</code>
Properties file	<code>\$OMNIHOME/probes/arch/probename.prop</code>
Rules file	<code>\$OMNIHOME/probes/arch/probename.rules</code>

In these paths:

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

For example, the abbreviated name for SunNet Manager is `snmlog` and the Probe for SunNet Manager executable file is named:

\$OMNIHOME/probes/arch/nco_p_snmlog

The properties file is named:

\$OMNIHOME/probes/arch/snmlog.prop

The rules file is named:

\$OMNIHOME/probes/arch/snmlog.rules

Probe architecture

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

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

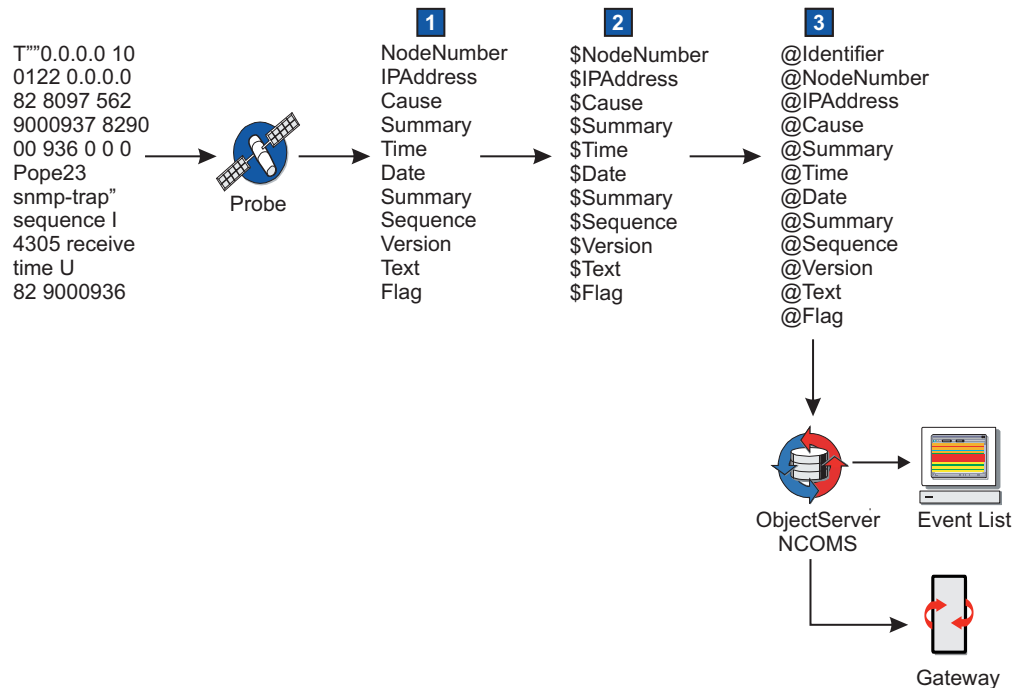


Figure 2. Event mapping using rules

The processing stages are as follows:

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

are forwarded to the ObjectServer, where they are stored and managed in the alerts.status table, and displayed in the event list.

The Identifier field is also generated by the rules file.

Related concepts

“How unique identifiers are constructed for events”

“Rules file” on page 6

How unique identifiers are constructed for events

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

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

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

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

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

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

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

Event deduplication with probes

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

Related concepts

“Probe architecture” on page 8

Related reference

Chapter 2, “Probe rules file syntax,” on page 17

“Update on deduplication function” on page 41

Modes of operation of probes

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

Store-and-forward mode for probes

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

Automatic store and forward

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

However, if you set the probe to run in *automatic* store-and-forward mode, it goes straight into store mode if the ObjectServer is not running, as long as the probe has been connected to the ObjectServer at least once before. Enable automatic store-and-forward mode by using the `-autosaf` command-line option or the **AutoSAF** property.

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

Legacy store and forward

When the probe detects that the ObjectServer is not present (usually because it cannot forward an alert to the ObjectServer), the probe switches to store mode. In this mode, the probe writes all of the messages that it would normally send to the ObjectServer to a store-and-forward file. This file name is constructed using the value that is specified for the **SAFFileName** property. A *.servername* extension is automatically appended to the **SAFFileName** value, where *servername* is the name of the ObjectServer to which the probe is attempting to send alerts. If the probe is configured to send alerts to multiple ObjectServers, individual store-and-forward files are therefore created for each ObjectServer.

If corrupted records are identified in a store-and-forward file, these records are ignored and the probe will forward only the valid records to the ObjectServer. You can indicate whether to automatically save a file that contains corrupted records for future diagnosis by using the **KeepLastBrokenSAF** property. If you set this property to 1, the file containing corrupted records is renamed **SAFFileName.servername.broken** (and will overwrite any previous *.broken* file).

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

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

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

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

Circular store and forward

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

The circular store-and-forward files are named **SAFFileName.servername** and **SAFFileName.servername_1**.

When the probe gets disconnected from the ObjectServer, the probe stores the timestamp of the last successful event and the ObjectServer name in a file that is named in the format **SAFFilename.DisconnectionTime**. This file is stored in the same directory as the store-and-forward files. If a backup ObjectServer is available for failover, the probe reconnects to the backup ObjectServer and replays events from the store-and-forward file that was earlier sent to the primary ObjectServer during the time period measured as one **RollSAFInterval** before the disconnection time. As a result, the probe will resend events that it might have sent to the primary ObjectServer, but which might not have been replicated in the backup ObjectServer before the primary ObjectServer went down.

If the probe is unable to connect to an ObjectServer, the probe automatically switches its handling of rolling store-and-forward files to the legacy store-and-forward behavior. The probe starts storing all events in a pool of store-and-forward files, where the size of the pool is defined by the **SAFFilePoolSize** property, and the maximum file size is defined by the **MaxSAFFileSize** property. During this time, the **RollSAFInterval** property is not used to roll over the store-and-forward files; instead, each file rolls over when it reaches the size specified by **MaxSAFFileSize**.

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

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

Important: If you want the probe to operate in circular store-and-forward mode, the **Server** property must be set to the name of the primary ObjectServer, and the **ServerBackup** property must be set to the name of the backup ObjectServer, if a backup is present. Do not use the definitions of virtual ObjectServer pairs for these properties.

For information about the use of the ProbeSubSecondId field in the deduplication trigger within the ObjectServer when the probe is running in circular

store-and-forward mode, see <https://www-304.ibm.com/support/docview.wss?uid=swg21469238>.

Summary of store-and-forward behavior

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

Table 3. Store-and-forward summary

ObjectServer status before probe startup	AutoSAF property	StoreAndForward property	Expected result
ObjectServer down	0	0	The probe does not start.
ObjectServer down	0	1	The probe does not start.
ObjectServer down	1	0	The probe starts writing events into the store-and-forward file. When the ObjectServer comes up, the probe forwards the store-and-forward file events and then stops writing events to the store-and-forward file. If the ObjectServer gets disconnected later, events will not be stored.
ObjectServer down	1	1	The probe starts writing events into the store-and-forward file. When the ObjectServer comes up, the probe forwards the store-and-forward file events. If the ObjectServer gets disconnected later, the probe will store events in store-and-forward files; these events will be forwarded on reconnection.
ObjectServer up	No effect of property	1	The probe forwards events in the store-and-forward files to the connected ObjectServer, and stores new events in the store-and-forward files only when disconnected from the ObjectServer.
ObjectServer up	No effect of property	0	The probe does not forward any events in the existing store-and-forward files, and does not store events in new store-and-forward files.
ObjectServer up	No effect of property	2	The probe forwards events in the store-and-forward files to the connected ObjectServer, and stores new events in the rolling store-and-forward files when connected. The probe stores all events in a pool of files when disconnected.

Related reference

“Multithreaded processing of alert data” on page 46

Chapter 5, “Common probe properties and command-line options,” on page 77

Raw capture mode for probes

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

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

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

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

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

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

For example:

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

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

Related reference

“Changing the value of the RawCapture property in the rules file” on page 19

Chapter 5, “Common probe properties and command-line options,” on page 77

Secure mode for probes

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

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

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

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

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

When in non-FIPS 140–2 mode, the password can be encrypted with the **nco_g_crypt** or **nco_aes_crypt** utilities. If you are encrypting passwords by using **nco_aes_crypt** in non-FIPS 140–2 mode, you can specify either AES_FIPS or AES as the encryption algorithm. Use AES only if you need to maintain compatibility with passwords that were encrypted using the tools provided in versions earlier than Tivoli Netcool/OMNIBus V7.2.1.

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

Related reference

Chapter 5, “Common probe properties and command-line options,” on page 77

Peer-to-peer failover mode for probes

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

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

To set up a peer-to-peer failover relationship, perform the following actions:

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

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

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

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

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

You can also switch the mode of a probe between master and slave in the rules file. There is a delay of up to one second before the mode change takes effect. This can result in duplicate events if two probe instances are switching from standard mode to master or slave; however, no data is lost.

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

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

Example properties file values for the master are as follows:

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

Example properties file values for the slave are as follows:

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

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

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

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

Chapter 2. Probe rules file syntax

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

Related concepts

“How unique identifiers are constructed for events” on page 9

“Rules file” on page 6

Elements, fields, properties, and arrays in rules files

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

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

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

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

Related concepts

“How unique identifiers are constructed for events” on page 9

Assigning values to ObjectServer fields

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

Examples are as follows:

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

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

- @Class=100
- @Class=0x64

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

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

Related reference

“Math functions” on page 36

Assigning temporary elements in rules files

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

For example:

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

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

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

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

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

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

```
$a=$b
```

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

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

The `extract` function is used to assign values to temporary elements `temp1` and `temp2`. Then these elements are concatenated (using the `+` concatenate operator) with a `.` separating them, and assigned to the `AlertKey` field. After these statements are run, the `AlertKey` field has the value `1.2`.

Related reference

“String functions” on page 33

“Math and string operators” on page 31

Assigning property values to fields

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

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

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

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

Assigning values to properties

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

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

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

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

Changing the value of the RawCapture property in the rules file

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

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

For example:

```
# Start rules processing
%RawCapture=0

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

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

Related concepts

“Raw capture mode for probes” on page 13

Related reference

“Control statements in rules files” on page 21

Using arrays

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

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

To define an array, use the following syntax:

```
array node_arr
```

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

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

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

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

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

Related reference

“Lookup table operations” on page 39

“Sending alerts to alternative ObjectServers and tables” on page 43

Rules file development guidelines

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

- Rules files must be of production quality and not require any additional modification.
- Rules files must not result in any additional modifications to the ObjectServer. That is, there must be no additional event fields other than those provided by Tivoli Netcool/OMNIBus.
- The basic structure of the rules files must be both easily maintainable and easily extendible, therefore enabling the addition of event handling for new devices without affecting existing rules.
- The basic textual-conventions used for the rules files must be consistent and therefore ensure that all newly created rules files share a common format.
- The rules files must be clearly documented to allow each event to be recognized without the need for any additional documentation.
- The events formatted by the rules files must be deduplicated properly by the Tivoli Netcool/OMNIBus ObjectServer. The Identifier field (@Identifier) must be set correctly to enable the granularity of deduplication to be directly controlled.
- The events formatted by the rules files must be compatible, whenever possible, with the ObjectServer's GenericClear Automation.
- Always make a backup copy of a rules file before modifying it. Save the file as a `.rules.date` file. For example, `snmp.rules.070131`. You need this file in case you have to perform a rollback.
- Any changes to the rules file must be commented out with a number sign (#) at the beginning of the line.
- Use the `details ($*)` function only when debugging rules files or writing rules files. After using `details ($*)` for long periods of time, the ObjectServer tables become very large and the performance of the ObjectServer suffers.
- The SWITCH and CASE constructs are processed more efficiently, and must therefore be used in preference to the IF and ELSE statements.
- Use lookup tables wherever possible. When multiple values are linked to a single key, use a multi-column lookup table. Lookup tables must be defined within an external file based table, specified with a `.lookup` file extension. This enables clear identification of the lookup tables. Additionally, the lookup tables must be the first elements of the rules files that are read by the probe. That is, in the basic rules file, locate the lookup file `include` statement at the top.
- Matching pairs of problem and resolution events must have identical @AlertGroup and @AlertKey values, and appropriate @Type and @Severity values.
- A Resolution event must have a severity alert of 1 (indeterminate) and a type of 2 (resolution). Do not set the severity of resolution events to 0 (Clear). This would prevent events being processed by the ObjectServer's GenericClear Automation.

Control statements in rules files

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

FOREACH statement

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

Syntax

The syntax of the FOREACH statement is as follows:

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

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

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

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

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

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

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

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

When looping through an array, note the following information:

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

Supported probe rules file functions

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

Restrictions

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

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

Examples of the looping function

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

Example 1: Looping through all elements

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

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

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

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

Example 2: Looping through a comma delimited list

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

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

Example 3: Loop through entries in an array

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

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

Example 4: Using IF with BREAK

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

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

For more information, see “BREAK statement” on page 27.

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

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

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

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

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

```
# Declare an empty array
array oids

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

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

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

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

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

Example 7: Nested loops

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

For more information, see “SWITCH statement” on page 26 and “BREAK statement” on page 27.

Related reference

“Rules file examples” on page 54

IF statement

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

The IF statement has the following syntax:

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

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

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

Related reference

"Logical operators" on page 32

"String functions" on page 33

SWITCH statement

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

The SWITCH statement has the following syntax:

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

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

```
switch($node)
```

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

```
case "jupiter":
```

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

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

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

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

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

The behaviour of a BREAK statement in a SWITCH statement case is identical to the behaviour of a BREAK statement inside an IF statement. If the SWITCH statement is inside the body of a loop statement then the process will exit the loop. If the SWITCH statement is not part of a loop body then the rules processing of the event is terminated at that point and the event is sent on to the ObjectServer.

BREAK statement

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

The behavior of the BREAK statement is as follows:

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

For additional information, see “Examples of the looping function” on page 22.

Embedding multiple rules files in a rules file

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

The format is as follows:

```
include "rulesfile"
```

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

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

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

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

Rules file functions and operators

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

The following table lists the rules file operators.

Table 4. Rules file operators

Operators	Description	Further details
*, /, -, +	Perform math and string operations.	"Math and string operators" on page 31
&, , ^, >>, <<	Perform bitwise operations.	"Bit manipulation operators" on page 31
==, !=, <>, <, >, <=, >=	Perform comparison operations.	"Comparison operators" on page 32
NOT (also !), AND (also &&), OR (also), XOR (also ^)	Perform logical (Boolean) operations.	"Logical operators" on page 32

The following table lists the rules file functions.

Table 5. Rules file functions

Function name	Description	Further details
charcount	Returns the number of characters in a string.	"String functions" on page 33
clear	Removes the elements of an array.	"String functions" on page 33
datetotime	Converts a string into a time data type.	"Date and time functions" on page 37
details	Adds information to the alerts.details table.	"Details function" on page 41
discard	Deletes an entire event.	"Elements and event functions" on page 33
exists	Tests for the existence of an element.	"Existence function" on page 33
expand	Returns a string (which must be a literal string) with escape sequences expanded.	"String functions" on page 33
extract	Returns the part of a string (which can be a field, element, or string expression) that matches the parenthesized section of the regular expression.	"String functions" on page 33
genevent	Enables you to: <ul style="list-style-type: none">• Create and send an alert from a rules file to a target ObjectServer.• Send the same alert to more than one ObjectServer or table.	"Sending alerts to alternative ObjectServers and tables" on page 43
getdate	Returns the current date as a date data type.	"Date and time functions" on page 37
getenv	Returns the value of an environment variable.	"Host and process utility functions" on page 38
geteventcount	Returns the number of events in the event window.	"Monitoring probe loads" on page 50

Table 5. Rules file functions (continued)

Function name	Description	Further details
gethostaddr	Returns the IP address of the host by using a naming service.	"Host and process utility functions" on page 38
gethostname	Returns the name of the host by using a naming service.	"Host and process utility functions" on page 38
getload	Measures the load on the ObjectServer.	"Monitoring probe loads" on page 50
getpid	Returns the process ID of a running probe.	"Host and process utility functions" on page 38
getplatform	Returns the operating system platform the probe is running on.	"Host and process utility functions" on page 38
hostname	Returns the name of the host on which the probe is running.	"Host and process utility functions" on page 38
int	Converts a numeric value into an integer.	"Math functions" on page 36
length	Returns the number of bytes in a string.	"String functions" on page 33
log	Enables you to log messages.	"Message logging functions" on page 42
lookup	Uses a lookup table to map additional information to an alert.	"Lookup table operations" on page 39
lower	Converts an expression to lowercase.	"String functions" on page 33
ltrim	Removes white space from the left of an expression.	"String functions" on page 33
match	Tests for an exact string match.	"String functions" on page 33
nmatch	Tests for a string match at the beginning of a specified string.	"String functions" on page 33
nvp_add	Enables probes to generate events that contain extended attributes, which are supplied as name-value pairs.	"String functions" on page 33
nvp_remove	Used with extended attributes. Removes specified keys from a name-value pair string, and returns the new name-value pair string.	"String functions" on page 33
printable	Converts any non-printable characters in an expression to a space character.	"String functions" on page 33
real	Converts a numeric value into a real number.	"Math functions" on page 36
recover	Recovers a discarded event.	"Elements and event functions" on page 33
regmatch	Performs full regular expression matching of a value in a regular expression in a string.	"String functions" on page 33

Table 5. Rules file functions (continued)

Function name	Description	Further details
regreplace	Uses regular expressions to perform search and replace operations on strings.	"Search and replace function" on page 47
remove	Removes an element from an event.	"Elements and event functions" on page 33
registertarget	Registers an ObjectServer so alerts can be sent to multiple ObjectServers.	"Sending alerts to alternative ObjectServers and tables" on page 43
rtrim	Removes white space from the right of an expression.	"String functions" on page 33
scanformat	Converts an expression according to the available formats, similar to the scanf family of routines in C.	"String functions" on page 33
setlog	Enables you to set the message log level.	"Message logging functions" on page 42
settarget, setdefaulttarget	Sets the ObjectServer to which alerts are sent.	"Sending alerts to alternative ObjectServers and tables" on page 43
service	Sets the status of a service.	"Service function" on page 49
split	Separates a string into elements of an array.	"String functions" on page 33
substr	Extracts a substring from an expression.	"String functions" on page 33
table	Defines a lookup table.	"Lookup table operations" on page 39
timetodate	Converts a time value into a string data type.	"Date and time functions" on page 37
toBase(numeric,numeric)	Converts a decimal numeric value into a different base.	"Math functions" on page 36
update	Indicates which fields are updated when an alert is deduplicated.	"Update on deduplication function" on page 41
updateload	Updates the load statistics for the ObjectServer.	"Monitoring probe loads" on page 50
upper	Converts an expression to uppercase.	"String functions" on page 33

Math and string operators

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

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

Table 6. Math operators

Operator	Description	Example
* /	Operators used to multiply (*) or divide (/) two operands.	\$eventid=int(\$eventid)*2
+ -	Operators used to add (+) or subtract (-) two operands.	\$eventid=int(\$eventid)+1

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

Table 7. String operator

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

Bit manipulation operators

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

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

Table 8. Bitwise operators

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

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

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

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

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

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

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

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

Related reference

"Math functions" on page 36

Comparison operators

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

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

Table 9. Comparison operators

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

Logical operators

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

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

Table 10. Logical operators

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

Existence function

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

Use the following syntax:

```
exists ($element)
```

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

Elements and event functions

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

The following table describes these functions.

Table 11. Deleting elements or events

Function	Description	Example
discard	Deletes an entire event. Note: This must be in a conditional statement; otherwise, all events are discarded.	<code>if match(@Node,"testnode") { discard }</code>
recover	Recovers a discarded event.	<code>if match(@Node,"testnode") { recover }</code>
<code>remove(element_name)</code>	Removes the element from the event.	<code>remove(test_element)</code>

String functions

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

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

Table 12. String functions

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

Table 12. String functions (continued)

Function	Description	Example
<code>expand("string")</code>	<p>Returns the string (which must be a literal string) with escape sequences expanded. Possible expansions are:</p> <p>\ " - double quote</p> <p>\NNN - octal value of NNN</p> <p>\\ - backslash</p> <p>\a - alert (BEL)</p> <p>\b - backspace</p> <p>\e - escape (033 octal)</p> <p>\f - form feed</p> <p>\n - new line</p> <p>\r - carriage return</p> <p>\t - horizontal tab</p> <p>\v - vertical tab</p> <p>This function cannot be used as the regular expression argument in the <code>regmatch</code> or <code>extract</code> functions.</p>	<p><code>log(debug, expand("Rules file with embedded \\\""))</code></p> <p>sends the following to the log:</p> <p>Sun Oct 21 19:56:15 2001 Debug: Rules file with embedded \"</p>
<code>extract(string, "regexp")</code>	Returns the part of the string (which can be a field, element, or string expression) that matches the parenthesized section of the regular expression.	<p><code>extract (\$expr, "ab([0-9]+)cd")</code></p> <p>If <code>\$expr</code> is "ab123cd" then the value returned is 123.</p>
<code>length(expression)</code>	Returns the number of bytes in a string.	<code>\$NodeLength = length(\$Node)</code>
<code>lower(expression)</code>	Converts an expression to lowercase.	<code>\$Node = lower(\$Node)</code>
<code>ltrim(expression)</code>	Removes white space from the left of an expression.	<code>\$TrimNode = ltrim(\$Node)</code>
<code>match(expression, "string")</code>	TRUE if the expression value matches the string exactly.	<code>if match(\$Node, "New")</code>
<code>nmatch(expression, "string")</code>	TRUE if the expression starts with the specified string.	<code>if nmatch(\$Node, "New")</code>
<code>nvp_add(string_nvp, \$name, \$value [, \$name2, \$value2,]*)</code> <code>nvp_add(\$*)</code>	<p>Creates or updates a name-value pair string of extended attributes. Multiple name-value pairs can be supplied for the string. Variables and their values can be added to, or replaced in, the name-value pair string.</p> <p>Creates a name-value pair string of all variables and their values when called as <code>nvp_add(\$*)</code>.</p>	<p><code>if (int(\$PercentFull) > 95)</code></p> <pre>{ @Severity = 5 @ExtendedAttr = nvp_add(@ExtendedAttr, "PercentFull", \$PercentFull, "Disk", \$Disk) }</pre> <p>If <code>\$PercentFull</code> is 97 and <code>\$Disk</code> is <code>/dev/sfa1</code>, <code>@ExtendedAttr</code> will be (assuming it was initially empty): <code>PercentFull="97";Disk="/dev/sfa1"</code></p>

Table 12. String functions (continued)

Function	Description	Example
<code>nvp_remove(string_nvp, string_key1 [, string_key2, [...]])</code>	<p>Used with extended attributes. Removes specified keys from a name-value pair string, and returns the new name-value pair string.</p> <p>Useful where the list of extended attributes to include is longer than the list of attributes to exclude. You can use <code>nvp_add(\$*)</code> to include all variables and their values, and then use <code>nvp_remove</code> to remove specific ones.</p>	<pre>\$interface = "eth0" \$network = "178.268.2.0" \$ipaddr = "178.268.2.64" \$netmask = "233.233.233.0" \$gateway = "178.268.2.1" @ExtendedAttr = nvp_add(\$*) @ExtendedAttr = nvp_remove(@ExtendedAttr, "network", "netmask")</pre> <p>This results in @ExtendedAttr being: interface="eth0";ipaddr="178.268.2.64"; gateway="178.268.2.1"</p>
<code>printable(expression)</code>	Converts any non-printable characters in the given expression into a space character.	<code>\$Print = printable(\$Node)</code>
<code>regmatch(expression, "regexp")</code>	Full regular expression matching.	<code>if (regmatch(\$enterprise, "^Acme Config:[0-9]"))</code>
<code>regreplace(expression, "regexp", string [, count])</code>	Uses a regular expression and a substitution string to perform a search and replace operation on an input string expression.	<code>\$result = regreplace(\$input, "([%'])", "")</code>
<code>rtrim(expression)</code>	Removes white space from the right of an expression.	<code>\$TrimNode = rtrim(\$Node)</code>
<code>scanformat(expression, "string")</code>	<p>Converts the expression according to the following formats, similar to the <code>scanf</code> family of routines in C. Conversion specifications are:</p> <p>%% - literal %; do not interpret</p> <p>%d - matches an optionally signed decimal integer</p> <p>%u - same as %d; no check is made for sign</p> <p>%o - matches an optionally signed octal number</p> <p>%x - matches an optionally signed hexadecimal number</p> <p>%i - matches an optionally signed integer</p> <p>%e, %f, %g - matches an optionally signed floating point number</p> <p>%s - matches a string terminated by white space or end of string</p>	<pre>\$element = "Lou is up in 15 seconds" [\$node, \$state, \$time] = scanformat(\$element, "%s is %s in %d seconds")</pre> <p>This sets \$node, \$state, and \$time to Lou, up, and 15, respectively.</p>

Table 12. String functions (continued)

Function	Description	Example
<code>num_returned_fields = split("string", destination_array, "field_separator")</code>	<p>Separates the specified string into elements of the destination array.</p> <p>The field separator separates the elements. The field separator itself is not returned. If you specify multiple characters in the field separator, when any combination of one or more of the characters is found in the string, a separation will occur.</p> <p>Regular expressions are not allowed in the string or field separator.</p>	<p><code>\$num_elements=split("bilbo:frodo:gandalf",names,":")</code></p> <p>creates an array with three entries:</p> <pre>names[1] = bilbo names[2] = frodo names[3] = gandalf</pre> <p><code>num_elements</code> is set to 3.</p> <p>You must define the <code>names</code> array at the start of the rules file, before any processing statements.</p>
<code>substr(expression,n, len)</code>	Extracts a substring, starting at the position specified in the second parameter, for the number of characters specified by the third parameter.	<p><code>\$Substring = substr(\$Node,2,10)</code></p> <p>extracts 10 characters from the second position of the <code>\$Node</code> element</p>
<code>upper(expression)</code>	Converts an expression to uppercase.	<code>\$Node = upper(\$Node)</code>

Related concepts

Appendix C, "Regular expressions," on page 143

Related reference

"Using arrays" on page 19

"Search and replace function" on page 47

Math functions

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

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

Table 13. Math functions

Function	Description	Example
<code>int(numeric)</code>	Converts a numeric value into an integer.	<code>if int(\$PercentFull) > 80</code>
<code>real(numeric)</code>	Converts a numeric value into a real number.	<code>@DiskSpace= (real(\$diskspace)/real(\$total))*100</code>
<code>toBase(base,value)</code>	Converts a decimal numeric value into a different base.	<p><code>toBase(2,16)</code> returns 10000</p> <p><code>toBase(16,14)</code> returns E</p> <p><code>toBase(16,\$a)</code> returns the value of the element <code>\$a</code> converted into base 16</p>

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

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

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

Example: Calculating the amount of disk space

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

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

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

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

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

Date and time functions

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

Times are specified in Coordinated Universal Time (UTC), as the number of elapsed seconds since 1 January 1970. The following table describes the date and time functions supported in rules files.

Table 14. Date and time functions

Function	Description	Example
<code>datetotime(string, conversion_specification)</code>	Converts a textual representation of a timestamp into UNIX epoch time (that is, the number of seconds since 00:00:00 1 Jan 1970 GMT). The POSIX format with % is deprecated.	<code>\$Date = datetotime("Tue Dec 19 18:33:11 GMT+00:00 2000", "EEE MMM dd HH:mm:ss vv yyyy")</code>
<code>getdate</code>	Takes no arguments and returns the current date as a date.	<code>\$tempdate = getdate</code>

Table 14. Date and time functions (continued)

Function	Description	Example
<code>timetodate(UTC, conversion_specification)</code>	Converts a time value into a string. Note: For the format used in <i>conversion_specification</i> , see the The POSIX format with % is deprecated.	<code>@StateChange = timetodate (\$StateChange, "HH:mm:ss, MM/dd/yy")</code>

Host and process utility functions

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

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

Table 15. Host and process utility functions

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

Table 15. Host and process utility functions (continued)

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

Lookup table operations

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

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

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

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

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

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

Defining lookup tables in the rules file

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

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

To create a lookup table:

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

```
table tablename={{ "key", "value" }, { "key", "value" } ... }
```
 - b. To create the lookup table with multiple columns, use the following format:

```
table tablename={{ "key1", "value1", "value2", "value3" },  

{ "key2", "val1", "val2", "val3" } }
```
 - c. To create the lookup table and specify a default option to handle an event that does not match any of the key values in the table, use the following format:

```

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

```

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

Example

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

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

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

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

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

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

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

Related tasks

“Re-reading the rules file” on page 7

Defining lookup tables in a separate file

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

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

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

For multiple values, the format is:

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

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

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

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

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

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

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

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

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

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

Update on deduplication function

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

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

The update function has the following syntax:

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

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

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

```
update(@Severity)
```

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

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

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

Details function

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

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

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

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

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

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

```
details($*)
```

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

Example: Using the details function

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

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

Message logging functions

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

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

Log function

The log function sends a message to the log file.

The syntax is:

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

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

Setlog function

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

The syntax is:

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

Example: Message logging

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

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

This produces log output of:

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

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

Sending alerts to alternative ObjectServers and tables

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

Registering target ObjectServers and setting targets for alerts

You can use the `registertarget` function to register one or more ObjectServers, and the corresponding tables, to which you might want to send alerts. You can use the `setdefaulttarget` and `settarget` functions to change the default ObjectServer, or specify an alternative, target ObjectServer.

Usually each alert is sent to only one alerts table, and optionally, a corresponding details table in any of the registered target ObjectServers (except where the `genevent` function is used).

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

```
target = registertarget(server_name, backupserver_name,  
                        database_table [, details_table ] )
```

In this statement:

- *target* is a meaningful label to help you identify the ObjectServer being registered. For example, you can specify a label that denotes the type or distribution of alerts to send to the ObjectServer; for example, `NCOMSalerts`, `FloodProtectionActiveAlert`, `HighAlerts`, `StatsInfoAlert`. This label must be unique across all `registertarget` statements in the rules file.
- *server_name* is the ObjectServer to which you want to send the alert and *backupserver_name* is the backup ObjectServer in the failover pair, if configured. Both the *server_name* and *backupserver_name* values must be enclosed in double quotation marks. To omit the backup ObjectServer, specify *backupserver_name* as an empty string `""`.
- *database_table* is a valid table in any database, into which the alert data should be inserted. This value must be enclosed in double quotation marks.
- *details_table* is the details table to which additional detail information for the alert should be inserted. This value must be enclosed in double quotation marks.

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

You must register all potential targets at the start of the rules file, before any processing statements. If you want to declare any lookup tables within the rules file, you must do so after all `registertarget` statements.

Note:

- The `registertarget` function requires the same user authorization for all the referenced ObjectServers.
- The default target ObjectServer, which is defined by the first (or only) `registertarget` statement will supersede any target ObjectServer that you specify by running the probe with the `-server` command-line option. For example, if

you have a single `registertarget` statement to register TEST1 as the default ObjectServer to which alerts are sent, and then run the probe with `-server` set to TEST2, alerts will be sent to TEST1.

In the following example, multiple targets are registered:

```
DefaultAlerts = registertarget( "TEST1", "", "alerts.status" )
HighAlerts = registertarget( "TEST2", "", "alerts.status" )
ClearAlerts = registertarget( "TEST3", "", "alerts.status" )
London = registertarget( "NCOMS", "NCOMSBACK", "alerts.london" )
```

When you register more than one target, the one registered first is initially the default target. In the preceding example, unless another command overrides the settings, the alert destination following these `registertarget` commands is the `alerts.status` table in the TEST1 ObjectServer.

You can use the `setdefaulttarget` function to change the default ObjectServer to which alerts are sent when a target is not specified.

The `settarget` function enables you to specify the ObjectServer to which an alert will be sent, without changing the default target. You can change both the default ObjectServer and the ObjectServer to which specific alerts are sent, as shown in the following example:

```
# When an event of Major severity or higher comes in,
# set the default ObjectServer to TEST2
if(int(@Severity) > 3)
{ setdefaulttarget(HighAlerts) }
# Send all clear events to TEST3
if (int(@Severity) = 0)
{ settarget(ClearAlerts) }
```

Related reference

“Details function” on page 41

“Lookup table operations” on page 39

“Sending alerts to multiple ObjectServers and tables”

Sending alerts to multiple ObjectServers and tables

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

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

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

ObjectServers. You can use the `genevent` function to send the alert data to all the ObjectServers. Note, however, that this type of usage is *not* intended as a replacement to the use of a gateway in a failover pair because the duplicated alerts will not be correctly associated with each other.

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

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

In this statement:

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

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

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

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

```
genevent(target)
```

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

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

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

Sending detail information and service status to targets

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

- If a `registertarget` statement specifies a details table to which detail information should be sent, a `genevent` statement that sends alerts to the same target will also send the detail information to the details table specified in the `registertarget` statement. This condition is true only if the details statement precedes the `genevent` statement.
- If you use the `service` function to define the status of a service, and the `service` statement precedes the `genevent` statement, the `genevent` statement will send the status information to its target `ObjectServer`.
- If more than one details or service statement precedes or follows a `genevent` statement, only the information from the last details or service statement directly above the `genevent` statement will be sent to the target. Information that is generated by any of the other details or service statements is associated with the main alert only, and is sent only to the relevant targets defined in the `registertarget` statements.

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

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

Related concepts

“Detecting event floods and anomalous event rates” on page 57

Related reference

“Registering target `ObjectServers` and setting targets for alerts” on page 43

“Details function” on page 41

“Service function” on page 49

Multithreaded processing of alert data

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

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

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

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

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

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

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

Related concepts

“Store-and-forward mode for probes” on page 10

Related reference

Chapter 5, “Common probe properties and command-line options,” on page 77

Search and replace function

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

The syntax is as follows:

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

Where:

- The *input* is a string expression. The `regreplace` function reads the input string from left to right.
- The *regularexpression* is a string. It cannot be a string expression. You can use parentheses () in the string to specify substrings that require specific matching. You can use multiple sets of parentheses in a string.
- The *substitution* is a string expression that specifies how strings that match *input* are to be written in the result. You can use metacharacters to reference matching substrings (in parentheses), as well as an entire matching string or strings. For example, `\1` matches the first group in the regular expression, `\2` the second, and so on, while `&` or `\0` match the entire string. Characters and strings that do not match the regular expression are copied to the result string.

- The *count* is an optional positive integer expression, and denotes the number of substitutions to be made on matching strings. If you do not provide a value for *count*, the substitutions continue until no more matching strings are found. If *count* is a non-integer expression, it is interpreted as 0, and the *input* is not changed. If *count* is a negative integer, a warning message is entered in the probe log, and the *input* is not changed.

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

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

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

The result of this expression is as follows:

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

Example: Reordering groups of characters in a string

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

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

The regular expression matches the substrings in the following order:

```
\1="aba"
```

```
\2="argle"
```

```
\3="aca"
```

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

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

Example: Using metacharacters to match an entire string

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

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

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

The result of this expression is as follows:

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

Related concepts

Appendix C, "Regular expressions," on page 143

Service function

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

The syntax is:

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

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

The following table lists the service status levels.

Table 16. Service function status levels

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

Example: Service function

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

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

```

default:
@Summary = "Ping Probe error details: " + $*
@Severity = "3"
service($host, marginal) # Service Entry
}

```

Monitoring probe loads

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

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

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

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

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

```
time_window_in_seconds.max_number_of_samples
```

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

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

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

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

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

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

Related reference

“Rules file examples” on page 54

Reserved words in the probe rules language

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

The following list shows the reserved words:




- and
- array
- bad
- break
- case
- char
- character
- charcount
- clear
- datetime
- datetotime
- debug
- decode
- default
- details
- discard
- double
- else
- error
- exists
- exit
- expand
- extract
- false
- fatal
- foreach
- genevent
- getdate
- good
- if
- in
- include
- info
- information
- int
- integer
- len
- length
- log
- lookup

- lower
- ltrim
- marginal
- match
- nmatch
- no
- not
- nvp_add
- nvp_remove
- off
- on
- or
- printable
- real
- recover
- registertarget
- regmatch
- regreplace
- remove
- rtrim
- scanformat
- service
- setdefaultobjectserver
- setdefaulttarget
- setlog
- setobjectserver
- settarget
- split
- string
- substr
- switch
- table
- timetodate
- true
- update
- upper
- warn
- warning
- xor
- yes

Testing rules files

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

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

-   \$NCHOME/omnibus/probes
-  %NCHOME%\omnibus\probes\win32

To run the Probe Rules Syntax Checker, enter the following command:

```
nco_p_syntax -rulesfile /rules_file_path/rules_file.rules
```

When running this command, use the `-rulesfile` command-line option to specify the full path and file name of the rules file.

Results

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

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

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

Debugging rules files

When making changes to the rules file, adding new rules, or creating lookup tables, it is useful to test the probe by running it in debug mode. This shows exactly how an event is being parsed by the probe and any possible problems with the rules file.

To change the message level of a running probe to run in debug mode, issue the command `kill -USR2 pid` on the probe process ID (PID). See the **ps** and **kill** man pages for more information.

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

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

You also can set the probe to run in debug mode on the command line or in the properties file. To enable debug mode on the command line, enter the command:

```
$OMNIHOME/probes/arch/probenname -messagelevel DEBUG -messagelog STDOUT
```

Alternatively, you can add the following entries to the properties file:

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

If you omit the **MessageLog** property or `-messageLog` command-line option, the debug information is sent to the probe log file in the `$OMNIHOME/log` directory rather than to the screen.

Tip: For changes to the rules file to take effect, the probe must be forced to re-read the rules file.

Related tasks

“Re-reading the rules file” on page 7

Rules file examples

These examples show typical rules file segments.

- “Example: Enhancing the Summary field”
- “Example: Populating multiple fields”
- “Example: Nested IF statements”
- “Example: Regular expression match” on page 55
- “Example: Regular expression extract” on page 55
- “Example: Numeric comparisons” on page 55
- “Example: Simple numeric expressions” on page 55
- “Example: Strings and numerics in one expression” on page 55
- “Example: Using load functions to monitor nodes” on page 55

Example: Enhancing the Summary field

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

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

Example: Populating multiple fields

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

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

Example: Nested IF statements

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

```
if( match($enterprise,"Acme") )
{
  if( match($trap-type, "Link-Up") )
```

```
{
@Summary= "Acme Link Up on " + @Node + " Port " + $ifIndex +
" Reason: "+$ifLocReason
} }
```

Example: Regular expression match

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

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

Example: Regular expression extract

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

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

Example: Numeric comparisons

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

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

Example: Simple numeric expressions

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

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

Example: Strings and numerics in one expression

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

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

Example: Using load functions to monitor nodes

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

```

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

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

if ( match("", loads[@Node]) ){
    loads[@Node] = "2.50"
}
if ( match("", %general_load) ){
    %general_load="2.50"
}
loads[@Node] = updateload(loads[@Node])
%general_load=updateload(%general_load)
if ( int(getload(loads[@Node]) ) > 5 ){
    log(WARN, $Node + " is creating more than 5 events per second")
}
if ( int(getload(%general_load)) > 80){
    log(WARN, "Probe is creating more than 80 events per second - switching to HIGHLOAD")
    settarget(HIGHLOAD)
}

```

Related reference

“Search and replace function” on page 47

“Examples of the looping function” on page 22

Chapter 3. Probe rules file customizations

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

Detecting event floods and anomalous event rates

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

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

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

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

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

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

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

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

Related reference

“Flood rules file” on page 62

“Flood configuration rules file” on page 59

Configuring probes to detect event floods and anomalous event rates

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

To enable these features for a probe:

1. Go to the `$NCHOME/omnibus/extensions/eventflood` directory.
2. Copy the `flood.config.rules` and `flood.rules` files to a preferred local or remote directory where the primary rules file for the probe or any secondary rules files are stored. Remove the default read-only permissions from the `flood.config.rules` and `flood.rules` files.
3. Edit the `flood.config.rules` and `flood.rules` files as appropriate for your requirements. You can comment out any unrequired sections. For example, if you do not want to discard alerts during an event flood, you can comment out the conditional statement in the `flood.rules` file. In the sample `flood.rules` file, event rates are calculated against all event sources that send data to the probe.
4. Embed the updated rules files in your probe rules file (typically `$NCHOME/omnibus/probes/arch/probename.rules`) by using `include` statements:
 - Include the `flood.config.rules` file at the very beginning of your set of rules, before any processing statements. This file contains an array declaration and `registertarget` statements, which must be defined at the start of a rules file.
 - Include the `flood.rules` file (within the primary rules file or another secondary rules file) in the section that defines your set of rules. If you want to conditionally discard or divert alerts with particular severity levels during the event flood, you must include the `flood.rules` file towards the end of the set of rules, at a position where the severity of the alert has already been determined.
5. After updating the probe rules file with the `include` statements, have the probe re-read the rules file.
6. In the probe properties file (`$NCHOME/omnibus/probes/arch/probename.props`), set the **MaxLogFileSize** property to a value that is large enough to accommodate the extra log file messages that are generated when the `flood.rules` file is processed.

Related tasks

“Re-reading the rules file” on page 7

Related reference

“Flood rules file” on page 62

“Flood configuration rules file”

“Embedding multiple rules files in a rules file” on page 27

Chapter 5, “Common probe properties and command-line options,” on page 77

Flood configuration rules file

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

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

Table 17. `flood.config.rules` file entries

Entry	Description	Action
<code>DefaultOS = registertarget("NCOMS", "", "alerts.status")</code>	This statement registers the default NCOMS ObjectServer as a target for alerts. In the flood rules file, this is the target ObjectServer to which an informational alert is sent when the current event rate from probe sources is unusually high or low, or when an event flood starts and ends. The default table to which the alert is sent is <code>alerts.status</code> .	Change the ObjectServer name and alerts table name to preferred valid names.
<code>#FloodEventOS = registertarget("NCOMS_BK", "", "alerts.status")</code>	This commented-out line registers an NCOMS_BK ObjectServer. In the flood rules file, this is an alternative target ObjectServer to which alerts with particular severity levels can be diverted during an event flood.	Uncomment this line if you want to divert alerts to this ObjectServer when an event flood is detected. Change the ObjectServer name and alerts table name to preferred valid names.
<code>array event_rate_array</code>	This array is defined to hold all the event rate calculation variables. These variables are used throughout the flood rules file.	N/A

Table 17. *flood.config.rules* file entries (continued)

Entry	Description	Action
<p>\$average_event_rate_time_window</p> <p>\$average_event_rate_max_sample_size</p>	<p>These elements store values that are used to calculate what is considered to be the average (or normal) rate of receipt of events:</p> <ul style="list-style-type: none"> • The \$average_event_rate_time_window element defines the maximum time window (in seconds) for which events are kept. This value depicts a rolling time window, which is updated by calling the <code>updateLoad</code> function. The \$average_event_rate_time_window element also sets the <i>training period</i>, which is the length of time the probe runs to determine the average or normal event rate. • The \$average_event_rate_max_sample_size element defines the maximum number of events to keep during the average event rate time window. <p>In the flood rules file, these elements are used to capture the event count in the last <i>n</i> seconds before the current time, and to calculate the average event rate during this period.</p>	<p>Change the default values as appropriate for your requirements.</p>
<p>\$flood_detection_time_window</p> <p>\$flood_detection_max_sample_size</p>	<p>These elements store values that are used to calculate the event flood detection rate, in order to determine whether an event flood is imminent:</p> <ul style="list-style-type: none"> • The \$flood_detection_time_window element defines the maximum time window (in seconds) for which events are kept. This value depicts a rolling time window, which is updated by calling the <code>updateLoad</code> function. • The \$flood_detection_max_sample_size element defines the maximum number of events to keep during this period. <p>In the flood rules file, these elements are used to capture the event count in the last <i>n</i> seconds before the current time, and to calculate the flood detection rate during this period.</p>	<p>Change the default values as appropriate for your requirements.</p>
\$flood_detection_startup_time	<p>This element defines the number of seconds over which the probe runs before event flood detection can begin.</p>	<p>Set a value.</p>

Table 17. *flood.config.rules* file entries (continued)

Entry	Description	Action
<code>\$anomaly_detection_time_window</code> <code>\$anomaly_detection_max_sample_size</code>	<p>These elements store values that are used to calculate the rate of receipt of events for detecting an anomalous flow:</p> <ul style="list-style-type: none"> The <code>\$anomaly_detection_time_window</code> element defines the maximum time window (in seconds) for which events are kept. This value depicts a rolling time window, which is updated by calling the <code>updateLoad</code> function. The <code>\$anomaly_detection_max_sample_size</code> element defines the maximum number of events to keep during this period. <p>In the flood rules file, these elements are used to capture the event count in the last <i>n</i> seconds before the current time, and to calculate the event rate during this period.</p>	Change the default values as appropriate for your requirements.
<code>\$flood_detection_event_rate_flood_threshold</code> <code>\$flood_detection_event_rate_normal_threshold</code>	<p>These elements store values that are used to specify event rate thresholds for detecting an event flood or a normal event rate.</p> <p>If the number of events received per second exceeds the value specified for the <code>\$flood_detection_event_rate_flood_threshold</code> element, event flood detection is triggered.</p> <p>If the number of events received per second is less than the value specified for the <code>\$flood_detection_event_rate_normal_threshold</code> element, a normal event rate is assumed.</p>	<p>Change the default values as appropriate for your requirements.</p> <p>Ensure that the value of <code>\$flood_detection_event_rate_normal_threshold</code> is lower than <code>\$flood_detection_event_rate_flood_threshold</code>.</p>
<code>\$lower_event_rate_threshold_multiplier</code> <code>\$upper_event_rate_threshold_multiplier</code>	<p>The <code>\$lower_event_rate_threshold_multiplier</code> element sets the multiplier value that is used to calculate the lower event rate threshold for detecting an anomalous event rate.</p> <p>The <code>\$upper_event_rate_threshold_multiplier</code> element sets the multiplier value that is used to calculate the upper event rate threshold for detecting an anomalous event rate.</p> <p>In the flood rules file, the average event rate is multiplied by these values to set the thresholds for determining unusually low or unusually high event rates.</p>	Change the default values as appropriate for your requirements.
<code>\$discard_event_during_flood</code>	<p>This element defines whether an alert is discarded during an event flood. A value of 1 equates to TRUE and a value of 0 equates to FALSE.</p> <p>In the flood rules file, if the <code>\$discard_event_during_flood</code> value is 1 and the alert is of a lower severity than the value specified for <code>\$forward_event_minimum_severity</code>, the alert will be discarded.</p>	Change the default value as appropriate for your requirements.

Table 17. *flood.config.rules* file entries (continued)

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

Related reference

"Flood rules file"

Flood rules file

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

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

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

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

Anomalous event rate calculations

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

At the end of this period, upper and lower event rate thresholds are calculated as percentages of the average event rate. The \$upper_event_rate_threshold_multiplier and

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

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

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

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

The upper event rate threshold can be calculated as follows:

average event rate * 5 = 100 events per second

The lower event rate threshold can be calculated as follows:

average event rate * 0.1 = 2 events per second

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

Flood detection calculations

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

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

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

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

Message logging

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

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

Related reference

“Flood configuration rules file” on page 59

Chapter 5, “Common probe properties and command-line options,” on page 77

“Sending alerts to alternative ObjectServers and tables” on page 43

Enabling self monitoring of probes

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

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

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

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

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

Related reference

“ProbeWatch and TSMWatch messages” on page 128

Configuration setup for self monitoring of probes

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

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

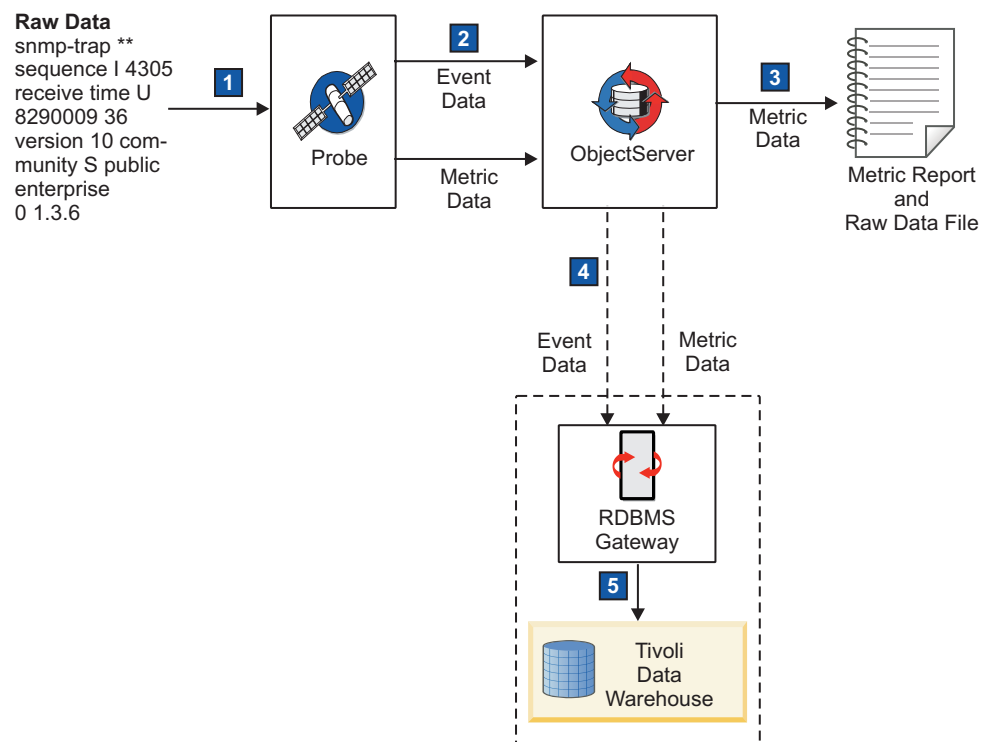


Figure 3. Configuration and data flow for probe self monitoring

The configuration flow is as follows:

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

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

Both sets of events are forwarded to the ObjectServer.

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

- 4** and **5**

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

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

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

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

Details of the configuration files are as follows:

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

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

- Individual metrics for each connected probe; for example, the number of events processed, generated, and discarded since the last reporting period
- A set of collated metrics; for example, the total number of `alerts.details` and `alerts.journal` inserts since the last reporting period

You can review this SQL file to familiarize yourself with the potential changes that will be applied to the ObjectServer.

- `probewatch.include` file: This customized rules file is provided for use with probes, and must be embedded within the main rules file for the probe. The `probewatch.include` file expands on the original default ProbeWatch-specific rules. This file contains new CASE statements for two additional ProbeWatch messages and for the ProbeWatch Heartbeat events, which act as a carrier for statistical data.

The `probewatch.include` rules file is generic to all probes. You can customize and share this file between multiple (or all) probes to centralize the administration of ProbeWatch Heartbeat events.

- `Omnibus_TDW_Reports_ROI.zip` file: This archive file contains a set of sample reports that require user customization, and integration with Tivoli Data

Warehouse and Tivoli Common Reporting. Working knowledge of these components is required to support this configuration.

A number of statistical properties are also added to the configuration for probes. These properties are used to collect usage and resource information that is specific to each probe. The statistical properties are different from the standard probe properties because they cannot be set to a meaningful value in the properties file, and they cannot be run as command-line options. These property names are all prefixed with **Op1Stats**, and are displayed in the output obtained when the probe is run with the `-dumpprops` command-line option.

The statistical properties are as follows:

Table 18. Statistical properties for probes

Property	Description
Op1StatsCPUTimeSec	The CPU time consumed by the probe in seconds. Example: If 6.002345 seconds CPU time has been consumed by the probe, Op1StatsCPUTimeSec = 6
Op1StatsCPUTimeUSec	The subsecond component of CPU time consumed by the probe, in millionths of a second. Example: If 6.002345 seconds CPU time has been consumed by the probe, Op1StatsCPUTimeUSec = 2345
Op1StatsRulesFileTimeSec	The time spent processing rules in seconds. Example: If 4.372700 seconds is spent processing rules, Op1StatsRulesFileTimeSec = 4
Op1StatsRulesFileTimeUSec	The subsecond component of time spent processing rules, in millionths of a second. Example: If 4.372700 seconds is spent processing rules, Op1StatsRulesFileTimeUSec = 372700
Op1StatsProbeStartTime	The time (in UNIX epoch time) at which the probe was started.
Op1StatsMemoryInUse	The memory footprint (in KB) of the probe.
Op1StatsNumberEvents	The number of events (including ProbeWatch events) that the probe has received from its event source since the probe started.
Op1StatsNumberEventsDiscarded	The number of events that are discarded after rules processing.
Op1StatsNumberEventsGenerated	The number of events that are generated using the <code>genevent</code> function in the rules file.

Configuring probes for self monitoring

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

To configure a probe to collect and process statistical data:

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

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

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

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

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

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

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

The code shown in bold text needs to be replaced with an include statement that enables you to embed the contents of the `probewatch.include` file, as instructed in step 5 on page 70.

4. Remove the default read-only permissions from your copy of the `probewatch.include` file and review the file to familiarize yourself with its contents. Then edit the file as follows:
 - Update any of the elements at the top of the file to define how a ProbeWatch Heartbeat event should be processed. Use the number sign (#) to comment

out any elements that you do not require. The processing logic for these elements is coded within the case "Heartbeat ..." statement in the ProbeWatch section of the file.

Table 19. Elements for ProbeWatch Heartbeat events

Element	Action
\$OplHeartbeat_discard	Set this value to 1 if you want to discard the ProbeWatch Heartbeat event. Set this value to 0 if you want to forward the ProbeWatch Heartbeat event to the ObjectServer.
\$OplHeartbeat_populate_master_probestats	Set this value to 1 to enable a new probe metrics event to be generated by using the genevent function, which is defined within the case "Heartbeat ..." statement. The event data consists a set of OplStats probe metrics, which are forwarded to the master.probestats table that was created when you ran the probestats.sql script. Set this value to 0 if you do not want to generate this event for insertion into the master.probestats table.
\$OplHeartbeat_write_to_probe_log	Set this value to 1 if you want to record the OplStats metrics in the probe log file. Details are logged at the INFO level. The metric details that are logged are defined in the case "Heartbeat ..." statement. Set this value to 0 if you do not want to record the metrics in the log file.
\$OplHeartbeat_generate_threshold_events	Set this value to 1 if you want to generate threshold events that indicate when a particular probe metric violates a defined threshold. By default, no code is provided for threshold events within this rules file because individual preferences can vary widely. If you require threshold events, you must first decide which thresholds you want to monitor. Then, within the case "Heartbeat ..." statement, provide the code for generating threshold events.

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

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

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

```

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

```

If you do not require these, use the discard function to prevent them from being sent to the ObjectServer.

- The final CASE statement (case "Heartbeat ...") contains a set of conditional statements for calculating the probe metrics and processing the data. IF statements are provided with the logic to discard events and to write the probe metrics to a log file. Some user input is also required:

Table 20. case "Heartbeat ..." sections that require user input

<p>Locate the section of code that begins with the following lines:</p> <pre> if(int(\$OpHeartbeat_populate_master_probestats) == 1) { log(DEBUG, "HEARTBEAT - SENDING PROBESTATS TO MASTER.PROBESTATS") ... </pre> <p>This section of code contains a genevent statement with a DefaultOS placeholder that identifies a target, registered ObjectServer. This target must be defined in a registertarget statement in the main rules file. Replace this placeholder with the target ObjectServer to which you want to send events.</p>
<p>Locate the section of code that begins with the following lines:</p> <pre> if(int(\$OpHeartbeat_generate_threshold_events) == 1) { # # Area to generate user defined threshold events using genevent ... </pre> <p>If you set the \$OpHeartbeat_generate_threshold_events element to 1 at the top of the file, you must enter the code for the type of threshold events that you want to monitor.</p> <p>You can ignore this section if you do not require threshold events.</p>

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

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

5. Embed the updated probewatch.include file in your main probe rules file (typically \$NCHOME/omnibus/probes/arch/probename.rules) by using an include statement. Ensure that the path in the include statement points to the location where the updated probewatch.include file is stored.

```

if( match( @Manager, "ProbeWatch" ) )
{
    include "directory_path/probewatch.include"
}

```

```

else
{
    ...probe specific rules...
}

```

6. Edit the `$NCHOME/omnibus/probes/arch/nco_p_probename.props` file to specify the interval (in seconds) at which the ProbeWatch Heartbeat events are generated. Set the **ProbeWatchHeartbeatInterval** property to a positive number to generate the events, or specify 0 (zero) or a negative number for no events.
7. Ensure that the stats_triggers trigger group is enabled. The triggers that are added by the `probestats.sql` file are assigned to this trigger group, which must be enabled for the triggers to run. You can enable the trigger group by using Netcool/OMNIBus Administrator or the ALTER TRIGGER GROUP command, as described in the *IBM Tivoli Netcool/OMNIBus Administration Guide*. Also enable the `probe_statistics_cleanup` trigger, which by default is set to delete probe statistics that are over an hour old. You can change this default period to increase the length of time for which statistics are stored.
8. Start the probe.
The probe metrics that are collected are recorded in the log file `$NCHOME/omnibus/log/server_name_probestats.log`, where `server_name` is the ObjectServer name.

Related concepts

Chapter 4, “Running probes,” on page 73

Related tasks

“Testing rules files” on page 53

Related reference

“Sending alerts to alternative ObjectServers and tables” on page 43

“Embedding multiple rules files in a rules file” on page 27

Chapter 5, “Common probe properties and command-line options,” on page 77

Chapter 4. Running probes

When running a probe, you can specify properties in a properties file or options at the command line to configure settings for the probe.

A probe has default values for each property. In an unedited properties file, all properties are listed with their default values, commented out with a hash symbol (#) at the beginning of the line.

You can edit the properties file before running the probe, or while the probe is running. If you edit the properties file while the probe is running, the changes that you make take effect the next time you start the probe. You can edit probe property values using a text editor. To override a default value, you must change the setting in the properties file and then remove the hash symbol.

If you change a property setting on the command line when starting a probe, this overrides both the default value and the setting in the properties file. To simplify the command that you type to run the probe, add as many properties as possible to the properties file instead of using the command-line options.

When running a probe, you must also set up your rules file to define how the probe should process event data. You can edit the rules file before running the probe, or while the probe is running. If you edit the rules file while the probe is running, you must force the probe to re-read the rules file, for the changes to take effect. You can edit the rules file using a text editor.

Tip: Always read the publication that is specific to the probe you are running for additional configuration information.

Related concepts

“Properties file” on page 5

“Rules file” on page 6

Related tasks

“Re-reading the rules file” on page 7

Running probes on UNIX

On UNIX, you can run probes from the command line, or under process control.

Note: Probes should be managed by process control. For further information about setting up a probe to run under process control, see the *IBM Tivoli Netcool/OMNIBus Administration Guide*.

After you install a probe, you must configure the properties and rules files to fit your environment. For example, if you are using a log file probe such as the HTTP Common Log Format Probe, you must set the **LogFile** property, so that the probe can connect to the event source.

To run a probe:

Enter the following command at the command line:

```
$OMNIHOME/probes/nco_p_probename [-option [ value ] ... ]
```

In this command, the *probename* is the abbreviated name of the probe that you want to run. The *-option* variable is a command-line option, and *value* is the value that you are setting the option to. Not every option requires you to specify a value. For example, to run the Sybase Probe in raw capture mode, enter:

```
$OMNIBUSHOME/probes/nco_p_sybase -raw
```

If you specify the *-name* command-line option, it determines the name used for the probe files described in the following table:

Table 21. Names of probe files

Type of file	Path and file name
Properties file	\$OMNIBUSHOME/probes/arch/probename.props
Rules file	\$OMNIBUSHOME/probes/arch/probename.rules
Store-and-forward file	\$OMNIBUSHOME/var/probename.store.server
Message Log file	\$OMNIBUSHOME/log/probename.log

In these paths, *arch* represents the name of the operating system on which the probe is installed; for example, *solaris2* when running on a Solaris system. If you specify the *-propsfile* command-line option, its value overrides the name setting for the properties file.

Note: If you are running a proxy server, connect your probes to the proxy server rather than to the ObjectServer. To do this, use the **Server** property or the *-server* command-line option and specify the name of the proxy server. For more information about the proxy server, see the *IBM Tivoli Netcool/OMNIBus Administration Guide*.

Related reference

Chapter 5, “Common probe properties and command-line options,” on page 77

Running probes on Windows

On Windows, you can run probes as console applications, as Windows services, or under process control.

Probes are installed as console applications by default.

For further information about setting up a probe to run under process control, see the *IBM Tivoli Netcool/OMNIBus Administration Guide*.

Related reference

Chapter 5, “Common probe properties and command-line options,” on page 77

Running a probe as a console application

Run a probe as a console application from the command line.

To run a probe as a console application, enter the following command from the probe directory:

```
nco_p_probename [ -option [ value ] ... ]
```

In this command, *probename* is the abbreviated name of the probe that you want to run. The *-option* variable is a command-line option, and *value* is the value that you are setting the option to. Not every option requires you to specify a value.

Additional command-line options are available for the Windows version of each probe. To display these, enter the following command:

nco_p_probename -?

The Windows-specific command-line options are described in the following table.

Table 22. Windows-specific probe command-line options

Command-line option	Description
-install	This option installs the probe as a Windows service.
-noauto	This option is used with the -install option. It disables automatic startup for the probe running as a service. If this option is used, the probe is not started automatically when the machine boots.
-remove	This option removes a probe that is installed as a service. It is the opposite of the -install command.
-group <i>string</i>	This option is used with the -depend command-line option. You can group all the probes together under the same group name. You can then force that group to be dependent on another service.
-depend <i>srv @grp ...</i>	This option specifies other services or groups that the probe is dependent on. If you use this option, the probe will not start until the services (<i>srv</i>) and groups (<i>@grp</i>) specified with this option have been run.
-cmdLine " <i>-option value...</i> "	This option specifies one or more command-line options to be set each time the probe service is restarted.

Related tasks

"Running a probe as a service"

Related reference

Chapter 5, "Common probe properties and command-line options," on page 77

Running a probe as a service

To run a probe as a service, use the -install command-line option when running the probe with the nco_p_probename command, where *probename* uniquely identifies the probe.

After setting up the service, you can configure how the probe starts by defining the Windows services settings as follows:

1. Click **Start > Control Panel**. The Control Panel opens.
2. Double-click the **Admin Tools** icon, then double-click the **Services** icon. The Services window opens.
The Services window lists all of the Windows services currently installed on your machine. All Tivoli Netcool/OMNIBus service names start with NCO.
3. Use the Services window to start and stop Windows services. Indicate whether the service is started automatically when the machine is booted by clicking the **Startup** button.

Note: If the ObjectServer and the probe are started as services, the probe may start first. The probe will not be able to connect to the ObjectServer until the ObjectServer is running.

Results

Related tasks

“Running a probe as a console application” on page 74

Use of OMNIHOME and NCHOME environment variables for probes

On Netcool/OMNIBus V7.0, or earlier, the UNIX environment variable \$OMNIHOME (%OMNIHOME% on Windows) is used in many configuration files. Netcool/OMNIBus V7.1, or later, uses the UNIX environment variable \$NCHOME (%NCHOME% on Windows) instead.

Configuration files containing the OMNIHOME environment variable will work on Netcool/OMNIBus V7.1, or later, as long as you set \$OMNIHOME to \$NCHOME/omnibus on UNIX, or set %OMNIHOME% to %NCHOME%\omnibus on Windows.

Chapter 5. Common probe properties and command-line options

A number of properties and command-line options are common to all probes and TSMs.

For the properties and command-line options that are specific to a particular probe or TSM, see the individual publications for each probe and TSM.

Tip: You can encrypt string values in a properties file by using property value encryption.

The following table lists the common properties and command-line options that are available to all probes, and provides their default settings.

Table 23. Common probe properties and command-line options

Property	Command-line option	Description
AuthPassword <i>string</i>	N/A	<p>Specifies the password associated with the user name that is used to authenticate the probe when it connects to a proxy server or an ObjectServer running in secure mode. The default is ''.</p> <p>When in FIPS 140–2 mode, the password can either be specified in plain text, or can be encrypted with the nco_aes_crypt utility. If you are encrypting passwords by using nco_aes_crypt in FIPS 140–2 mode, you must specify AES_FIPS as the encryption algorithm.</p> <p>When in non-FIPS 140–2 mode, the password can be encrypted with the nco_g_crypt or nco_aes_crypt utilities. If you are encrypting passwords by using nco_aes_crypt in non-FIPS 140–2 mode, you can specify either AES_FIPS or AES as the encryption algorithm. Use AES only if you need to maintain compatibility with passwords that were encrypted using the tools provided in versions earlier than Tivoli Netcool/OMNIBus V7.2.1.</p>
AuthUserName <i>string</i>	N/A	<p>Specifies a user name used to authenticate the probe when it connects to a proxy server or an ObjectServer running in secure mode. The default is ''.</p>

Table 23. Common probe properties and command-line options (continued)

Property	Command-line option	Description
AutoSAF 0 1	-autosaf -noautosaf	<p>Specifies whether automatic store-and-forward mode is enabled. In this mode, if the probe starts but is unable to send events to the ObjectServer, the probe goes into store mode instead of terminating.</p> <p>By default, automatic store-and-forward mode is not enabled (0).</p> <p>Note: For automatic store-and-forward to work, the probe must previously have been connected at least once to the ObjectServer so that it knows the format in which to store events for that ObjectServer. If the probe is trying to connect to a virtual pair of ObjectServers and both of the ObjectServers are down, the probe checks the AutoSAF property setting. If automatic store-and-forward is enabled, the probe begins to store events in the store-and-forward file; otherwise, the probe terminates.</p>
BeatInterval <i>integer</i>	-beatinterval <i>integer</i>	Specifies the heartbeat interval for peer-to-peer failover. The default is 2 seconds.
BeatThreshold <i>integer</i>	-beatthreshold <i>integer</i>	Specifies the extra period that the slave probe in a peer-to-peer failover relationship waits for before switching to active mode. The default is 1 second.
Buffering 0 1	-buffer -nobuffer	<p>Specifies whether buffering is used when sending alerts to the ObjectServer. By default, buffering is not enabled (0).</p> <p>Note: All alerts sent to the same table are sent in the order in which they were processed by the probe. If alerts are sent to multiple tables, the order is preserved for each table, but not across tables.</p> <p>If multithreaded processing is in operation (the default), a separate communication thread is used to send data to each registered target ObjectServer, and a separate text buffer is therefore maintained for each ObjectServer.</p>
BufferSize <i>integer</i>	-buffersize <i>integer</i>	Specifies the number of alerts the probe buffers. The default is 10.

Table 23. Common probe properties and command-line options (continued)

Property	Command-line option	Description
ConfigCryptoAlg <i>string</i>	N/A	<p>Specifies the cryptographic algorithm to use for decrypting string values (including passwords) that were encrypted with the nco_aes_crypt utility and then stored in the properties file. Set the <i>string</i> value as follows:</p> <ul style="list-style-type: none"> When in FIPS 140–2 mode, use AES_FIPS. When in non-FIPS 140–2 mode, you can use either AES_FIPS or AES. Use AES only if you need to maintain compatibility with passwords that were encrypted by using the tools provided in versions earlier than Tivoli Netcool/OMNIbus V7.2.1. <p>The value that you specify must be identical to that used when you ran nco_aes_crypt with the -c setting, to encrypt the string values.</p> <p>Use this property in conjunction with the ConfigKeyFile property.</p>
ConfigKeyFile <i>string</i>	N/A	<p>Specifies the path and name of the key file that contains the key used to decrypt encrypted string values (including passwords) in the properties file.</p> <p>The key is used at run time to decrypt string values that were encrypted with the nco_aes_crypt utility. The key file that you specify must be identical to the file used to encrypt the string values when you ran nco_aes_crypt with the -k setting.</p> <p>Use this property in conjunction with the ConfigCryptoAlg property.</p>
N/A	-help	Displays the supported command-line options and exits.
KeepLastBrokenSAF 0 1	-keeplastbrokensaf -dontkeeplastbrokensaf	<p>Specifies whether to automatically save corrupted store-and-forward records for future diagnosis.</p> <p>If set to 1, corrupted store-and-forward records are automatically saved. The default is 0.</p> <p>Use this property in conjunction with the StoreSAFRejects property.</p>
LogFilePoolSize <i>integer</i>	-logfilepoolsize <i>integer</i>	<p>Specifies the number of log files to use if the logging system is writing to a pool of files. This property works only when the LogFileUsePool property is set to TRUE. The pool size can range from 2 to 99.</p> <p>The default is 10.</p> <p>Note: This option is supported only on Windows operating systems.</p>

Table 23. Common probe properties and command-line options (continued)

Property	Command-line option	Description
LogFileUsePool 0 1	-logfileusepool -nologfileusepool	<p>Specifies whether to use a pool of log files for logging messages.</p> <p>If set to 1, the logging system opens the number of files specified for the pool at startup, and keeps them open for the duration of its run. (You define the number of files in the pool by using the LogFilePoolSize property.) When a file in the pool reaches its maximum size (as specified by the MaxLogFileSize property), the logging system writes to the next file. When all the files in the pool are at maximum size, the logging system truncates the first file in the pool and starts writing to it again. Files in the pool are named using the format <i>probename.log_ID</i>, where <i>ID</i> is a two-digit number starting from 01, to the maximum number specified for the LogFilePoolSize property. When the logging system starts to use a file pool, the system writes to the lowest-available file number, regardless of which file it was writing to when it last ran.</p> <p>The default is 0. When set to 0, the default <i>probename.log</i> file is renamed <i>probename.log_OLD</i> and a new log file is started when the maximum size is reached. If the file cannot be renamed, for example, because of a read lock on the <i>_OLD</i> file, and LogFileUseStdErr is set to 0, the logging system automatically starts using a pool of log files. If the file cannot be renamed, and LogFileUseStdErr is set to 1, messages are logged to the console if the probe was run from the command line. If the file cannot be renamed, and LogFileUseStdErr is set to 1, messages are logged to a file named %NCHOME%\omnibus\log\probename.err if the probe is running as a Windows service. Note: This option is supported only on Windows operating systems.</p>
LogFileUseStdErr 0 1	-logfileusestderr -nologfileusestderr	<p>Specifies whether to use stderr as an output stream for logging messages.</p> <p>The default is 1, which causes messages to be logged to the console only if the probe was run from the command line.</p> <p>If set to 0, the logging system writes to the default log file or to a pool of log files, as set by the LogFileUsePool property. Note: The LogFileUsePool property setting takes precedence over the LogFileUseStdErr setting. Note: This option is supported only on Windows operating systems.</p>

Table 23. Common probe properties and command-line options (continued)

Property	Command-line option	Description
LookupTableMode <i>integer</i>	<code>-lookupmode integer</code>	<p>Specifies how table lookups are performed. It can be set to 1, 2, or 3. The default is 3.</p> <p>If set to 1, all external lookup tables are assumed to have a single value column. Tabs are not used as column delimiters.</p> <p>If set to 2, all external lookup tables are assumed to have multiple columns. If the number of columns on each line is not the same, an error is generated that includes the file name and the line on which the error occurred.</p> <p>If set to 3, the rules engine attempts to determine the number of columns in the external lookup table. An error is generated for each line that has a different column count from the previous line. The error includes the file name and the line on which the error occurred.</p>
Manager <i>string</i>	<code>-manager string</code>	Specifies the value of the Manager field for the alert. The default value is determined by the probe.
MaxLogFileSize <i>integer</i>	<code>-maxlogfilesize integer</code>	Specifies the maximum size that the log file can grow to, in Bytes. The default is 1 MB. When the log file reaches the size specified, a second log file is started. When the second file reaches the maximum size, the first file is overwritten with a new log file and the process starts again.
MaxRawFileSize <i>integer</i>	N/A	Specifies the maximum size of the raw capture file, in KB. The default is unlimited (-1).
MaxSAFFileSize <i>integer</i>	<code>-maxsaffilesize integer</code>	Specifies the maximum size (in Bytes) that the store-and-forward file can grow to when disconnected from the ObjectServer. The default is 1 MB.
MessageLevel <i>string</i>	<code>-messagelevel string</code>	<p>Specifies the message logging level. Possible values are: debug, info, warn, error, and fatal. The default level is warn.</p> <p>Messages that are logged at each level are as follows:</p> <p>fatal: fatal only.</p> <p>error: fatal and error.</p> <p>warn: fatal, error, and warn.</p> <p>info: fatal, error, warn, and info.</p> <p>debug: fatal, error, warn, info, and debug.</p>
MessageLog <i>string</i>	<code>-messagelog string</code>	<p>Specifies where messages are logged. The default is <code>\$OMNIHOME/log/probename.log</code>.</p> <p>MessageLog can also be set to stdout or stderr.</p>

Table 23. Common probe properties and command-line options (continued)

Property	Command-line option	Description
Mode <i>string</i>	-master -slave	Specifies the role of the instance of the probe in a peer-to-peer failover relationship. The value of the property can be set to: master: This instance is the master. slave: This instance is the slave. standard: There is no failover relationship. The default is standard.
MsgDailyLog 0 1	-msgdailylog 0 1	Specifies whether daily logging is enabled. By default, the daily backup of log files is not enabled (0). Note: Because the time is checked regularly, when MsgDailyLog is set there is a slight reduction in performance.
MsgTimeLog <i>string</i>	-msgtimelog <i>string</i>	Specifies the time after which the daily log is created. The default is 0000 (midnight). If MsgDailyLog set to 0, this value is ignored.
Name <i>string</i>	-name <i>string</i>	Specifies the name of the probe. This value determines the names of the properties file, rules file, message log file, store-and-forward file, and raw capture file. Note: You can specify alternative file names by using the PropsFile , RulesFile , MessageLog , SAFFilename , and RawCaptureFile properties. If you want to set any of these file names in the properties file, they must be specified after the Name property. Otherwise, the Name property will override any previous setting of the files names.
NetworkTimeout <i>integer</i>	-networktimeout <i>integer</i>	Specifies the length of time (in seconds) that the probe can wait without a response; after this time, the connection to the ObjectServer times out. The maximum value is 2147483, and the default is 0, meaning that no timeout occurs. If a timeout occurs, the probe attempts to connect to the backup ObjectServer, identified by the ServerBackup property. If a timeout occurs and no backup ObjectServer is specified, the probe enters store-and-forward mode. The NetworkTimeout setting overrides the operating system-level TCP/IP timeout setting.

Table 23. Common probe properties and command-line options (continued)

Property	Command-line option	Description
OldTimeStamp TRUE FALSE	-oldtimestamp TRUE FALSE	<p>Specifies the timestamp format to use in the log file.</p> <p>Set the value to TRUE to display the timestamp format that is used in Tivoli Netcool/OMNIBUS V7.2.1, or earlier. For example: dd/MM/YYYY hh:mm:ss AM or dd/MM/YYYY hh:mm:ss PM when the locale is set to en_GB on a Solaris 9 computer.</p> <p>Set the value to FALSE to display the ISO 8601 format in the log file. For example: YYYY-MM-DDThh:mm:ss, where T separates the date and time, and hh is in 24-hour clock. The default is FALSE.</p>
PeerHost <i>string</i>	-peerhost <i>string</i>	Specifies the host name of the network element acting as the counterpart to this probe instance in a peer-to-peer failover relationship. The default is localhost.
PeerPort <i>integer</i>	-peerport <i>integer</i>	Specifies the port through which the master and slave communicate in a peer-to-peer failover relationship. The default port is 99.
PollServer <i>integer</i>	-pollserver <i>integer</i>	<p>If connected to a backup ObjectServer because failover occurred, a probe periodically attempts to reconnect to the primary ObjectServer. This property specifies the frequency in seconds at which the probe polls for the return of the primary ObjectServer. It does this by disconnecting and then reconnecting to the primary ObjectServer if available, or to the secondary ObjectServer if the primary is not available. Polling is the only way that the probe can determine if the primary ObjectServer is available. The default is 0, meaning that no polling occurs.</p> <p>When a probe connects to an ObjectServer, the probe checks the BackupObjectServer property setting of the ObjectServer to which it is connecting. Polling occurs only if this property is set to TRUE, indicating a backup ObjectServer.</p> <p>Note: A probe can go into store-and-forward mode when the primary ObjectServer becomes unavailable. The first alert is not forwarded to the backup ObjectServer until the second alert opens the connection to the backup. If PollServer is set to less than the average time between alerts, the ObjectServer connection is polled before an alert is sent, and the probe does not go into store-and-forward mode. For controlled failback, set PollServer to 0 to disable automatic failback of a probe that is connected to a failover pair of ObjectServers.</p>

Table 23. Common probe properties and command-line options (continued)

Property	Command-line option	Description
ProbeWatchHeartbeatInterval <i>integer</i>	<code>-probewatchheartbeatinterval</code> <i>integer</i>	Generates a ProbeWatch Heartbeat event if this property is set to a positive number. The number defines the interval (in seconds) at which the heartbeats are generated. If set to 0 (zero), or a negative number, no ProbeWatch heartbeats are generated.
Props.CheckNames TRUE FALSE	N/A	When TRUE, the probe does not run if any specified property is invalid. The default is TRUE.
PropsFile <i>string</i>	<code>-propsfile</code> <i>string</i>	Specifies the name of the properties file. The default is \$OMNIBUSHOME/probes/ <i>arch</i> / <i>probename</i> .props, where <i>probename</i> is the name of the probe and <i>arch</i> represents the operating system.
RawCapture 0 1	<code>-raw</code> <code>-noraw</code>	Controls the raw capture mode. Raw capture mode is usually used at the request of IBM Software Support. By default, raw capture mode is disabled (0). Note: Raw capture can generate a large amount of data. By default, the raw capture file can grow indefinitely, although you can limit the size using the MaxRawFileSize property. Raw capture can also slow probe performance due to the amount of disk activity required for a busy probe.
RawCaptureFile <i>string</i>	<code>-capturefile</code> <i>string</i>	Specifies the name of the raw capture file. The default is \$OMNIBUSHOME/var/ <i>probename</i> .cap, where <i>probename</i> is the name of the probe.
RawCaptureFileAppend 0 1	<code>-rawcapfileappend</code> <code>-norawcapfileappend</code>	Specifies whether new data is appended to the existing raw capture file, instead of overwriting it. By default, the file is overwritten (0).
RegexLibrary <i>string</i>	<code>-regexplib</code> <i>string</i>	Defines which regular expression library to use. Possible values are: NETCOOL and TRE. The default value of TRE enables the use of the extended regular expression syntax on single-byte and multi-byte character languages. This setting results in decreased system performance. The NETCOOL value is useful for single-byte character processing and provides optimal system performance.
RetryConnectionCount <i>integer</i>	N/A	Specifies the number of events the probe processes in store-and-forward mode before trying to reconnect to the ObjectServer. The default is 15.
RetryConnectionTimeout <i>integer</i>	N/A	Specifies the number of seconds that the probe processes events in store-and-forward mode before trying to reconnect to the ObjectServer. The default is 30.

Table 23. Common probe properties and command-line options (continued)

Property	Command-line option	Description
RollSAFInterval <i>integer</i>	<code>-rollsafinterval integer</code>	<p>Used when the probe is connected to an ObjectServer, and circular store and forward is enabled by setting StoreAndForward to 2.</p> <p>Specifies the time interval in seconds after which a store-and-forward file is rolled over to the next file in the pool of two files that are used to store a copy of events that are sent to a connected ObjectServer.</p> <p>To minimize event loss during failover and failback, set the time interval to a value that is greater than or equal to the granularity of the ObjectServer. In case of failure, the probe will have a copy of events from the last granularity period, which can be replayed to the backup ObjectServer.</p> <p>The default is 90 seconds, which is 1.5 times greater than the default granularity period of 60 seconds that is set for an ObjectServer.</p>
RulesFile <i>string</i>	<code>-rulesfile string</code>	<p>Specifies the name of the rules file.</p> <p>This can be a file name or Web address that specifies a rules file located on a remote server that is accessible using HTTP.</p> <p>The default is <code>\$OMNIHOME/probes/arch/probenamename.rules</code>, where <i>probenamename</i> is the name of the probe.</p>
SAFFilename <i>string</i>	<code>-saffilename string</code>	<p>Specifies the name of the store-and-forward file.</p> <p>The default is <code>\$OMNIHOME/var/probenamename.store</code>, where <i>probenamename</i> is the name of the probe.</p> <p>A <i>.servername</i> extension is automatically appended to the file name, where <i>servername</i> is the name of the target ObjectServer.</p> <p>A separate store-and-forward file is created for each registered target ObjectServer.</p>
SAFPoolSize <i>integer</i>	<code>-safpoolsize integer</code>	<p>Used when the probe is not connected to an ObjectServer.</p> <p>Specifies the number of store-and-forward files in a pool of files that can be used to store alerts. The default is 3.</p> <p>Each file rolls over to the next when it reaches the maximum size specified by the MaxSAFFileSize property.</p>

Table 23. Common probe properties and command-line options (continued)

Property	Command-line option	Description
SecureLogin 0 1	-securelogin -nosecurelogin	Specifies whether the probe uses an encrypted secure login to access the host system: <ul style="list-style-type: none"> • 0: The probe does not use an encrypted secure login. • 1: The probe uses an encrypted secure login. The default is 0. Note: Secure login is not available in FIPS 140-2 mode. SSL is more secure than secure login.
Server <i>string</i>	-server <i>string</i>	Specifies the name of the primary ObjectServer or the proxy server to which alerts are sent. The default is NCOMS. If you want the probe to operate in circular store-and-forward mode, do not specify a virtual ObjectServer definition as the value of this property.
ServerBackup <i>string</i>	N/A	Specifies the name of a backup ObjectServer to which the probe should connect if the primary ObjectServer connection fails. If NetworkTimeout is set, use ServerBackup to identify a backup ObjectServer. If you want the probe to operate in circular store-and-forward mode, do not specify a virtual ObjectServer definition as the value of this property.
SingleThreadedComms TRUE FALSE	-singlethreadedcomms	Specifies whether multithreaded or single-threaded processing is used to process and send alerts to the target ObjectServers. The default is FALSE, which enables multithreaded communication. You can also use the SingleThreadedComms property to enforce an order for sending alerts to ObjectServers. With multithreaded processing, alerts are simultaneously sent to the target ObjectServers. In single-threaded mode, the order is defined by the order in which the registertarget statements are listed in the rules file.
SSLServerCommonName <i>string1,...</i>	N/A	If the probe is connecting to an ObjectServer using SSL, and the Common Name field of the received certificate does not match the name specified by the Server property, use this property to specify a comma-separated list of acceptable SSL Common Names. The default setting is to use the Server property.

Table 23. Common probe properties and command-line options (continued)

Property	Command-line option	Description
StoreAndForward <i>integer</i>	<code>-saf integer</code>	Controls the store and forward operations. Possible values for the property are: <ul style="list-style-type: none"> • 0: Do not use store and forward. • 1: Use legacy store and forward, which stores alerts in a store-and-forward file only if the alerts cannot be sent to an ObjectServer. • 2: Use circular store and forward, which stores all generated alerts in a rolling pool of store-and-forward files while the probe is connected to an ObjectServer. If the probe is disconnected, the circular store and forward behavior is similar to the legacy store and forward behavior. By default, the legacy store-and-forward mode is enabled (1).
StoreSAFRejects 0 1	<code>-storesafrejects</code> <code>-dontstoresafrejects</code>	Specifies whether the probe should continuously save the individual corrupted store-and-forward records for analysis. If set to 1, corrupted store-and-forward records are continuously saved. The default is 0. Use this property in conjunction with the KeepLastBrokenSAF property.
N/A	<code>-version</code>	Displays version information and exits.

Related concepts

“Probe property versus probe command-line option usage” on page 6

“Probe property types” on page 5

“Store-and-forward mode for probes” on page 10

“Raw capture mode for probes” on page 13

“Secure mode for probes” on page 13

“Peer-to-peer failover mode for probes” on page 14

Related reference

“Multithreaded processing of alert data” on page 46

“Lookup table operations” on page 39

Chapter 6. About gateways

Tivoli Netcool/OMNIBus gateways enable you to exchange alerts between ObjectServers and complementary third-party applications, such as databases and helpdesk or Customer Relationship Management (CRM) systems.

You can use gateways to replicate alerts or to maintain a backup ObjectServer. Application gateways enable you to integrate different business functions. For example, you can configure a gateway to send alert information to a helpdesk system. You can also use a gateway to archive alerts to a database.

The following figure shows an example gateway architecture.

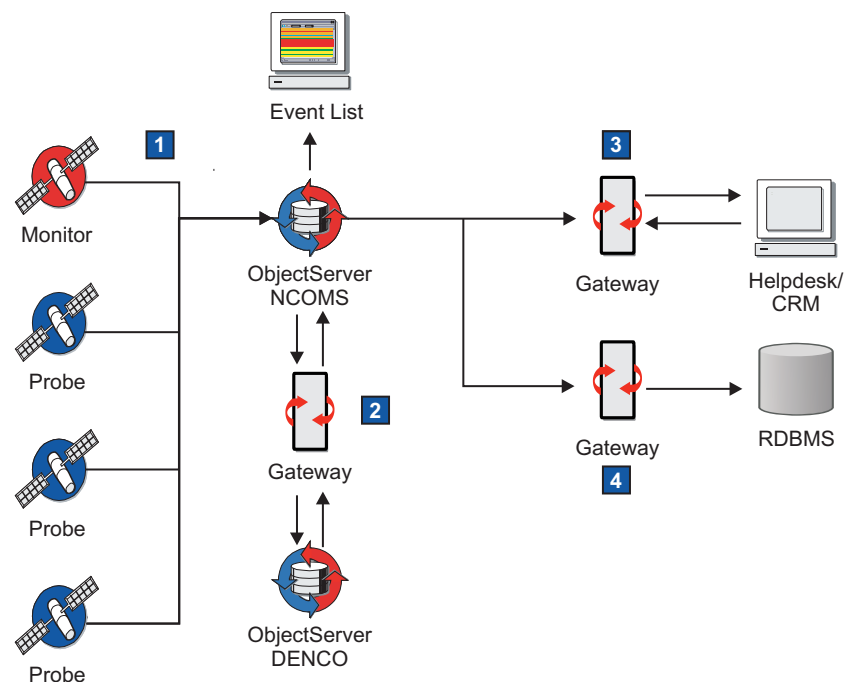


Figure 4. Gateways in the Tivoli Netcool/OMNIBus architecture

The preceding figure illustrates how to use gateways for a variety of purposes:

- 1** Probes send alerts to the local ObjectServer.
- 2** The ObjectServer Gateway replicates alerts between ObjectServers in a failover configuration.
- 3** The Helpdesk gateway integrates the Network Operations Center (NOC) and the helpdesk by converting trouble tickets to alerts, and alerts to trouble tickets.
- 4** The RDBMS gateway stores critical alerts in a relational database management system (RDBMS) so that you can analyze network performance.

After a gateway is correctly installed and configured, the transfer of alerts is transparent to operators. For example, alerts are forwarded from an ObjectServer to a database automatically without user intervention.

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

<http://publib.boulder.ibm.com/infocenter/tivihelp/v8r1/index.jsp>

Related concepts

“Types of gateways”

Types of gateways

There are two main types of gateways: unidirectional gateways and bidirectional gateways.

Unidirectional gateways allow alerts to flow in only one direction. Changes made in the source ObjectServer are replicated in the destination ObjectServer or application, but changes made in the destination ObjectServer or application are not replicated in the source ObjectServer. Unidirectional gateways can be considered as *archiving* tools.

Bidirectional gateways allow alerts to flow from the source ObjectServer to the target ObjectServer or application, and also allow feedback to the source ObjectServer. In a bidirectional gateway configuration, changes made to the contents of a source ObjectServer are replicated in a destination ObjectServer or application, and the destination ObjectServer or application replicates its alerts in the source ObjectServer. Bidirectional gateways can be considered as *synchronization* tools.

Gateways can send alerts to a variety of targets:

- Another ObjectServer
- A database
- A helpdesk application
- Other applications or devices

ObjectServer gateways are used to exchange alerts between ObjectServers. This is useful when you want to create a distributed installation, or when you want to install a backup ObjectServer.

Database gateways are used to store alerts from an ObjectServer. This is useful when you want to keep a historical record of the alerts forwarded to the ObjectServer.

Helpdesk gateways are used to integrate Tivoli Netcool/OMNIBus with a range of helpdesk systems. This is useful when you want to correlate the trouble tickets raised by your customers with the networks and systems you are using to provide their services.

Other gateways are specialized applications that forward ObjectServer alerts to other applications or devices (for example, a flat file or socket).

Note: Only gateways that send alerts to certain targets can be bidirectional.

ObjectServer gateways

ObjectServer gateways can be unidirectional and bidirectional.

Unidirectional ObjectServer Gateway

A unidirectional ObjectServer Gateway allows alerts to flow from a source ObjectServer to a destination ObjectServer. Changes made in the source ObjectServer are replicated in the destination ObjectServer, but changes made in the destination ObjectServer are not replicated in the source ObjectServer.

The following figure shows the configuration of a unidirectional ObjectServer Gateway.

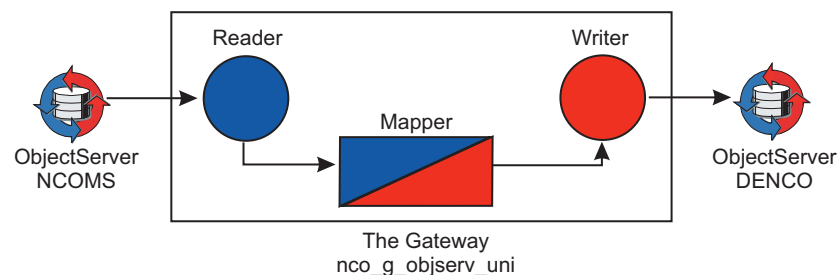


Figure 5. Unidirectional ObjectServer Gateway

In this figure, changes made in the NCOMS ObjectServer are replicated in the DENCO ObjectServer, but changes made in the DENCO ObjectServer are not replicated in the NCOMS ObjectServer.

The unidirectional ObjectServer Gateway is described in detail in the *IBM Tivoli Netcool/OMNibus ObjectServer Gateway Reference Guide*, SC14-7609.

Related concepts

“Bidirectional ObjectServer Gateway”

Bidirectional ObjectServer Gateway

A bidirectional ObjectServer Gateway allows alerts to flow from a source ObjectServer to a destination ObjectServer. Changes made to the contents of a source ObjectServer are replicated in a destination ObjectServer, and the destination ObjectServer replicates its alerts in the source ObjectServer.

This enables you, for example, to maintain a system with two ObjectServers configured as a failover pair.

The following figure shows the configuration of a bidirectional ObjectServer Gateway:

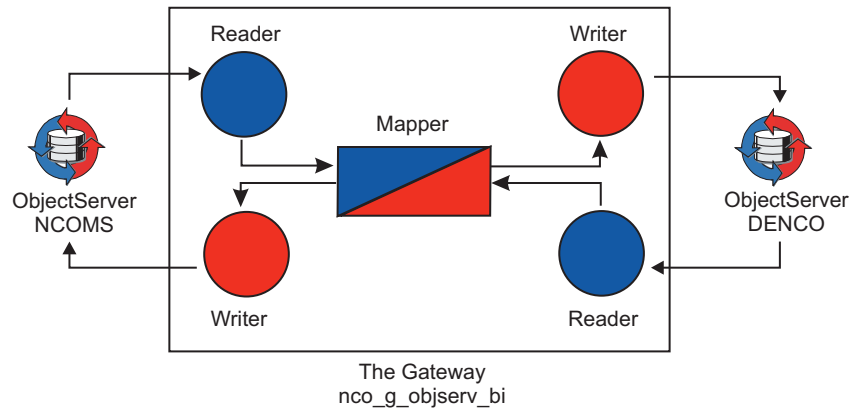


Figure 6. Bidirectional ObjectServer Gateway

In this figure, changes made in the NCOMS ObjectServer are replicated in the DENCO ObjectServer. Changes made in the DENCO ObjectServer are also replicated in the NCOMS ObjectServer.

The bidirectional ObjectServer Gateway is described in detail in the *IBM Tivoli Netcool/OMNIBus ObjectServer Gateway Reference Guide*, SC14-7609.

Related concepts

“Unidirectional ObjectServer Gateway” on page 91

ObjectServer Gateway writers and failback (alert replication between sites)

Failover occurs when a gateway loses its connection to the primary ObjectServer; this allows the gateway to connect to a backup ObjectServer. Failback functionality allows the gateway to reconnect to the primary ObjectServer when it becomes active again.

Because bidirectional ObjectServer gateways are used to resynchronize failover pairs, failback is automatically disabled. This is because one half of the gateway can legitimately be connected to a backup server and so should not be forced to keep failing back to the primary ObjectServer.

However, if a bidirectional gateway is being used to share data between two separate sites, and each site has a failover pair operating, you can manually enable failback on each server. When enabled, the writer automatically enables failback on its counterpart reader.

ObjectServer gateway failover and failback are described in detail in the *IBM Tivoli Netcool/OMNIBus ObjectServer Gateway Reference Guide*, SC14-7609.

Database, helpdesk, and other gateways

Most database, helpdesk, and other gateways use a standard architecture, but each gateway has its own binary file, with additional modules to handle the communication with the target applications, devices, or files.

For information about specific gateways and their architectures, see the individual gateway publications.

Gateway components

Gateways have *reader* and *writer* components. Readers extract alerts from the ObjectServer. Writers forward alerts to another ObjectServer or to other applications.

There is only one type of reader, but there are various types of writers depending on the destination application.

Routes specify the destination to which a reader forwards alerts. One reader can have multiple routes to different writers, and one writer can have multiple routes from different readers.

Gateway *filters* and *mappings* configure alert flow. Filters define the types of alerts that can be passed through a gateway. Mappings define the format of these alerts.

Readers, writers, routes, filters, and mappings are defined in the gateway configuration file.

Related concepts

“Gateway configuration” on page 98

Unidirectional gateways

A unidirectional database, helpdesk, or other gateway allows alerts to flow from a source ObjectServer to a destination application. Changes made in the source ObjectServer are replicated in the destination application, but changes made in the destination application are not replicated in the source ObjectServer.

A simple example of a unidirectional gateway is the Flat File Gateway, which reads alerts from an ObjectServer and writes them to a flat file. This example architecture is shown in the following figure.

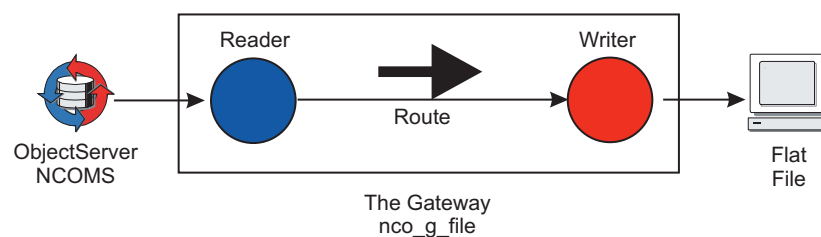


Figure 7. Example Flat File Gateway architecture

Related concepts

“Bidirectional gateways” on page 94

Bidirectional gateways

In a bidirectional database, helpdesk, or other gateway configuration, changes made to the alerts in a source ObjectServer are replicated in a destination application, and the destination application replicates changes to its alerts in the source ObjectServer.

This enables you, for example, to raise trouble tickets in a helpdesk system for certain alerts. Changes made to the tickets in the helpdesk system can then be sent back to the ObjectServer.

Bidirectional gateways have a similar configuration to unidirectional gateways, with an additional COUNTERPART attribute for the writers. The COUNTERPART attribute defines a link between a gateway writer and reader.

The following figure shows an example bidirectional gateway configuration.

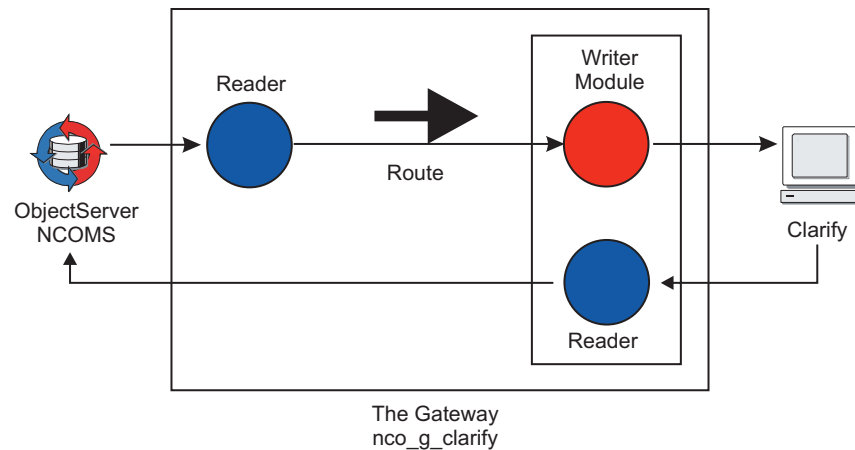


Figure 8. Bidirectional Clarify Gateway

Related concepts

“Unidirectional gateways” on page 93

Reader component

A reader extracts alerts from an ObjectServer. There is only one type of reader: the ObjectServer reader.

When the reader starts, the gateway attempts to open a connection to the source ObjectServer. If the gateway succeeds in opening the connection, it immediately starts to read alerts from the ObjectServer.

Writer modules

Writer modules manage communications between gateways and third-party applications, and format the alert correctly for entry into the application.

The writer module generates log files that can help debug the gateway.

Communication between the writer module and the third-party application uses helper applications, which interact directly with the application through its APIs or other interfaces. These processes are transparent to the user (though they are visible using the **ps** command or similar utility).

The writer module uses a reference number cache to track the alerts and their associated reference number in the target application. For each alert, the cache stores the following:

- The serial number of the alert
- A reference number from the target application (for example, Clarify Cases or ServiceCenter Tickets)

When a ticket is raised in response to an alert, the writer module enters the reference number in the cache and returns it to the ObjectServer where the alert is updated to include the reference number.

The following figure shows a simplified example of the writer module architecture.

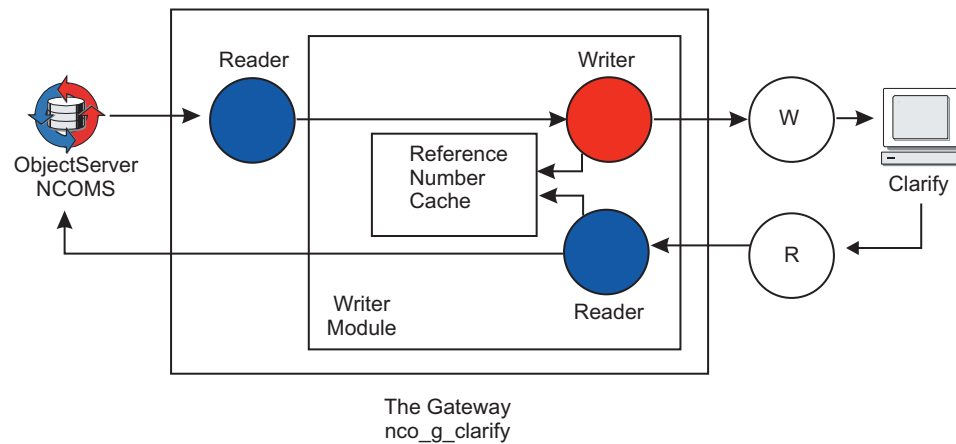


Figure 9. Reader/writer module architecture

Related reference

“Gateway debugging” on page 102

Routes

Routes create the link between the source reader and the destination writer.

Any alerts received by the source ObjectServer are read by the reader, passed through the route to the writer, and written into the destination ObjectServer or application.

Alert updates from the helpdesk

When a helpdesk operator makes additional changes to a ticket, these are forwarded to the gateway which runs the corresponding action .sql file to update the alert in the ObjectServer.

Typically the following action .sql files are provided:

- open.sql
- update.sql
- journal.sql
- close.sql

For detailed information on configuring alert updates from the helpdesk, see the individual gateway publications.

Modes of operation of gateways

Gateways can operate in store-and-forward mode and secure mode.

Store-and-forward mode for gateways

If there is a problem with the gateway target, the ObjectServer and database writers can continue to run using store-and-forward mode.

When the writer detects that the target ObjectServer or database is not present or is not functioning (usually because the writer is unable to write an alert), it switches into *store* mode. In this mode, the writer stores everything it would normally send to the database in a file named:

```
$OMNIHOME/var/writername.destserver.store
```

In this file name, *writername* is the name of the writer and *destserver* is the name of the server to which the gateway is attempting to send alerts.

When the gateway detects that the destination server is back on line, it switches into *forward* mode and sends the alert information held in the *.store* file to the destination server. After all of the alerts in the *.store* file have been forwarded, the writer returns to normal operation.

Store-and-forward mode only works when a connection to the ObjectServer or database destination has been established, used, and then lost. If the destination server is not running when the gateway starts, store-and-forward mode is not triggered and the gateway terminates.

If the gateway connects to the destination ObjectServer and a store-and-forward file already exists, the gateway replays the contents of the store-and-forward file before it sends new alerts.

Store-and-forward mode is configured using the attributes `STORE_AND_FORWARD` and `STORE_FILE`.

Note: See the individual gateway publications to determine whether an individual gateway supports store-and-forward mode. Store and forward does not work with bidirectional gateway configurations, with the exception of the bidirectional ObjectServer Gateway.

Secure mode for gateways

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

Note: This section applies to gateways that use configuration, rather than properties, files. If you require information about running the ObjectServer Gateway in secure mode and FIPS 140-2 mode, see the *IBM Tivoli Netcool/OMNIBus Installation and Deployment Guide* and the *IBM Tivoli Netcool/OMNIBus ObjectServer Gateway Reference Guide*, SC14-7609.

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

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

When connecting to a secure ObjectServer, the gateway must have the AUTH_USER and AUTH_PASSWORD commands in the gateway configuration file. You can choose any valid user name for the AUTH_USER gateway command. To generate the encrypted AUTH_PASSWORD, use the nco_g_crypt utility. The command takes the unencrypted password and displays the encrypted password to be entered for the AUTH_PASSWORD command.

The AUTH_USER and AUTH_PASSWORD commands must precede any reader commands in the gateway configuration file. Before running the gateway, add the user name and corresponding encrypted password to the configuration file, for example:

```
AUTH_USER 'Gate_User'  
AUTH_PASSWORD 'Crypt_Password'
```

Note: For Tivoli Netcool/OMNIBus V7.1, or later, you must set the gateway properties for the user name and password when connecting to a secure ObjectServer from the ObjectServer Gateway. For information about ObjectServer gateways, see the *IBM Tivoli Netcool/OMNIBus ObjectServer Gateway Reference Guide*, SC14-7609.

Encrypting target system passwords

You can use the **nco_g_crypt** utility to encrypt plain text login passwords (this method uses DES encryption). The gateways use these encrypted passwords to log into their target systems. Encrypted passwords are decoded by the gateway before they are used to log in to the target system.

Note: You cannot use nco_g_crypt in FIPS 140-2 mode. Use nco_aes_crypt (specifying the algorithm AES_FIPS) to protect the passwords specified in a properties file. Use operating system protections to protect both the properties file and the encryption key. Use SSL to protect passwords and other data over the network.

The user name and password are stored in the USERNAME and PASSWORD attributes in the gateway writer.

Note: If you are using a helpdesk gateway, substitute USER for USERNAME.

To encrypt a plain text password for a gateway target system:

1. Use the **nco_g_crypt** utility to obtain an encrypted version of the password.
2. Update the gateway writer in the gateway configuration file by copying the user name into the USERNAME attribute value and the encrypted password created in step 1 into the PASSWORD attribute value.

For example:

```
START WRITER SYBASE_WRITER  
(  
  TYPE = SYBASE,  
  REVISION = 1,  
  SERVER = DARKSTAR,  
  MAP = SYBASE_MAP,  
  USER = 'SYSTEM',  
  PASSWORD = 'MKFGHIFE',  
  FORWARD_DELETES = TRUE  
);
```

3. Run the gateway.

Related concepts

“Gateway configuration”

Chapter 7, “Running gateways,” on page 105

Gateway configuration

The configuration file, or files, define the environment in which the gateway operates and how the gateway maps data onto tables within the ObjectServer.

Note: Most gateways are configured using a single configuration file with the .conf extension. The following gateways are configured using a properties file which replaces this configuration file:

- ObjectServer Gateway
- ODBC Gateway
- Gateway for Oracle

For information about configuring these gateways, see the *IBM Tivoli Netcool/OMNIbus ObjectServer Gateway Reference Guide, SC14-7609*, *IBM Tivoli Netcool/OMNIbus ODBC Gateway Guide, SC23-9572*, and *IBM Tivoli Netcool/OMNIbus Gateway for Oracle Guide, SC23-7669*, respectively.

Related concepts

“Gateway components” on page 93

Chapter 7, “Running gateways,” on page 105

Related tasks

“Encrypting target system passwords” on page 97

Gateway configuration file

Most gateways have a configuration file, with the .conf extension. When a gateway of this type starts, it processes the commands in its configuration file. This defines the connections between the source ObjectServer and the alert destinations.

For example, by default, the Gateway for Clarify uses the following configuration file:

```
$OMNIHOME/etc/G_CLARIFY.conf
```

Before running a gateway, you must add the relevant commands to the configuration file to specify the reader, writer, route, mapping, and filter definitions.

When running a gateway, you do not need to specify the configuration file if you are using the default file that resides in the \$OMNIHOME/etc location. If you want to use a different configuration file, you can use the -config command-line option to specify the full path and name of the alternative configuration file. For example:

```
$OMNIHOME/bin/nco_g_clarify -config /path/G_CLARIFY.conf
```

Where *path* is the full path and name of the alternative configuration file.

Note: The ObjectServer gateway does not use a .conf configuration file. Instead, it uses a properties file and a number of other configuration files. For information

about ObjectServer gateway configuration, see the *IBM Tivoli Netcool/OMNIBus ObjectServer Gateway Reference Guide*, SC14-7609.

Related concepts

Chapter 7, “Running gateways,” on page 105

Related reference

“Filter commands” on page 116

“Mapping commands” on page 115

“Reader commands” on page 111

“Route commands” on page 117

“Writer commands” on page 112

Reader configuration

A reader extracts alerts from an ObjectServer. Readers are started using the START READER command, which defines the name of the reader and the name of the ObjectServer from which to read.

For example, to start a reader for an NCOMS ObjectServer, add the following command to the configuration file:

```
START READER NCOMS_READ CONNECT TO NCOMS;
```

After this command is issued, the reader starts and the gateway attempts to open a connection to the source ObjectServer. If the gateway succeeds in opening the connection, it immediately starts to read alerts from the ObjectServer. For the reader to forward these alerts to their destination, you must define an associated route and writer.

Related reference

“Reader commands” on page 111

Writer configuration

Writers send the alerts acquired by a reader to the destination application or ObjectServer. Writers are created using the START WRITER command, which defines the name of the writer and the information that allows it to connect to its destination.

For example, to create the writer for a Flat File Gateway, add the following command to the configuration file:

```
START WRITER FILE_WRITER
(
    TYPE = FILE,
    REVISION = 1,
    FILE = '/tmp/omnibus/log/NCOMS_alert.log',
    MAP = FILE_MAP,
    INSERT_HEADER = 'INSERT: ',
    UPDATE_HEADER = 'UPDATE: ',
    DELETE_HEADER = 'DELETE: ',
    START_STRING = '',
    END_STRING = '',
    INSERT_TRAILER = '\n',
    UPDATE_TRAILER = '\n',
    DELETE_TRAILER = '\n'
);
```

After the START WRITER command is issued, the gateway attempts to establish the connection to the alert destination (either an application or another ObjectServer). The writer sends alerts received from the source ObjectServer until the STOP WRITER command is issued.

Related reference

“Writer commands” on page 112

Route configuration

Routes create the link between readers and writers. Routes are created using the ADD ROUTE command. This command defines the name of the route, the source reader, and the destination writer.

For example, to create the route between the NCOMS ObjectServer reader and the writer for a Flat File Gateway, add the following command to the configuration file:

```
ADD ROUTE FROM NCOMS_READ TO FILE_WRITER;
```

After this command is issued, the connection between a reader and writer is established. Any alerts received by the source ObjectServer are read by the reader, passed through the route to the writer, and written into the destination ObjectServer or application.

Related reference

“Route commands” on page 117

Mapping configuration

Mappings define how alerts received from the source ObjectServer should be written to the destination ObjectServer or application. Each writer has a different mapping that is defined using the CREATE MAPPING command.

For example, to create the mapping between the ObjectServer reader and the writer for a Flat File Gateway, add the following command to the configuration file:

```
CREATE MAPPING FILE_MAP
(
  '@Identifier',
  '@Serial',
  '@Node',
  '@Manager',
  '@FirstOccurrence' CONVERT TO DATE,
  '@LastOccurrence'   CONVERT TO DATE,
  '@InternalLast'     CONVERT TO DATE,
  '@Tally',
  '@Class',
  '@Grade',
  '@Location',
  '@ServerName',
  '@ServerSerial'
);
```

In this example, the mapping name is FILE_MAP.

Each line between the parentheses defines how the gateway writes alerts into the file. For the Flat File Gateway, the CREATE MAPPING command defines the fields from which data is written into each alert in the output file. The alert fields from the source ObjectServer are represented by the @ symbol.

The following example shows INSERT and UPDATE commands using the FILE_MAP mapping shown in the preceding example.

```
INSERT: "Downlink6LinkMon4Link",127,"sfo4397","Netcool Probe",12/05/03 15:39:23,
12/05/03 15:39:23,12/05/03 15:30:53,1,3300,0,"","NCOMS",127
UPDATE: "muppetMachineMon2Systems",104,"sfo4397","Netcool Probe",12/05/03 12:29:34,
12/05/03 15:40:06,12/05/03 15:31:36,11,3300,0,"","NCOMS",104
UPDATE: "muppetMachineMon4Systems",93,"sfo4397","Netcool Probe",12/05/03 12:29:11,
12/05/03 15:40:35,12/05/03 15:32:05,12,3300,0,"","NCOMS",93
```

Other gateways (with the exception of the Socket Writer Gateway) require a field in the target to be specified for each source ObjectServer field. For example, in the Gateway for Remedy ARS, source ObjectServer fields are mapped to Remedy ARS fields, which are identified with long integer values rather than field names. In the following example, the ARS field 536870913 maps to the Serial field from the ObjectServer:

```
536870913 = '@Serial' ON INSERT ONLY
```

The ON INSERT ONLY clause controls when the field is updated. Fields with the ON INSERT ONLY clause are forwarded only once, when the alert is created for the first time in the ObjectServer. Fields that do not have the ON INSERT ONLY clause are updated each time the alert changes.

Related reference

“CREATE MAPPING” on page 115

Filter configuration

You might not always want to send all of the alerts that are read by a reader to the destination application. Filters define which of the alerts read by the ObjectServer reader should be forwarded to the destination.

For example, you may want to send only alerts that have a severity level of Critical.

You create filters using the CREATE FILTER command and apply them using the START READER command. For example, to create a filter that forwards only critical alerts to the destination application or ObjectServer, add the following command to the configuration file:

```
CREATE FILTER CRITONLY AS 'Severity = 5';
```

This command creates a filter named CRITONLY, which forwards alerts with a severity level of Critical (5) only.

To apply the filter to an ObjectServer reader, add the following command to the configuration file:

```
START READER NCOMS_READ CONNECT TO NCOMS USING FILTER CRITONLY;
```

Note: To perform string comparisons with filters, you must escape the quotes in the CREATE FILTER command with backslashes. For example, to create a filter that forwards only alerts from a node called fred, the CREATE FILTER command is:

```
CREATE FILTER FREDONLY AS 'NODE = \'fred\'';
```

Creating multiple filters and multiple readers

If you need more than one filter for the same ObjectServer, you can create multiple readers for it.

For example, to create a reader that forwards all critical alerts and another that forwards everything else, use the following commands:

```
CREATE FILTER CRITONLY AS 'Severity = 5';  
CREATE FILTER NONCRIT AS 'Severity < 5';  
START READER CRIT_NCOMS CONNECT TO NCOMS USING FILTER CRITONLY;  
START READER NONCRIT_NCOMS CONNECT TO NCOMS USING FILTER NONCRIT;
```

Loading filters created using the Filter Builder

You can load and use filters that are created in the Filter Builder.

For example:

```
LOAD FILTER FROM '/usr/filters/myfilt.elf';
```

This command loads the file `/usr/filters/myfilt.elf` as a filter. This filter name is defined by the Filter Builder **Name** field.

Note: The **Name** field must be alphabetical and must not contain spaces.

Related reference

“Filter commands” on page 116

“START READER” on page 111

Gateway debugging

When debugging, you should initially check the log file:

```
$OMNIHOME/log/NCO_GATENAME.log
```

Where *GATENAME* is the name of the gateway.

You might receive an error message such as the following:

```
error in srv_select () - file descriptor x is no longer active!
```

This type of error message indicates that the gateway has aborted because one of the reader or writer modules failed. In this case, check the following log files:

```
NCO_GATENAME_XRWY_WRITE.log
```

or

```
NCO_GATENAME_XRWY_READ.log
```

Where *X* identifies the name of the gateway and *Y* identifies the version of that gateway.

Related concepts

“Writer modules” on page 94

Gateway writers and failback

The ObjectServer reader can fail over and fail back between source ObjectServers without shutting down. This ability is not supported by all gateway writers.

If a writer does not support this mode of failback and failover, the writer, on detection of the reader failover or failback, will shut down the gateway and rely on the process agent to restart the gateway.

Writers that support reader failover and failback without shutting down are:

- ObjectServer writer
- Sybase database writer
- Sybase Reporter writer
- SNMP writer
- ServiceView writer
- Socket writer
- Flat file writer
- Informix® database writer

Writers that support failover and failback with shutdown are:

- Remedy ARS writer
- Siebel eCommunications writer
- Oracle database writer
- Oracle Reporter writer
- Peregrine writer
- Clarify writer
- HP ITSM writer
- Peoplesoft Vantive writer
- HP Service Desk writer
- ODBC database writer

Creating conversion tables

You can create conversion tables to enable certain data conversions to take place between fields.

For example, if you are using the Gateway for Remedy, you can create a table within the ObjectServer to insert values for the Severity field found in Remedy.

To do this, you must use ObjectServer SQL commands. You can run ObjectServer SQL commands using the following tools:

- UNIX SQL interactive interface (**nco_sql**)
- Windows SQL interactive interface (**isql**)
- Netcool/OMNIbus Administrator

Conversion table on Tivoli Netcool/OMNIbus V7.0, or later

The following example ObjectServer SQL creates the table `remedy` in a Tivoli Netcool/OMNIbus V7.0, or later, ObjectServer, and inserts six values and corresponding descriptions for the Severity field:

```

create database conversions;
use database conversions;
create table conversions.remedy persistent (
  KeyField varchar(255) primary key,
  Colname  varchar(255),
  OSValue  varchar(255),
  Conversion varchar(255)
);
go

insert into conversions.remedy values ('Severity0','Severity','0','Clear');
insert into conversions.remedy values ('Severity1','Severity','1','Indeterminate');
insert into conversions.remedy values ('Severity2','Severity','2','Warning');
insert into conversions.remedy values ('Severity3','Severity','3','Minor');
insert into conversions.remedy values ('Severity4','Severity','4','Major');
insert into conversions.remedy values ('Severity5','Severity','5','Critical');
go

```

Chapter 7. Running gateways

A gateway requires an entry in the Server Editor. You must also create your gateway configuration file. After you have defined the gateway communications and created your configuration file, you can run the gateway.

Related concepts

“Gateway configuration file” on page 98

“Gateway configuration” on page 98

Related tasks

“Encrypting target system passwords” on page 97

Running gateways on UNIX

On UNIX, you can run gateways from the command line.

To run a gateway with a default configuration, enter:

```
$OMNIHOME/bin/nco_g_gatename
```

This runs a gateway with the default name *GATENAME* and the default configuration file *\$OMNIHOME/etc/G_GATENAME.conf*.

To run a gateway with a different name and configuration file, use command-line options. For example:

```
$OMNIHOME/bin/nco_g_gatename -name GATE2 -config $OMNIHOME/etc/GATE2.conf
```

This runs a gateway named *GATE2* using a configuration file named *GATE2.conf*.

You must configure gateways to run under process control.

Related reference

“Common gateway command-line options” on page 109

Running gateways on Windows

Gateways on Windows can be run as console applications or as services.

Running a gateway as a console application

You run a gateway as a console application from the command line.

To run a gateway as a console application, enter the following command:

```
%OMNIHOME%\bin\nco_g_gatename
```

This runs a gateway with the default name *GATENAME* and the default configuration file *%OMNIHOME%\etc\G_GATENAME.conf*.

To run a gateway as a console application with your own configuration, use the command-line options. For example:

```
%OMNIHOME%\bin\ncg_gatename -name GATE2 -config %OMNIHOME%\etc\GATE2.conf
```

This runs a gateway named GATE2 using a configuration file named GATE2.conf.

Related reference

“Common gateway command-line options” on page 109

Running a gateway as a service

To run a gateway as a service, use the `-install` command-line option.

You can configure how gateways are started by changing the Windows services settings as follows:

1. Click **Start** > **Control Panel**. The Control Panel opens.
2. Double-click the **Admin Tools** icon, then double-click the **Services** icon. The Services window opens.
The Services window lists all of the Windows services currently installed on your machine. All Tivoli Netcool/OMNIBus services start with **NCO**.
3. Use the Services window to start and stop Windows services. Define whether the service is started automatically when the machine is booted by clicking the **Startup** button.

Configuring gateways interactively

You can change the configuration of a gateway while it is running by using the SQL interactive interface (`ncg_sql`).

Note: To connect to a gateway on UNIX using `ncg_sql`, you must specify the user name and password of a member of the UNIX user group that is allowed to log into a gateway. This user group is specified using the `-admingroup` command-line option. By default, this is the `ncadmin` user group. You might need to ask your system administrator to create this group. Also, the user running the gateway must have access to the appropriate file that is used to verify passwords so that the members of `ncadmin` can be authenticated when logging into the gateway using `ncg_sql`.

Use the SQL interactive interface to connect to a gateway as a specific user, as shown in the following table.

Table 24. Connecting to the gateway using the SQL interactive interface

On	Enter the following command
UNIX	<code>\$OMNIHOME/bin/ncg_sql -server servername -user username</code>
Windows	<code>%OMNIHOME%\bin\redist\isql -S servername -U username</code>

In these commands, *servername* is the name of the gateway and *username* is a valid user name. If you do not specify a user name, the default is the user running the command.

You are prompted to enter a password. On UNIX, the default is to enter your UNIX password. To authenticate users using other methods, use the `-authenticate` command-line option.

After connecting with a user name and password, a numbered prompt is displayed.

1>

You can enter commands to configure the gateway dynamically. The following example shows a session in which new routes are added:

```
$ nco_sql -server REMEDY
Password:
User 'admin' logged in.
1> ADD ROUTE FROM Denco_READ TO ARS_WRITER;
2> ADD ROUTE FROM Denco_READ TO OS_WRITER;
3> go
```

1>

If you want to disable interactive configuration, add the following line to the end of the gateway configuration file:

```
SET CONNECTIONS FALSE;
```

Related reference

“Common gateway command-line options” on page 109

Saving configurations interactively

You can save the interactive gateway configuration with the command:

```
SAVE CONFIG TO 'filename';
```

In this command, *filename* is the name of a file on a local file system.

You can then use the saved configuration file for other gateways.

Dumping and loading gateway configurations interactively

You can load gateway configurations interactively.

First stop any running readers and writers manually with the STOP command. Then use the DUMP CONFIG command to discard the current configuration.

The DUMP CONFIG command will not discard the configuration if any readers and writers are running or if the configuration has been changed interactively, unless you use the FORCE option. To determine if the configuration has been changed interactively, use the SHOW SYSTEM command.

After you have dumped the configuration, you can load a new configuration with the command:

```
LOAD CONFIG FROM 'filename';
```

In this command, *filename* is the name of a file on a local file system.

Related reference

“SHOW SYSTEM” on page 119

“Configuration commands” on page 118

Use of OMNIHOME and NCHOME environment variables for gateways

On Netcool/OMNIbus V7.0, or earlier, the UNIX environment variable \$OMNIHOME (%OMNIHOME% on Windows) is used in many configuration files. Netcool/OMNIbus V7.1, or later, uses the UNIX environment variable \$NCHOME (%NCHOME% on Windows) instead.

Configuration files containing the OMNIHOME environment variable will work on Netcool/OMNIbus V7.1, or later, as long as you set \$OMNIHOME to \$NCHOME/omnibus on UNIX, or set %OMNIHOME% to %NCHOME%\omnibus on Windows.

Chapter 8. Gateway commands and command-line options

A number of gateway commands and command-line options are common to all gateways configured using a single configuration file with the .conf extension.

Resynchronization commands are valid only for the ObjectServer Gateway. These commands are described in the *IBM Tivoli Netcool/OMNIBus ObjectServer Gateway Reference Guide*.

Additional information about specific gateways is available in the publication for each gateway.

Note: The commands and command-line options described in this chapter do not apply to the following gateways:

- ObjectServer Gateway
- ODBC Gateway
- Gateway for Oracle

For information about configuring these gateways, see the *IBM Tivoli Netcool/OMNIBus ObjectServer Gateway Reference Guide*, *IBM Tivoli Netcool/OMNIBus ODBC Gateway Guide*, and *IBM Tivoli Netcool/OMNIBus Gateway for Oracle Guide*, respectively.

Common gateway command-line options

A number of command-line options are common to all gateways.

For the command-line options that are specific to a particular gateway, see the individual publications for each gateway.

The following table lists the command-line options that are common to all gateways, and provides the default settings.

Table 25. Common gateway command-line options

Command-line option	Description
-admingroup <i>string</i>	Specifies the name of the UNIX user group that has administrator privileges. Members of this group can log into the gateway. The default group name is ncoadmin.
-authenticate UNIX PAM HPTCB	<p>Specifies the authentication mode to use to verify user credentials. The options are UNIX, PAM, and HPTCB.</p> <p>The default authentication mode is UNIX, which means that the Posix getpwnam or getswnam function is used to verify user credentials on UNIX operating systems. Depending on system setup, passwords are verified using the /etc/passwd file, the /etc/shadow shadow password file, NIS, or NIS+.</p> <p>If PAM is specified as the authentication mode, Pluggable Authentication Modules are used to verify user credentials. The service name used by the gateway when the PAM interface is initialized is netcool. PAM authentication is available on Linux, Solaris, and HP-UX 11 operating systems only.</p> <p>If HPTCB is specified as the authentication mode, this HP-UX password protection system is used. This option is only available on HP trusted (secure) systems.</p>

Table 25. Common gateway command-line options (continued)

Command-line option	Description
-config <i>string</i>	Specifies the name of the configuration file to be read when the gateway starts. The default is \$OMNIHOME/etc/ <i>gatename</i> .conf.
-connections	The number of permitted connections. The default is 30.
-debug	When specified, debug mode is enabled.
-help	Displays help information about the command-line options and exits.
-ipctimeout	IPC Session timeout. The default is 60 seconds.
-logfile <i>string</i>	Specifies the name of the log file. If omitted, the default is \$OMNIHOME/log/ <i>gatename</i> .log.
-logsize <i>integer</i>	Specifies the maximum size of the log file in KB. The minimum is 16 KB. The default is 1 MB.
-messagelevel	The level of messages to be logged. The default is warn: <ul style="list-style-type: none"> • debug • info • warn • error • fatal
-messagelog	The path to the message log file. The default is: /export/build/nco/Developers/jlawder/OMNIbii/Solaris731/omnibus/log/NCO_GATE.log
-name <i>string</i>	Specifies the gateway name. Specify this name following the -server command-line option to connect to the gateway using nco_sql . If omitted, the default is <i>GATENAME</i> .
-notruncate	Specifies that the log file is not truncated.
-oldtimestamp	
-propsfile	
-queue <i>integer</i>	Specifies the size of the internal queues. The default is 1024. Do not modify unless advised by IBM Software Support.
-stacksize <i>integer</i>	Specifies the size of the internal threads. The default is 256 KB. Do not modify unless advised by IBM Software Support.
-uniqueolog	If -logfile is not set, this option forces the log file to be uniquely named by appending the process ID of the gateway to the end of the default log file name. If -logfile is set, this has no effect.
-version	Displays version information and exits.

Reader commands

A number of reader commands are available for gateways.

Related concepts

“Gateway configuration file” on page 98

“Reader configuration” on page 99

START READER

Use the START READER command to start a reader named *reader_name* that connects to an ObjectServer named *server_name*.

Syntax

```
START READER reader_name CONNECT TO server_name [ USING FILTER filter_name ]  
[ ORDER BY 'column,... [ ASC | DESC ]' ] [ AFTER IDUC DO 'update_command' ]  
[ IDUC = integer ] [ JOURNAL_FLUSH = integer ] [ IDUC_ORDER ];
```

The optional USING FILTER clause, followed by the name of a filter that has been created using the CREATE FILTER command, enables you to restrict the number of rows affected by gateway updates. The filter replaces an SQL WHERE clause, so the gateway only updates the rows selected by the filter.

The optional ORDER BY clause instructs the gateway to display the results in sequential order, depending on the values of one or more column names, in either descending (DESC) or ascending (ASC) order. If the ORDER BY clause is not specified, no ordering is used.

The optional AFTER IDUC clause instructs the gateway to perform the update specified in the *update_command* in the ObjectServer when it places alerts in the writer queue. This is used to provide feedback when alerts pass through a gateway.

Note: The update command that follows an AFTER IDUC DO statement in the START READER command must be a simple UPDATE statement. It must not use conditions (for example, WHERE or HAVING); these are not supported in this context.

The value specified in the optional IDUC clause indicates an IDUC interval for gateways that is more frequent than the value of the **Granularity** property set in the source ObjectServer. This enables gateway updates to be forwarded to the target more rapidly without causing overall system performance to deteriorate.

The value specified in the optional JOURNAL_FLUSH clause indicates a delay in seconds between when the IDUC update occurs in the ObjectServer (every *Granularity* seconds) and when the journal entries are retrieved by the gateway. Normally, only journal entries that have been made in the last *Granularity* seconds are retrieved. When the system is under heavy load, set this clause so journal entries are retrieved for the last *integer* + *Granularity* seconds. This prevents the loss of any journal entries that are created after the IDUC update but before the gateway retrieves the entries. Any duplicate journal entries retrieved are eliminated by deduplication.

The optional IDUC_ORDER clause specifies the order in which the IDUC data is processed. The default processing mode for gateways is to process DELETE statements, followed by UPDATE statements, followed by INSERT statements. Do

not change this clause unless you have been advised to do so by IBM Software Support.

Example

This example uses the Grade field as a state field. Initially, all probes set Grade to 0. The gateway filters any alerts that have a Grade of 1. After the alerts have passed through the gateway, the AFTER IDUC update provides alert state feedback by changing the value of the Grade field to 2.

```
START READER NCOMS_READER CONNECT TO NCOMS USING FILTER CRIT_ONLY  
ORDER BY 'SERIAL ASC' AFTER IDUC DO 'update alerts.status set Grade=2';
```

Related concepts

“Filter configuration” on page 101

STOP READER

Use the STOP READER command to stop the reader named *reader_name*.

Syntax

```
STOP READER reader_name;
```

This command does not stop the reader if the reader is in use with any routes.

Example

```
STOP READER NCOMS_READ;
```

SHOW READERS

Use the SHOW READERS command to list all the current readers that have been started, and which are running on the gateway.

Syntax

```
SHOW READERS;
```

This command can only be used interactively.

Example

```
SHOW READERS;
```

Writer commands

A number of writer commands are available for gateways.

Related concepts

“Gateway configuration file” on page 98

“Writer configuration” on page 99

START WRITER

Use the START WRITER command to start a writer named *writer_name*.

Syntax

```
START WRITER writer_name
( TYPE=writer_type , REVISION=number
[ , keyword_setting [ , keyword_setting ] ...] );
```

The START WRITER command is followed by a list of comma-separated keyword settings in parentheses. The first setting must be a TYPE setting indicating the *writer_type*. The next setting must be a REVISION setting. This is currently set to 1 for all writers. The remaining keywords and their settings depend on the type of writer.

Example

This example starts the writer for a Socket Writer Gateway.

```
START WRITER SOCKET_WRITER
(
    TYPE = SOCKET,
    REVISION = 1,
    HOST = 'sfo768',
    PORT = 4010,
    MAP = SOCKET_MAP,
    INSERT_HEADER = 'INSERT: ',
    UPDATE_HEADER = 'UPDATE: ',
    DELETE_HEADER = 'DELETE: ',
    START_STRING = '',
    END_STRING = '',
    INSERT_TRAILER = '\n',
    UPDATE_TRAILER = '\n',
    DELETE_TRAILER = '\n'
);
```

STOP WRITER

Use the STOP WRITER command to stop the writer called *writer_name*.

Syntax

```
STOP WRITER writer_name;
```

If any route is using this writer, the writer does not stop.

Example

```
STOP WRITER ARS_WRITER;
```

SHOW WRITERS

Use the SHOW WRITERS command to list all the current writers in the gateway.

Syntax

```
SHOW WRITERS;
```

This command can only be used interactively.

Example

```
1>SHOW WRITERS;
2>GO
Name           Type Routes Msgq Id Mutex Id Thread
```

```
-----
SNMP_WRITER SNMP 1      15      0      0x001b8cd0
```

```
1>
```

SHOW WRITER TYPES

Use the SHOW WRITER TYPES command to list all the currently known types of writers that are supported by the gateway.

Syntax

```
SHOW WRITER TYPES;
```

This command can only be used interactively.

Example

```
1> SHOW WRITER TYPES;
```

```
2> GO
```

Type	Revision	Description
-----	-----	-----
ARS	1	Action Request System V3.0
OBJECT_SERVER	1	Netcool/OMNIBus ObjectServer V7
SYBASE	1	Sybase SQL Server 10.0 RDBMS
SNMP	1	SNMP Trap forwarder
SERVICE_VIEW	1	Service View

SHOW WRITER ATTRIBUTES

Use the SHOW WRITER ATTRIBUTES command to show all the settings (or attributes) of the writer named *writer_name*.

Syntax

```
SHOW WRITER { ATTRIBUTES | ATTR } FOR writer_name;
```

The ATTRIBUTES keyword is interchangeable with the abbreviated ATTR keyword.

This command can only be used interactively.

Example

```
1> SHOW WRITER ATTR FOR SNMP_WRITER;
```

```
2> GO
```

Attribute	Value
-----	-----
MAP	SNMP_MAP
TYPE	SNMP
REVISION	1
GATEWAY	penelope

```
1>
```

Mapping commands

A number of mapping commands are available for gateways.

Related concepts

“Gateway configuration file” on page 98

CREATE MAPPING

Use the CREATE MAPPING command to create a mapping file named *mapping_name*, for use by a writer.

Syntax

```
CREATE MAPPING mapping_name ( mapping [ , mapping ] );
```

Mapping lines have the following syntax:

```
{ string | integer } = { string | integer | name | real | boolean }  
[ ON INSERT ONLY ] [ CONVERT TO { INT | STRING | DATE } ]
```

The first argument is an identifier for the destination field and the second argument is an identifier for the source field (or a preset value).

The right side of the mapping is dependent on the writer with which the mapping is to be used. (For gateway-specific details, see the writer section of the individual gateway publications.)

The optional ON INSERT ONLY clause determines the update behavior of the mapping. Without the ON INSERT ONLY clause, the field is updated every time a change is made to an alert. With the ON INSERT ONLY clause, the field is inserted at creation time (that is, when the alert appears for the first time) but is not updated on subsequent updates of the alert even if the field value is changed.

The optional CONVERT TO type clause allows the mapping to define a forced conversion for situations where a source field may not match the type of the destination field. The type can be INT, STRING, or DATE. This forces the source field to be converted to the specified data type.

Example

```
CREATE MAPPING SYBASE_MAP  
(  
  'Node'='@Node' ON INSERT ONLY,  
  'Summary'='@Summary' ON INSERT ONLY,  
  'Severity'='@Severity' );
```

DROP MAPPING

Use the DROP MAPPING command to remove the mapping named *mapping_name* from the gateway.

Syntax

```
DROP MAPPING mapping_name;
```

This command does not drop the map if it is being used by a writer.

Example

```
DROP MAPPING SYBASE_MAP;
```

SHOW MAPPINGS

Use the SHOW MAPPINGS command to list all the mappings that are currently created in the gateway.

Syntax

```
SHOW MAPPINGS;
```

This command can only be used interactively.

Example

```
1> SHOW MAPPINGS;
2> GO
Name                               Writers
-----
SNMP_MAP                           1
1>
```

SHOW MAPPING ATTRIBUTES

Use the SHOW MAPPING ATTRIBUTES command to show the mappings (or attributes) of the mapping named *mapping_name*.

Syntax

```
SHOW MAPPING { ATTRIBUTES | ATTR } FOR mapping_name;
```

The ATTRIBUTES keyword is interchangeable with the abbreviated ATTR keyword. This command can only be used interactively.

Example

```
SHOW MAPPING ATTR FOR SYBASE_MAP;
```

Filter commands

A number of filter commands are available for gateways.

Related concepts

“Gateway configuration file” on page 98

“Filter configuration” on page 101

CREATE FILTER

Use the CREATE FILTER command to create a filter named *filter_name* for use by a reader.

Syntax

```
CREATE FILTER filter_name AS filter_condition;
```

The filter specification *filter_condition* is an SQL condition.

Example

```
CREATE FILTER HIGH_TALLY_LOG AS 'Tally > 100';
CREATE FILTER NCOMS_FILTER AS 'Agent = \'NNM\'';
```


LOAD FILTER

Use the LOAD FILTER command to load a filter from a file.

Syntax

```
LOAD FILTER FROM 'filename';
```

Include the path to the file. Filter files have a .elf file extension.

Example

```
LOAD FILTER FROM '/disk/filters/newfilter.elf';
```

DROP FILTER

Use the DROP FILTER command to remove the filter named *filter_name* from the gateway.

Syntax

```
DROP FILTER filter_name;
```

The filter is not dropped if it is being used by a reader.

Example

```
DROP FILTER HIGH_TALLY_LOG;
```

Route commands

A number of route commands are available for gateways.

Related concepts

“Gateway configuration file” on page 98

“Route configuration” on page 100

ADD ROUTE

Use the ADD ROUTE command to add a route between a reader named *reader_name* and a writer named *writer_name*, to allow alerts to pass through the gateway.

Syntax

```
ADD ROUTE FROM reader_name TO writer_name;
```

Example

```
ADD ROUTE FROM NCOMS_READER TO ARS_WRITER;
```

REMOVE ROUTE

Use the REMOVE ROUTE command to remove an existing route between a reader named *reader_name* and a writer named *writer_name*.

Syntax

```
REMOVE ROUTE FROM reader_name TO writer_name;
```

Example

```
REMOVE ROUTE FROM NCOMS_READER TO ARS_WRITER;
```

SHOW ROUTES

Use the SHOW ROUTES command to show all currently-configured routes in the gateway.

Syntax

```
SHOW ROUTES;
```

This command can only be used interactively.

Example

```
1> SHOW ROUTES;
2> GO
Reader                               Writer
-----
NCOMS_READER                        SNMP_WRITER

1>
```

Configuration commands

A number of configuration commands are available for gateways.

LOAD CONFIG

Use the LOAD CONFIG command to load a gateway configuration file from a file named *filename*.

Syntax

```
LOAD CONFIG FROM 'filename';
```

Example

```
LOAD CONFIG FROM '/disk/config/gateconf.conf';
```

SAVE CONFIG

Use the SAVE CONFIG command to save the current configuration of the gateway into a file named in *filename*.

Syntax

```
SAVE CONFIG TO 'filename';
```

Example

```
SAVE CONFIG TO '/disk/config/newgate.conf';
```

DUMP CONFIG

Use the DUMP CONFIG command to clear the current configuration.

Syntax

```
DUMP CONFIG [ FORCE ];
```

If the gateway is active and forwarding alerts, this command does not clear the configuration unless the optional keyword FORCE is used.

Example

```
DUMP CONFIG;
```

General commands

A number of general commands are available for gateways.

SHUTDOWN

Use the SHUTDOWN command to instruct the gateway to shut down; all readers and writers are stopped.

Syntax

```
SHUTDOWN [ FORCE ];
```

By default, the gateway is not shut down if interactive changes to the configuration have not been saved.

If the optional FORCE keyword is used, the gateway is shut down, even if the configuration has been changed interactively.

Example

```
SHUTDOWN;
```

SET CONNECTIONS

Use the SET CONNECTIONS command to enable or disable connections to the gateway using the SQL interactive interface.

Syntax

```
SET CONNECTIONS { TRUE | FALSE | YES | NO };
```

When set to FALSE or NO, it is not possible to connect to the gateway with `nco_sql`. When set to TRUE or YES, it is possible to connect to the gateway with `nco_sql`. This command determines whether interactive reconfiguration is allowed.

Example

```
SET CONNECTIONS TRUE;
```

SHOW SYSTEM

Use the SHOW SYSTEM command to display information about the current gateway settings.

Syntax

```
SHOW SYSTEM;
```

The parameters returned are shown in the following table.

Table 26. Show system parameters

System Parameter	Description
Version	Version number of the gateway.
Server Type	Type of server. Set to Gateway.
Connections	Status of the SET CONNECTIONS flag.
Debug Mode	Status of the SET DEBUG MODE flag.

Table 26. Show system parameters (continued)

System Parameter	Description
Multi User	Gateway multi-user mode. Set to YES.
Configuration Changed	If the configuration has been changed interactively, this is set to YES.

More parameters can be returned when in debug mode. This command can only be used interactively.

Example

```
1> SHOW SYSTEM;
2> GO
```

System Parameter	Value
Version	7.0
Server Type	Gateway
Connections	ENABLED
Debug Mode	NO
Multi User	YES

Related reference

“SET CONNECTIONS” on page 119

“SET DEBUG MODE”

SET DEBUG MODE

Use the SET DEBUG MODE command to set the debugging mode of the gateway.

Syntax

```
SET DEBUG MODE { TRUE | FALSE | YES | NO };
```

When set to TRUE or YES, debugging messages are sent to the log file. The default setting is NO or FALSE. Use this command only under the advice of IBM Software Support.

Example

```
SET DEBUG MODE NO;
```

TRANSFER

Use the TRANSFER command to transfer the contents of one database table to another database table.

Syntax

```
TRANSFER 'tablename' FROM readername TO writername [ AS 'tableformat' ]
{ DELETE | DELETE condition | DO NOT DELETE }
[ USE TRANSFER_MAP ] [ USING FILTER filter_clause ];
```

You can use this command to transfer tables between Sybase, Oracle, Informix, ODBC, CORBA, and Socket Writer gateways.

The AS *tableformat* clause specifies the format of the destination table if it is different from the source table format.

The DELETE and DO NOT DELETE clauses define how the destination table is processed. By default, the contents of the destination table are deleted before the

contents of the source table are transferred. You can optionally specify a condition that determines whether the deletion occurs. If you specify the DO NOT DELETE clause, the contents of the destination table are not deleted before the contents of the source table are transferred.

Note: The DELETE clause does not function with the Socket Writer Gateway and the CORBA gateways.

The USE TRANSFER_MAP clause instructs the gateway to use the mapping definition that is assigned as the map to the writer used in the TRANSFER command. The USE TRANSFER_MAP clause is only available for use with the Oracle Gateway.

An optional filter clause can be applied by specifying USING FILTER followed by the filter. Enter a valid filter.

Example

```
TRANSFER 'alerts.conversions' FROM NCO_READER TO SYBASE_WRITER AS  
'alerts.conversions' DELETE;  
TRANSFER 'alerts.status' FROM NCOMS_READ TO Denco_WRITE AS 'ncoms.status'  
USING FILTER 'ServerName = \'NCOMS\'' DELETE USE TRANSFER_MAP;
```

Related reference

“CREATE FILTER” on page 116

Appendix A. Probe error messages and troubleshooting techniques

A number of error messages are common to all probes. This includes ProbeWatch and TSMWatch messages. Troubleshooting information is also available for probes.

See the individual probe publications for information about probe-specific messages.

Generic error messages

Probes can generate the following types of messages: fatal, error, warning, information, and debug.

Fatal-level messages

The probe automatically terminates when a fatal message is issued.

Table 27. Fatal level probe messages

Message	Description	Action
Connection to ObjectServer marked DEAD - aborting...	The connection to the ObjectServer ceased (and store and forward is not enabled in the probe).	Check that the ObjectServer is available.
Failed to access OMNIHOME directory: "directory name" Failed to set interfaces file location	The probe was unable to locate the interfaces file.	Check that the OMNIHOME environment variable is set to the correct destination.
Failed to connect - aborting	The ObjectServer is not available.	Check that the ObjectServer is running, that the interfaces file on the system where the probe is installed has an entry for the ObjectServer, and that there is no networking problem between the two systems.
Failed to create property Failed to define argument Failed to initialise Failed to set property Failed to process arguments Session create failed - aborting	Internal errors.	See your support contract for information about contacting IBM Software Support.
Failed to read rules - aborting	A property or command-line option is pointing to a non-existent rules file.	Check that the command-line option or properties file refers to the correct rules file.
Field "field name" not found in status table No matching field found for "field name"	The rules file being used refers to a field of the format @fieldname which does not exist in the status table.	Check the rules file and correct the problem.

Table 27. Fatal level probe messages (continued)

Message	Description	Action
Unknown data type returned from ObjectServer	The ObjectServer has returned unknown data.	See your support contract for information about contacting IBM Software Support.

Error-level messages

The probe is likely to terminate when an error message is issued.

Table 28. Error level probe messages

Message	Description	Action
Can't set generic property "property name" via command line Property "property name" for option "option name" does not exist	An option in the probe is not mapped correctly to a property.	Check the properties file for the named property and see the probe publication for supported properties.
Could not send alert	The probe was unable to send an alert (usually an internal alert) to the ObjectServer.	Check that the ObjectServer is available.
Could not set "fieldname" field	The probe was unable to set a field value. This may be because the ObjectServer tables have been modified so that default fields are no longer present.	Check if the ObjectServer tables have been modified.
CreateAndSet failed CreateAndSet failed for attr: "element name"	The probe is unable to create an element.	See your support contract for information about contacting IBM Software Support.
Error Setting SIGINT Handler Error Setting SIGQUIT Handler Error Setting SIGTERM Handler	The probe was unable to set up a signal handler for either an INT , QUIT, or TERM.	See your support contract for information about contacting the IBM Software Support.
Failed to open file: "file name"	A file referred to in the rules file (for example, with the table function) does not exist.	Check the rules file and ensure the file is available.
Failed to open message log: "file name"	The probe is unable to open the specified log file.	Check the command line or properties file and correct the problem.
Failed to open Properties file: "properties file name"	The probe is unable to open the properties file.	Check the properties file or command line to ensure the properties file is in the specified location.
Failed to open Rules file: "rules file name" The rules file for the probe is not available or incorrectly specified.	The probe is unable to open the rules file.	Check the properties file or command line to ensure the rules file is in the specified location.

Table 28. Error level probe messages (continued)

Message	Description	Action
No extraction data for "regexp" - missing ()'s? Regexp doesn't match for "string"	A regular expression being used in the extract function may be missing parentheses. The string data that is being used to extract may not match the regular expression. The extract function is unable to extract data.	Check the rules file and correct the problem.
Option "option name" used without argument	The option used expects a value which has not been supplied.	Check the probe publication and the contents of the command line.
OS Error: "error message" Procedure "procedure name": "error message" Server "server name": "error message"	There is an error in the Sybase connection. There should be a subsequent message from the probe which details the effect of this error.	See your support contract for information about contacting IBM Software Support.
Properties file: "error description" at line "line no"	There is an error in the format of the properties file.	Check the properties file at the specified line number and correct the problem.
PropGetValue failed	A required property has not been set.	Check the properties file.
Regular Expression Error: "regexp"	A regular expression is incorrectly formed in the rules file.	Check the rules file for the regular expression and correct the problem.
Results processing failed Unexpected return from results processing Unexpected value during results processing	There is a problem with the ObjectServer.	See your support contract for information about contacting IBM Software Support.
Rules file: "error description" at line "line no"	There is an error in the rules file format or syntax.	Check the rules file at the specified line number and correct the problem.
SendAlert failed	The probe was unable to send an alert to the ObjectServer.	Check that the ObjectServer is available.
SessionProcess failed	The probe was unable to process the alert against the rules file.	See your support contract for information about contacting IBM Software Support.
Unknown message level "message level string" - using WARNING level	The properties file or command line specified a message level which is not supported.	Check the properties file or command line and use a supported message level (debug, info, warning, error, fatal).
Unknown option: "option name"	An option has been used on the command line to start the probe which is not supported by the probe.	Check the probe documentation and the contents of the command line.
Unknown property "property name" - ignored	A property specified in the properties file does not exist in the probe.	Check the properties file for the named property and see the probe publication for supported properties.

Warning-level messages

These messages are issued as warnings but should not cause the probe to terminate.

Table 29. Warning level probe messages

Message	Description	Action
Failed to install Client Message Callback Failed to install Server Message Callback Failed to retrieve connection status - attempting to continue Results processing failed	There is a problem with the ObjectServer.	The probe will try to continue.
Failed to set SYBASE in environment	The probe was unable to override the SYBASE environment variable.	Check that the SYBASE environment variable is correctly set.
New value for field "field name" truncated to "number" characters	A string being copied into an alert field has had to be truncated to fit the field.	Check the rules file.
Type mismatch for property "property name" - new value ignored	A property has been set with the wrong data type.	Check the properties file or command line to ensure that the property is correctly set.

Information-level messages

This message is for information purposes.

Table 30. Information level probe messages

Message	Description	Action
Using stderr for logging	The probe was unable to open a log file.	No action required. The probe is writing messages to stderr.

Debug-level messages

Debug level messages provide information about the internal functions of the probe. These messages are aimed at probe developers but are listed here for completeness.

Table 31. Debug level probe messages

Message	Description	Action
A value for "string" doesn't exist in lookup table "table name"	A value requested from a lookup table is not available.	No action required. The function in the rules file returns an empty string.

Table 31. Debug level probe messages (continued)

Message	Description	Action
<p>Attempted to duplicate NULL string</p> <p>Attempted to free NULL pointer</p> <p>Attempted to realloc NULL pointer</p> <p>Failed to allocate memory (Requested size was "number" bytes)</p> <p>Failed to duplicate string</p> <p>Failed to reallocate memory block at address "hex address" (Requested size was "number" bytes)</p>	<p>An error or problem has occurred in the memory allocation or string handling components of the probe library.</p>	<p>No action required. The library handles the problem.</p>
<p>Failed to allocate command structure</p> <p>Failed to allocate context structure</p> <p>Failed to bind column</p> <p>Failed to connect</p> <p>Failed to describe column</p> <p>Failed to fetch number of columns</p> <p>Failed to initialise Sybase internals: "number"</p> <p>Failed to send command</p> <p>Failed to set appname</p> <p>Failed to set command query</p> <p>Failed to set hostname</p> <p>Failed to set password</p> <p>Failed to set username</p> <p>Got a row fail - continuing</p> <p>No columns in result set</p>	<p>A problem or error has occurred at the Sybase or ObjectServer connection level.</p>	<p>N/A</p>
<p>Failed to flush alerts before EXIT</p> <p>Problem during disconnect before EXIT</p> <p>Problem during session destruction before EXIT</p> <p>Problem during shutdown before EXIT</p>	<p>A problem has occurred during probe shutdown.</p>	<p>N/A</p>
<p>New value for field "field name" is "value"</p>	<p>A field value has been set.</p>	<p>N/A</p>

Table 31. Debug level probe messages (continued)

Message	Description	Action
OpIInitialise() called more than once	Multiple calls have been made to the OpIInitialise C probe API function, which can only be called once.	N/A

ProbeWatch and TSMWatch messages

In some situations, a probe or TSM generates events of its own. These events can provide information (such as startup or shutdown messages) or identify problems.

A number of elements are common to all ProbeWatch and TSMWatch messages.

ProbeWatch and TSMWatch messages are processed in the rules file and converted into alerts like other events. The following table shows the elements that are common to ProbeWatch and TSMWatch events.

Table 32. Common ProbeWatch and TSMWatch elements

Element name	Description
Summary	Summary string, described in the following tables.
Node	Name of the node on which the probe or TSM is running.
Agent	Name of the probe or TSM.
Manager	ProbeWatch or TSMWatch.

The following table describes summary strings that are common to all probes and TSMs.

Table 33. Common ProbeWatch and TSMWatch summary strings

ProbeWatch/TSMWatch message	Description	Cause
Going down ...	The probe or TSM is shutting down.	The probe or TSM is running a shutdown routine.
Running ...	The probe or TSM has started running.	The probe or TSM has just been started.
Unable to get events ...	The probe or TSM encountered a problem while listening for events.	There was a problem initializing the connection or there was a license or connection failure after some events were received. See your support contract for information about contacting IBM Software Support.
Rules file reread upon SIGHUP successful ...	The probe successfully re-read its rules file on receipt of a SIGHUP signal.	The probe received a SIGHUP signal.
Rules file reread upon SIGHUP failed ...	The probe could not re-read its rules file on receipt of a SIGHUP signal.	The probe received a SIGHUP signal.
Heartbeat ...	Heartbeat event	Not applicable

See the individual probe publications for additional summary strings for each probe.

TSMWatch messages are in the same format as ProbeWatch messages. The following table describes summary strings that are common to all TSMs.

Table 34. Common TSMWatch summary strings

TSMWatch message	Description	Action
Connection Attempted ... Connection Succeeded ... Connection Failed ... Connection Timed out ... Connection Lost ...	Messages relating to the establishment of a TCP/IP connection.	N/A
Disconnection Attempted ... Disconnection Succeeded ... Disconnection Failed ...	Messages relating to relinquishing a TCP/IP connection.	N/A
Wakeup Attempted ... Wakeup Succeeded ... Wakeup Failed ...	Messages relating to wake up functionality.	N/A
Login Attempted ... Login Succeeded ... Login Timed out ... Login Failed ...	Messages relating to host login.	N/A
Logout Attempted ... Logout Succeeded ... Logout Timed out ... Logout Failed ...	Messages relating to host logout.	N/A
Heartbeat Sent ... Heartbeat Received ... Heartbeat Timed out ...	Messages relating to sending and receiving heartbeat messages to and from the host.	N/A
Resynchronisation Attempted ... Resynchronisation Succeeded ... Resynchronisation Failed ...	Messages relating to synchronizing current alerts between the switch and Tivoli Netcool/OMNIBus.	N/A

Troubleshooting probes

This topic describes some of the common problems experienced by Tivoli Netcool/OMNIbus users and explains possible causes and solutions.

This troubleshooting information is divided into two areas:

- Common problem causes
- What to do if

Table 35. Troubleshooting probes

Area	Description
Common problem causes	This information contains a list of common problem causes. If you are unsure what your problem is, you should start by reading this part and following the instructions. If you cannot solve your problem by following the instructions in this part, move on to the "What to do if" information.
What to do if	This information describes common symptoms caused by probe problems and step-by-step instructions to help you locate and solve the problem. If none of the headings match the symptoms of your problem, read through the lists of instructions and make sure that you have tried all of the most likely solutions listed there.

Common problem causes

The most common causes of probe problems are:

- Incorrectly set OMNIHOME environment variable
- Errors in the rules file, particularly in extract statements
- Configuration errors in the properties file

For information about setting the OMNIHOME environment variable, see the *IBM Tivoli Netcool/OMNIbus Installation and Deployment Guide*.

Check that all of the properties are set correctly in the probe properties file. For example, check that the **Server** property contains the correct ObjectServer or proxy server name and that the **RulesFile** property contains the correct rules file name.

If you cannot solve the problem, read through the next section and make sure that you have tried all of the most likely solutions listed there.

What to do if

The headings in this topic describe the most common symptoms of probe problems. Find the heading that most closely describes your problem and follow the instructions until you have located the cause and solved the problem.

If none of the headings match the symptoms of your problem, read through the lists of instructions and make sure that you have tried all of the most likely solutions listed there. If you have tried all of the suggested problem solutions and your probe still does not work, See your support contract and contact IBM Software Support.

The probe does not start

If the probe does not start:

1. Run the probe in debug mode.
2. Check that the ObjectServer is running by trying to connect using **nco_ping** or **nco_sql**.
If you can connect successfully, the ObjectServer is running. If the ObjectServer is not running, this is likely to be the cause of the problem.
3. Check that there are no other probes running with the same configuration using the commands:

```
ps -ef | grep nco_p
```


A list of probe processes is displayed. Check that none of the processes correspond to the same type of probe. You cannot run two identical probe configurations because this duplicates all of the events forwarded to the ObjectServer.
4. Check that you are using the correct probe for the current version of the target software.
5. Check that there are no syntax errors in the rules file.
6. Check that your system has not run out of system resources and can launch more processes. You can do this using **df -k** or **top**. See the **df** and **top** man pages for more information about using these commands.
7. Check to see if the `$OMNIHOME/var/probename.saf` store-and-forward file exists. If it exists, check that it has not become too large. If your disk is full, the probes and ObjectServers are not able to work properly.
Attention: Store and forward is not designed to handle very large numbers of events. Left unattended, a store-and-forward file will continue to grow until it runs out of disk space.
8. Check that the store-and-forward file has not been corrupted. If the store-and-forward file has been corrupted there should be an error message in the log file (`$OMNIHOME/log/probename.log`). If the file is corrupted, delete it and restart the probe.
9. Check that the probe binary you are trying to run is the correct one for the current architecture by entering:

```
$OMNIHOME/bin/arch/probename -version
```


Check that the probe version matches your system architecture.
If you are running the probe on a remote host:
10. Check that the probe host can connect to the ObjectServer host using the **ping** command. Try to ping the ObjectServer host machine using the hostname and the IP address. See the **ping** man page for more information about how to do this.
If you cannot connect to the ObjectServer host using the **ping** command, there is a problem with the connection between your probe host and your ObjectServer host.
11. Check that the ObjectServer has been configured correctly in the Server Editor (**nco_xigen**) and that the interfaces information has been distributed to the ObjectServer and probe hosts.
12. Check to see if there is a firewall between the probe host and the ObjectServer host. If there is, make sure that the firewall allows traffic between the probe and the ObjectServer.

Related tasks

“Testing rules files” on page 53

“Debugging rules files” on page 53

Related reference

Chapter 5, “Common probe properties and command-line options,” on page 77

The probe is not sending alerts to the ObjectServer

If the probe is not sending alerts to the ObjectServer:

1. Check that the probe is running by entering:

```
ps -ef | grep nco_p
```

A list of probe processes is displayed. If the probe is not running, start the probe from the command line.
2. Check that there are no other probes running with the same configuration by entering:

```
ps -ef | grep nco_p
```

A list of probe processes is displayed. Check that none of the processes correspond to the same type of probe. You cannot run two identical probe configurations because this duplicates all of the events forwarded to the ObjectServer.
3. Read the probe properties file and check that all of the properties are set correctly. For example, check that the **Server** property contains the correct ObjectServer name and that the **RulesFile** property contains the correct rules file name.
4. Check that the probe event source has events to send to the ObjectServer.
5. Check that the ObjectServer you are logged in to is the same ObjectServer that the probe is forwarding events to.
6. Check that the event source you are trying to probe is working correctly. See the documentation supplied with your element manager for more information about how to do this.
7. Check that you are using the correct probe.
8. Check that the probe is not running in store-and-forward mode. To do this, check the `$OMNIHOME/var/probename.saf` and `$OMNIHOME/var/probename.reco` files to see if they are growing. If they are, disable store-and-forward mode.
9. Check that your system has not run out of system resources and can launch more processes. You can do this using `df -k` or `top`. See the `df` and `top` man pages for more information about using this command.
10. Check for any discard functions in the probe rules file. The discard function must be in a conditional statement; otherwise, all events are discarded.
If you are running the probe on a remote host:
11. Check that the probe host can connect to the ObjectServer host using the ping command. Try to ping the ObjectServer host machine using the hostname and the IP address. See the ping man page for more information about how to do this.

If you cannot connect to the ObjectServer host using the ping command, there is a problem with the connection between your probe host and your ObjectServer host.
12. Check that the ObjectServer has been configured correctly through the Server Editor (**nco_xigen**) and that the interfaces information has been distributed to the ObjectServer and probe hosts.

13. Check to see if there is a firewall between the probe host and the ObjectServer host. If there is, make sure that the firewall allows traffic between the probe and the ObjectServer.

Related concepts

“Store-and-forward mode for probes” on page 10

The probe is losing events

If not all of the events are being forwarded to the ObjectServer:

1. Run the probe in debug mode.
2. Check that the event source you are trying to probe is working correctly. See the documentation supplied with your element manager for more information about how to do this.
3. Check that the probe event source has events to send to the ObjectServer.
4. Check that all of the properties in the properties file are set correctly. For example, check that the **Server** property contains the correct ObjectServer name and that the **RulesFile** property contains the correct rules file name.
5. Check for any discard functions in the probe rules file. The discard function discards events based on specified conditions.

Related tasks

“Debugging rules files” on page 53

The probe is consuming too much CPU time

If the probe is consuming too much CPU time:

1. Run the probe in debug mode.
2. Check that the probe can connect to the event source.
3. Check to see if the `$OMNIHOME/var/probename.saf` store-and-forward file exists. If it exists, check that it has not become too large. If your disk is full, the probes and ObjectServer will not be able to work properly.
Attention: Store and forward is not designed to handle very large numbers of alerts. Left unattended, a store-and-forward file will continue to grow until it runs out of disk space.
4. Check that the store-and-forward file has not been corrupted. If the store-and-forward file has been corrupted there should be an error message in the probe log file (`$OMNIHOME/log/probename.log`). If the store-and-forward file is corrupted, delete it and restart the probe.

The event list is not being populated properly

If the probe is detecting events and forwarding them to the ObjectServer but the event list fields are not being populated correctly:

1. Run the probe in debug mode.
2. Check that fields which are not being populated properly are being correctly mapped to elements in the rules file.
3. Check that it is not a GUI problem by querying the alerts.status table using ObjectServer SQL.

Appendix B. Common gateway error messages

A number of error messages are common to all gateways. The *gateway_name* in each error message refers to the individual gateway name and indicates which gateway generated the error.

Table 36. Common gateway error messages

Error	Description	Action
<i>Gateway_name</i> Writer: HashAlloc failure in <i>_gateway_name</i> CacheAdd(). <i>Gateway_name</i> Writer: MemStrDup() failure in <i>_gateway_name</i> CacheAdd().	The gateway failed to allocate memory.	Try to free more memory.
<i>Gateway_name</i> Writer: Failed to allocate memory. <i>Gateway_name</i> Writer <i>writer_name</i> : Memory allocation failed. <i>Gateway_name</i> Writer: Memory allocation failure. <i>Gateway_name</i> Writer: Memory allocation error. <i>Gateway_name</i> Writer: Memory reallocation error. Failed to allocate memory in writer <i>writer_name</i> .	The gateway failed to allocate memory.	Try to free more memory.
<i>Gateway_name</i> Writer <i>writer_name</i> : Could not create serial cache - memory problems. <i>Gateway_name</i> Writer <i>writer_name</i> : Failed to allocate memory for a GPCModule handle.	The gateway failed to allocate memory.	Try to free more memory.
<i>Gateway_name</i> Writer: Failed to lock connection mutex.	The writer failed to lock the ObjectServer feedback connection in order to access the connection and feed back problem ticket data for the associated alert. This lock failure may be due to insufficient resources or as a result of the underlying threading system preventing a deadlock between multiple threads that are contending for the resource.	Refer to your support contract for information about contacting the helpdesk.
<i>Gateway_name</i> Writer: Failed to re-acquire alert details from OS.	This error message comes from the gateway cache reclamation subsystem. This message indicates that the gateway failed to re-acquire the trouble ticket number and reclaim its internal cache entry from the ObjectServer.	Refer to your support contract for information about contacting the helpdesk.

Table 36. Common gateway error messages (continued)

Error	Description	Action
<i>Gateway_name</i> Writer: Invalid datatype for problem number feedback field.	The data type is invalid.	Refer to the <i>IBM Tivoli Netcool/OMNIBus Administration Guide</i> for information about data types.
<i>Gateway_name</i> Writer: Serial x already in serial Cache. Cannot add.	The gateway tried to add a serial number that already exists.	Refer to your support contract for information about contacting the helpdesk.
<i>Gateway_name</i> Writer: Serial x not found in serial cache. Cannot Delete.	The gateway could not delete this alert because it has already been deleted in Tivoli Netcool/OMNIBus.	You do not need to do anything.
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to construct path to <i>gateway_name</i> Read/Write Module.	The gateway could not locate the reader or writer module application.	Check that the module is installed in the correct location.
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to construct the argument list for <i>gateway_name</i> Module.	Failed to construct the argument list for gateway module.	Check that the arguments in the configuration file are set correctly.
<i>Gateway_name</i> Writer <i>writer_name</i> : GPCModule creation failed.	Failed to create the GPCModule due to insufficient memory.	Try to free more memory.
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to start the OS- <i>gateway_name</i> Writer. <i>Gateway_name</i> Writer <i>writer_name</i> : Failed to start the <i>gateway_name</i> -OS Reader.	Failed to start the ObjectServer gateway reader or writer module.	Check that the module is installed in the correct location and that the file permissions are set correctly.
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to shutdown <i>gateway_name</i> Writer.	Failed to stop gateway writer module.	Check the writer log file for more information.
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to construct path to <i>gateway_name</i> Read/Write Module.	Failed to construct the path to the gateway reader or writer module application.	Check that the module is installed in the correct location and that the file permissions are set correctly.
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to find the <i>gateway_name</i> Read/Write Module [x].	Cannot find the module binary.	Check that the module is installed in the correct location and that the file permissions are set correctly.
<i>Gateway_name</i> Writer <i>writer_name</i> : Incorrect permissions on the <i>gateway_name</i> module binary [x].	The module's file permissions are set incorrectly.	Check that the module is installed in the correct location and that the file permissions are set correctly.
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to create the Serial Cache Mutex.	The gateway writer failed to create the necessary data protection structure for the internal serial number cache due to insufficient resources. This is generally due to insufficient memory.	Try to free more memory.
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to create the Conn Mutex.	The gateway writer failed to create the necessary data protection structure for the ObjectServer connection due to insufficient resources.	Try to free more memory.
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to start the <i>gateway_name</i> -to-OS service thread.	The gateway failed to spawn the service thread.	Check that the gateway can access the ObjectServer.

Table 36. Common gateway error messages (continued)

Error	Description	Action
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to send a shutdown request to the <i>gateway_name</i> Writer.	The gateway did not shut down cleanly.	Check the writer log file for more information.
Failed to install SIGCHLD handler. Failed to install SIGPIPE handler.	The gateway failed during handler installation.	Refer to your support contract for information about contacting the helpdesk.
No <mapname> attribute for <i>gateway_name</i> writer <i>writer_name</i> .	The gateway could not find the map name.	Check the configuration file.
<mapname> attribute is not a name for <i>gateway_name</i> writer <i>writer_name</i> .	Incorrect writer name given.	Check the configuration file.
A MAP called <map> does not exist for <i>gateway_name</i> writer <i>writer_name</i> .	The gateway could not find the specified map.	Check the configuration file.
MAP <map> is invalid for <i>gateway_name</i> writer <i>writer_name</i> .	The given map is not valid.	Check the configuration file.
Map <map> is not the journal map and cannot contain the <journal map name> map item in <i>gateway_name</i> Writer <i>writer_name</i> .	If this map is not the journal map, then the JOURNAL_MAP_NAME attribute is set incorrectly.	Check the JOURNAL_MAP_NAME attribute in the gateway configuration file.
<i>Gateway_name</i> Writer: Failed to send <i>gateway_name</i> Event to the <i>gateway_name</i> Writer module.	The gateway failed to send a given event.	Check the log files for more information.
<i>Gateway_name</i> Writer: Failed to wait for return from the <i>gateway_name</i> Writer module.	There was an error in retrieving the success statement.	Check the log files for more information.
<i>Gateway_name</i> Writer: Failed to read the status return message from the <i>gateway_name</i> Writer module.	The gateway failed to retrieve the status of a module.	Check the log files for more information.
<i>Gateway_name</i> Writer: Failed to send event to <i>gateway_name</i> .	The module failed to send the event to gateway.	Check the log files for more information.
<i>Gateway_name</i> Writer: <i>gateway_name</i> Writer Module experienced Fatal Error.	There was a fatal error.	Check the log files for more information.
<i>Gateway_name</i> Writer: Failed to send event to <i>gateway_name</i> . Unknown type.	The gateway received unexpected type.	Refer to your support contract for information about contacting the helpdesk.
<i>Gateway_name</i> Writer: Failed to build serial index.	The gateway failed to build indexes.	Check that the Serial column exists in the ObjectServer alerts.status table.
Incorrect data type for the Serial column.	The gateway did not receive the correct data type.	Check that the data type for the Serial column in the ObjectServer alerts.status table is an integer.
<i>Gateway_name</i> Writer: Failed to build server serial index.	The gateway failed to get the server serial index.	Check that the ServerSerial column exists in the ObjectServer alerts.status table.
Incorrect data type for the Server Serial column.	The gateway did not receive the correct data type.	Check that the data type for the ServerSerial column in the ObjectServer alerts.status table is an integer.

Table 36. Common gateway error messages (continued)

Error	Description	Action
<i>Gateway_name</i> Writer: Failed to build server name index.	The gateway failed to get the server name index.	Check that the <code>ServerName</code> column exists in the <code>ObjectServer alerts.status</code> table.
Incorrect data type for the <code>ServerName</code> column.	The gateway did not receive the correct data type.	Check that the data type for the <code>ServerName</code> column in the <code>ObjectServer alerts.status</code> table is a string.
<i>Gateway_name</i> Writer: Failed to find field <fieldnumber> in <i>gateway_name</i> Event.	The gateway could not find the field number it was looking for.	Refer to your support contract for information about contacting the helpdesk.
<i>Gateway_name</i> Writer: Invalid field name for expansion on action SQL [<field>].	The gateway received an invalid field name.	Refer to the <i>IBM Tivoli Netcool/OMNIBus Administration Guide</i> for information about ObjectServer SQL.
<i>Gateway_name</i> Writer: Unenclosed field expansion request in action SQL [<sql action>].	The gateway did not find an enclosing bracket.	Check the <code>action.sql</code> file.
<i>Gateway_name</i> Writer: Failed to turn counter-part notification back-on. Fatal error in <i>gateway_name</i> -to-OS Feedback. <i>Gateway_name</i> Writer: Failed to turn counter-part notification off. <i>Gateway_name</i> -to-OS Feedback failed.	The gateway failed to send a notify command.	This is an internal error. Refer to your support contract for information about contacting the helpdesk.
<i>Gateway_name</i> Writer: Failed to send SQL command to ObjectServer. <i>Gateway_name</i> -to-OS Feedback failed.	The gateway failed to send the SQL command to the ObjectServer.	Check the ObjectServer log file.
Failed to find the column <column_name> in map <map_name>.	The gateway failed to find the given column.	Check that the given column name is entered correctly in the configuration file and that it is shown in the <code>ObjectServer alerts.status</code> table.
<i>Gateway_name</i> Writer: Failed to lock the cache mutex.	The writer failed to lock the ObjectServer feedback connection in order to access the connection and feed back problem ticket data changes for the associated alert.	This lock failure may be due to insufficient resources or as a result of the underlying threading system preventing a deadlock between multiple threads that are contending for the resource.
Failed to find cached problem ticket for serial <serial number> using map <map name>.	The gateway failed to find the specified cache problem ticket number.	Check that the specified ticket was originally created by the gateway.
<i>Gateway_name</i> Writer: Failed to unlock the cache mutex.	After access to the cache, an attempt to unlock the data structures protection lock failed. This message indicates that the gateway is in a position which will lead to a deadlock situation.	Refer to your support contract for information about contacting the helpdesk.
<i>Gateway_name</i> Writer: Cache add error.	The gateway could not add the serial to the serial cache due to insufficient resources.	Try to free more memory.

Table 36. Common gateway error messages (continued)

Error	Description	Action
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to create <i>gateway_name</i> Event for journal update.	The gateway failed to create the journal event update.	Check the writer log file.
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to send journal update event to <i>gateway_name</i> .	The gateway failed to send journal event update.	Check the writer log file.
<attribute name> attribute is not a string for <i>gateway_name</i> writer <i>writer_name</i> .	An attribute in the writer was of an incorrect data type.	Check the writer definition in the configuration file.
No <attribute name> attribute for <i>gateway_name</i> writer <i>writer_name</i> given.	The gateway failed to find the attribute.	Add the attribute to the writer definition in the configuration file.
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to find the <counterpart attribute> attribute for the writer. This is necessary due to bi-directional nature.	An attempt to find the necessary counterpart attribute failed.	Check the configuration file.
<i>Gateway_name</i> Writer <i>writer_name</i> : Is not a name for an Object Server reader.	The gateway found an incorrect data type.	Check the configuration file.
<i>Gateway_name</i> Writer <i>writer_name</i> : Reader <reader> was not found for counter part.	The reader was not found.	Check the counterpart configuration in the configuration file.
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to send SKIP Command.	This command failed to disable IDUC on a bidirectional connection.	Refer to your support contract for information about contacting the helpdesk.
Connection to feedback server failed.	The gateway failed to make a connection.	Check the ObjectServer log file.
Failed to set the death call on the feedback connection.	The gateway failed to set the necessary property.	This is an internal error. Refer to your support contract for information about contacting the helpdesk.
Writer counterpart error.	The gateway failed to find the counterpart attribute for gateway writer.	Check the counterpart configuration in the configuration file.
<i>Gateway_name</i> Writer: Failed to stat() the action SQL file " <i>filename</i> ".	The gateway failed to stat the file in order to determine its size.	Check the file access permissions for the specified action file.
<i>Gateway_name</i> Writer: Empty action SQL file " <i>filename</i> ".	File " <i>filename</i> " is empty.	Check the action SQL file.
<i>Gateway_name</i> Writer: Failed to open the action SQL file " <i>filename</i> ".	The gateway failed to open the file.	Check the file permissions.
<i>Gateway_name</i> Writer: Failed to read the action SQL file " <i>filename</i> ".	The gateway failed to read the file.	Check the file permissions.
<i>Gateway_name</i> Writer: No Action SQL find in file " <i>filename</i> ".	There is no action SQL in the file.	Check the file.
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to read the conversions table.	The gateway failed to read the conversions table.	Check the file permissions.

Table 36. Common gateway error messages (continued)

Error	Description	Action
<i>Gateway_name</i> Writer: Failed to find PM %s in cache for return PMO event.	The gateway has received a Problem Management Open return event from gateway for the problem ticket. When an attempt was made to look up the problem ticket number in the writer's cache, in order to determine the serial number of the ticket's associated alert, no record could be reclaimed or found.	Refer to your support contract for information about contacting the helpdesk.
<i>Gateway_name</i> Writer: Open Feedback Failed.	The gateway failed to construct the open action SQL statement or send the SQL action command to the server.	Check the ObjectServer SQL file.
<i>Gateway_name</i> Writer: No Update action SQL for <i>gateway_name</i> Update event.	There is no update action SQL statement.	Check the configuration file.
<i>Gateway_name</i> Writer: Failed to find PM %s in cache for return PMU event.	The gateway has received a Problem Management Update return event from gateway for the problem ticket. When an attempt was made to look up the problem ticket number in the writer's cache in order to determine the serial number of the ticket's associated alert, no record could be reclaimed or found.	Refer to your support contract for information about contacting the helpdesk.
<i>Gateway_name</i> Writer: Update Feedback Failed.	The gateway failed to construct the open action SQL statement or send the SQL action command to the server.	Check the ObjectServer log file.
<i>Gateway_name</i> Writer: Failed to find PM %s in cache for return PMJ event.	The gateway has received a Problem Management Journal return event from gateway for the problem ticket. When an attempt was made to look up the problem ticket number in the writer's cache in order to determine the serial number of the ticket's associated alert, no record could be reclaimed or found.	Refer to your support contract for information about contacting the helpdesk.
<i>Gateway_name</i> Writer: Journal Feedback Failed.	The gateway failed to construct the open action SQL statement or send the SQL action command to the server.	Check the ObjectServer log file.
<i>Gateway_name</i> Writer: Failed to find PM %s in cache for return PMC event.	The gateway has received a Problem Management Close return event from gateway for the problem ticket. When an attempt was made to look up the problem ticket number in the writer's cache in order to determine the serial number of the ticket's associated alert, no record could be reclaimed or found.	Refer to your support contract for information about contacting the helpdesk.

Table 36. Common gateway error messages (continued)

Error	Description	Action
<i>Gateway_name</i> Writer: Close Feedback Failed.	The gateway failed to construct the open action SQL statement or send the SQL action command to the server.	Check the ObjectServer log file.
Received error code <code> from Reader/Writer Module - [<message>].	The gateway received an error message.	Check the module log files.
<i>Gateway_name</i> Writer: Failed to read <i>gateway_name</i> event from <i>gateway_name</i> Reader Module.	The gateway failed to read the event sent by the gateway reader module.	Check the reader log files.
<i>Gateway_name</i> Writer: Received event of type <event type> which was unexpected.	The gateway received an unknown event type.	Refer to your support contract for information about contacting the helpdesk.
<i>Gateway_name</i> Writer: Received invalid known message from Reader/Writer Module for this system.	The gateway received an invalid known message.	Refer to your support contract for information about contacting the helpdesk.
<i>Gateway_name</i> Writer: Received unknown message from Reader/Writer Module.	The gateway received an invalid unknown message.	Refer to your support contract for information about contacting the helpdesk.
<i>Gateway_name</i> Writer: Failed to block on data feed from <i>gateway_name</i> Reader Module.	The gateway failed to block due to a shutdown request. This message is displayed when the gateway is shutting down.	Refer to your support contract for information about contacting the helpdesk.
<i>Gateway_name</i> Writer: Fatal thread termination. Stopping gateway.	A thread exited unexpectedly.	Check the gateway log files.
<attribute name> attribute is not a string for <i>gateway_name</i> writer <i>writer_name</i> - IGNORED	An attribute name is not recognized. The gateway will ignore it.	Check the gateway log files.
<attribute name> attribute must be set to TRUE or FALSE for writer <i>writer_name</i> .	An attribute name has not been set to TRUE or FALSE.	Check the gateway configuration file.
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to shutdown <i>gateway_name</i> Reader/Writer Modules.	The gateway failed to shut down the reader and writer modules.	Check the module log file.
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to disconnect feedback connection.	The disconnect of feedback channel failed.	Check the ObjectServer log file.
Failed to create <i>gateway_name</i> event structure for a problem management open event in writer <i>writer_name</i> .	The gateway writer failed to allocate a gateway event structure for a problem management open event due to insufficient memory resources.	Try to free more memory.
<i>Gateway_name</i> Writer: FEEDBACK FAILED!!	The gateway failed to store the problem number.	Check the ObjectServer log file.
Failed to create journal for <i>gateway_name</i> writer <i>writer_name</i> (from INSERT)	The gateway failed to create journal.	Check the writer log file.

Table 36. Common gateway error messages (continued)

Error	Description	Action
Failed to create <i>gateway_name</i> event structure for a problem management update event in writer <i>writer_name</i> .	The gateway writer failed to allocate a gateway event structure for a problem management update event due to insufficient memory resources.	Try to free more memory.
<i>Gateway_name</i> Writer <i>writer_name</i> : Failed to delete problem ticket from cache for serial <serial number>.	The gateway failed to delete serial number from cache.	This is an internal error. You can ignore it.
Failed to create <i>gateway_name</i> event structure for a PMC event in writer <i>writer_name</i> .	The gateway writer failed to allocate a gateway event structure for a Problem Management Close event due to insufficient memory resources.	Try to free more memory.

Appendix C. Regular expressions

Tivoli Netcool/OMNibus supports the use of regular expressions in search queries that you perform on ObjectServer data. Regular expressions are sequences of *atoms* that are made up of normal characters and metacharacters.

An atom is a single character or a pattern of one or more characters in parentheses. Normal characters include uppercase and lowercase letters, and numbers. Metacharacters are non-alphabetic characters that possess special meanings in regular expressions.

Two types of regular expression libraries are available for use with the ObjectServer:

- **NETCOOL:** This library is useful for single-byte character processing.
- **TRE:** This library enables use of the POSIX 1003.2 extended regular expression syntax, and provides support for both single-byte and multi-byte character languages. When the UTF-8 encoding is enabled on Windows, only the characters within Unicode plane 0, the Basic Multilingual Plane (BMP), are supported in regular expression patterns. Any character outside of the BMP, which is found in the pattern, will result in an error. The matching strings for the regular expression pattern can contain any UTF-8 character.

Note: Use of the TRE library can lead to a marked decrease in system performance. Optimal system performance is achieved with the NETCOOL library.

You can use the ObjectServer property **RegexpLibrary** to specify which library should be used for regular expression matching. The NETCOOL regular expression library is enabled by default.

NETCOOL regular expression library

If your system supports single-byte character languages, you can use the NETCOOL regular expression library to run search queries on your data. You obtain optimal system performance with this library, over the TRE regular expression library.

The NETCOOL regular expression library supports the use of normal characters and metacharacters. The following table describes the set of metacharacters supported by the NETCOOL regular expression library.

Table 37. Metacharacters

Metacharacter	Description	Examples
*	Matches zero or more instances of the preceding atom. Matches as many instances as possible.	goo* matches my godness, my goodness, and my goodness, but not my gdness.
+	Matches one or more instances of the preceding atom. Matches as many instances as possible.	goo+ matches my goodness and my goodness, but not my godness.

Table 37. Metacharacters (continued)

Metacharacter	Description	Examples
?	Matches zero or more instances of the preceding atom.	goo? matches my godness, my goodness, and my gooodness, but not my gdnness. colou?r matches color and colour. end-?user matches enduser and end-user.
\$	Matches the end of the string.	end\$ matches the end, but not the ending.
^	Matches the beginning of the string.	^severity matches severity level 5, but not The severity is 5.
.	Matches any single character.	b.at matches baat, bBat, and b4at, but not bat or bB4at.
[abcd]	Matches any character in the square brackets.	[nN][oO] matches no, n0, No, and NO. gr[ae]y matches both spellings of the word 'grey'; that is, gray and grey.
[a-d]	Matches any character in the range of characters separated by a hyphen (-).	[0-9] matches any decimal digit. [ab3-5] matches a, b, 3, 4, and 5. ^[A-Za-z]+\$ matches any string that contains only upper or lowercase characters.
[^abcd] [^a-d]	Matches any character except those in the square brackets or in the range of characters separated by a hyphen (-).	[^0-9] matches any string that does not contain any numeric characters.
()	Indicates that the characters within the parentheses should be treated as a character pattern.	A(boo)+Z matches AbooZ, AboobooZ, and AboobooZ, but not AboZ or AboooZ. Jan(uary)? matches Jan and January.
	Matches one of the atoms on either side of the pipe character.	A(B C)D matches ABD and ACD, but not AD, ABCD, ABBD, or ACCD. (AB CD) matches AB and CD, but not ABD and ACD.
\	Indicates that the metacharacter following should be treated as a regular character. The metacharacters listed in this table require a backslash escape character as a prefix to switch off their special meaning.	* matches the * character. \. matches the \ character. \. matches the . character. \[[0-9]*\] matches an opening square bracket, followed by any digits or spaces, followed by a closed bracket.

Related concepts

“TRE regular expression library” on page 145

TRE regular expression library

Use the TRE regular expression library to run search queries on both single-byte and multi-byte character languages.

The TRE regular expression library supports usage of the POSIX 1003.2 extended regular expression syntax in the form of:

- Metacharacters
- Minimal or non-greedy quantifiers
- Bracket expressions
- Constructs for multicultural support
- Backslash sequences

Restriction: A marked decrease in system performance might be observed when using this library.

Related reference

“NETCOOL regular expression library” on page 143

Metacharacters

Metacharacters are non-alphabetic characters that possess special meanings in regular expressions.

The set of metacharacters that can be used in extended regular expression syntax is as follows:

`* + ? $ ^ . () | \ { } [`

The following table describes all of these metacharacters except the square bracket `[` metacharacter. You can use the `[` metacharacter to construct bracket expressions.

Table 38. Metacharacters

Metacharacter	Description	Examples
<code>*</code>	Matches zero or more instances of the preceding atom. Matches as many instances as possible.	<code>goo*</code> matches <code>my godness</code> , <code>my goodness</code> , and <code>my gooodness</code> , but not <code>my gdness</code> .
<code>+</code>	Matches one or more instances of the preceding atom. Matches as many instances as possible.	<code>goo+</code> matches <code>my goodness</code> and <code>my gooodness</code> , but not <code>my godness</code> .
<code>?</code>	Matches zero or more instances of the preceding atom.	<code>goo?</code> matches <code>my godness</code> , <code>my goodness</code> , and <code>my gooodness</code> , but not <code>my gdness</code> . <code>colou?r</code> matches <code>color</code> and <code>colour</code> . <code>end-?user</code> matches <code>enduser</code> and <code>end-user</code> .
<code>\$</code>	Matches the end of the string.	<code>end\$</code> matches the end, but not the ending.
<code>^</code>	Matches the beginning of the string. The <code>^</code> metacharacter can also be used in bracket expressions.	<code>^severity</code> matches <code>severity level 5</code> , but not <code>The severity is 5</code> .

Table 38. Metacharacters (continued)

Metacharacter	Description	Examples
.	Matches any single character.	b.at matches baat, bBat, and b4at, but not bat or bB4at.
()	Indicates that the characters within the parentheses should be treated as a character pattern.	A(boo)+Z matches AbooZ, AboobooZ, and AboobooZ, but not AboZ or AboooZ. Jan(uary)? matches Jan and January.
	Matches one of the atoms on either side of the pipe character.	A(B C)D matches ABD and ACD, but not AD, ABCD, ABBD, or ACCD. (AB CD) matches AB and CD, but not ABD and ACD.
\	Indicates that the metacharacter following should be treated as a regular character. The metacharacters listed in this section require a backslash escape character as a prefix to switch off their special meaning. The \ metacharacter can also be used to construct backslash sequences.	* matches the * character. \ matches the \ character. \. matches the . character.
{m , n}	Matches from m to n instances of the preceding atom, where m is the minimum and n is the maximum. Matches as many instances as possible. Note: m and n are unsigned decimal integers between 0 and 255.	f{1,2}ord matches ford and fford. N/{1,3}A matches N/A, N//A, and N///A, but not NA or N///A.
{m , }	Matches m or more instances of the preceding atom.	Z{2,} matches two or more repetitions of Z.
{m}	Matches exactly m instances of the preceding atom.	a{3} matches aaa. 1{2} matches 11.

Related reference

“Bracket expressions” on page 148

“Backslash sequences” on page 149

Minimal or non-greedy quantifiers

Regular expressions are generally considered *greedy* because an expression with repetitions will attempt to match as many characters as possible. The asterisk (*), plus (+), question mark (?), and curly braces ({}) metacharacters exhibit 'repetitious' behavior, and attempt to match as many instances as possible.

To make a subexpression match as few characters as possible, a question mark (?) can be appended to these metacharacters to make them *minimal* or *non-greedy*. The following table describes the non-greedy quantifiers.

Table 39. Minimal/non-greedy quantifiers

Quantifier	Description	Examples
*?	Matches zero or more instances of the preceding atom. Matches as few instances as possible.	Given an input string of Netcool Tool Library: <ul style="list-style-type: none"> The first group in <code>^(.*1).*</code> matches Netcool Tool . The first group in <code>^(.*?1).*</code> matches Netcool.
+?	Matches one or more instances of the preceding atom. Matches as few instances as possible.	Given an input string of little: <ul style="list-style-type: none"> <code>.*?l</code> matches l. <code>^.+l</code> matches littl.
??	Matches zero or one instance of the preceding atom. Matches as few instances as possible.	<code>??b</code> matches ab in abc, and b in bbb. <code>?b</code> matches ab in abc, and bb in bbb.
{m , n} ?	Matches from m to n instances of the preceding atom, where m is the minimum and n is the maximum. Matches as few instances as possible. Note: m and n are unsigned decimal integers between 0 and 255.	Given an input string of Netcool Tool Cool Fool Library: <ul style="list-style-type: none"> <code>^((.*?ool)*).*</code> matches Netcool Tool Cool Fool. <code>^((.*?ool)+).*</code> matches Netcool Tool Cool Fool. <code>^((.*?ool)?).*</code> matches Netcool. <code>^((.*?ool){2,5}).*</code> matches Netcool Tool Cool Fool. <code>^((.*?ool){2,5}?).*</code> matches Netcool Tool. <code>^((.*?ool){2,5}) [FL].*</code> matches Netcool Tool Cool Fool. <code>^((.*?ool){2,5}?) [FL].*</code> matches Netcool Tool Cool.
{m , } ?	Matches m or more instances of the preceding atom. Matches as few instances as possible.	Given an input string of Netcool Tool Cool Fool Library: <ul style="list-style-type: none"> <code>^((.*?ool){2,}).*</code> matches Netcool Tool Cool Fool. <code>^((.*?ool){2,}?).*</code> matches Netcool Tool. <code>^((.*?ool){2,}) [FL].*</code> matches Netcool Tool Cool Fool. <code>^((.*?ool){2,}?) [FL].*</code> matches Netcool Tool Cool.

Bracket expressions

Bracket expressions can be used to match a single character or collating element.

The following table describes how to use bracket expressions.

Table 40. Bracket expressions

Expression	Description	Examples
[abcd]	Matches any character in the square brackets.	[nN] [oO] matches no, nO, No, and NO. gr[ae]y matches both spellings of the word 'grey'; that is, gray and grey.
[a-d]	Matches any character in the range of characters separated by a hyphen (-).	[0-9] matches any decimal digit. [ab3-5] matches a, b, 3, 4, and 5. [0-9]{4} matches any four-digit string. ^[A-Za-z]+\$ matches any string that contains only upper or lowercase characters. \[[0-9]*\] matches an opening square bracket, followed by any digits or spaces, followed by a closed bracket.
[^abcd] [^a-d]	Matches any character except those in the square brackets or in the range of characters separated by a hyphen (-).	^[^0-9] matches any string that does not contain any numeric characters.
[.ab.]	Matches a multi-character collating element.	[.ch.] matches the multi-character collating sequence ch (if the current language supports that collating sequence).
[=a=]	Matches all collating elements with the same primary sort order as that element, including the element itself.	[=e=] matches e and all the variants of e in the current locale.

Note the following points:

- The caret character (^) only has a special meaning when included as the first character after the open bracket ([). Otherwise, it is treated as a normal character.
- The hyphen character (-) is treated as a normal character only under either of the following conditions:
 - The hyphen character is the first or last character within the square brackets, for example, [ab-] or [-xy].
 - The hyphen character is the only (both first and last) character; that is, [-].
- To match a closing square bracket within a bracketed expression, the closing bracket must be the first character within the enclosing brackets; for example, [] [xy] matches], [, x, and y.
- Other metacharacters are treated as normal characters within square brackets, and do not need to be escaped; for example, [ca\$] will match c, a, or \$.

Related reference

“Metacharacters” on page 145

Constructs for multicultural support

The sort order of characters (and any of their variants) is locale-dependent, so different regular expressions are generally required to match characters of the same class, in different locales. To facilitate multicultural support, a set of predefined names enclosed in `[:` and `:]` can be used to represent characters of the same class.

The set of valid names depends on the value of the `LC_CTYPE` environment variable of the current locale, but the names shown in the following table are valid in all locales.

Table 41. Multicultural constructs

Construct	Description
<code>[a1num:]</code>	Matches any alphanumeric character.
<code>[alpha:]</code>	Matches any alphabetic character.
<code>[blank:]</code>	Matches any blank character - that is, space and TAB.
<code>[cntrl:]</code>	Matches any control characters; these are non-printable.
<code>[digit:]</code>	Matches any decimal digits.
<code>[graph:]</code>	Matches any printable character except space.
<code>[lower:]</code>	Matches any lowercase alphabetic character.
<code>[print:]</code>	Matches any printable character including space.
<code>[punct:]</code>	Matches any printable character that is not a space or alphanumeric; that is, punctuation.
<code>[space:]</code>	Matches any whitespace character.
<code>[upper:]</code>	Matches any uppercase alphabetic character.
<code>[xdigit:]</code>	Matches any hexadecimal digit.

Example: Multicultural constructs

`[[:lower:]]AB` matches the lowercase letters and uppercase A and B.

`[[:space:]][[:alpha:]]` matches any character that is either whitespace or alphabetic.

`[[:alpha:]]` matches to `[A-Za-z]` in the English locale (`en`), but would include accented or additional letters in another locale.

Backslash sequences

When constructing regular expressions, the backslash character can be used in a variety of ways.

The backslash character (`\`) can be used to:

- Turn off the special meaning of metacharacters so they can be treated as normal characters.
- Include non-printable characters in a regular expression.
- Give special meaning to some normal characters.
- Specify backreferences. *Backreferences* are used to specify that an earlier matching subexpression is matched again later.

Note: The backslash character cannot be the last character in a regular expression.

The following table describes how to specify backslash sequences for non-printable characters and backreferences. This table also shows how to use backslash sequences to apply special meaning to some normal characters.

Table 42. Backslash sequences

Backslash sequence	Description
\a	Matches the bell character (ASCII code 7).
\e	Matches the escape character (ASCII code 27).
\f	Matches the form-feed character (ASCII code 12).
\n	Matches the new-line or line-feed character (ASCII code 10).
\r	Matches the carriage return character (ASCII code 13).
\t	Matches the horizontal tab character (ASCII code 9).
\v	Matches the vertical tab character.
\<	Matches the beginning of a word, or the beginning of an identifier, defined as the boundary between non-alphanumerics and alphanumerics (including underscore). This matches no characters, only the context.
\>	Matches the end of a word or identifier.
\b	Matches a word boundary; that is, matches the empty string at the beginning or end of an alphanumeric sequence. Enables a 'whole words only' search.
\B	Matches a non-word boundary; that is, matches the empty string not at the beginning or end of a word.
\d	Matches any decimal digit. Equivalent to [0-9] and [[:digit:]].
\D	Matches any non-digit character. Equivalent to [^0-9] or [^[:digit:]].
\s	Matches any whitespace character. Equivalent to [\t\n\r\f\v] or [[:space:]].
\S	Matches any non-whitespace character. Equivalent to [^ \t\n\r\f\v] or [^[:space:]].
\w	Matches a word character; that is, any alphanumeric character or underscore. Equivalent to [a-zA-Z0-9_] or [[:alnum:]].
\W	Matches any non-alphanumeric character. Equivalent to [^a-zA-Z0-9_] or [^[:alnum:]].
\[1-9]	A backslash followed by a single non-zero decimal digit <i>n</i> is termed a <i>backreference</i> . Matches the same set of characters matched by the <i>n</i> th parenthesized subexpression.

Example backslash constructs

`\bcat\b` matches `cat` but not `cats` or `bobcat`.

`\d\s` matches a digit followed by a whitespace character.

`[\d\s]` matches any digit or whitespace character.

`.([XY]).([XY]).` matches `aXbXc` and `aYbYc`, but also `aXbYc` and `aYbXc`. However, `.([XY]).\1.` will only match `aXbXc` and `aYbYc`.

Related reference

“Metacharacters” on page 145

Appendix D. ObjectServer tables and data types

This appendix contains ObjectServer database table information.

alerts.status table

The alerts.status table contains status information about problems that have been detected by probes.

The following table describes the columns in the alerts.status table.

Table 43. Columns in the alerts.status table

Column name	Data type	Mandatory	Description
Identifier	varchar(255)	Yes	<p>Controls ObjectServer deduplication. The Identifier field controls the deduplication feature of the ObjectServer, and also supports compatibility with the GenericClear automation by ensuring resolution events are properly inserted into the ObjectServer and not deduplicated with their respective problem events.</p> <p>The following identifier correctly identifies repeated events in a typical environment:</p> <pre>@Identifier=@Node+ " "+@AlertKey+ "+@AlertGroup+ " "+@Type+ " "+@Agent+ "+@Manager</pre> <p>Additional information might need to be appended to the Identifier field to ensure correct deduplication and compatibility with the GenericClear automation. For example, if an SNMP specific trap contains a status enumeration value in one of its variable bindings, the specific trap number and the value of the relevant varbind must be appended to the Identifier field as follows:</p> <pre>@Identifier=@Node + " "+ @AlertKey+ "+@AlertGroup+ " "+@Type+ " "+@Agent+ "+@Manager+ " "+\$specific-trap+ "+\$2</pre>
Serial	incr	Yes	The Tivoli Netcool/OMNIBus serial number for the row.

Table 43. Columns in the *alerts.status* table (continued)

Column name	Data type	Mandatory	Description
Node	varchar(64)	Yes	<p>Identifies the managed entity from which the alarm originated. This could be a device or host name, service name, or other entity.</p> <p>For IP network devices or hosts, the Node column contains the resolved name of the device or host. In cases where the name cannot be resolved, the Node column must contain the IP address of the device or host.</p> <p>For non-IP network devices or hosts, alarms must contain similar information to the IP device or host. That is, the Node column must contain the name of the device or host which allows direct communication, or can be resolved to allow direct communication, with the device or host.</p>
NodeAlias	varchar(64)	No	<p>The alias for the node. For network devices or hosts, this should be the logical (layer-3) address of the entity. For IP devices or hosts, this must be the IP address.</p> <p>For non-IP devices or hosts, there are several addressing schemes that could be used. When selecting a value for the NodeAlias field, the value should allow for direct communication with the device or host. For example, a device managed by TL-1. The NodeAlias field may be populated by a lookup table or Netcool/Impact policy, with the IP address and port number of the terminal server through which the TL-1 device can be reached.</p>
Manager	varchar(64)	Yes	<p>The descriptive name of the probe that collected and forwarded the alarm to the ObjectServer. This can also be used to indicate the host on which the probe is running. Ideally this is set in the properties file of the probe, however the rules file should check to ensure it is set correctly, and modify if necessary.</p> <p>For example, the following syntax can be used to define the Manager field:</p> <pre>@Manager="MTTrapd Probe on" + hostname()</pre>

Table 43. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
Agent	varchar(64)	No	<p>The descriptive name of the sub-manager that generated the alert.</p> <p>Probes which process SNMP traps must set the Agent field to either the name of the vendor or the standards body which defined the trap, and provide a description of the MIB, or MIB Definition Name, where the trap is defined. It must be presented in the following format: vendor-MIB description</p> <p>For example::</p> <p>Cisco-Accounting Control, Cisco-Health Monitor, IETFBRIDGEMIB, ATMF-ATM-FORUM-MIB</p> <p>Optionally, vendor-specific information, such as device model numbers, can be appended to the Agent field for vendor-specific implementations of standard traps.</p> <p>The Syslog probe should set the Agent field to the name of the vendor which defined the received message, and provide any logical description for the family of messages to which the received message belongs.</p> <p>For example, Cisco defines messages received from IOS-based devices in separate documentation from messages received from the PIX Firewall. The format of the messages is also slightly different. Therefore the Syslog messages received from Cisco will have the Agent field set to either Cisco-IOS or Cisco- PIX Firewall.</p> <p>The TL-1 TSM should set the Agent field to the name of the vendor which defined the received message, and provide any logical description for the family of messages to which the received message belongs.</p>
AlertGroup	varchar(255)	No	<p>The descriptive name of the failure type indicated by the alert. For example:</p> <p>Interface Status or CPU Utilization).</p> <p>The AlertGroup field must contain the same value for related problem and resolution events.</p> <p>For example, SNMP trap 2 (linkDown) and trap 3 (linkUp) must both contain the same AlertGroup value of Link Status.</p> <p>The AlertGroup field for a TL-1 message will be set to the value of the message's alarm type.</p>

Table 43. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
AlertKey	varchar(255)	Yes	<p>The descriptive key that indicates the managed object instance referenced by the alert. For example, the disk partition indicated by a file system full alert or the switch port indicated by a utilization alert.</p> <ul style="list-style-type: none"> SNMP Trap-related probes: Probes that process SNMP traps should set the AlertKey field to one of the following values (in order of preference): <ul style="list-style-type: none"> The SNMP instance of the managed object which is represented by the alarm. This is normally obtained by extracting the instance from the OID of one of the variable bindings of the trap. Additionally, it might also be contained in a combination of one or more of the trap's variable binding values. For example, the first variable binding of a linkDown trap contains the ifIndex value (interface number) of the interface which failed. The AlertKey can be set with either of the following: <pre>@AlertKey = extract(\$OID1, "\.([0-9]+)\$")</pre> <pre>@AlertKey = \$1</pre> A textual description of the instance derived from the trap name or trap description. For example, a device with two power supplies (A and B) might be able to send two separate specific traps, without variable bindings, to indicate the failed status of either power supply. The appropriate power supply instance would need to be derived from the trap definitions of the MIB and then encoded in the rules file: <pre>switch(\$specific-trap) { case "1": @AlertKey = "A" case "2": @AlertKey = "B" default: }</pre> A mixed combination of variable binding values and information derived from the trap name or trap description. Therefore, any instance information that is not available in the previous two methods, but is required to ensure correct deduplication and GenericClear compatibility.

Table 43. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
			<ul style="list-style-type: none"> The Syslog Probe: The Syslog Probe should set the AlertKey to a textual description of the instance derived from the log message text. Ideally this is a textual name of the same managed entity. For example: Nov 20 13:12:57 device.customer.net 195.180.208.193 19986: 37w0d: %LINK-3-UPDOWN: Interface FastEthernet0/13, changed state to down In the previous example, the AlertKey would be set to FastEthernet0/13 using the following syntax: @AlertKey = extract(\$Details, "Interface (.*), changed") TL-1 TSM: Typically the AlertKey field for a TL-1 message is set to the value of the message's alarm location.

Table 43. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
Severity	integer	Yes	<p>Indicates the alert severity level, which indicates how the perceived capability of the managed object has been affected. The color of the alert in the event list is controlled by the severity value:</p> <p>0: Clear. The Clear severity level indicates the clearing of one or more previously reported alarms. The alarms have either been cleared manually by a network operator, or automatically by a process which has determined the fault condition no longer exists. Automatic processes, for example the GenericClear Automation process, typically clear all alarms for a managed object (the AlertKey) that have the same Alarm Type and/or probable cause (the Alert Group).</p> <p>1: Indeterminate. The Indeterminate severity level indicates that the severity level cannot be determined. Additionally, all problem resolving alarms are initially defined as indeterminate until they have been correlated with problem indicating alarms (for example by the GenericClear Automation), when all correlated alarms are set to Clear.</p> <p>2: Warning. The Warning severity level indicates the detection of potential or impending service affecting faults. If necessary, a further investigation of the fault should be made to prevent it from becoming more serious.</p> <p>3: Minor. The Minor severity level indicates the existence of a non-service affecting fault condition. Corrective action should be taken to prevent it from becoming a more serious fault. This severity level may be reported, for example, when the detected alarm condition is not currently degrading the capacity of the managed object.</p> <p>4: Major. The Major severity level indicates that a service affecting condition has developed and corrective action is urgently required. This severity level may be reported, for example, when there is a severe degradation in the capability of the managed object, and its full capability must be restored.</p> <p>5: Critical. The Critical severity level indicates that a service affecting condition has occurred, and corrective action is immediately required. This severity level may be reported, for example, when a managed object is out of service, and its capability must be restored.</p>

Table 43. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
Summary	varchar(255)	Yes	<p>Contains text which describes the alarm condition and the affected managed object instance.</p> <ul style="list-style-type: none"> You must ensure that the information presented in the Summary field is concise and sufficiently detailed. The Summary field must contain, in parenthesis, a description of the managed object instance provided by the available alarm data. For example, a linkDown trap from a Cisco device will contain the ifDescr value in the 2nd variable binding. The text summary of such an event would be similar to: "Link Down (FastEthernet0/13)" For alarms that relate to thresholds containing the compared or threshold values, you should select one of the following formats based on the available data: <ul style="list-style-type: none"> No values provided: "Link Utilization High (BRI2/0:1)" Compared value name provided: "Link Utilization High: inOctets Exceeded Threshold (BRI2/0:1)" Compared value name and value provided: "Link Utilization High: inOctets, 7100, Exceeded Threshold (BRI2/0:1)" Threshold name provided: "Link Utilization High: inOctetsMax Exceeded (BRI2/0:1)" Threshold Value provided: "Link Utilization High: inOctetsMax, 7000, Exceeded (BRI2/0:1)" Compared value and threshold value provided: "Link Utilization High: 7100 Exceeded 7000 (BRI2/0:1)" Both names and values provided: "Link Utilization High: inOctets, 7100, Exceeded inOctetsMax,7000 (BRI2/0:1)"
StateChange	time	Yes	An automatically-maintained ObjectServer timestamp of the last insert or update of the alert from any source.
FirstOccurrence	time	Yes	The time in seconds (from midnight January 1, 1970) when this alert was created or when polling started at the probe.
LastOccurrence	time	Yes	The time when this alert was last updated at the probe.
InternalLast	time	Yes	The time when this alert was last updated at the ObjectServer.
Poll	integer	No	The frequency of polling for this alert in seconds.

Table 43. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
Type	integer	No	<p>The type of alarm, where type refers to the problem or resolution state of the Alarm. This field is important for the correct correlation of events by the GenericClear Automation. The following values are valid for the Type field:</p> <p>0: Type not set</p> <p>1: Problem</p> <p>2: Resolution</p> <p>3: Netcool/Visionary problem</p> <p>4: Netcool/Visionary resolution</p> <p>7: Netcool/ISMs new alarm</p> <p>8: Netcool/ISMs old alarm</p> <p>11: More Severe</p> <p>12: Less Severe</p> <p>13: Information</p> <p>Some scenarios cannot be categorized as either a Problem or Resolution. For example, events which are increasingly becoming an issue but do not currently represent a failure, and events which are becoming less of an issue but do not currently indicate the failure has been completely resolved. In which case, the Type field must be set to Problem, More Severe or Less Severe to maintain compatibility with the GenericClear Automation.</p> <p>For example, the following rule file logic is used for handling traps associated with BGP Peer Connection Status:</p> <pre> switch (\$bgpPeerState) { case "1": ### idle @Severity = 4 @Type = 1 case "2": ### connect @Severity = 2 @Type = 12 case "3": ### active @Severity = 2 @Type = 12 case "4": ### opensent @Severity = 2 </pre>

Table 43. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
			<pre>@Type = 12 case "5": ### openconfirm @Severity = 2 @Type = 12 case "6": ### established @Severity = 1 @Type = 2 default: @Severity = 2 @Type = 1 }</pre>
Tally	integer	Yes	Automatically-maintained count of the number of inserts and updates of the alert from any source. This count is affected by deduplication.
Class	integer	Yes	The alert class used to identify the probe or vendor from which the alert was generated. Controls the applicability of context-sensitive event list tools.
Grade	integer	No	Indicates the state of escalation for the alert: 0: Not Escalated 1: Escalated
Location	varchar(64)	No	Indicates the physical location of the device, host, or service for which the alert was generated.
OwnerUID	integer	Yes	The user identifier of the user who is assigned to handle this alert. The default is 65534, which is the identifier for the nobody user.
OwnerGID	integer	No	The group identifier of the group that is assigned to handle this alert. The default is 0, which is the identifier for the public group.
Acknowledged	integer	Yes	Indicates whether the alert has been acknowledged: 0: No 1: Yes Alerts can be acknowledged manually by a network operator or automatically by a correlation or workflow process.
Flash	integer	No	Enables the option to make the event list flash.
EventId	varchar(255)	No	The event ID (for example, SNMPTRAP-link down). Multiple events can have the same event ID. The event ID is populated by the probe rules file and used by IBM Tivoli Network Manager IP Edition.
ExpireTime	integer	Yes	The number of seconds from the time this alert was last received by the ObjectServer (LastOccurrence) until it is cleared automatically. Used by the Expire automation.

Table 43. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
ProcessReq	integer	No	Indicates whether the alert should be processed by IBM Tivoli Network Manager IP Edition. This is populated by the probe rules file and used by IBM Tivoli Network Manager IP Edition.
SuppressEscl	integer	Yes	Used to suppress or escalate the alert: 0: Normal 1: Escalated 2: Escalated-Level 2 3: Escalated-Level 3 4: Suppressed 5: Hidden 6: Maintenance The suppression level is manually selected by operators from the event list.
Customer	varchar(64)	No	The name of the customer affected by this alert.
Service	varchar(64)	No	The name of the service affected by this alert.
PhysicalSlot	integer	No	The slot number indicated by the alert.
PhysicalPort	integer	No	The port number indicated by the alert.
PhysicalCard	varchar(64)	No	The card name or description indicated by the alert.
TaskList	integer	Yes	Indicates whether a user has added the alert to the Task List: 0: No 1: Yes Operators can add alerts to the Task List from the event list.
NmosSerial	varchar(64)	No	The serial number of the event that is suppressing the current event. Populated by IBM Tivoli Network Manager IP Edition.
NmosObjInst	integer	No	Populated by IBM Tivoli Network Manager IP Edition during alert processing.
NmosCauseType	integer	No	The alert state, populated by IBM Tivoli Network Manager IP Edition as an integer value: 0: Unknown 1: Root cause 2: Symptom

Table 43. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
NmosDomainName	varchar(64)	No	<p>The name of the IBM Tivoli Network Manager IP Edition domain that is managing the event.</p> <p>By default, this column is populated only for events that are generated by IBM Tivoli Network Manager IP Edition polls. To populate this column for other event sources such as probes, you must modify the rules files.</p>
NmosEntityId	integer	No	<p>A unique numerical ID that identifies the IBM Tivoli Network Manager IP Edition topology entity with which the event is associated.</p> <p>This column is similar to the NmosObjInst column, but is more granular. For example, the NmosEntityId value can represent the ID of an interface within a device.</p>
NmosManagedStatus	integer	No	<p>The managed status of the network entity for which the event was raised. Can apply to events from IBM Tivoli Network Manager IP Edition and from any probe.</p> <p>You can use this column to filter out events from interfaces that are not considered relevant.</p>
NmosEventMap	varchar(64)	No	<p>Contains the required IBM Tivoli Network Manager IP Edition V3.9 or later, eventMap name and optional precedence for the event, which indicates how IBM Tivoli Network Manager IP Edition should process the event.</p> <p>The optional precedence number can be concatenated to the end of the value, following a period (.). If the precedence is not supplied, it is set to 0. The following examples show the configuration for an event map with an explicit event precedence of 900, and another where the precedence defaults to 0:</p> <ul style="list-style-type: none"> • ItnmLinkdownIfIndex.900 • PrecisionMonitorEvent
LocalNodeAlias	varchar(64)	Yes	The alias of the network entity indicated by the alert. For network devices or hosts, this is the logical (layer-3) address of the entity, or another logical address that enables direct communication with the device. For use in managed object instance identification.
LocalPriObj	varchar(255)	No	The primary object referenced by the alert. For use in managed object instance identification.
LocalSecObj	varchar(255)	No	The secondary object referenced by the alert. For use in managed object instance identification.
LocalRootObj	varchar(255)	Yes	An object that is equivalent to the primary object referenced in the alarm. For use in managed object instance identification.
RemoteNodeAlias	varchar(64)	Yes	The network address of the remote network entity. For use in managed object instance identification.

Table 43. Columns in the *alerts.status* table (continued)

Column name	Data type	Mandatory	Description
RemotePriObj	varchar(255)	No	The primary object of a remote network entity referenced by an alarm. For use in managed object instance identification.
RemoteSecObj	varchar(255)	No	The secondary object of a remote network entity referenced by an alarm. For use in managed object instance identification.
RemoteRootObj	varchar(255)	Yes	An object that is equivalent to the remote entity's primary object referenced in the alarm. For use in managed object instance identification.
X733EventType	integer	No	Indicates the alert type: 0: Not defined 1: Communications 2: Quality of Service 3: Processing error 4: Equipment 5: Environmental 6: Integrity violation 7: Operational violation 8: Physical violation 9: Security service violation 10: Time domain violation
X733ProbableCause	integer	No	Indicates the probable cause of the alert.
X733SpecificProb	varchar(64)	No	Identifies additional information for the probable cause of the alert. Used by probe rules files to specify a set of identifiers for use in managed object instance identification.
X733CorrNotif	varchar(255)	No	A listing of all notifications with which this notification is correlated.
ServerName	varchar(64)	Yes	The name of the originating ObjectServer. Used by gateways to control propagation of alerts between ObjectServers.
ServerSerial	integer	Yes	The serial number of the alert on the originating ObjectServer (if it did not originate on this ObjectServer). Used by gateways to control the propagation of alerts between ObjectServers.
URL	varchar(1024)	No	Optional URL which provides a link to additional information in the vendor's device or ENMS.

Table 43. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
ExtendedAttr	varchar(4096)	No	<p>Holds name-value pairs (of Tivoli Enterprise Console® extended attributes) or any other additional information for which no dedicated column exists in the alerts.status table.</p> <p>Use this column only through the nvp_get, nvp_set, and nvp_exists SQL functions.</p> <p>An example of a name-value string is: Region="EMEA";host="sf01392w"; Error="errno=32: ""Broken pipe"""</p> <p>In this example, the Region attribute has a value of EMEA, the host attribute has a value of sf01392w, and the Error attribute has a value of errno=32: "Broken pipe".</p> <p>Notice that quotation marks are escaped by doubling them, as shown with the Error attribute value.</p> <p>In name-value pairs, the value is always enclosed in quotation marks (" ") and embedded quotation marks are escaped by doubling them. The separator between name-value pairs is a semicolon (;). No whitespace is allowed around the equal sign (=) or semicolon.</p> <p>Note: The column can hold only 4096 bytes, so there will be fewer than 4096 characters if multi-byte characters are used.</p>
OldRow	integer	No	<p>Maintains the local state of the row in each ObjectServer during resynchronization in the failover pair. This column must not be added to the gateway mapping files.</p> <p>The value of OldRow is changed to 1 in the destination ObjectServer for the duration of resynchronization if the Gate.ResyncType property of the gateway is set to Minimal.</p> <p>The default is 0.</p>
ProbeSubSecondId	integer	No	<p>For those alerts that a probe sends within the same one-second interval, and which therefore have the same LastOccurrence value, an incremental value, starting at 1, is added to the ProbeSubSecondId field to differentiate the LastOccurrence time. The default is 0.</p>
MasterSerial	integer	No	<p>Identifies the master ObjectServer if this alert is being processed in a desktop ObjectServer environment.</p> <p>This column is added when you run the database initialization utility nco_dbinit with the -desktopserver option.</p> <p>Note: MasterSerial must be the last column in the alerts.status table if you are using a desktop ObjectServer environment.</p>

Table 43. Columns in the alerts.status table (continued)

Column name	Data type	Mandatory	Description
BSM_Identity	varchar(1024)	No	The unique identifier of the resource from where the event originates, and is used to correlate the event to that resource in IBM Tivoli Business Service Manager (TBSM).

Note: You can display only columns of type CHAR, VARCHAR, INCR, INTEGER, and TIME in the event list. Do not add columns of any other type to the alerts.status table.

alerts.details table

The alerts.details table contains the detail attributes of the alerts in the system.

The following table describes the columns in the alerts.details table.

Table 44. Columns in the alerts.details table

Column name	Data type	Description
KeyField	varchar(255)	<p>Internal sequencing string for uniqueness.</p> <p>The Keyfield value is composed of an Identifier value plus four # plus a sequence number starting at a count of 1; for example:</p> <p><i>Identifier####1</i></p> <p>Where <i>Identifier</i> is a data type of varchar(255), which is used to relate details to entries in the alerts.status table.</p> <p>If the Identifier value is over a certain length, there is a possibility that the Keyfield value could exceed its defined 255 limit, resulting in truncation of the sequence number. Keyfield values could therefore no longer be unique, and the unintended duplication could cause inserts into the alerts.details table to fail.</p> <p>Tip: To prevent an overflow in KeyField (and ensure uniqueness), the length of the Identifier value must be sufficiently less than 255 to allow the four # and a sequence number (of one or more digits) to be appended.</p>
Identifier	varchar(255)	<p>Identifier to relate details to entries in the alerts.status table.</p> <p>The Identifier is used to compute the Keyfield value, and is required to be less than a certain length to ensure that each computed Keyfield value remains unique. For guidelines on the maximum length of the Identifier value, see the tip in the preceding KeyField row.</p>
AttrVal	integer	Boolean; when false (0), just the Detail column is valid. Otherwise, the Name and Detail columns are both valid.
Sequence	integer	Sequence number, used for ordering entries in the event list Event Information window.
Name	varchar(255)	Name of attribute stored in the Detail column.
Detail	varchar(255)	Attribute value.

alerts.journal table

The alerts.journal table provides a history of work performed on alerts.

The following table describes the columns in the alerts.journal table.

Table 45. Columns in the alerts.journal table

Column name	Data type	Description
KeyField	varchar(255)	Primary key for table.
Serial	integer	Serial number of alert that this journal entry is related to.
UID	integer	User identifier of user who made this entry.
Chrono	time	Time and date that this entry was made.
Text1	varchar(255)	First block of text for journal entry.
Text2	varchar(255)	Second block of text for journal entry.
Text3	varchar(255)	Third block of text for journal entry.
Text4	varchar(255)	Fourth block of text for journal entry.
Text5	varchar(255)	Fifth block of text for journal entry.
Text6	varchar(255)	Sixth block of text for journal entry.
Text7	varchar(255)	Seventh block of text for journal entry.
Text8	varchar(255)	Eighth block of text for journal entry.
Text9	varchar(255)	Ninth block of text for journal entry.
Text10	varchar(255)	Tenth block of text for journal entry.
Text11	varchar(255)	Eleventh block of text for journal entry.
Text12	varchar(255)	Twelfth block of text for journal entry.
Text13	varchar(255)	Thirteenth block of text for journal entry.
Text14	varchar(255)	Fourteenth block of text for journal entry.
Text15	varchar(255)	Fifteenth block of text for journal entry.
Text16	varchar(255)	Sixteenth block of text for journal entry.

service.status table

The service.status table is used to control the additional features required to support IBM® Tivoli Composite Application Manager for Internet Service Monitoring.

The following table describes the columns in the service.status table.

Table 46. Columns in the service.status table

Column name	Data type	Description
Name	varchar(255)	Name of the service.
CurrentState	integer	Indicates the state of the service: 0: Good 1: Bad 2: Marginal 3: Unknown

Table 46. Columns in the service.status table (continued)

Column name	Data type	Description
StateChange	time	Indicates the last time the service state changed.
LastGoodAt	time	Indicates the last time the service was Good (0).
LastBadAt	time	Indicates the last time the service was Bad (1).
LastMarginalAt	time	Indicates the last time the service was Marginal (2).
LastReportAt	time	Time of the last service status report.

ObjectServer data types

Each column value in the ObjectServer has an associated data type. The data type determines how the ObjectServer processes the data in the column.

For example, the plus operator (+) adds integer values or concatenates string values, but does not act on Boolean values. The data types supported by the ObjectServer are listed in the following table:

Table 47. ObjectServer data types

SQL type	Description	Default value	ObjectServer ID for data type
INTEGER	32-bit signed integer.	0	0
INCR	32-bit unsigned auto-incrementing integer. Applies to table columns only, and can only be updated by the system.	0	5
UNSIGNED	32-bit unsigned integer.	0	12
BOOLEAN	TRUE or FALSE.	FALSE	13
REAL	64-bit signed floating point number.	0.0	14
TIME	Time, stored as the number of seconds since midnight January 1, 1970. This is the Coordinated Universal Time (UTC) international time standard.	Thu Jan 1 01:00:00 1970	1
CHAR(integer)	Fixed size character string, <i>integer</i> characters long (8192 Bytes is the maximum). The char type is identical in operation to varchar, but performance is better for mass updates that change the length of the string.	' '	10
VARCHAR(integer)	Variable size character string, up to <i>integer</i> characters long (8192 Bytes is the maximum). The varchar type uses less storage space than the char type and the performance is better for deduplication, scanning, insert, and delete operations.	' '	2
INTEGER64	64 bit signed integer.	0	16
UNSIGNED64	64 bit unsigned integer.	0	17

Note: You can display only columns of type CHAR, VARCHAR, INCR, INTEGER, and TIME in the event list. Do not add columns of any other type to the

alerts.status table.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
958/NH04
IBM Centre, St Leonards
601 Pacific Hwy
St Leonards, NSW, 2069
Australia

IBM Corporation
896471/H128B
76 Upper Ground
London SE1 9PZ
United Kingdom

IBM Corporation
JBF1/SOM1
294 Route 100
Somers, NY, 10589-0100
United States of America

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

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

Trademarks

AIX, IBM, the IBM logo, ibm.com[®], Informix, Netcool, System z, Tivoli, and Tivoli Enterprise Console are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Adobe, Acrobat, Portable Document Format (PDF), PostScript, and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.



Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

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

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

- \$ symbol
 - in probe rules files 17
- @ symbol
 - in gateway mappings 100
 - in probe rules files 8, 17
- @Identifier 8, 9
- @Tally 9
- % symbol
 - in probe rules files 18

A

- accessibility viii
- ADD ROUTE gateway command 100, 117
- alerts.details table 166
- alerts.journal table 167
- alerts.status table 153
- anomalous event rates
 - configuring 58
- API probes 3
- arch
 - operating system directory viii
- arithmetic functions
 - in probe rules files 36
- arithmetic operators
 - in probe rules files 31
- atoms
 - description 143
- audience v

B

- backslash sequences
 - regular expressions 149
- bidirectional gateways 90, 91
- bit manipulation operators
 - in probe rules files 31
- BOOLEAN data type 168
- bracket expressions
 - regular expressions 148

C

- CHAR data type 168
- command line options
 - gateways 109
 - probes 77
- comparison operators
 - in probe rules files 32
- configuration commands
 - gateways 118
- configuration files
 - gateways 98
- configuring
 - anomalous event rates 58
 - event flood 58
 - probe statistics 68

- conventions, typeface viii
- CORBA probes 3
- correlation of events 9
- COUNTERPART attribute in
 - gateways 94
- CREATE FILTER gateway
 - command 101, 116
- CREATE MAPPING gateway
 - command 115

D

- data types 168
- database probes 3
- date functions
 - in probe rules files 37
- debugging
 - probes 13, 130
 - rules files 53
- deduplication 9, 41
- deleting
 - elements in probe rules files 33
- details function
 - in probe rules files 41
- device probes 2
- DROP FILTER gateway command 117
- DROP MAPPING gateway
 - command 115
- DUMP CONFIG gateway
 - command 107, 118

E

- editing
 - probe properties 73
- education
 - see Tivoli technical training viii
- elements
 - in probe rules files 17
- encrypting
 - passwords for gateway target systems 97
 - passwords for the ObjectServer 13, 96
- environment variables, notation viii
- error messages
 - gateways 135
 - probes 123
- event flood
 - configuring 58
- exists function
 - in probe rules files 33

F

- fields
 - Identifier 9
 - in probe rules files 17
 - Tally 9

- filters
 - commands 116
 - in gateways 101
- flood configuration rules file 59
- flood rules file 62
- flood.config.rules 59
- flood.rules 62
- functions
 - rules files 28

G

- gateways
 - ADD ROUTE command 117
 - bidirectional 90, 91
 - command line options 109
 - configuration commands 118
 - COUNTERPART attribute 94
 - CREATE FILTER command 116
 - CREATE MAPPING command 115
 - DROP FILTER command 117
 - DROP MAPPING command 115
 - DUMP CONFIG command 118
 - dumping configurations
 - interactively 107
 - encrypting target system
 - passwords 97
 - error messages 135
 - filter commands 116
 - filter description 101
 - general commands 119
 - LOAD CONFIG command 118
 - LOAD FILTER command 117
 - loading configurations
 - interactively 107
 - log files 102
 - mapping commands 115
 - mapping description 100
 - overview 89
 - reader commands 111
 - reader description 94, 99
 - reader/writer modules 94
 - REMOVE ROUTE command 117
 - route commands 117
 - route description 95, 100
 - SAVE CONFIG command 118
 - saving configurations
 - interactively 107
 - secure mode 96
 - SET CONNECTIONS command 119
 - SET DEBUG MODE command 120
 - SHOW MAPPING ATTRIBUTES
 - command 116
 - SHOW MAPPING command 116
 - SHOW READERS command 112
 - SHOW ROUTES command 118
 - SHOW SYSTEM command 119
 - SHOW WRITER ATTRIBUTES
 - command 114
 - SHOW WRITER TYPES
 - command 114

gateways (*continued*)

- SHOW WRITERS command 113
- SHUTDOWN command 119
- START READER command 111
- START WRITER command 113
- STOP READER command 112
- STOP WRITER command 113
- store-and-forward mode 96
- TRANSFER command 120
- types 90
 - unidirectional 90, 93
 - writer commands 112
 - writer description 99

Generic probe 13

genevent 43, 44

I

Identifier field 9

IDUC 111

IF statements in rules files 26

include files

- in probe rules files 27

INCR data type 168

INTEGER data type 168

INTEGER64 data type 168

L

LOAD CONFIG gateway command 107, 118

LOAD FILTER gateway command 117

log file probes 2

log function

- in probe rules files 42

logical operators

- in probe rules files 32

lookup tables 39

- in probe rules files 39

M

manuals vi

mappings

- commands 115
- in gateways 100

math functions

- in probe rules files 36

math operators

- in probe rules files 31

messagelevel command line option 53

messagelog command line option 53

metacharacters

- regular expressions 145

minimal quantifiers

- regular expressions 147

miscellaneous probes 4

multicultural constructs

- regular expressions 149

multithreaded processing 46

N

nco_aes_crypt 13, 97

nco_g_crypt 13, 96, 97

nco_objserv 73

ncoadmin user group 106, 109

NETCOOL regular expression library 143

non-greedy quantifiers

- regular expressions 147

O

ObjectServer

- data types 168

ObjectServer tables

- alerts.details 166
- alerts.journal 167
- alerts.status 153
- service.status 167

ON INSERT ONLY flag in gateways 100

online publications vi

operating system directory

- arch viii

operators

- rules files 28

ordering publications vi

P

password encryption 13, 96

peer-to-peer failover mode

- probes 14

Ping probe 5

probe rules language

- reserved words 51

probe self monitoring

- resources 66

probe statistics

- configuring 68

probes

- anomalous event rates 58
- API 3
- arithmetic functions in rules files 36
- arithmetic operators in rules files 31
- bit manipulation operators in rules files 31
- command line options 77
- comparison operators in rules files 32
- components 4
- CORBA 3
- customizations 57
- database 3
- date functions in rules files 37
- debugging 13, 130
- debugging rules files 53
- deduplication in rules files 41
- deleting elements in rules files 33
- details function in rules files 41
- device 2
- editing properties 73
- elements in rules files 17
- error messages 123
- event flood detection 58
- executable file 4
- fields in rules files 17
- Identifier field 9
- IF statements in rules files 26
- include files in rules files 27

probes (*continued*)

- log file 2
- log function in rules files 42
- logical operators in rules files 32
- lookup tables 39
- lookup tables in rules files 39
- math functions in rules files 36
- math operators in rules files 31
- metric data collection 68
- miscellaneous 4
- operation 10
- overview 1
- peer-to-peer failover mode 14
- properties 5
- properties file 5
- properties in rules files 18
- raw capture 13
- rules file 6
- rules file processing 17
- search and replace function in rules files 47
- secure mode 13
- self monitoring 64, 68
- self monitoring setup 65
- service function in rules files 49
- setlog function in rules files 42
- store and forward 10
- string functions in rules files 33
- string operators in rules files 31
- SWITCH statement in rules files 26
- temporary elements in rules files 18
- testing rules files 53
- time functions in rules files 37
- troubleshooting 130
- types 2
- update function in rules files 41
- using a specific probe 7

properties

- in probe rules files 18
- probes 77

publications vi

R

raw capture mode in probes 13

readers

- commands 111
- in gateways 94, 99

REAL data type 168

RegexpLibrary property 143

registertarget 43

regular expressions

- atoms 143
- backslash sequences 149
- bracket expressions 148
- metacharacters 145
- minimal quantifiers 147
- multicultural constructs 149
- NETCOOL library 143
- non-greedy quantifiers 147
- overview 143
- RegexpLibrary property 143
- TRE library 145

REMOVE ROUTE gateway command 117

reserved words

- probe rules language 51

- routes
 - commands 117
 - in gateways 95, 100
- rules file processing 17
 - bit manipulation operators 31
 - comparison operators 32
 - date functions 37
 - deduplication 9
 - deleting elements 33
 - details function 41
 - exists function 33
 - IF statements 26
 - log function 42
 - logical operators 32
 - lookup tables 39
 - math functions 36
 - math operators 31
 - rules file examples 54
 - search and replace function 47
 - setlog function 42, 49
 - string functions 33
 - string operators 31
 - SWITCH statement 26
 - time functions 37
 - update function 41
- rules files 53
 - functions 28
 - operators 28

S

- SAVE CONFIG gateway command 107, 118
- saving
 - gateway configurations
 - interactively 107
- search and replace function
 - in probe rules files 47
- secure mode
 - for gateways 96
 - for probes 13
- self monitoring
 - probes 64
- service function
 - in probe rules files 49
- service.status table 167
- SET CONNECTIONS gateway
 - command 106, 119
- SET DEBUG MODE gateway
 - command 120
- setdefaulttarget 43
- setlog function
 - in probe rules files 42
- settarget 43
- SHORT data type 168
- SHOW MAPPING ATTRIBUTES gateway
 - command 116
- SHOW MAPPINGS gateway
 - command 116
- SHOW READERS gateway
 - command 112
- SHOW ROUTES gateway command 118
- SHOW SYSTEM gateway command 107, 119
- SHOW WRITER ATTRIBUTES gateway
 - command 114

- SHOW WRITER TYPES gateway
 - command 114
- SHOW WRITERS gateway
 - command 113
- SHUTDOWN gateway command 119
- START READER gateway command 99, 111
- START WRITER gateway command 99, 113
- STOP gateway command 107
- STOP READER gateway command 112
- STOP WRITER gateway command 99, 113
- store-and-forward mode
 - in gateways 96
 - in probes 10
- string functions
 - in probe rules files 33
- string operators
 - in probe rules files 31
- support information viii
- SWITCH statement in rules files 26

T

- Tally field 9
- temporary elements
 - in probe rules files 18
- testing
 - rules files 53
- time functions
 - in probe rules files 37
- Tivoli software information center vi
- Tivoli technical training viii
- training, Tivoli technical viii
- TRANSFER gateway command 120
- TRE regular expression library 145
- troubleshooting
 - gateways 135
 - probes 130
- typeface conventions viii

U

- unidirectional gateways 90
- UNSIGNED data type 168
- UNSIGNED64 data type 168
- update function
 - in probe rules files 41
- UTC data type 168

V

- VARCHAR data type 168
- variables, notation for viii

W

- writers
 - commands 112
 - in gateways 99



Printed in the Republic of Ireland

SC14-7608-00

