IBM® Tivoli® Netcool/OMNIbus Probe for Message Bus
Version 13.0

Reference Guide
September 25, 2020

SC27-8701-14

Note

Before using this information and the product it supports, read the information in Appendix A, “Notices and Trademarks,” on page 101.

Edition notice

This edition (SC27-8701-14) applies to version 13.0 of IBM Tivoli Netcool/OMNIbus Probe for Message Bus and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC27-8701-13.

© Copyright International Business Machines Corporation 2015, 2020.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

About this guide ..... **v**
Document control page v
Conventions used in this guide x

Chapter 1. Probe for Message Bus **1**
Summary 1
Installing probes 4
Migrating to the Probe for Message Bus 4
Identifying new and changed features of the Probe for Message Bus 4
Configuring the probe 7
Enabling the max_line_length property 8
Enabling the max_http_payload_size property 8
Configuring the parser with a different JsonMessageDepth 10
Generating events from a nested JSON 11
Specifying multiple parser configurations to parse different JSON structures 12
Upgrading to Probe for Message Bus version 5 or newer 15
Using the transport module 17
Configuring the transport properties files 18
Configuring the JMS transport 18
Configuring the data file transport 21
Configuring the MQTT transport 21
Configuring the Web Socket transport 22
Configuring the Web Hook transport 32
Configuring the Cometd transport 40
Configuring the socket transport 42
Configuring the Kafka transport 43
Message Bus Probe integrations with event sources 47
Probe integration for Amazon Web Services 48
Probe integration for Ciena Blue Planet MCP 50
Probe integration for IBM Cloud Platform Common Services (CS) Monitoring 52
Probe integration for IBM Event Streams for IBM Cloud 58
Probe integration for iDirect Pulse 60
Probe integration for Kafka 61
Probe integration for Microsoft Azure Monitoring 63
Probe integration for Nokia 1350 OMS 65
Probe integration for Nokia NSP 66
Using the transformer module 66
Using XSLT files to transform events 66
Using the transformer testing tool 68
Configuring the transformer definition file 69
Using the XML validation tool 71
Running the probe 71
Data acquisition 72
Peer-to-peer failover functionality 72
HTTP/HTTPS command interface 72
Properties and command line options 74
Properties and command line options provided by the Java Probe Integration Library (probe-sdk-java) version 11.0 80
Elements 83
Error messages 87

Common error messages....88
ProbeWatch messages....92
Using the probe with the Gateway for Message Bus....92
Requirements....92
Sample implementation using JMS....93
Frequently asked questions....94
Troubleshooting....94
Known issues with the Probe for Message Bus....95

Appendix A. Notices and Trademarks....101
Notices....101
Trademarks....102

About this guide

The following sections contain important information about using this guide.

Document control page

Table 1. Document modification history

Document version	Publication date	Comments
SC27-8701-00	December 10, 2015	First IBM publication. The Probe for Message Bus addresses the following enhancement requests: **RFE 59587**: Request to provide filtering support on the Java Message Service (JMS) Transport Module. **RFE 71751**: Request to add support for the REST API protocol within the JMS Transport Module. **RFE 63631**: Request to enhance the event parser to process JSON messages.
SC27-8701-01	July 28, 2016	Version updated to 2.0 in “Summary” on page 1. Updated “Elements” on page 83 with new element `$resync_event`. Updated “Enabling the max_line_length property” on page 8 and “Enabling the max_http_payload_size property” on page 8 command from %JAVA_USED% to %JAVA_EXEC_USED%. Updated “Properties and command line options” on page 74. Added descriptions for the following property: **`JsonNestedPayload`**. Version 2.0 of the probe addresses the following enhancement request: **RFE 80771**: Request to enhance the Probe for Message Bus for JSON parsing.

© Copyright IBM Corp. 2015, 2020

Table 1. Document modification history (continued)

Document version	Publication date	Comments
SC27-8701-02	November 24, 2016	Version updated to 3.0 in "Summary" on page 1. The following topics added: • "Authenticating the probe using REST or WebSocket" on page 29 • "Resynchronizing the probe with the REST API" on page 31 • "Subscribing to receive notifications using WebSocket" on page 31 • "Restarting the probe and re-connecting with the persistent URI" on page 31 • "Configuring the parser with a different JsonMessageDepth" on page 10 • "Generating events from a nested JSON" on page 11 Descriptions for the following WebSocket transport properties were added to "Configuring the Web Socket transport" on page 22: • **httpHeaders** • **refreshRetryCount** • **securityProtocol** • **webSocketRefreshInterval** • **webSocketRefreshMessage** • **webSocketPeristentURI** Descriptions for the following probe properties were added to "Properties and command line options" on page 74: • **JsonMessageDepth** • **JsonNestedHeader** • **MessageHeader** • **RecordData**
SC27-8701-03	March 14, 2017	Version updated to 4.0 in "Summary" on page 1. Support for the **disconnectProbe** and **shutdownProbe** commands using HTTP added. See "HTTP/HTTPS command interface" on page 72. "Properties and command line options provided by the Java Probe Integration Library (probe-sdk-java) version 11.0" on page 80 updated to reflect the probe's usage of version 11 of the Probe Framework. Error message added to "Error messages" on page 87. Issue added to "Known issues with the Probe for Message Bus" on page 95.

Table 1. Document modification history (continued)

Document version	Publication date	Comments
SC27-8701-04	July 20, 2017	Version updated to 5.0 in “Summary” on page 1. Added support for Ciena's Blue Planet MCP Release 17.02. “Probe integration for Ciena Blue Planet MCP” on page 50 added. Description of the deprecated **WebSocketId** property removed from “Properties and command line options” on page 74. Addition of the **keepTokens** property to the `restWebSocketTransport.properties` file to enable the probe to extract multiple attributes from the incoming JSON data. Addition of the **webSocketSubscribeMessage** property to the `restWebSocketTransport.properties` file to specify the message that the probe sends after successfully connecting through the WebSocket channel. Addition of the following properties to the `restWebSocketTransport.properties` file to enable the overriding of the global HTTP header settings: • **loginRequestHeaders** • **loginRefreshHeaders** • **logoutRequestHeaders** • **resyncRequestHeaders** • **subscribeRequestHeaders** • **subscribeRefreshHeaders** • **webSocketSubscribeMessage**
SC27-8701-05	August 18, 2017	Guide updated to describe how to upgrade to version 5 of the probe to make use of the enhanced JSON functionality. The following topics added: • “Upgrading to Probe for Message Bus version 5 or newer” on page 15 • “Migrating the probe parser configuration to the JSON parser configuration file” on page 15 • “Migrating the WebSocketID property to the Websocket transport properties file” on page 17 The following topics updated: • “Configuring the probe” on page 7 • “Example configuration” on page 13

Table 1. Document modification history (continued)

Document version	Publication date	Comments
SC27-8701-06	November 23, 2017	Version updated to 6.0 in "Summary" on page 1. "Using the transport module" on page 17 updated. WebSocket transport enhanced to support OAuth authentication and extended to replace the HTTP transport for alarm subscription and notification. The following topics added: • "Configuring the Web Hook transport" on page 32 • "Probe integration for IBM Cloud Platform Common Services (CS) Monitoring" on page 52 • "Authenticating the probe Using OAuth authentication" on page 29 • "Specifying a callback URL" on page 39 • "Subscribing to receive notifications using an HTTP server" on page 39 • "Configuring the Cometd transport" on page 40 • "Probe integration for Nokia 1350 OMS" on page 65 "Known issues with the Probe for Message Bus" on page 95 has been updated. The guide has also been reorganized for clarity.
SC27-8701-07	April 12, 2018	Version updated to 7.0 in "Summary" on page 1. Support added for Kafka. The following topic was added "Probe integration for Kafka" on page 61: "Probe integration for IBM Cloud Platform Common Services (CS) Monitoring" on page 52 updated. The following topics were added • "Configuring the Message Bus Probe to receive notifications from Prometheus" on page 52: • "Configuring Prometheus in Kubernetes from the command line" on page 53: • "Configuring the Message Bus Probe to receive notifications from Logstash" on page 56: • "Configuring Logstash in ICP from the command line" on page 57:

Table 1. Document modification history (continued)

Document version	Publication date	Comments
SC27-8701-08	October 11, 2018	Version updated to 8.0 in "Summary" on page 1. Updated the table in "Using the transport module" on page 17. Description for **webSocketHeaders** added to "Configuring the Web Socket transport" on page 22. Updated "Probe integration for Kafka" on page 61. Descriptions for **respondWithContent**, **validateBodySyntax**, **validateRequestURI**, and **idleTimeout** added to "Configuring the Web Hook transport" on page 32. Added the following topics: • "Configuring the Kafka transport" on page 43.
SC27-8701-09	February 28, 2019	Version updated to 9.0 in "Summary" on page 1. "Message Bus Probe integrations with event sources" on page 47 added. "Probe integration for Amazon Web Services" on page 48 and "Probe integration for Microsoft Azure Monitoring" on page 63 added. HTTP/HTTPS transport removed from the Message Bus Probe, the functionality being replaced by the Webhook Transport.
SC27-8701-10	August 29, 2019	Version updated to 10.0 in "Summary" on page 1. Probe updated to enable the transport properties file encryption mechanism. Descriptions for new Webhook transport properties added to "Configuring the Web Hook transport" on page 32 Running the probe with the Webhook transport and HTTP transport updated to provide event sender as a token, adding HTTP headers to be parsed into Message payload. **Note:** XML payload is currently not supported. Details about the integration with Nokia NSP moved to the new *Probe Integration for Nokia NSP Reference Guide*.

Table 1. Document modification history (continued)		
Document version	**Publication date**	**Comments**
SC27-8701-11	January 31, 2020	Version updated to 11.0 in "Summary" on page 1. "Message Bus Probe integrations with event sources" on page 47 updated. "Probe integration for Ciena Blue Planet MCP" on page 50 updated. "Probe integration for IBM Event Streams for IBM Cloud" on page 58 added. "Probe integration for Kafka" on page 61 updated. "Probe integration for Microsoft Azure Monitoring" on page 63 renamed. "Troubleshooting" on page 94 added.
SC27-8701-12	March 20, 2020	Version updated to 12.0 in "Summary" on page 1. "Probe integration for IBM Event Streams for IBM Cloud" on page 58 updated.
SC27-8701-13	April 30, 2020	"Properties and command line options" on page 74 updated.
SC27-8701-14	September 25, 2020	Version updated to 13.0 in "Summary" on page 1.

Note: The IBM Tivoli Netcool/OMNIbus Probe for Message Bus was previously documented in the *IBM® Tivoli® Netcool/OMNIbus Probe for Message Bus* reference guide (SC14-7649-04).

Conventions used in this guide

All probe guides use standard conventions for operating system-dependent environment variables and directory paths.

Operating system-dependent variables and paths

All probe guides use standard conventions for specifying environment variables and describing directory paths, depending on what operating systems the probe is supported on.

For probes supported on UNIX and Linux operating systems, probe guides use the standard UNIX conventions such as **$***variable* for environment variables and forward slashes (**/**) in directory paths. For example:

```
$OMNIHOME/probes
```

For probes supported only on Windows operating systems, probe guides use the standard Windows conventions such as **%***variable***%** for environment variables and backward slashes (****) in directory paths. For example:

```
%OMNIHOME%\probes
```

For probes supported on UNIX, Linux, and Windows operating systems, probe guides use the standard UNIX conventions for specifying environment variables and describing directory paths. When using the Windows command line with these probes, replace the UNIX conventions used in the guide with Windows conventions. If you are using the bash shell on a Windows system, you can use the UNIX conventions.

Note: The names of environment variables are not always the same in Windows and UNIX environments. For example, %TEMP% in Windows environments is equivalent to $TMPDIR in UNIX and Linux environments. Where such variables are described in the guide, both the UNIX and Windows conventions will be used.

Operating system-specific directory names

Where Tivoli Netcool/OMNIbus files are identified as located within an *arch* directory under NCHOME or OMNIHOME, *arch* is a variable that represents your operating system directory. For example:

$OMNIHOME/probes/*arch*

The following table lists the directory names used for each operating system.

Note: This probe may not support all of the operating systems specified in the table.

Table 2. Directory names for the arch variable

Operating system	**Directory name represented by *arch***
AIX® systems	aix5
Red Hat Linux® and SUSE systems	linux2x86
Linux for System z	linux2s390
Solaris systems	solaris2
Windows systems	win32

OMNIHOME location

Probes and older versions of Tivoli Netcool/OMNIbus use the OMNIHOME environment variable in many configuration files. Set the value of OMNIHOME as follows:

- On UNIX and Linux, set $OMNIHOME to $NCHOME/omnibus.
- On Windows, set %OMNIHOME% to %NCHOME%\omnibus.

Chapter 1. Probe for Message Bus

The IBM Tivoli Netcool/OMNIbus Probe for Message Bus can acquire XML and JSON events from various sources using Java™ Message Service (JMS), Webhook, Message Queue Telemetry Transport (MQTT), Web Socket or data files. It then converts these events into Netcool/OMNIbus events and sends them to the ObjectServer.

The probe can also be used with the IBM Tivoli Netcool/OMNIbus Gateway for Message Bus. If you wish to use the probe with the gateway, see "Using the probe with the Gateway for Message Bus" on page 92 before installing or configuring the probe.

This guide contains the following sections:

- "Summary" on page 1
- "Installing probes" on page 4
- "Migrating to the Probe for Message Bus" on page 4
- "Configuring the probe" on page 7
- "Using the transport module" on page 17
- "Message Bus Probe integrations with event sources" on page 47
- "Using the transformer module" on page 66
- "Running the probe" on page 71
- "Data acquisition" on page 72
- "Properties and command line options" on page 74
- "Properties and command line options provided by the Java Probe Integration Library (probe-sdk-java) version 11.0" on page 80
- "Elements" on page 83
- "Error messages" on page 87
- "ProbeWatch messages" on page 92
- "Using the probe with the Gateway for Message Bus" on page 92
- "Frequently asked questions" on page 94
- "Troubleshooting" on page 94
- "Known issues with the Probe for Message Bus" on page 95

Summary

Each probe works in a different way to acquire event data from its source, and therefore has specific features, default values, and changeable properties. Use this summary information to learn about this probe.

Probe for Message Bus

Table 3. Summary

Probe target	XML or JSON event sources
Probe executable name	`nco_p_message_bus`
Installation package	`omnibus_`*`arch`*`_probe_nco_p_message_bus_`*`version`*
Package version	13.0

© Copyright IBM Corp. 2015, 2020

Table 3. Summary (continued)	
Probe supported on	For details of supported operating systems, see the following Release Notice on the IBM Software Support website: http://www-01.ibm.com/support/docview.wss?uid=swg21970413
Properties files	The probe is supplied with the following properties files installed in the $OMNIHOME/probes/*arch* directory: message_bus.props message_bus_aws.props message_bus_azure.props message_bus_ciena_mcp.props message_bus_contrail.props message_bus_iDirect_pulse.props message_bus_kafka.props message_bus_logstash.props message_bus_nokia_oms1350.props message_bus_nokia_nfmp.props message_bus_prometheus.props
Rules file	The probe is supplied with the following rules files installed in the $OMNIHOME/probes/*arch* directory: message_bus.rules message_bus_aws.rules message_bus_azure.rules message_bus_cbe.rules message_bus_ciena_mcp.rules message_bus_ciena_notificationMap.rules message_bus_ciena_resyncMap.rules message_bus_contrail.rules message_bus_iDirect_pulse.rules message_bus_kafka.rules message_bus_logstash.rules message_bus_netcool.rules message_bus_nokia_nfmp.rules message_bus_nokia_oms1350.rules message_bus_prometheus.rules message_bus_wbe.rules message_bus_wef.rules

Table 3. Summary (continued)	

Transport properties files	The probe supports the following transport properties file installed in the `$OMNIHOME/java/conf/` directory:
	`awsWebhookTransport.properties`
	`cienaMcpTransport.properties`
	`cometdTransport.properties`
	`eventSourceTransport.properties`
	`fileTransport.properties`
	`iDirectPulseTransport.properties`
	`jmsTransport.properties`
	`kafkaClient.properties`
	`kafkaConnectionProperties.json`
	`kafkaTransport.properties`
	`logstashWebhookTransport.properties`
	`message_bus_azure_WebhookTransport.properties`
	`messageHubKafkaClient.properties`
	`messageHubKafkaJavaSys.properties`
	`messageHubKafkaTransport.properties`
	`mqttTransport.properties`
	`nokiaOms1350CometdTransport.properties`
	`nokiaNspKafkadTransport.properties`
	`prometheusWebhookTransport.properties`
	`restWebSocketTransport.properties`
	`restWebhookTransport.properties`

Requirements	The MQTT transport has dependency on wmqtt.jar. You can download the JAR file from the following site: http://www-01.ibm.com/support/docview.wss?rs=203&uid=swg24006006 Copy the JAR file to `$OMNIHOME/java/jars` For details of any additional software that this probe requires, refer to the README file that is supplied in its download package.
Connection method	JMS, Webhook, MQTT, WebSocket with REST API, data file, EventSource, Kafka
Multicultural support	Available For information about configuring multicultural support, including language options, see the *IBM Tivoli Netcool/OMNIbus Installation and Deployment Guide*.
Peer-to-peer failover functionality	Available

Table 3. Summary (continued)	
IP environment	IPv4 and IPv6 For communications between the probe and the XML event source, or between the probe and the JSON event source, the probe supports the IPv6 environment on all operating systems except Windows XP and Windows 2003.

Installing probes

All probes are installed in a similar way. The process involves downloading the appropriate installation package for your operating system, installing the appropriate files for the version of Netcool/OMNIbus that you are running, and configuring the probe to suit your environment.

The installation process consists of the following steps:

1. Downloading the installation package for the probe from the Passport Advantage Online website.

   Each probe has a single installation package for each operating system supported. For details about how to locate and download the installation package for your operating system, visit the following page on the IBM Tivoli Knowledge Center:

   http://www-01.ibm.com/support/knowledgecenter/SSSHTQ/omnibus/probes/all_probes/wip/reference/install_download_intro.html
2. Installing the probe using the installation package.

   The installation package contains the appropriate files for all supported versions of Netcool/OMNIbus. For details about how to install the probe to run with your version of Netcool/OMNIbus, visit the following page on the IBM Tivoli Knowledge Center:

   http://www-01.ibm.com/support/knowledgecenter/SSSHTQ/omnibus/probes/all_probes/wip/reference/install_install_intro.html
3. Configuring the probe.

   This guide contains details of the essential configuration required to run this probe. It combines topics that are common to all probes and topics that are peculiar to this probe. For details about additional configuration that is common to all probes, see the *IBM Tivoli Netcool/OMNIbus Probe and Gateway Guide*.

Migrating to the Probe for Message Bus

This topic describes how to migrate from the legacy XML Probe to the Probe for Message Bus.

The following topics describe the steps required to migrate to the Probe for Message Bus:

- "Identifying new and changed features of the Probe for Message Bus" on page 4.
- "Configuration files" on page 6.

Identifying new and changed features of the Probe for Message Bus

When migrating from the legacy probe to the Probe for Message Bus, there are some important points to note about the features that are available and the properties that are required to configure them:

- Some properties form the legacy probe have been deprecated in the Probe for Message Bus.
- Some features, and the properties required to configure them, have been added to the Probe for Message Bus.

The following table compares the features, and the properties required to configure them, of the legacy XML Probe and the Probe for Message Bus.

Table 4. Comparison of properties between the legacy XML Probe and the Probe for Message Bus

Legacy XML Probe property	Message Bus Probe property	Functional description
ConnectionCheckInterval	**HeartbeatInterval**	Interval between successive connection status checks.
Not available in the legacy probe.	**Cookie**	HTTP cookie name to be retrieved from the probe store.
DisconnectionOnInactivity	**Inactivity**	Indicates whether the probe disconnects from the target system when the inactivity timeout period is reached.
Not available in the legacy probe.	**EnableSSL**	Indicates whether SSL connectivity.
FlushBufferInterval	**FlushBufferInterval**	Interval at which the probe flushes all alerts in the buffer to the ObjectServer.
Not available in the legacy probe.	**Host**	Host name or IP address of the instance of the XML or JSON event source to which the probe connects.
Inactivity	**Inactivity**	Time (in seconds) that the probe allows the port to receive no incoming data before disconnecting.
Not available in the legacy probe.	**InitialResync**	Indicates whether the probe requests all active alarms from the host server on startup.
Not available in the legacy probe.	**JsonNestedPayload**	Indicates whether nested parsing on JSON data is enabled. You can specify either XML or JSON tree structure.
Not available in the legacy probe.	**KeyStore**	Location of the keystore file that contains the client certificate for the SSL and trusted authority certificate.
Not available in the legacy probe.	**KeyStorePassword**	Password required to access the certificate specified by the **Keystore** property.
Not available in the legacy probe.	**MaxEventQueueSize**	Maximum number of events that can be queued between the non native process and the ObjectServer.
Not available in the legacy probe.	**MessagePayload**	Type of message payload, either XML or the JSON tree.
Not available in the legacy probe.	**Password**	Password associated with the **Username** property for logging into the XML event source.

Table 4. Comparison of properties between the legacy XML Probe and the Probe for Message Bus (continued)

Legacy XML Probe property	Message Bus Probe property	Functional description
Not available in the legacy probe.	**Port**	Port of the instance of the XML or JSON event source to which the probe connects.
Not available in the legacy probe.	**ResyncInterval**	Interval at which the probe makes successive resynchronization requests.
Retry	**RetryCount** and **RetryInterval**	Indicates whether the probe retries connecting to the target system before shutting down.
StreamCapture	**StreamCapture**	Indicates whether or not the probe stores the XML or JSON event data in a stream capture file.
StreamCaptureFile	**StreamCaptureFile**	Location of the stream capture file.
TransformerFile	**TransformerFile**	Location of the transformer properties file.
TransportFile	**TransportFile**	Location of the transport properties file.
TransportType	**TransportType**	Transport method to be used or the name of the transport module class to use.
Not available in the legacy probe.	**Username**	User account for logging into the XML or JSON event source.
Not available in the legacy probe.	**WebSocketID** (Deprecated in version 5.0 of the Message Bus Probe.)	JSON field name where the WebSocket subscription ID is stored as a field value in the response message during a subscription request. **Note:** This property is deprecated in version 5.0 of Message Bus Probe. To enable the probe to extract multiple attributes from incoming JSON data, use the **keepTokens** property in the `restWebSocketTransport.properties` file.

Configuration files

The configuration files have been renamed. If you update those file accordingly if they have some custom settings in the legacy probe.

The following table shows the configuration files that have been renamed.

Table 5. Comparison of configuration file names between the legacy XML Probe and the Probe for Message Bus	
Legacy XML Probe property	**Message Bus Probe property**
`xml.rules`	`message_bus.rules`
`xml_cbe.rules`	`message_bus_cbe.rules`
`xml_netcool.rules`	`message_bus_netcool.rules`
`xml_wbe.rules`	`message_bus_wbe.rules`
`xml_wef.rules`	`message_bus_wef.rules`
`xml.props`	`message_bus.props`
`nco_p_xml.bat`	`nco_p_message_bus.bat`
`nco_p_xml.env`	`nco_p_message_bus.env`

Configuring the probe

Before running the probe for the first time, you must specify a minimum set of properties.

To run the probe successfully, the following properties are the minimum that you must specify in the `message_bus.props` file:

- **Server** - This generic property specifies the name of the primary ObjectServer or the proxy server to which alerts are sent. The default is `NCOMS`.
- **TransformerFile** - This probe-specific property specifies the location of the transformer properties file. For XML events, specify the XML transformer properties file. For JSON events, specify the JSON parser configuration file, for more details on the JSON configuration format, refer to "Configuring the JSON parser to parse different JSON structures" on page 13.
- **TransportFile** - This probe-specific property specifies the location of the transport properties file.
- **TransportType** - This probe-specific property specifies the transport method to be used (JMS, MQTT, Webhook, WebSocket, or data file).

For installations of the probe on Windows operating systems, you must edit the values of the following properties to specify the full directory paths to their respective files:

- **PropsFile**
- **RulesFile**
- **StreamCaptureFile**
- **TransformerFile**
- **TransportFile**

For example, the value of the **TransformerFile** property might be:

```
'C:\\IBM\\Tivoli\\Netcool\\omnibus\\java\\conf\\transformers.xml'
```

or

```
'C:\\IBM\\Tivoli\\Netcool\\omnibus\\probes\\win32\
\message_bus_parser_config.json'
```

Enabling the max_line_length property

The -Dcom.ibm.csi.netcool.integrations.max_line_length property of the Transport module allows you to limit the amount of data that can be read at one time. Making the reading from files bounded in this way allows you to prevent unbounded reads from files being exploited in a denial of service attack.

Note: The max_line_length property is specified in bytes.

UNIX and Linux operating system

If you are running the probe on UNIX or Linux operating systems, customize the probe scripts using the following steps:

1. Update nco_p_message_bus.env by adding the following line at the end of the file:

```
NCO_PROBE_JAVA_ARGS=-Dcom.ibm.csi.netcool.integrations.max_line_length=xxxx
```

2. Update nco_jprobe by updating the following command:

```
exec "$JAVA" $NCO_PROBE_JAVA_ARGS $NCO_JPROBE_JAVA_FLAGS -cp "$CLASSPATH" -
DOMNIHOME="$OMNIHOME" -DKERN_ARCH="$KERN_ARCH"
com.ibm.tivoli.netcool.omnibus.oidk.Probe "$@"
```

Windows operating system

If you are running the probe on Windows operating systems, update nco_p_message_bus.bat using the following steps:

1. Add the following line before the line discussed in Step 2:

```
set TRANSPORT_EXTRA_ARGS=-
Dcom.ibm.csi.netcool.integrations.max_line_length=xxxx
```

2. Add the TRANSPORT_EXTRA_ARGS variable to the following line:

```
%JAVA_EXEC_USED% -DOMNIHOME=%NEWOMNIHOME% ^ -cp %PROBE_CLASSPATH
%;%CP_CLASSPATH%;%FW_CLASSPATH%;%PS_CLASSPATH%;%NS_CLASSPATH%;%TS_CLASSPATH
%;%MESSAGEBUS_CLASSPATH%;%TRANSPORT_EXTRA_ARGS% ^
com.ibm.tivoli.netcool.omnibus.oidk.Probe %*
```

Selecting a value for the max_line_length property

If you set max_line_length to a relatively small value (for example 100 bytes), and if the probe reads a line from the data file that is greater than max_line_length, the probe shuts down and writes the following exception to the error log:

```
Error: E-JPR-000-000: Line length exceeds maximum size of %d chars. Increase
this value using the property com.ibm.csi.netcool.integrations.max_line_length.
Snippet of parsed line: '%s'
```

If the **RetryCount** property is set to a value greater than 0, the probe restarts and reopens the data file. When the probe reaches the line that triggered the IOException, it shuts down again and restarts.

To avoid this type of shutdown and restart loop, perform one of the following steps:

- Either remove the lines from the data file that trigger the probe restarting.
- Or set max_line_length to a value greater than the longest line length in the data file.

Enabling the max_http_payload_size property

The -Dcom.ibm.csi.netcool.integrations.max_http_payload_size property of the HTTP Transport module allows you to limit the amount of data embedded in the body of an HTTP POST request. The data in the HTTP POST must be an XML message. If a valid XML message is too large, its

transformation speed is relatively slow. In a worst case scenario, very large messages may stall at some point before getting to the transformation part.

Although the probe can still run and process other HTTP POST requests, the mounting stalled points will eventually crash the probe due to memory depletion. To prevent a denial of service attack, the `max_http_payload_size` property allows you to bar large XML messages from entering the pipeline.

Note: The `max_http_payload_size` property is specified in bytes.

UNIX and Linux operating system

If you are running the probe on UNIX or Linux operating systems, customize the probe scripts using the following steps:

1. Update `nco_p_message_bus.env` by adding the following line at the end of the file:

   ```
   NCO_PROBE_JAVA_ARGS=-
   Dcom.ibm.csi.netcool.integrations.max_http_payload_size=xxxx
   ```

2. Update `nco_jprobe` by updating the following command:

   ```
   exec "$JAVA" $NCO_PROBE_JAVA_ARGS $NCO_JPROBE_JAVA_FLAGS -cp "$CLASSPATH" -
   DOMNIHOME="$OMNIHOME" -DKERN_ARCH="$KERN_ARCH"
   com.ibm.tivoli.netcool.omnibus.oidk.Probe "$@"
   ```

Windows operating system

If you are running the probe on Windows operating systems, update `nco_p_message_bus.bat` using the following steps:

1. Add the following line before the line discussed in Step 2:

   ```
   set TRANSPORT_EXTRA_ARGS=-
   Dcom.ibm.csi.netcool.integrations.max_http_payload_size=xxxx
   ```

2. Add the TRANSPORT_EXTRA_ARGS variable to the following line:

   ```
   %JAVA_EXEC_USED% -DOMNIHOME=%NEWOMNIHOME% ^ -cp %PROBE_CLASSPATH
   %;%CP_CLASSPATH%;%FW_CLASSPATH%;%PS_CLASSPATH%;%NS_CLASSPATH%;%TS_CLASSPATH
   %;%MESSAGEBUS_CLASSPATH%;%TRANSPORT_EXTRA_ARGS% ^
   com.ibm.tivoli.netcool.omnibus.oidk.Probe %*
   ```

Selecting a value for the max_http_payload_size property

The payload size specified must be between 5120 and 2097152.

If you set `max_http_payload_size` to a value outside of this range (for example, `12300000`), when the probe starts it uses the maximum allowable payload value instead and writes the following exception to the error log:

```
Error: D-JPR-000-000: [HttpParser]: Property
[com.ibm.csi.netcool.integrations.max_http_payload_size]'s value: 12300000 is
out of range.Valid range is from 5120 to 2097152; Max valid value will be used
as the http payload size limit.
```

If the probe receives an XML message that exceeds the value set for `max_http_payload_size`, the probe writes the following exception to the error log:

```
Error message for xml message's size exceeding max_http_payload_size:
Information: I-JPR-104-000: [stderr]
com.ibm.tivoli.netcool.integrations.transportmodule.http.HttpException:
HttpParse: Abort reading payload, number of byte exceeds the limit (2097152):
%d
```

Note: %d will indicate the actual size of the input HTTP payload.

Configuring the parser with a different JsonMessageDepth

Changing the value set for the **JsonMessageDepth** property changes the tokens that the parser generates.

The table that follows the example JSON data below illustrates how changing the **JsonMessageDepth** property in the `message_bus.props` file determines which key-value pairs the parser generates.

```
{
    "properties": {
        "storage": {
            "type": "object",
            "oneOf": [
                  {"$ref": "#/definitions/diskDevice"},
                  {"$ref": "#/definitions/diskUUID"},
                  {"$ref": "#/definitions/nfs"},
                  {"$ref": "#/definitions/tmpfs"}
             ]
        },
        "fstype": {
            "enum": ["ext3", "ext4", "btrfs"]
        },
        "options": {
            "type": "array",
            "minItems": 1,
            "items": {
                "type": "string"
            },
            "uniqueItems": true
        }
    }
}
```

Table 6. Tokens genarated

JSON parser properties	Tokens generated

`MessagePayload = “json.properties”`
`JsonMessageDepth = 1`
`MessageHeader = “”`
`JsonNestedPayload = “”`
`JsonNestedHeader = “”`
	None. There is no field or key-value pair in the first level.

`MessagePayload = “json.properties”`
`JsonMessageDepth = 2`
`MessageHeader = “”`
`JsonNestedPayload = “”`
`JsonNestedHeader = “”`
	`options.minItems=1`
	`options.type=array`
	`options.uniqueItems=true`
	`resync_event=false`
	`storage.type=object`

`MessagePayload = “json.properties”`
`JsonMessageDepth = 3 (the default)`
`MessageHeader = “”`
`JsonNestedPayload = “”`
`JsonNestedHeader = “”`
	`fstype.enum=ext3,ext4,btrfs`
	`options.items.type=string`
	`options.minItems=1`
	`options.type=array`
	`options.uniqueItems=true`
	`resync_event=false`
	`storage.type=object`
	
	Note: Array values such as the `enum` object are treated as fields or key-value pairs at the same level as the node. With this configuration, the `oneOf` array of objects is not generated because it is one level deeper.

Table 6. Tokens genarated (continued)

JSON parser properties	Tokens generated
MessagePayload = “json.properties” JsonMessageDepth = **4** MessageHeader = “” JsonNestedPayload = “” JsonNestedHeader = “”	fstype.enum=ext3,ext4,btrfs options.items.type=string options.minItems=1 options.type=array options.uniqueItems=true resync_event=false storage.oneOf.0.$ref= #/definitions/diskDevice storage.oneOf.1.$ref= #/definitions/diskUUID storage.oneOf.2.$ref= #/definitions/nfs storage.oneOf.3.$ref= #/definitions/tmpfs storage.type=object **Note:** With this configuration, the `oneOf` array of objects is generated.

Note: The `resync_event` token is not part of the JSON message, but is generated by the probe for internal use.

Generating events from a nested JSON

Some event sources send events as a nested JSON in a JSON message. The parser can be configured to extract and parse the nested JSON.

Given the following data which contains a nested JSON in one of its objects, `payload`, the parser needs to be configured using both the **`MessagePayload`** property and the **`JsonNestedPayload`** property. Example parser configuration and the tokens generated are shown in the table that follows.

```
{
"payload" :"{\"properties\": {\"storage\": {\"type\": \"object\",\"oneOf\": [
{\"$ref\": \"#\/definitions\/diskDevice\"}, {\"$ref\":
\"#\/definitions\/diskUUID\"},{\"$ref\": \"#\/definitions\/nfs\"},{\"$ref\":
\"#\/definitions\/tmpfs\"}]},\"fstype\":{\"enum\":[\"ext3\",\"ext4\",\"btrfs\"]},
\"options\":{\"type\":\"array\",\"minItems\":\"1\",\"items\": {\"type\":
\"string\"},\"uniqueItems\": \"true\"}}}",

    "header": {"options" : "none"},
    "log":{"message":"Alert"}
}
```

Table 7. Tokens generated

Json parser properties	Tokens generated
MessagePayload = “json.payload” JsonMessageDepth = **1** MessageHeader = “” JsonNestedPayload = “json.properties.storage” JsonNestedHeader = “”	resync_event=false type=object
MessagePayload = “json.payload” JsonMessageDepth = **2** MessageHeader = “” JsonNestedPayload = “json.properties.storage” JsonNestedHeader = “”	enum=ext3,ext4,btrfs resync_event=false type=object

Table 7. Tokens generated (continued)

Json parser properties	Tokens generated
MessagePayload = “json.payload” JsonMessageDepth = **3** MessageHeader = “” JsonNestedPayload = “json.properties.storage” JsonNestedHeader = “”	oneOf.0.$ref=#/definitions/diskDevice oneOf.1.$ref=#/definitions/diskUUID oneOf.2.$ref=#/definitions/nfs oneOf.3.$ref=#/definitions/tmpfs resync_event=false type=object
MessagePayload = “json.payload” JsonMessageDepth = 3 MessageHeader = “” JsonNestedPayload = “json.properties.storage” JsonNestedHeader = “**json.properties.fstype**”	enum=ext3,ext4,btrfs oneOf.0.$ref=#/definitions/diskDevice oneOf.1.$ref=#/definitions/diskUUID oneOf.2.$ref=#/definitions/nfs oneOf.3.$ref=#/definitions/tmpfs resync_event=false type=object **Note:** The `fstype` object is not under the ‘storage’ object but is also parsed because it is specified as the Header object.
MessagePayload = “json.log” JsonMessageDepth = 3 MessageHeader = “**json.header**” JsonNestedPayload = “” JsonNestedHeader = “”	message=Alert options=none resync_event=false

Note: The `resync_event` token is not part of the Json message but is generated by the probe for internal use.

Specifying multiple parser configurations to parse different JSON structures

If the probe’s transport is configured to retrieve JSON alarms using resynchronization and notification, and has different alarm structures, you can configure the probe’s JSON parser with different settings to parse the JSON alarms and generate elements from them.

The following configuration file is provided with the probe as a template:

```
{
  "eventSources" : [ {
    "endpoint" : "/notification",
    "name" : "NotificationAlarmParser",
    "config" : {
      "dataToRecord" : [ ],
      "messagePayload" : "json",
      "messageHeader" : "",
      "jsonNestedPayload" : "",
      "jsonNestedHeader" : "",
      "messageDepth" : 3
    }
  }, {
    "endpoint" : "/resync",
    "name" : "ResyncAlarmParser",
    "config" : {
      "dataToRecord" : [ ],
      "messagePayload" : "json",
      "messageHeader" : "",
      "jsonNestedPayload" : "",
      "jsonNestedHeader" : "",
      "messageDepth" : 3
    }
  }, {
    "name" : "OtherAlarmParser",
    "type" : "ANY",
    "config" : {
      "dataToRecord" : [ ],
      "messagePayload" : "json",
      "messageHeader" : "",
```

```
      "jsonNestedPayload" : "",
      "jsonNestedHeader" : "",
      "messageDepth" : 5
    }
  }]
}
```

Each alarm that the probe receives from the transport contains an `endpoint` value to indicate the source of the alarm. For example, for the Web Socket transport, the `endpoint` for resynchronization alarms is `resync` and the `endpoint` for notification is the Web Socket URI. To determine the `endpoint` value to use in the configuration file, you should run the probe with debug level logging and use the endpoint logged when an alarm is received.

When a probe receives an alarm from the source, the following messages are logged.

Note: The endpoint for this case is set to the Web Socket URI path: `/lab/notification/api`. So the `endpoint` attribute should be set to `/lab/notification/api`.

```
2017-06-23T12:06:51: Debug: D-JPR-000-000: Received message from websocket.
2017-06-23T12:06:51: Debug: D-JPR-000-000: Received message with length of 824
from endpoint /lab/notification/api:  <MESSAGE > …
```

The probe parser uses the following criteria, in order, when performing the parser configuration lookup:

1. Exact match of the parser's `endpoint` attribute with the `endpoint` value in the message.
2. Parser whose `endpoint` matches the start of the `endpoint` value in the message.

   For example, a parser with `endpoint: /lab/notification/api` will be used to parse messages containing `endpoint= /lab/notification/api`
3. Parser whose `endpoint` is ANY.

   This is the last parser configuration used and if there is no ANY parser configuration set, the probe considers the message as unparsed.

Configuring the JSON parser to parse different JSON structures

To configure multiple parser configurations, use the following steps:

1. Create a parser configuration file (`.json`) in `$OMNIHOME/probes/<arch>/ParserConfig.json`

   A template is provided in `$OMNIHOME/probes/<arch>/message_bus_parser_config.json`. This template contains three parser configurations: notification, resynchronized alarm, and a common parser for any alarms that does not match the endpoint criteria of the other two parsers. This `OtherAlarmParser` is indicated by the configuration with `endpoint = "ANY"`.
2. Update the `endpoint` attribute to be the same as the endpoint set by the transport used by the probe.

   For example, for the `restWebSocket` transport, the endpoints for alarms retrieved from resynchronization and notifications are the `resync` and `webSocketURI` transport property values respectively.
3. Update "`messagePayload`", "`messageHeader`", "`jsonNestedPayload`", "`jsonNestedHeader`", "`messageDepth`", and "`dataToRecord`" to suit the alarm structure to be parsed.
4. Optionally, update the `OtherAlarmParser` configuration so that the parser traverses the JSON event and generates elements on the leaf nodes instead of flattening the JSON event.
5. In the probe properties file, set the **`TransfomerFile`** property to the parser configuration file created in Step 1.

   `TransformerFile : '$OMNIHOME/probes/<arch>/ParserConfig.json'`

Example configuration

Given the following JSON alarms received by the probe transport using notification and resynchronization respectively, the parser can be configured to parse both alarm structures.

Sample Notification Structure

```
{
  "payload": {
    "id": 1,
    "body": {
      "value": "{\"header\":{\"timestamp\":\"2017-07-
06T09:07:56Z\",\"event\":{\"alarm\":{\"id\":\"1234567\",\"resource\":\"4\",\"node-
id\":\"node01\",\"condition-type\":\"Link Down\",\"condition-
severity\":\"WARNING\",\"condition-source\":\"NETWORK\",\"condition-
state\":\"ACTIVE\",\"additional-text\":\"Link Down\",\"first-raise-time\":\"2000-01-
01T00:00:58.000+0000\",\"last-raise-time\":\"2000-01-01T00:00:58.000+0000\",\"number-
of-occurrences\":1,\"acknowledge-state\":\"ACKNOWLEDGED\",\"acknowledge-update-
time\":\"2017-07-06T09:07:56.463+0000\",\"additional-attrs\":{\"source\":\"EMS-
1\"}},\"_type\":\"alarmAcknowledged\"}}",
      "attributes": 1
    }
  },
  "event": "sample_alarm”
}
```

Sample resynchronized alarm structure

```
{
  "data": [
    {
      "id": "1222383102379613532",
      "type": "FilteredAlarm",
      "attributes": {
        "id": "1234567",
        "node-id": "node01",
        "state": "ACTIVE",
        "resource": "SWITCH",
        "condition-severity": "CRITICAL",
        "first-raise-time": "2017-07-06T13:43:19.000+0000",
        "last-raise-time": "2017-07-06T13:43:19.000+0000",
        "number-of-occurrences": 1,
        "acknowledge-state": "NOT_ACKNOWLEDGED"
      }
    },
    {
      "id": "-4067638085156070319",
      "type": "FilteredAlarm",
      "attributes": {
        "id": "7654321",
        "node-id": "node02",
        "state": "ACTIVE",
        "resource": "SWITCH",
        "condition-severity": "CRITICAL",
        "first-raise-time": "2017-07-06T13:42:59.000+0000",
        "last-raise-time": "2017-07-06T13:42:59.000+0000",
        "number-of-occurrences": 1,
        "acknowledge-state": "NOT_ACKNOWLEDGED"
      }
    }
  ]
}
```

The configuration to parse the above alarms and generate elements from the alarms (italics) as shown below.

Note: The endpoint is configured to the correct source.

```
{
  "eventSources": [
    {
      "endpoint": "/lab/notification/api",
      "name": "NotificationAlarmParser",
      "config": {
        "dataToRecord": [],
        "messagePayload": "json.payload.body.value",
        "messageHeader": "json",
        "jsonNestedPayload": "json.event.alarm",
        "jsonNestedHeader": "json.header",
        "messageDepth": 5
       }
     },
     {
```

```
      "endpoint": “resync",
      "name": ”ResynchAlarmParser",
      "config": {
        "dataToRecord": [],
        "messagePayload": "json.payload.body.value",
        "messageHeader": "json",
        "jsonNestedPayload": "json.event.alarm",
        "jsonNestedHeader": "json.header",
        "messageDepth": 5
      }
 }]
}
```

The parser will generate the following elements for rules files consumption.

```
acknowledge-state=ACKNOWLEDGED
acknowledge-update-time=2017-07-06T09:07:56.463+0000
additional-attrs.source=EMS-1
additional-text=Link Down
condition-severity=WARNING
condition-source=NETWORK
condition-state=ACTIVE
condition-type=Link Down
first-raise-time=2000-01-01T00:00:58.000+0000
id=1234567
last-raise-time=2000-01-01T00:00:58.000+0000
node-id=node01
number-of-occurrences=1
resource=4
resync_event=false
timestamp=2017-07-06T09:07:56Z
```

Upgrading to Probe for Message Bus version 5 or newer

Version 5 of the Probe for Message Bus has an enhanced JSON parser that supports multiple parser configurations using a parser configuration file. This enables the probe to recognize the JSON structure by the endpoint and use a specific configuration for the JSON events received.

This is useful when the JSON event structure of resynchronized events is different from that of the events received from notification or subscription channels. For some event sources, the resynchronized alarm is usually sent in batches or an array of events while notification events are sent as a single event.

Note: To support this new feature, the order of precedence of the **MessagePayload** and **TransformerFile** probe properties has changed. The **TransformerFile** property now takes precedence over the **MessagePayload** property in order to determine the type of event source that the probe will receive. It will be either XML or JSON depending on the file extension.

If your existing probe configuration has the **MessagePayload** property set or starts with `json` and the default **TransformerFile** property value is used, uncomment the **TransformerFile** property and set it to an empty string (`''`) to bypass it. The probe will then load your existing parser configuration as is. However, you should migrate the configuration to the new JSON parser configuration file as shown in “Migrating the probe parser configuration to the JSON parser configuration file” on page 15.

In this version of the probe, the **WebSocketID** probe property has been deprecated and replaced with a new property in the WebSocket transport, see “Migrating the WebSocketID property to the Websocket transport properties file” on page 17.

Migrating the probe parser configuration to the JSON parser configuration file

To migrate the probe parser configuration from the probe properties file to the new JSON parser configuration file format, use the following steps:

1. Check the current probe properties file to determine whether probe is configured to receive JSON or XML events. View the probe properties file and check the **MessagePayload** probe property in the properties file: `$OMNIHOME/probes/<arch>/message_bus.props`

   If the **MessagePayload** property is set to, or starts with, `json`, the probe is configured to consume JSON events. Proceed to step 2.

If the **MessagePayload** property is set to `xml`, the probe is configured to consume XML events. Leave this value as it is and verify that the **TransformerFile**property is set to the XML transformer file path.

Note: The remaining steps do not apply if the probe is configured to parse XML events.

2. If the probe is configured to consume JSON events, the probe properties are mapped to the JSON configuration as shown in the following table:

Table 8. Mapping of probe properties to JSON configuration attributes

Probe property name	JSON parser configuration attribute name
MessagePayload	`messagePayload`
MessageHeader	`messageHeader`
JsonNestedPayload	`jsonNestedPayload`
JsonNestedHeader	`jsonNestedHeader`
MessageDepth	`messageDepth`
RecordData	`dataToRecord`

For example, if the probe is configured with the following properties:

- **MessagePayload**: `json.payload.body.value`
- **MessageHeader**: `json`
- **JsonNestedPayload**: `json.event.alarm`
- **JsonNestedHeader**: `json.header`
- **MessageDepth**: 5
- **RecordData**: `first-raise-time,last-raise-time`

The JSON parser configuration file will be:

```
{
  "eventSources": [
    {
      "endpoint": "/lab/notification/api",
      "name": "AllJsonEventSource",
      "type": "ANY",
      "config": {
        "dataToRecord": [
          "first-raise-time",
          "last-raise-time"
        ],
        "messagePayload": "json.payload.body.value",
        "messageHeader": "json",
        "jsonNestedPayload": "json.event.alarm",
        "jsonNestedHeader": "json.header",
        "messageDepth": 5
      }
    }
  ]
}
```

3. Edit the template JSON parser configuration file provided in `$OMNIHOME/probes/<arch>/message_bus_parser_config.json` and copy the above JSON configuration into the file.

Note: The `name` attribute can be used to assign a name to the event source. If there is only one event source configured in the JSON configuration, the `endpoint` attribute will be ignored.

4. In the `message_bus.props` properties file, set the **TransformerFile** property to the JSON parser configuration file used in step 3.

5. Restart the probe and verify that the probe starts successfully and parses the JSON events correctly.

Migrating the WebSocketID property to the Websocket transport properties file
The **WebSocketID** property is often used to keep an attribute received from a JSON event and used in variable substitution during runtime. The **WebSocketID** property is specific to the WebSocket transport. Hence, it is deemed more suitable as a transport property rather than a probe property.

A new WebSocket transport property, **keepTokens**, was introduced to replace the **WebSocketID** probe property and allows you to specify more than one attribute to keep. If the WebSocket transport type is used and the **WebSocketID** property is enabled, the probe will not start due to an unrecognized property. To resolve this, use the following steps:

1. View the Message Bus Probe properties file (`$OMNIHOME/probes/<arch>/message_bus.props`) and check the values set for the **TransportType** and **WebSocketID** properties.

   If the **TransportType** property is set to `WebSocket`, then proceed with the following steps.
2. Copy the **WebSocketID** property value if it has been set.
3. Edit the WebSocket transport property file (`$OMNIHOME/java/conf/restWebSocketTransport.properties`)
4. Uncomment the **keepTokens** property and set the value to the value copied from the probe properties file in Step 2. For example:

   `keepTokens=subscriptionId`
5. Save the `restWebSocketTransport.properties` file.
6. Restart the probe and verify that the probe runs successfully.

Using the transport module

The transport module enables the probe to acquire XML and JSON formatted events.

Before using the transport module, configure the transport properties file appropriate to the transport protocol you are using to transfer events. The probe supports the use of Java Message Service (JMS), Webhook, Message Queue Telemetry Transport (MQTT), Web Socket, and data files.

The transport properties files are located in the following directory:

`$OMNIHOME/java/conf/`

After installation, the probe defaults to using the JMS properties file (`jmsTransport.properties`) located in `$OMNIHOME/java/conf`. To change this setting, specify a new transport type and transport properties by changing the values set for the following properties:

- **TransportType**
- **TransportFile**

The following table lists the properties file associated with each available protocol:

Table 9. Transport properties files

Transport protocol	Transport type	Properties file
Java Message Service (JMS)	JMS	jmsTransport.properties
Data file	File	fileTransport.properties
EventSource	EventSource	eventSourceTransport.properties
Message Queue Telemetry Transport (MQTT)	MQTT	mqttTransport.properties

Table 9. Transport properties files (continued)		
Transport protocol	**Transport type**	**Properties file**
WebSocket	`WebSocket`	`restWebSocketTransport.properties`
WebHook	`Webhook`	`restWebhookTransport.properties`
CometD	`Cometd`	`cometdTransport.properties`
Socket	`Socket`	`socketTransport.properties`
Kafka	`KAFKA`	`kafkaTransport.properties`

Configuring the transport properties files

Transport properties files define how the probe receives events using the transport module.

The following sections describe the property settings available for each transport protocol:

- "Configuring the JMS transport" on page 18
- "Configuring the data file transport" on page 21
- "Configuring the MQTT transport" on page 21
- "Configuring the Web Socket transport" on page 22
- "Configuring the Web Hook transport" on page 32
- "Configuring the Cometd transport" on page 40
- "Configuring the socket transport" on page 42
-

Note: On Windows platforms, the delimiter in all the paths defined in the Java properties files should be escaped, for example, if you are setting the path for the key store, the path should be defined as follows:

```
keyStore=C:\\IBM\\Tivoli\\Netcool\\omnibus\\java\\security\\client.jks
```

Configuring the JMS transport

The JMS transport allows the probe to connect to and receive messages from a JMS provider.

JMS transport properties

The following table describes the properties used to configure the `jmsTransport.properties` file.

Table 10. JMS transport properties	
Property name	**Description**
`jmsFilter`	Use this property to specify the expression for JMS servers for sending the intended messages or events to the probe. **Note:** This property is only used if the probe subscribes to a topic.

Table 10. JMS transport properties (continued)	
Property name	**Description**
initialContextFactory	Use this property to specify the context factory class name of the JMS provider. **Note:** The **initialContextFactory** is the initial context factory for the Java Naming Directory Interface (JNDI) provider being used. The default is the Websphere JNDI provider.
providerURL	Use this property to specify the URL of the JMS provider. You can specify the URL of the data store for the JNDI provider, or you can specify the path to a binding file. **Note:** If you specify a binding file, always create that file using the IBM WebSphere® MQ JMS Administration tool. In addition, copy the WebSphere jar files to the `$OMNIHOME/java` directory. For information about creating binding files and JMS Administration tool jar files, see the IBM WebSphere MQ knowledge center.
queueConnectionFactory	Use this property to specify the identifier of the queue connection factory.

queueName	Use this property to specify the names of the queues on which the consumer receives messages.
	You can enter more than one **queueName** property in the JMS properties file. For example:
	`queueName=topicOne`
	`queueName=topicTwo`

topicName	Use this property to specify the names of the topics to which the consumer subscribes for messages.
	You can enter more than one **topicName** property in the JMS properties file. For example:
	`topicName=topicOne`
	`topicName=topicTwo`

topicConnectionFactory	Use this property to specify the identifier of the topic connection factory.
username	Use this property to specify the user name for the JMS connection.
password	Use this property to specify the password for the JMS connection. **Note:** The transport property files do not support encrypted passwords. You must specify the password in plain text.

Connecting to ActiveMQ using SSL

You can configure the probe to communicate with ActiveMQ that uses a Secure Socket Layer (SSL) encrypted connection. A Java keystore file is required for this connection that you create using the Java `keytool` command.

To connect to ActiveMQ using an SSL connection:

- Determine the SSL port and certificate from the ActiveMQ system
- Import the SSL certificate and enable the SSL connection on the probe system

Determining the SSL port and certificate from the Active MQ system

To determine the SSL port that ActiveMQ uses and export the SSL certificate, use the following steps:

1. Log in to the system that hosts ActiveMQ and open the following file:

   `$ACTIVEMQ_HOME/conf/activemq.xml`

   Where `$ACTIVEMQ_HOME` is the installation directory of ActiveMQ.
2. In `activemq.xml`, locate the section beginning with the `<amq:transportConnections>` element. For example:

   ```
   <amq:transportConnectors>
   <amq:transportConnector uri="ssl://localhost:61616" />
   </amq:transportConnectors>
   ```
3. Note the port number (in this example, 61616) in the `uri` attribute of the `<amq:transportConnector>` element and close the file.
4. Check what the alias of the certificate (`broker.ks`) is:

   `keytool -list -v -keystore broker.ks`
5. Locate the SSL certificate and export that to a file using the following command:

   `keytool -export -alias broker -keystore broker.ks -file broker_certs`

   This creates file named `broker_certs` that contains the SSL certificate.

Importing the SSL certificate and enable the SSL connection

To import the SSL certificate and enable SSL communications on the probe, use the following steps:

1. Copy the `broker_cert` file to the following location on the system running the probe:

   `$NCHOME/platform/`*`arch`*`/`*`jre-directory`*`/lib/security`

   Where *`arch`* is a variable that represents your operating system directory and *`jre-directory`* is the name of the directory that contains the Java Runtime Environment. For example:

   `$NCHOME/platform/solaris2/jre_1.5.6/lib/security`
2. Import the SSL certificate:

   `keytool -import -alias broker -keystore cacerts -file broker_cert -storepass `*`storepassword`*

   Where *`storepassword`* is the password for the certificate store (`cacerts`).
3. Edit the `jmsTransport.props` file to define the SSL connection:
   a. Edit the file `$NCHOME/java/confjmsTransport.props`.
   b. Set the value of the **`providerURL`** property as follows:

      `providerURL = ssl://`*`activemqurl`*`:`*`port`*

      Where *activemqurl* is the URL of the ActiveMQ system and *port* is the port number that the ActiveMQ system uses for SSL connections.
   c. Save the file.

Configuring the data file transport

The data file transport allows the probe to read XML files, probe stream capture, and JSON files.

Data File transport properties

The following table describes the properties used to configure the `fileTransport.properties` file.

Note: You can specify a value for either the **`jsonFilename`** property, the **`streamfilename`** property or the **`xmlfilename`** property, but you cannot specify a value for more than more than one of these properties.

Table 11. Data file transport properties

Property name	Description
`jsonFilename`	Use this property to specify the full path to a JSON file.
`streamFilename`	Use this property to specify a stream capture file to run through the probe.
`xmlFilename`	Use this property to run a standard stream of XML through the probe.
`sleepInterval`	Use this property to specify the interval (in milliseconds) that the probe waits between iterations when tailing the source file. The default is 1000 milliseconds.

Configuring the MQTT transport

The MQTT transport allows the probe to connect to and receive messages from an MQTT provider.

MQTT transport properties

The following table describes the properties used to configure the `mqttTransport.properties` file.

Table 12. MQTT transport properties

Property name	Description
`connectionURL`	Use this property to specify the URL of the MQTT bus provider. The default is `tcp://localhost:1883/`.
`clientId`	Use this property to specify an ID for each MQTT client connection to the MQTT provider. The **`clientId`** of each MQTT client connection must be unique. The default is `MQTTClient`.

Table 12. MQTT transport properties (continued)	
Property name	**Description**
topicName	Use this property to specify the name of a topic to subscribe to for messages. To subscribe to multiple topics, you can define multiple topic names. The default is "". **Note:** The MQTT wildcard character # can be used to define a wildcard set of topics to subscribe to. For example, a value of # would register interest in all topics known to the MQTT provider.
cleanStart	Use this property to specify whether or not the connection should perform a clean start. The default is `false`. If you specify a value of `false`, any message stored by the MQTT provider for a connection client ID will be returned on startup.
keepAlive	Use this property to specify the frequency (in seconds) with which the MQTT provider connection is polled during periods of inactivity. Polling keeps the connection alive and active. The default is `30`.

The following is an example of an MQTT properties file:

```
# Example format of MQTT properties file
# Uncomment the relevant lines and change the settings accordingly
#
# connectionURL - the URL of the MQTT provider
connectionURL=tcp://example.sbank.uk.ibm.com:1883
#
# clientId - the id to be used for this instance of an MQTT client
# (must be unique for each client)
clientId=MQTTProbeClient
#
# topicName - name of a topic on which to subscribe for messages
# (can be more than one)
topicName=topica
topicName=topicb
#
# cleanStart - true or false - decide whether to do a clean start
# when reading messages
cleanStart=false
#
# keepAlive - duration, in seconds, to keep the connection alive
# with no activity
keepAlive=30
#
```

Configuring the Web Socket transport

The Web Socket transport allows the probe to connect to the target device to send the following types of HTTP requests:

- `loginRequest`: This is sent after the OAuth access token request to log into the target system.
- `loginRefresh` This is sent periodically to refresh the login on the target device.
- `logoutRequest` This is sent before disconnecting from the target device.

- `resynchRequest` This is sent to request an event resynchronization with the target device.
- `subscribeRequest` This is sent to subscribe to received events from the target device as they are created.
- `subscribeRefresh` This is sent periodically to refresh the event subscription on the target device.

Web Socket transport properties

The following table describes the properties in the `restWebSocketTransport.properties` file. The `restWebSocketTransport.properties` file controls the integration with the Message Bus Probe using Web Socket.

Note: When using the WebSocket transport, you must specify values for the **`Host`** and **`Port`** properties in the `message_bus.props` file.

Table 13. Properties in the `restWebSocketTransport.properties` file

Property name	Description
`httpVersion`	Use this property to specify the version of the HTTP protocol that the target system supports. The default is `1.1`.
`httpHeaders`	Use this property to specify the HTTP header options to use in all HTTP requests. This property accepts a comma separated list of key-value pairs using the equals sign (=) as the value separator. For options that accept multiple values, use the double-quote (") character around the value and a use semi-colon (;) as the value separator; for example: `Keep-alive="timeout 30;max 10",Connection=Keep-alive` For the authorization header option, the credentials that follow the single whitespace after `Basic` are encoded as a Base-64 encoded string. For example, to authenticate with the username `Me`, and the password `MyPassword`, set the **`httpHeader`** property to `Authorization=Basic Me:MyPassword`. The probe passes this as an authorization header with every request. The default value is "". **Note:** The **`httpHeaders`** property sets the headers for all HTTP requests. However, you can override these global HTTP header options using the following properties: • **`loginRequestHeaders`** • **`loginRefreshHeaders`** • **`logoutRequestHeaders`** • **`resyncRequestHeaders`** • **`subscribeRequestHeaders`** • **`subscribeRefreshHeaders`**

Table 13. Properties in the `restWebSocketTransport.properties` file (continued)	
Property name	**Description**
responseTimeout	Use this property to specify how long (in seconds) the probe waits for a response from the target system before timing out. The default is 60 seconds.
loginRequestURI	Use this property to specify the URI that the probe uses to request a login.
loginRequestMethod	Use this property to specify the message type that the probe sends to request a login.
loginRequestContent	Use this property to specify any additional information that the probe sends with the login request.
loginRequestHeaders	Use this property to specify an HTTP header to send with all login requests. This overrides the global HTTP header options configured by the **httpHeader** property.
loginRefreshURI	Use this property to specify the URI that the probe uses to refresh the login on the target device.
loginRefreshMethod	Use this property to specify the message type that the probe sends to refresh the login on the target device.
loginRefreshContent	Use this property to specify any additional information that the probe sends with the login refresh request.
loginRefreshHeaders	Use this property to specify an HTTP header to send with all login refresh requests. This overrides the global HTTP header options configured by the **httpHeader** property.
loginRefreshInterval	Use this property to specify the interval (in seconds) that the probe leaves between successive login refresh requests. This can be disabled by setting it to "" or leaving it commented out. If set to a negative value or an valid value such as `String`, it will default to 60 seconds.
logoutRequestURI	Use this property to specify the URI that the probe uses to request a logout from the target device.
logoutRequestMethod	Use this property to specify the message type that the probe sends to request a logout from the target device.
logoutRequestContent	Use this property to specify any additional information that the probe sends with the logout request.
logoutRequestHeaders	Use this property to specify an HTTP header to send with all logout requests. This overrides the global HTTP header options configured by the **httpHeader** property.

Table 13. Properties in the `restWebSocketTransport.properties` file (continued)	
Property name	**Description**
resyncRequestURI	Use this property to specify the URI that the probe uses to request a resynchronization with the target system at startup. You can append the URI that you specify with this property with a query filter that limits the scope of the events that are returned by the request. **Note: resyncRequestURI** and **resyncRequestMethod=GET** can be used independently, together with **httpversion** and **responsetimeout**, to perform the resynchornization. However, either **initialResync** or **resyncinterval** must also be specified.
resyncRequestMethod	Use this property to specify the message type that the probe sends to request a resynchronization with the target system. **Note: resyncRequestURI** and **resyncRequestMethod=GET** can be used independently, together with **httpversion** and **responsetimeout**, to perform the resynchornization. However, either **initialResync** or **resyncinterval** must also be specified.
resyncRequestContent	Use this property to specify any additional information that the probe sends with the resynchronization request.
resyncRequestHeaders	Use this property to specify an HTTP header to send with all resynchronization requests. This overrides the global HTTP header options configured by the **httpHeader** property.
subscribeRequestURI	Use this property to specify the URI that the probe uses to request a subscription to receive new alarms as they are created in the target system.
subscribeRequestMethod	Use this property to specify the message type that the probe sends to request a subscription to receive new alarms.
subscribeRequestContent	Use this property to specify any additional information that the probe sends with the subscription request.
subscribeRequestHeaders	Use this property to specify an HTTP header to send with all subscription requests. This overrides the global HTTP header options configured by the **httpHeader** property.
subscribeRefreshURI	Use this property to specify the URI that the probe uses to request a subscription refresh.
subscribeRefreshMethod	Use this property to specify the message type that the probe sends to request a subscription refresh.

Table 13. Properties in the `restWebSocketTransport.properties` file (continued)	
Property name	**Description**
subscribeRefreshContent	Use this property to specify any additional information that the probe sends with the subscription refresh request.
subscribeRefreshHeaders	Use this property to specify an HTTP header to send with all subscription refresh requests. This overrides the global HTTP header options configured by the **httpHeader** property.
subscribeRefreshInterval	Use this property to specify the interval (in seconds) that the probe leaves between successive subscription refresh requests. This can be disabled by setting it to "" or leaving it commented out. If it is set to a negative value or an valid value such as `String`, it will default to 60 seconds.
keepTokens	Use this property to specify a comma-separated list of the attributes that the probe extracts from the incoming JSON data. These data items can be used in token substitution throughout the runtime of the probe.
webSocketHeaders	Use this property to specify a comma-separated list of HTTP headers to send with the initial Websocket handshake request. This overrides the global HTTP header options configured by the **httpHeader** property. Example use: By setting `"webSocketHeaders=Authorization=Bearer ++token++"`, the Authorization Header will be included in the handshake to provide a Bearer token for authentication. The `"++token++"` will be substituted with a value retrieved from prior requests such as login or resync requests. This must be used with the **keepTokens** property, for example: `"keepTokens=token"` so that the value of the `"token"` attribute received from the HTTP response is kept in memory.
webSocketURI	Use this property to specify the URI that the probe uses to request a WebSocket connection with the REST API. **Note: webSocketURI** can work independently allowing the probe to subscribe to the target system.
webSocketPeristentURI	Use this property to specify the persistent URI that the probe uses to re-establish a WebSocket connection. **Note:** This property takes precedence over **webSocketURI** if both are set. But, the probe uses **webSocketURI** if **webSocketPeristentURI** contains tokens that can not be resolved to construct the final URI to use to establish the WebSocket connection.
webSocketSubProtocol	Use this property to specify the WebSocket subprotocol to use.

Table 13. Properties in the `restWebSocketTransport.properties` file (continued)	
Property name	**Description**
webSocketSubscribeMessage	Use this property to specify the subscribe message that the probe sends to the server through the WebSocket channel after successfully establishing the connection.
webSocketRefreshMessage	Use this property to specify the refresh message to send in the `Ping` frame that the probe sends to the WebSocket.
webSocketRefreshInterval	Use this property to specify the interval (in seconds) that the probe leaves between sending successive refresh requests to the WebSocket. This can be disabled by setting it to "" or leaving it commented out. If it is set to a negative value or an valid value such as `String`, it will default to 60 seconds.
webSocketMaxFramePayloadLength	Use this property to configure the size in bytes of the payload frame of the WebSocket. Specify a value between 1 and 512000 bytes (512KB). The default value is 65536 (bytes). If the probe receives a websocket frame that is bigger than the value set for this property, the probe fails and writes a message to the error log.
refreshRetryCount	Use this property to specify the maximum number of times that the probes sends a refresh request. If set to an invalid value such as a string or a value less than zero, it will default to zero and no limit is applied. The default is 0.
securityProtocol	Use this property to specify the security protocol to use when retrieving events from the REST API.
tokenEndpointURI	Use this property to specify the URI that the probe uses to obtain an access token for the target device. This is the path on the remote host to request an access token, for example: `tokenEndpointURI=/oauth/token` By default, this property is not set, which disables the OAuth token request; no access token request will be sent to the server.
basicAuthenticationUsername	Use this property to specify the basic authentication username that the probe should use in the authentication header to gain access to the target device.
basicAuthenticationPassword	Use this property to specify the password associated with the basic authentication username that the probe should use in the authentication header.

Table 13. Properties in the restWebSocketTransport.properties file (continued)	
Property name	**Description**
clientId	Use this property to specify the `Client ID` registered with the `Oauth` server. This parameter will be sent in the HTTP request-body. Leave this property empty if this should be omitted to use the `Basic Authentication` client authentication method.
clientSecret	Use this property to specify the `Client Secret` string registered to the **clientId**. This parameter will be sent in the HTTP request-body. Leave this property empty if this should be omitted to use the `Basic Authentication` client authentication method.
scope	Use this property to specify the level of access to the target server that the probe is requesting. You can specify a comma-separated list of scopes, for example: `scope=read,write`

The following is an example of a `restWebSocketTransport.properties` file:

```
# Example format of Web Socket properties file
#httpVersion=1.1
#httpHeaders=
#responseTimeout=60
#loginRequestURI=
#loginRequestMethod=
#loginRequestContent
#loginRequestHeaders=
#loginRefreshURI=
#loginRefreshMethod=
#loginRefreshContent=
#loginRefreshHeaders=
#loginRefreshInterval=
#logoutRequestURI=
#logoutRequestMethod=
#logoutRequestContent=
#logoutRequestHeaders=
#resyncRequestURI=
#resyncRequestMethod=
#resyncRequestContent=
#resyncRequestHeaders=
#subscribeRequestURI=
#subscribeRequestMethod=
#subscribeRequestContent=
#subscribeRequestHeaders=
#subscribeRefreshURI=
#subscribeRefreshMethod=
#subscribeRefreshContent=
#subscribeRefreshHeaders=
#subscribeRefreshInterval=
#keepTokens=
#webSocketURI=
#webSocketPersistentURI=
#webSocketSubProtocol=
#webSocketSubscribeMessage=
#webSocketRefreshMessage=
#webSocketRefreshInterval=
#webSocketMaxFramePayloadLength=65536
#refreshRetryCount=0
#securityProtocol=
#tokenEndpointURI=
#basicAuthenticationUsername=
#basicAuthenticationPassword=
#clientId=
```

```
#clientSecret=
#scope=
```

Authenticating the probe Using OAuth authentication
The Web Hook and Web Socket transports support OAuth authentication with an OAuth server to request access to a restricted resource using an access token.

To request an access token, the transport uses the resource owner password credential grant method, whereby the probe sends a token request to the token end point specified by the **tokenEndpointURI** property in the `restWebhookTransport.properties` file.

Along with the token request, the probe sends the resource credentials specified by the **Username** and **Password** properties in the `message_bus.props` file to log into the OAuth server.

In response, the token end point sends to the probe the `access_token` and `refresh_token`.

The probe will send the `access_token` to gain access to the restricted resource and will send the `refresh_token` to refresh the access token shortly before it expires.

For OAuth servers that require a client to authenticate using the basic authentication method, set the <code> **basicAuthenticationUsername** and **basicAuthenticationPassword** properties.

Set the **clientId** and **clientSecret** properties to send the credentials in the P0ST request body.

Set the **scope** property to specify a comma-separated list of scopes to set in the access token request.

To use the access token in another transport property, use the `++0Auth.access_token++` variable name in the transport property. For example, to use this in the Web Hook transport **httpHeader** property: `httpHeader=Authorization=Bearer ++0auth.access_token++`

Authenticating the probe using REST or WebSocket
The probe can authenticate with the client by passing the username and password with every request it makes to the service as an HTTP basic authentication header.

To specify HTTP header options to use in all HTTP requests, use the **httpHeaders** property in the `restWebSocketTransport.properties` file. The **httpHeader** property allows you to specify a list of (comma-separated) HTTP header options.

To authenticate the probe, include `Authorization` in the list of header options. For example, to authenticate using the username `Me` and the password `MyPassword`, include in the value set for the **httpHeader** property the following header option:

`Authorization=Basic Me:MyPassword`.

The probe encodes the `Username:Password` credentials that follow the single whitespace after `Basic` as a Base-64 encoded string. The probe prefixes to the resulting string `Basic` and a space, and passes this as an authorization header with every request. The probe passes this as an authorization header with every request.

Connecting to WebSocket using SSL
The probe supports Secure Sockets Layer (SSL) connections between the probe and WebSocket. SSL connections provide additional security when the probe retrieves alarms from the target systems.

To enable SSL connections, obtain any required SSL certificates and Trusted Authority certificates for WebSocket. Add the certificates to a local Java keystore so that they can be referenced by the **KeyStore** property.

Prerequisites

The following tools are available to create the keystore:

- The OpenSSL toolkit.

  This is available from http://www.openssl.org/.
- The IBM KeyMan utility.

This is available from http://www.alphaworks.ibm.com/tech/keyman/download.

- The Keytool toolkit.

  This is available in the JRE package.

Converting the key and certificate into PKCS12 format

If you have a key and a certificate from the server in separate files, you must combine them into a single PKCS12 format file to load into a new keystore. To convert the server certificate into PKCS12 format, use the following OpenSSL toolkit command:

```
openssl pkcs12 -export -inkey key_file-in cert_file-out cert_pkcs12
```

Where

key_file is the key file retrieved from the server.

cert_file is the certificate retrieved from the server.

cert_pkcs12 is the combined file in PKCS12 format for loading into the keystore.

Creating the SSL keystore

You can create a Java keystore using either the KeyMan utility or the `Keytool` utility.

To create a Java keystore using the KeyMan utility, follow these steps:

1. Start the KeyMan utility.
2. Click **Create New** and select the **Keystore token** option.
3. Click **File** > **Import** and choose the certificate that you retrieved from the server.

   This imports the certificate into the keystore.
4. Click **File** > **Save** and enter a password and name for the keystore; for example, *trusted_keystore*.jks.

To create a Java keystore using the `Keytool` utility, follow these steps:

1. Generate a keystore and self-signed certificate using the following command:

   ```
   keytool -genkey -keyalg RSA -alias alias_name -keystore keystore_file -
   storepass keystore_password -validity 360 -keysize 2048
   ```
2. Import the WebSocket SSL certificate into the newly created Java keystore file using the following command:

   ```
   keytool -import -trustcacerts -alias alias_name -file cert_file -keystore
   keystore_file
   ```
3. Verify that the certificates are in a Java keystore using the following command:

   ```
   keytool -list -v -keystore keystore_file
   ```

Enabling SSL connections

To enable SSL-based connections between the probe and the Element Management System (EMS) server, make the following changes to the probe's properties file:

1. Set the **EnableSSL** property to `true`.

   When the **EnableSSL** property is set to `true`, the following properties are enabled:

   - **KeyStore**
   - **KeyStorePassword**
2. Use the **KeyStore** property to specify the location of the keystore file.
3. Use the **KeyStorePassword** property to specify a password for the keystore.

Note: You can encrypt the keystore file password using the `nco_aes_crypt` utility (for FIPS 104-2 mode security).

4. Set the **`Port`** property to the port that the probe uses for HTTPS connections.

Resynchronizing the probe with the REST API
The probe can create an ad hoc resynchronization request using REST API.

To do this, set the following properties in the `restWebSocketTransport.properties` file:

```
resyncRequestURI=/api/resynchronization
resyncRequestMethod=GET
resyncRequestContent=
```

Configuring HTTP requests
Each HTTP request can be configured by using the corresponding **`URI`**, **`Method`**, **`Content`** and **`Header`** transport properties.

For example, to configure the `loginRequest` use the following steps:

1. Specify the path in the **`loginRequestURI`** property, for example: `/login`
2. Specify the HTTP Method in the **`loginRequestMethod`** property, for example: `POST`, `GET`, `PATCH`, and so forth.
3. If required, specify the HTTP body in the **`loginRequestContent`** property, for example: `{\"sample\":\"json\"}`
4. If required, specify additional HTTP headers in the **`loginRequestRequestHeaders`** property. This will override any headers set in the **`httpHeader`** property.

Note: For `loginRefresh` and `subscribeRefresh` requests, you can use the **`loginRefreshInterval`** and **`subscribeRefreshInterval`** properties respectively to enable the HTTP request to be sent periodically.

Subscribing to receive notifications using WebSocket
You can instruct the probe to receive subscriptions through WebSocket.

To do this, set the following properties in the `restWebSocketTransport.properties` file:

```
webSocketURI=/websocket
httpHeaders=Authorization=Basic ++Username++:++Password++,Content-Type=application/json
httpVersion=1.1
responseTimeout=2
webSocketRefreshMessage=Ping
webSocketRefreshInterval=10
```

Note:

Leave the following properties empty:

```
subscribeRequestURI=
subscribeRequestMethod=
subscribeRequestContent=
subscribeRefreshURI=
subscribeRefreshMethod=
subscribeRefreshContent=
subscribeRefreshInterval=
```

Restarting the probe and re-connecting with the persistent URI
If the probe shuts down, you can specify that when it restarts, it resumes the subscription through the WebSocket with a different URI if necessary. For example, you can specify a URI with a query filter to query new alarms from a specific time.

You specify the URI with which the probe resumes the subscription using the **`webSocketPeristentURI`** property in the `restWebSocketTransport.properties` file. When the probe restarts, it resumes the subscription with this URI. If there are tokens used in the URI, it reconstructed with the last value

recorded in the **DataBackupFile**. This value is recorded in the file specified by the **DataBackupFile** property which is loaded during probe startup.

The content of the file specified by **DataBackupFile** is determined by the **RecordData** property. This property allows you to specify a comma-separated list of attributes from the event that the probe records in the data backup file.

Configuring the Web Hook transport

The Web Hook transport allows the probe to send the following types of HTTP requests to the target device:

- `loginRequest`: This is sent after the OAuth access token request to log into the target system.
- `loginRefresh` This is sent periodically to refresh the login on the target device.
- `logoutRequest` This is sent before disconnecting from the target device.
- `resynchRequest` This is sent to request an event resynchronization with the target device.
- `subscribeRequest` This is sent to subscribe to received events from the target device as they are created.
- `subscribeRefresh` This is sent periodically to refresh the event subscription on the target device.

The transport also enables the probe to create a callback URL to which the target server or an HTTP client can sed notifications using POST or GET requests.

Web Hook transport properties

The following table describes the properties in the `restWebhookTransport.properties` file.

Note: When using the Web Hook transport, you must specify values for the **Host** and **Port** properties in the `message_bus.props` file.

Table 14. Properties in the `restWebHookTransport.properties` file

Property name	Description
httpVersion	Use this property to specify the version of the HTTP protocol that the target system supports. The default is `1.1`.

Table 14. Properties in the `restWebHookTransport.properties` file (continued)	
Property name	**Description**
httpHeaders	Use this property to specify the HTTP header options to use in all HTTP requests. This property accepts a comma separated list of key-value pairs using the equals sign (=) as the value separator. For options that accept multiple values, use the double-quote (") character around the value and a use semi-colon (;) as the value separator; for example: `Keep-alive="timeout 30;max 10",Connection=Keep-alive` For the authorization header option, the credentials that follow the single whitespace after `Basic` are encoded as a Base-64 encoded string. For example, to authenticate with the username `Me`, and the password `MyPassword`, set the **httpHeader** property to `Authorization=Basic Me:MyPassword`. The probe passes this as an authorization header with every request. The default value is "". **Note:** The **httpHeaders** property sets the headers for all HTTP requests. However, you can override these global HTTP header options using the following properties: • **loginRequestHeaders** • **loginRefreshHeaders** • **logoutRequestHeaders** • **resyncRequestHeaders** • **subscribeRequestHeaders** • **subscribeRefreshHeaders**
responseTimeout	Use this property to specify how long (in seconds) the probe waits for a response from the target system before timing out. The default is 60 seconds.
loginRequestURI	Use this property to specify the URI that the probe uses to request a login.
loginRequestMethod	Use this property to specify the message type that the probe sends to request a login.
loginRequestContent	Use this property to specify any additional information that the probe sends with the login request.
loginRequestHeaders	Use this property to specify an HTTP header to send with all login requests. This overrides the global HTTP header options configured by the **httpHeader** property.
loginRefreshURI	Use this property to specify the URI that the probe uses to refresh the login on the target device.

Table 14. Properties in the `restWebHookTransport.properties` file (continued)	
Property name	**Description**
loginRefreshMethod	Use this property to specify the message type that the probe sends to refresh the login on the target device.
loginRefreshContent	Use this property to specify any additional information that the probe sends with the login refresh request.
loginRefreshHeaders	Use this property to specify an HTTP header to send with all login refresh requests. This overrides the global HTTP header options configured by the **httpHeader** property.
loginRefreshInterval	Use this property to specify the interval (in seconds) that the probe leaves between successive login refresh requests. This can be disabled by setting it to "" or leaving it commented out. If set to a negative value or an valid value such as `String`, it will default to 60 seconds.
logoutRequestURI	Use this property to specify the URI that the probe uses to request a logout from the target device.
logoutRequestMethod	Use this property to specify the message type that the probe sends to request a logout from the target device.
logoutRequestContent	Use this property to specify any additional information that the probe sends with the logout request.
logoutRequestHeaders	Use this property to specify an HTTP header to send with all logout requests. This overrides the global HTTP header options configured by the **httpHeader** property.
resyncRequestURI	Use this property to specify the URI that the probe uses to request a resynchronization with the target system at startup. You can append the URI that you specify with this property with a query filter that limits the scope of the events that are returned by the request. **Note: resyncRequestURI** and **resyncRequestMethod=GET** can be used independently, together with **httpversion** and **responsetimeout**, to perform the resynchornization. However, either **initialResync** or **resyncinterval** must also be specified.
resyncRequestMethod	Use this property to specify the message type that the probe sends to request a resynchronization with the target system. **Note: resyncRequestURI** and **resyncRequestMethod=GET** can be used independently, together with **httpversion** and **responsetimeout**, to perform the resynchornization. However, either **initialResync** or **resyncinterval** must also be specified.

Table 14. Properties in the `restWebHookTransport.properties` file (continued)	
Property name	**Description**
resyncRequestContent	Use this property to specify any additional information that the probe sends with the resynchronization request.
resyncRequestHeaders	Use this property to specify an HTTP header to send with all resynchronization requests. This overrides the global HTTP header options configured by the **httpHeader** property.
serverBasicAuthenticationUsername	Use this property to specify the webhook (server) basic authentication username.
serverBasicAuthenticationPassword	Use this property to specify the webhook (server) basic authentication password. This is used together with **serverBasicAuthenticationUsername** to support basic authentication for the server component and is used for client authentication using the HTTP Basic Authentication scheme. Both the **serverBasicAuthenticationUsername** and **serverBasicAuthenticationPassword** properties must be set to enable basic authentication. If these properties are set, the probe will accept HTTP requests with the correct credentials specified in the Authorization header. If the credentials do not match, the probe will reject the request with the error code: `401 Unauthorized`. An empty password is not supported. **Note:** You can encrypt the password using the `nco_aes_crypt` utility within Netcool/OMNIbus. The transport module only supports `AES_FIPS`, so you must use `-c AES_FIPS`.
subscribeRequestURI	Use this property to specify the URI that the probe uses to request a subscription to receive new alarms as they are created in the target system.
subscribeRequestMethod	Use this property to specify the message type that the probe sends to request a subscription to receive new alarms.
subscribeRequestContent	Use this property to specify any additional information that the probe sends with the subscription request.
subscribeRequestHeaders	Use this property to specify an HTTP header to send with all subscription requests. This overrides the global HTTP header options configured by the **httpHeader** property.
subscribeRefreshURI	Use this property to specify the URI that the probe uses to request a subscription refresh.
subscribeRefreshMethod	Use this property to specify the message type that the probe sends to request a subscription refresh.

Table 14. Properties in the `restWebHookTransport.properties` file (continued)	
Property name	**Description**
subscribeRefreshContent	Use this property to specify any additional information that the probe sends with the subscription refresh request.
subscribeRefreshHeaders	Use this property to specify an HTTP header to send with all subscription refresh requests. This overrides the global HTTP header options configured by the **httpHeader** property.
subscribeRefreshInterval	Use this property to specify the interval (in seconds) that the probe leaves between successive subscription refresh requests. This can be disabled by setting it to "" or leaving it commented out. If it is set to a negative value or an valid value such as `String`, it will default to 60 seconds.
keepTokens	Use this property to specify a comma-separated list of the attributes that the probe extracts from the incoming JSON data. These data items can be used in token substitution throughout the runtime of the probe.
refreshRetryCount	Use this property to specify the maximum number of times that the probes sends a refresh request. If set to an invalid value such as a string or a value less than zero, it will default to zero and no limit is applied. The default is 0.
securityProtocol	Use this property to specify the security protocol to use when retrieving events from the REST API.
tokenEndpointURI	Use this property to specify the URI that the probe uses to obtain an access token for the target device. This is the path on the remote host to request an access token, for example: `tokenEndpointURI=/oauth/token` By default, this property is not set, which disables the OAuth token request; no access token request will be sent to the server.
basicAuthenticationUsername	Use this property to specify the basic authentication username that the probe should use in the authentication header to gain access to the target device.
basicAuthenticationPassword	Use this property to specify the password associated with the basic authentication username that the probe should use in the authentication header.

Table 14. Properties in the `restWebHookTransport.properties` file (continued)	
Property name	**Description**
clientId	Use this property to specify the `Client ID` registered with the `Oauth` server. This parameter will be sent in the HTTP request-body. Leave this property empty if this should be omitted to use the `Basic Authentication` client authentication method.
clientSecret	Use this property to specify the `Client Secret` string registered to the **clientId**. This parameter will be sent in the HTTP request-body. Leave this property empty if this should be omitted to use the `Basic Authentication` client authentication method.
scope	Use this property to specify the level of access to the target server that the probe is requesting. You can specify a comma-separated list of scopes, for example: `scope=read,write`

webhookURI	Use this property to specify the path of the Web Hook URI on the local server into which the target device will POST notifications.
	This property can be set to a URL to specify the scheme (HTTP or HTTPS), port number, and path, for example:
	`webhookURI=/probe/webhook`
	or
	`webhookURI=http://hostname:80/probe/webhook`
	If only the path is specified, the HTTP(s) server will bind to the local port number specified in the **Port** property of the `message_bus.props` file. Make sure the local port is free.
	If unset or empty, the probe will generate a path.
	By default this property is set to `OFF`, the HTTP(s) server is disabled.

respondWithContent	Use this property to specify whether the probe includes the HTTP body received from the client HTTP request in the HTTP response. Set this property to `ON` to configure probe webhook to include the HTTP body. The default is `OFF`.
validateBodySyntax	Use this property to specify that the probe performs a JSON and XML syntax format check on the HTTP request body. The default is `ON`.

Table 14. Properties in the restWebHookTransport.properties file (continued)	
Property name	**Description**
validateRequestURI	Use this property to specify whether the probe checks the URI paths. Set this property to ON to enable URI path check. Setting this property to OFF disables the URI check and the webhook will accept all HTTP request regardless of the path set. The default is ON.
idleTimeout	Use this property to specify the time (in seconds) to allow an idle HTTP client to be connected. When the timeout period has elapsed, the idle client is disconnected to free up resources. The default is 180.

The following is an example of a `restWebHookTransport.properties` file:

```
# Example format of Web Hook properties file

#httpVersion=1.1
#httpHeaders=
#responseTimeout=60
#loginRequestURI=
#loginRequestMethod=
#loginRequestContent
#loginRequestHeaders=
#loginRefreshURI=
#loginRefreshMethod=
#loginRefreshContent=
#loginRefreshHeaders=
#loginRefreshInterval=
#logoutRequestURI=
#logoutRequestMethod=
#logoutRequestContent=
#logoutRequestHeaders=
#resyncRequestURI=
#resyncRequestMethod=
#resyncRequestContent=
#resyncRequestHeaders=
#subscribeRequestURI=
#subscribeRequestMethod=
#subscribeRequestContent=
#subscribeRequestHeaders=
#subscribeRefreshURI=
#subscribeRefreshMethod=
#subscribeRefreshContent=
#subscribeRefreshHeaders=
#subscribeRefreshInterval=
#keepTokens=
#refreshRetryCount=0
#securityProtocol=
#tokenEndpointURI=
#basicAuthenticationUsername=
#basicAuthenticationPassword=
#clientId=
#clientSecret=
#scope=
#webhookURI=OFF
#respondWithContent=OFF
#validateBodySyntax=ON
#validateRequestURI=ON
#idleTimeout=180
```

Authenticating the probe Using OAuth authentication
The Web Hook and Web Socket transports support OAuth authentication with an OAuth server to request access to a restricted resource using an access token.

To request an access token, the transport uses the resource owner password credential grant method, whereby the probe sends a token request to the token end point specified by the **tokenEndpointURI** property in the `restWebhookTransport.properties` file.

Along with the token request, the probe sends the resource credentials specified by the **Username** and **Password** properties in the `message_bus.props` file to log into the OAuth server.

In response, the token end point sends to the probe the `access_token` and `refresh_token`.

The probe will send the `access_token` to gain access to the restricted resource and will send the `refresh_token` to refresh the access token shortly before it expires.

For OAuth servers that require a client to authenticate using the basic authentication method, set the <code> **basicAuthenticationUsername** and **basicAuthenticationPassword** properties.

Set the **clientId** and **clientSecret** properties to send the credentials in the POST request body.

Set the **scope** property to specify a comma-separated list of scopes to set in the access token request.

To use the access token in another transport property, use the `++OAuth.access_token++` variable name in the transport property. For example, to use this in the Web Hook transport **httpHeader** property: `httpHeader=Authorization=Bearer ++Oauth.access_token++`

Specifying a callback URL
The Web Hook transport allows you to enable the HTTP(s) server capabilities (Web Hook) of the probe and to specify a callback URL to which a target server or an HTTP client can POST notifications.

Use the **webhookURI** property in the `restWebhookTransport.properties` file to specify a callback URL for the client to POST notifications.

The **webhookURI** property can specify the scheme (HTTP or HTTPS), port number, and path of the callback URL, for example:

`webhookURI=/probe/webhook`

or

`webhookURI=http://hostname:80/probe/webhook`

If you leave the **webhookURI** property blank, the probe will randomly generate a URL.

If you want to disable the Web Hook capabilities of the probe and only the use probe as an HTTP client, set the **webhookURI** property to OFF.

Subscribing to receive notifications using an HTTP server
You can use the Web Hook transport to create an HTTP endpoint to listen to notifications from a target system by sending the probe's webhook URL in a HTTP request body, such as a POST request.

To do this, set the following properties in the `restWebHookTransport.properties` file:

```
httpHeader=Authorization=Bearer ++Oauth.access_token++,Accept=application/json,
    Content-Type=application/json
subscribeRequestURI=/monitoring/external/api/v1/faults
subscribeRequestMethod=POST
subscribeRequestBody={"severities":["minor","major","critical","clear"],"notification-target":
     "++WebhookUrl++"}
```

Where `++WebhookUrl++` is a reserved variable that will be substituted with the full URL at runtime.

Configuring the Cometd transport

The Cometd transport allows the probe to connect to a target system using a Bayeux client.

CometD is a scalable HTTP-based event routing bus that uses an AJAX push technology pattern known as Comet. It implements the Bayeux protocol.

Cometd transport properties

The following table describes the properties used to configure the `cometdTransport.properties` file.

Table 15. Cometd transport properties

Property name	Description
bayeuxClientTransport	Use this property to which transport the probe uses to connect to the Bayeux server. This property can take the following values: `long-polling`: The probe only uses the long-polling transport. `websocket`: The probe uses the Websocket transport first and falls back to long-polling. The default is `long-polling`.
serverUri	Use this property to specify the Cometd server path to include in the URL used to connect to the comet server. The default is `/cometd/cometd`
subscriptionChannel	Use this property to specify the channel on the Cometd server to which the probe subscribes to receive events. The default is `/channel`
loginRequests	Use this property to send REST requests to login or authenticate with the target system. The **loginRequest** property takes in a JSON with the following format and constructs an HTTP request to send to the server right after the probe connects to the server: `{"requests":[{"uri":"/some/uri","method":"POST","header":{"Content-Type":"application/json","Authorization":"Basic ++Username++:++Password++"},"data":"{\"username\":\"++Username++\",\"password\":\"++Password++\""},{"uri":"/some/other/uri","method":"POST","data":"{}","header":{"Content-Type":"application/json","Connection":"Keep-Alive"}}]}`

Table 15. Cometd transport properties (continued)	
Property name	**Description**
resyncRequests	Use this property to request a resynchronization with the target system. The **resyncRequests** property takes the same JSON format as the **loginRequests** property. Responses from **resyncRequests** will be added to the probe's message queue for parsing.
logoutRequests	Use this property to un-subscribe or initiate a logout from a system. The **logoutRequests** property takes the same JSON format as the loginRequests property. Logout requests will be sent to the server before the probe disconnects.
securityProtocol	Use this property to specify the security protocol to use when retrieving events from the target system.
connectTimeout	Use this property to specify how long (in seconds) the probe waits for a Web Socket connection to be opened before timing out. The default is 320.
responseTimeout	Use this property to specify how long (in seconds) the probe waits for an exchange or reply from the server before timing out. The default is 20. **Note:** You should set this parameter to a value greater than the estimated network delay.
idleTimeout	Use this property to specify how long (in seconds) the probe allows the HTTP Client component of the CometD transport to be idle before timing out. The default is 20.
maxMessageSize	Use this property to specify the maximum number of bytes allowed for each WebSocket message. The default is 8192.
stickyReconnect	Use this property to specify whether to keep using the Web Socket transport when the probe detects a Web Socket transport failure after the Web Socket transport successfully connected to the server. The default is true.

Table 15. Cometd transport properties (continued)	
Property name	**Description**
httpHeaders	Use this property to set or override any HTTP header options such as Authorization header for basic authentication. This property is in JSON format. Example value for this property: `{"Accept":"application/json","User-Agent":"IBM Netcool/OMNIbus Message Bus Probe"}`

The following is an example of an Cometd properties file:

```
# Example format of CometD properties file

# bayeuxClientTransport=long-polling
# serverUri=/cometd/cometd
# subscriptionChannel=/channel
# loginRequests=
# resyncRequests=
# logoutRequests=
# securityProtocol = "TLSv1.2"
# connectTimeout=75
# responseTimeout=320
# idleTimeout=20
# maxMessageSize=8192
# stickyReconnect=true
# httpHeaders=
```

Configuring the socket transport

The socket transport allows the probe to connect to a target system and subscribe to receive all events sent to a specific port.

Socket transport properties

The following table describes the properties in the `socketTransport.properties` file.

Note: When using the Socket transport, you must specify values for the **Host** and **Port** properties in the `message_bus.props` file.

Table 16. Properties in the `socketTransport.properties` file	
Property name	**Description**
serverPort	Use this property to specify the port on the server to which the target system sends events.
clientAddress	Use this property to specify the client address to which the probe connects.
subscribeMessages	Use this property to specify a list of the messages that the probe will send when subscribing as an asynchronous client.

Table 16. Properties in the socketTransport.properties file (continued)	
Property name	**Description**
subscribeResponses	Use this property to specify a list of response processors for the subscribe messages. **Note:** This property must contain the same number of entries as the **subscribeMessages**property.
disconnectMessages	Use this property to specify the messages sent when disconnecting an asynchronous client.
disconnectResponses	Use this property to specify response processors for the disconnect messages.
activeAlarmMessages	Use this property to specify the messages sent when requesting active alarms as an asynchronous client.
messageTerminator	Use this property to specify the regular expression that marks the end of each alert from the target system.
readResponseAttempts	Use this property to specify the number of times the probe should attempt to read a response sent as a result of subscribing, disconnecting, or getting active alarms.
readResponseDelay	Use this property to specify the interval (in seconds) that the probe should wait between successive read attempts.
threadPoolSize	Use this property to specify the maximum number of threads to use to process connections made to the probe when it is running as a server.

The following is an example of a socketTransport.properties file:

```
# Example format of Socket properties file
# serverPort=12345
# clientAddress=some.host.com:port
# subscribeMessages=/first/message.txt:/second/message.xml
# subscribeResponses=none:xslt("/second/response_processor.xsl","\n\n")
# disconnectMessages=/first/message.txt:/second/message.xml
# disconnectResponses=none:xslt("/second/response_processor.xsl","\n\n")
# activeAlarmMessages=/first/message.txt:/second/message.xml
# messageTerminator=\n\n
# readResponseAttempts = 60
# readResponseDelay = 1
# threadPoolSize = 10
```

Configuring the Kafka transport

The Kafka transport allows the probe to integrate with a Kafka server to consume events.

Kafka transport properties table

The following table describes the properties used to configure the kafkaTransport.properties file.

Table 17. Kafka transport properties	
Property name	**Description**
kafkaClientMode	Use this property to set the transport as a Kafka client to run as a consumer or a producer. This property takes the following values: CONSUMER: A Kafka consumer reads data from topics. PRODUCER: A Kafka producer writes data to topics.
connectionPropertiesFile	Use this property to specify the JSON file holding the Kafka connection properties.

Kafka connection properties table

The following table describes the properties used to configure the kafkaConnectionProperties.json file.

Table 18. Kafka connection properties		
Property name		**Description**
zookeeper_client	**target**	Use this property to specify the ZooKeeper endpoint. When this property is empty, the transport will not initiate connection to ZooKeeper. The default is `empty`.
	properties	Use this property to specify the path to a file holding ZooKeeper client properties in key-value format, for example: `key=value` The default is `empty`.
	java_sys_props	Use this property to specify the path to a file holding ZooKeeper client Java system properties required in a secure connection. The default is `empty`.
	topic_watch	Use this property to enable the ZooKeeper topic watch service. Valid values are: `true`: Enable the ZooKeeper topic watch service. `false`: Disable the ZooKeeper topic watch service. The default is `true`.
	broker_watch	Use this property to enable the ZooKeeper broker watch service. Valid values are: `true`: Enable the ZooKeeper broker watch service. `false`: Disable the ZooKeeper broker watch service. The default is `true`.
brokers		Use this property to specify broker endpoints in a comma-separated list. For example: “`localhost:9092, localhost:9093, localhost:9094`” The brokers must belong to the same cluster managed by a zookeeper. The default is `empty`.
topics		Use this property to specify topics in a comma-separated list. For example: “`topic1, topic2, topic3`” The default is `empty`.

Table 18. Kafka connection properties (continued)		
Property name		**Description**
Kafka_client	**properties**	Use this property to specify the path to a file holding Kafka client properties. The default is `empty`.
	java_sys_props	Use this property to specify the path to a file holding the Kafka client's Java system properties required in a secure connection. The default is `empty`.

Kafka configuration for different connection protocols

Kafka supports following types of connection protocol:

- SASL_PLAINTEXT
- SASL_SSL

The following table describes the configuration required by each connection protocol.

Table 19. Connection protocol configuration	
Connection protocol	**Configuration required**
SASL_PLAINTEXT	**Kafka producer properties** `security.protocol=SASL_PLAINTEXT` **Note:** Must combine with SASL-specific configurations.

SASL_SSL	**Kafka producer properties**
	`acks=all`
	`security.protocol=SASL_SSL`
	`sasl.mechanism=PLAIN`
	`ssl.protocol=TLSv1.2`
	`ssl.enabled.protocols=TLSv1.2`
	`ssl.truststore.location=<path>\<trust_store_file>`
	`ssl.truststore.password=<trust_store_password>`
	`ssl.truststore.type=JKS`
	Java system properties
	`java.security.auth.login.config=<path>/user_jaas.conf`
	`https.protocols=TLSv1.2`
	Note: Must combine with SASL-specific configurations.

The following table describes SASL-specific configurations.

Table 20. SASL-specific configuration

SASL: Kafka user access control	SASL: Kerberos

Java system properties
`java.security.auth.login.config=<path>/user_jaas.conf`
Example user_jass.conf
`KafkaClient {`
`org.apache.kafka.common.security.plain.PlainLoginModule required`
`  serviceName="kafka"`
`  username="myUserName"`
`  password="myPasword";`
`};`
	Java system properties
	`java.security.auth.login.config=<path>/user_jass.conf`
	`java.security.krb5.conf=<path>/krb5.conf`
	Example user_jass.conf
	When using IBM JDK
	`KafkaClient {`
	`com.ibm.security.auth.module.Krb5LoginModule required`
	`  debug=true`
	`  credsType=both`
	`  useKeytab="<path>/kafka.keytab"`
	`  principal="username/instance@realm";`
	`};`
	When using Oracle JDK
	`KafkaClient {`
	`com.sun.security.auth.module.Krb5LoginModule required`
	`  debug=true`
	`  renewTicket=true`
	`  serviceName="kafka"`
	`  useKeyTab=true`
	`  keyTab="<path>/kafka.keytab"`
	`  principal="username/instance@realm";`
	`};`
	Note: This is the generic format of principal: *username/instance@realm*. Some organizations might use *servicename* instead of *username* or without *username*
	`principal="servicename/instance@realm"`
	`principal="instance@realm`
	Consult your organization administrator for principal information.

Note:

Kafka producer properties are configured in the file specified in the **kafka_client.properties** field.

Java system properties are configured in the file specified in the **kafka_client.java_sys_props** field.

In broker list configuration, a broker endpoint without a protocol prefix is assumed to be using the protocol configured in the **security.protocol** property. An unconfigured **security.protocol** denotes PLAINTEXT.

Message Bus Probe integrations with event sources

You can use the Message Bus Probe to integrate Netcool/OMNIbus with XML or JSON event sources.

The probe comes with configuration files for integrating with the following event sources:

- Amazon Web Services

- Ciena Blue Planet MCP
- IBM Cloud Private
- IBM Event Streams for IBM Cloud
- iDirect Pulse
- Juniper Contrail Alert API

  Note: The integration with Juniper Contrail Alert API is described in a separate guide, which you can access using the above link.
- Kafka
- Microsoft Azure Monitoring
- Nokia 1350 OMS
- Nokia Network Services Platform (NSP)

  Note: The integration with Nokia NSP is described in a separate guide, which you can access using the above link.

Probe integration for Amazon Web Services

The Message Bus Probe can be configured to obtain events from Amazon Web Services. The integration requires the following items to be installed on the probe's server:

- `Curl`
- `Perl`
- Scope-based event grouping

  For details see: https://www.ibm.com/support/knowledgecenter/en/SSSHTQ_8.1.0/com.ibm.netcool_OMNIbus.doc_8.1.0/omnibus/wip/install/task/omn_con_ext_installingscopebasedegrp.html

The following configuration files are supplied with the probe for the integration:

- `aws_NHttp_MsgBusProbe.pl`
- `aws_create_MsgBusTools.sql`
- `aws_remove_MsgBusTools.sql`
- `awsWebhookTransport.properties`
- `message_bus_aws.props`
- `message_bus_aws.rules`
- `message_bus_aws.json`

Note: On UNIX, make `aws_NHttp_MsgBusProbe.pl` an executable file using the command `chmod +x <filename>`.

To enable the Message Bus Probe to receive notifications from AWS using the Webhook transport, use the following steps:

1. Update the following property in the `awsWebhookTransport.properties` file:

   `webhookURI=/probe/aws`

   Note: The value set for **webhookURI** must be consistent with the message endpoint configured on AWS.
2. Set the following properties in the `message_bus_aws.props` file:

```
.
.
.
Server                          : '<YOUR_OBJECT_SERVER>'
JsonParserName                  : 'AWS'
TransformerFile                  : '${OMNIHOME}/probes/<platform>/message_bus_aws.json'
```

```
RulesFile                          : '$OMNIHOME/probes/<platform>/message_bus_aws.rules'

Host                              : '<probe_server_IP>'
Port                              : <probe_server_port>
TransportFile                     : '${OMNIHOME}/java/conf/awsWebhookTransport.properties'
TransportType                     : 'Webhook'

EnableSSL                         : 'true'
KeyStore                          : '<PATH_TO_YOUR_KEYSTORE_FILE>.jks'
KeyStorePassword                  : '<YOUR_KEYSTORE_PASSWORD>'

NHttpd.EnableHTTP                : TRUE
NHttpd.ExpireTimeout             : 30
NHttpd.ListeningPort             : 8899
                           # <this is an example; other available ports can be used>
.
.
.
```

3. Create MessageBus probe tools for AWS:

   ```
   $OMNIHOME/bin/nco_sql -user user -password password <
   aws_create_MsgBusTools.sql
   ```

   The command performs the following steps:

   a. Adds two columns to the `alerts.status` table: **AWSTargetConfirmation**, **AWSAutoConfirm**.

   b. Creates a trigger group `aws_triggers` holding **aws_target_confirmation** and **aws_process_target_status**.

4. The forwarding of AWS notifications requires confirmation of SubscribeURL. Before running the probe, decide the option for the AWS target confirmation: auto or manual.

   a. Edit `message_bus_aws.rules` under the section:

   ```
   if (match( $(json.Type), "SubscriptionConfirmation" ))
   ```

   b. Amend the `@AWSAutoConfirm` assignment with `1` for auto, `0` for manual.

   The act of confirmation is achieved by sending an HTTP command to the AWS's SubscribeURL or opening the SubscribeURL using a web browser.

   Auto confirmation method:

   This involves `message_bus_aws.rules` and the two triggers: `aws_target_confirmation` and `aws_process_target_status`.

   Right after where the SubscriptionConfirmation message is inserted at the ObjectServer, the trigger will invoke the `aws_NHttp_MsgBusProbe.pl` script to send a **GET** command to the SubscribeURL, and feedback the HTTP response to the probe rules file for status update.

   Manual confirmation method:

   Copy the SubscribeURL from the **ExtendedAttr** field to a web browser, and open the page.

5. Go to the AWS system and configure the notification destination (subscribe the probe's webhook).

   During this time the probe must be alive so that the SubscriptionConfirmation message can reach the ObjectServer through the probe.

   To run the PA daemon, use the following command:

   ```
   $OMNIHOME/bin/nco_pad -name NCO_PA -configfile $OMNIHOME/etc/nco_pa.conf -
   admingroup root
   ```

   To check the PA status, use the following command:

   ```
   $OMNIHOME/bin/nco_pa_status -server NCO_PA -user <unix_root> -password
   <unix_root_password>
   ```

6. In the Event List, the SubscriptionConfirmation message appears as an event with the Summary starting `AWS Subscription...` followed by a brief description of the confirmation state. The SubscribeURL string is stored in the `ExtendedAttr` field.

Note: SubscribeURL can comprise a few hundred characters, hence the use of **ExtendedAttr** (length: 4096) as its storage.

7. If the SubscriptionConfirmation message Summary indicates that the URL has not yet been successfully confirmed (regardless of the message being fresh or auto-confirmation attempt having failed), use the manual method.
8. After SubscribeURL confirmation, verify that the probe can receive AWS notifications.
9. You can choose to keep or to delete the SubscriptionConfirmation message in the ObjectServer after the integration with AWS is established.
10. To stop AWS sending notifications to the probe, unsubscribe the probe's webhook in AWS.

    The probe will receive an UnsubscribeConfirmation message.

Probe integration for Ciena Blue Planet MCP

The Message Bus Probe can be configured to obtain events from Blue Planet MCP Release 3.0.

The following configuration files are supplied with the probe for the integration:

- message_bus_ciena_mcp.props
- message_bus_ciena_mcp.rules
- message_bus_ciena_notificationMap.rules
- message_bus_ciena_resyncMap.rules
- cienaMcpTransport.properties
- ciena_mcp_parser_config.json

To enable the Message Bus Probe to connect to the Blue Planet MCP REST API, and to subscribe to notifications using Web Socket, use the following steps:

1. Configure the probe properties in the message_bus_ciena_mcp.props file.

```
Username           : 'username'
Password           : 'password'
Host               : 'mcp.ciena.com'
Port               : 443
EnableSSL          : 'true'
KeyStore           : '/home/netcool/keystore.jks'
KeyStorePassword   : 'password'

RulesFile          : '$OMNIHOME/probes/linux2x86/message_bus_ciena_mcp.rules'
PropsFile          : '$OMNIHOME/probes/linux2x86/message_bus_ciena_mcp.props'
TransportFile      : '$OMNIHOME/java/conf/cienaMcpTransport.properties'
TransportType      : 'WebSocket'
TransformerFile    : '$OMNIHOME/probes/linux2x86/ciena_mcp_parser_config.json'
MessageLog         : '$OMNIHOME/log/message_bus_ciena_mcp.log'

InitialResync      : 'true'
HeartbeatInterval  : 1
```

   Update the properties with the values appropriate for your system.

2. Configure the transport properties in the cienaMcpTransport.properties file.

```
## Get the client token from /tron/api/v1/tokens
loginRequestURI=/tron/api/v1/tokens
loginRequestMethod=POST
loginRequestContent={\"username\":\"++Username++\",\"password\":\"++Password++\"}
```

   ++Username++ and ++Password++ used in loginRequestContent will be substituted with the values specified for **Username** and **Password** in the probe properties file. When correctly authenticated, the response will contain a token which will be used in subsequent API queries.

```
#Use with pagination
resyncRequestURI=/nsa/api/v2_0/alarms/filter/filteredAlarms?filter%5Bstate%5D%5B
%5D=ACTIVE&filter%5Bseverity%5D%5B%5D=CRITICAL%2CMAJOR%2CMINOR%2CWARNING&offset=+
+{0,100,100}++&pageSize=100
```

```
resyncRequestMethod=GET
resyncRequestHeaders=Authorization=Bearer ++token++
```

These properties specify the Alarms (v2) REST API query to retrieve network alarms for resynchronisation purposes. This query filters only for active alarms having `Critical`, `Major`, `Minor` and `Warning` severities. The token value will be substituted with the token value retrieved from successful authentication with Blue Planet MCP.

The query also uses an offset pagination functionality to enable the probe to request for data in batches. The offset value `++{start, stop, step}++` will be substituted with the correct number at runtime by the probe. In the example above, the offset number is derived by adding the start and step numbers namely: `0+100=100`. You should specify `0` for the `start` value to indicate that the start of the returned data is from the first record and specify similar values for the `step` and `pageSize`. The `stop` value is currently not used to calculate the offset value and can be ignored. When 100 is specified for the `step` and `pageSize`, MCP returns 100 records for each queried batch.

```
## Connecting to Frostpush websocket
webSocketURI=/kafkacomet/socket/websocket?user_id=++user++&vsn=1.0.0
webSocketHeaders=Authorization=Bearer ++token++
webSocketSubscribeMessage={\"topic\":\"topics:bp.aeprocessor.v1.alarms\",\"ref\":0,\"event
\":\"phx_join\",\"payload\":{}}
webSocketRefreshMessage={\"topic\":\"topics:bp.aeprocessor.v1.alarms\",\"event\":\"heartbeat
\",\"payload\":{},\"ref\":\"1\"}
webSocketRefreshInterval=20
```

These properties configure the probe to subscribe to notifications using WebSocket. The `++user++` variable will be substituted with the value specified for **`Username`** in the probe properties file.

To keep the WebSocket connection open, a heartbeat must be sent with a frequency of less than every 30 seconds. If the Blue Planet MCP server does not receive a heartbeat every 30 seconds, in the absence of any other messages, the connection will be terminated. The default `webSocketRefreshInterval` value is 20 seconds.

```
## Refresh the tokens every hour , change the interval if necessary.
loginRefreshURI=/tron/api/v1/tokens
loginRefreshMethod=POST
loginRefreshContent={\"username\":\"++Username++\",\"password\":\"++Password++\"}
loginRefreshInterval=3600
refreshRetryCount=5
```

Blue Planet MCP sets the session inactivity timeout value per system or user, this can be viewed and configured in the Blue Planet MCP UI **System > Security > Inactivity** tab. For the HTTP channel to be kept alive, `loginRefreshInterval` should be set to a value lower than the MCP session inactivity timeout value. The default `loginRefreshInterval` value is 3600 seconds. The default `refreshRetryCount` is 5 times.

3. Configure the transformer properties in the `ciena_mcp_parser_config.json` file. Update the endpoint attribute of `NotificationAlarmParser` to match the `webSocketURI` used in the `cienaMcpTransport.properties` file.

```
{
  "eventSources" : [ {
    "endpoint" : "/kafkacomet/socket/websocket",
    "name" : "NotificationAlarmParser",
    "config" : {
      "dataToRecord" : [ ],
      "messagePayload" : "json.payload.body.value",
      "messageHeader" : "",
      "jsonNestedPayload" : "json.event.alarm",
      "jsonNestedHeader" : "",
      "messageDepth" : 4
    }
  }, {
    "endpoint" : "resync",
    "name" : "ResyncAlarmParser",
    "config" : {
      "dataToRecord" : [ ],
      "messagePayload" : "json.data",
      "messageHeader" : "",
      "jsonNestedPayload" : "",
```

```
      "jsonNestedHeader" : "",
      "messageDepth" : 3
    }
  }, {
    "name" : "OtherAlarmParser",
    "type" : "ANY",
    "config" : {
      "dataToRecord" : [ ],
      "messagePayload" : "json",
      "messageHeader" : "",
      "jsonNestedPayload" : "",
      "jsonNestedHeader" : "",
      "messageDepth" : 5
    }
  }]
}
```

Probe integration for IBM Cloud Platform Common Services (CS) Monitoring

The Message Bus Probe can be configured to integrate with IBM Cloud Platform Common Services (CS) Monitoring by using either Logstash or Prometheus to forward CS events to Netcool/OMNIbus.

CS uses Logstash to collect and normalize container logs running on CS. A Logstash pipeline can be configured to forward these normalized events in JSON format to the probe. Prometheus is an open-source system monitoring and alerting toolkit. It collects metrics from CS and allows users to configure alerting rules to send alerts when a condition is triggered.

You can choose to receive CS events via either Logstash, Prometheus or both. Two sets of configuration files are provided with the probe: one for Logstash and one for Prometheus. To receive CS events for both, two probes are required.

Note: A containerised version of the Probe for Message Bus is also available. It is delivered in the form of helm packages: one for Logstash and one for Prometheus. Each package pulls the probe Docker image and deploys a cluster of containerised Message Bus Probes within the Kubernetes platform environment. The probes process events and alerts from Logstash or Prometheus and send them to a NOI operational dashboard. For details about the helm package used to deploy a cluster of Probes for Message Bus onto Kubernetes see https://www.ibm.com/support/knowledgecenter/en/SSSHTQ/omnibus/helms/cloud_monitoring/wip/concept/cemh_intro.html.

Configuring the Message Bus Probe to receive notifications from Prometheus

The Message Bus Probe can be configured to integrate with IBM Cloud Platform Common Services on Red Hat Open Shift Container Platform (OCP) using Prometheus to forward cluster alerts to Netcool/OMNIbus.

The following configuration files are supplied with the probe for the integration with CS Monitoring using Prometheus:

- `message_bus_prometheus.props`
- `message_bus_prometheus.rules`
- `message_bus_prometheus_parser.json`
- `prometheusWebhookTransport.properties`

Note: The default probe rules require OMNIbus event grouping triggers to be installed.

To integrate the Message Bus Probe with CS Monitoring using Prometheus, use the following steps:

1. Install/update the Message Bus Probe using IBM Installation Manager, see “Installing probes” on page 4.
2. Edit the probe configuration in the following probe properties file:

   `$OMNIHOME/probes/<arch>/message_bus_prometheus.props`

   Where *<arch>* is the architecture directory, for example `linux2x86`
3. Update the following property values with the appropriate path:

```
PropsFile         : '$OMNIHOME/probes/linux2x86/message_bus_prometheus.props'
RulesFile         : '$OMNIHOME/probes/linux2x86/message_bus_prometheus.rules'
```

```
TransportType     : 'Webhook'
TransportFile     : '$OMNIHOME/java/conf/prometheusWebhhookTransport.properties'
```

4. Edit the **Port** property to a free local port number, for example: 80.
5. Edit the Webhook transport configuration in the following transport properties file:

   $OMNIHOME/probes/java/conf/prometheusWebhookTransport.properties
6. Update the following property value with the appropriate path:

```
webhookURI=/probe/webhook/prometheus
```

7. Update the endpoint attribute in $OMNIHOME/probes/*<arch>*linux2x86/message_bus_prometheus_parser.json to be the same as the **webhookURI** path if you have updated the **webhookURI** property.
8. If you require secure connection, refer to "Connecting to WebSocket using SSL" on page 29.
9. Start the probe with the propsfile option to specify the Prometheus properties file using the following command:

   $OMNIHOME/probes/nco_p_message_bus -propsfile $OMNIHOME/probes/linux2x86/message_bus_prometheus.props
10. Verify that the probe is running.

    The Webhook URL will be logged in the probe log file for reference. This is the URL of the probe Webhook.
11. Re-configure Prometheus in ICP to forward HTTP POST notifications to the probe host with the path specified in the **webhookURI** property.

    For example: http://<probehost>:80/probe/webhook/prometheus

    See "Configuring Prometheus in Kubernetes from the command line" on page 53.
12. Optional. Verify the probe version info with the version command line option using the following command:

    $OMNIHOME/probes/nco_p_message_bus -version

Configuring Prometheus in Kubernetes from the command line

Procedure for configuring Prometheus to point to the probe's webhook running on Red Hat Open Shift Container Platform (OCP). It can also be used with the on-premises version of the probe.

Modifying Prometheus Alert Manager and Alert Rules Configuration for OCP Monitoring

1. Determine the Prometheus Alert Manager configuration secret in the cluster. The default Secret that contains the Alert Manager configuration is in openshift-monitoring. See Applying custom Alertmanager configuration
2. A sample Alert Manager configuration with the probe webhook config applied is shown below. The sample endpoint http://<probehost>:80/probe/webhook/prometheus.

```
global:
  resolve_timeout: '5m'
receivers:
- name: 'null'
- name: 'netcool_probe'
  webhook_configs:
  - url: 'http://<probehost>:80/probe/webhook/prometheus'
    send_resolved: true
route:
  group_by:
  - alertname
  group_interval: 5m
  group_wait: 30s
  receiver: netcool_probe
  repeat_interval: 5s
  routes:
  - receiver: netcool_probe
```

```
    match:
      alertname: Watchdog
```

3. Apply the updated Alert Manager configuration file.
4. For details about applying custom alerting rules, see Managing cluster alerts.
5. Verify that your probe is receiving the OCP Monitoring alerts and events appear on the Netcool/OMNIbus Event List.

Modifying Prometheus Alert Manager and Alert Rules on IBM Cloud Platform Common Services in Red Hat OCP 4.2

To modify the default CS Monitoring configuration, use the following steps:

1. Determine the `Prometheus Alert Manager` config map in the `kube-system` namespace. In the default configmaps in the `kube-system` namespace it is: `monitoring-prometheus-alertmanager`.
2. Edit the **`Prometheus Alert Manager`** config map to add a new receiver in the `receivers` section. The default Prometheus deployment config map name is `monitoring-prometheus-alertmanager` in the `kube-system` namespace. If a separate Prometheus or CS Monitoring instance is deployed, determine the `alertmanager` config map and add the new receiver. To do this from the command line, configure the `kubectl` client and follow the steps below.
3. Load the config map into a file using the following command:

   ```
   kubectl get configmap monitoring-prometheus-alertmanager --namespace=kube-system -o yaml > alertmanager.yaml
   ```

4. Edit the `alertmanager.yaml` file and add the configuration as shown below:

   ```
   route:
       receiver: 'netcool_probe'

   receivers:
   - name: 'netcool_probe'
     webhook_configs:
     - url: 'http://<probehost>:80/probe/webhook/prometheus'
       send_resolved: true
   ```

   Replace the `url` parameter with the probe's webhook URL. This can be the probe's webhook URL deployed either on Kubernetes or on-premises.

5. Save the changes in the file and replace the config map using the following command:

   ```
   $ kubectl replace configmaps monitoring-prometheus-alertmanager --namespace=kube-system -f alertmanager.yaml

   configmap "monitoring-prometheus-alertmanager" replaced
   ```

6. Review the sample alert rules CRD YAML below. You may update the rules or add more rules to generate more alerts to monitor your cluster. The Message Bus Probe rules file expects the following attributes from the alerts generated by Prometheus Alert Manager:
   - `labels.severity`: The severity of the alert. Should be set to `critical`, `major`, `minor`, or `warning`. This is mapped to the `Severity` field in the ObjectServer `alerts.status` table.
   - `labels.instance`: The instance generating the alert. This is mapped to the `Node` field in the ObjectServer `alerts.status` table.
   - `labels.alertname`: The alert rule name. This is mapped to the `AlertGroup` field in the ObjectServer `alerts.status` table.
   - `annotations.description`: (Optional) The full description of the alert. This is mapped to the `Summary` field in the ObjectServer `alerts.status` table.
   - `annotations.summary`: A short description or summary of the alert. This is mapped to the `Summary` field in the ObjectServer `alerts.status` table if `annotations.description` is unset.

- `annotations.type`: The alert type. For example, "`Container`", "`Service`", or "`Service`". This is mapped to the `AlertKey` field in the ObjectServer `alerts.status` table.
- `labels.release`: (Optional) If set, will be mapped to the `ScopeId` field in the ObjectServer `alerts.status` table which will be used as the first level group to group related events.
- `labels.job`: (Optional) If set, will be mapped to the `SiteName` field in the ObjectServer `alerts.status` table which will be used as the sub-group to group related events.

Note: Sample alert-rules CRD. This file is also available in the included CloudPak under `pak_extensions/prometheus-rules`.

```
# File: netcool-rules.yaml
# Please modify these rules to monitor specific workloads,
# containers, services or nodes in your cluster
apiVersion: monitoringcontroller.cloud.ibm.com/v1
kind: AlertRule
metadata:
  name: netcool-rules
spec:
  enabled: true
  data: |-
    groups:
    - name: alertrules.rules
      rules:
      ## Sample workload monitoring rules
      - alert: jenkins_down
        expr: absent(container_memory_usage_bytes{pod_name=~".*jenkins.*"})
        for: 30s
        labels:
          severity: critical
        annotations:
          description: Jenkins container is down for more than 30 seconds.
          summary: Jenkins down
          type: Container
      - alert: jenkins_high_cpu
        expr: sum(rate(container_cpu_usage_seconds_total{pod_name=~".*jenkins.*"}[1m]))
          / count(node_cpu_seconds_total{mode="system"}) * 100 > 70
        for: 30s
        labels:
          severity: warning
        annotations:
          description: Jenkins CPU usage is {{ humanize $value}}%.
          summary: Jenkins high CPU usage
          type: Container
      - alert: jenkins_high_memory
        expr: sum(container_memory_usage_bytes{pod_name=~".*jenkins.*"}) > 1.2e+09
        for: 30s
        labels:
          severity: warning
        annotations:
          description: Jenkins memory consumption is at {{ humanize $value}}.
          summary: Jenkins high memory usage
          type: Container
      ## End - Sample workload monitoring rules.
      ## Sample container monitoring rules
      - alert: container_restarts
        expr: delta(kube_pod_container_status_restarts_total[1h]) >= 1
        for: 10s
        labels:
          severity: warning
        annotations:
          description: The container {{ $labels.container }} in pod {{ $labels.pod }}
            has restarted at least {{ humanize $value}} times in the last hour on instance
            {{ $labels.instance }}.
          summary: Containers are restarting
          type: Container
      ## End - Sample container monitoring rules.
      ## Sample node monitoring rules
      - alert: high_cpu_load
        expr: node_load1 > 1.5
        for: 30s
        labels:
          severity: critical
        annotations:
          description: Docker host is under high load, the avg load 1m is at {{ $value}}.
            Reported by instance {{ $labels.instance }} of job {{ $labels.job }}.
          summary: Server under high load
          type: Server
```

```
      - alert: high_memory_load
        expr: (sum(node_memory_MemTotal_bytes) - sum(node_memory_MemFree_bytes +
node_memory_Buffers_bytes
          + node_memory_Cached_bytes)) / sum(node_memory_MemTotal_bytes) * 100 > 85
        for: 30s
        labels:
          severity: warning
        annotations:
          description: Docker host memory usage is {{ humanize $value}}%. Reported by
            instance {{ $labels.instance }} of job {{ $labels.job }}.
          summary: Server memory is almost full
          type: Server
      - alert: high_storage_load
        expr: (node_filesystem_size_bytes{fstype="aufs"} -
node_filesystem_free_bytes{fstype="aufs"})
          / node_filesystem_size_bytes{fstype="aufs"} * 100 > 85
        for: 30s
        labels:
          severity: warning
        annotations:
          description: Docker host storage usage is {{ humanize $value}}%. Reported by
            instance {{ $labels.instance }} of job {{ $labels.job }}.
          summary: Server storage is almost full
          type: Server
      - alert: monitor_service_down
        expr: up == 0
        for: 30s
        labels:
          severity: critical
        annotations:
          description: Service {{ $labels.instance }} is down.
          summary: Monitor service non-operational
          type: Service
     ## End - Sample node monitoring rules.
```

7. Use the following command to create a new AlertRule in the kube-system namespace.

   ```
   $ kubectl apply -f netcool-rules.yaml --namespace kube-system
   ```

   Note: It usually takes a couple of minutes for Prometheus to reload the updated config maps and apply the new configuration.

8. Verify that Prometheus events appear on the OMNIbus Event List.

Configuring the Message Bus Probe to receive notifications from Logstash

The Message Bus Probe can be configured to integrate with IBM Cloud Private (ICP) using Logstash to forward ICP events to Netcool/OMNIbus.

The following configuration files are supplied with the probe for the integration with ICP using Logstash:

- `message_bus_logstash.props`
- `message_bus_logstash.rules`
- `message_bus_logstash_parser.json`
- `logstashWebhookTransport.properties`

Note: The default probe rules require OMNIbus event grouping triggers to be installed.

To integrate the Message Bus Probe with ICP using Logstash, use the following steps:

1. Install/update the Message Bus Probe using IBM Installation Manager, see “Installing probes” on page 4.
2. Edit the probe configuration in the following probe properties file:

   `$OMNIHOME/probes/<arch>/message_bus_logstash.props`

   Where *<arch>* is the architecture directory, for example `linux2x86`
3. Update the following property values with the appropriate path:

```
PropsFile         : '$OMNIHOME/probes/linux2x86/message_bus_logstash.props'
RulesFile         : '$OMNIHOME/probes/linux2x86/message_bus_logstash.rules'
TransportType     : 'Webhook'
TransportFile     : '$OMNIHOME/java/conf/logstashWebhhookTransport.properties'
```

4. Edit the **Port** property to a free local port number, for example: 80.
5. Edit the Webhook transport configuration in the following transport properties file:

   $OMNIHOME/probes/java/conf/logstashWebhookTransport.properties
6. Update the following property value with the appropriate path:

```
webhookURI=/probe/webhook/logstash
```

7. Update the endpoint attribute in $OMNIHOME/probes/*<arch>*linux2x86/message_bus_logstash_parser.json to be the same as the **webhookURI** path if you have updated the **webhookURI** property.
8. If you require secure connection, refer to "Connecting to WebSocket using SSL" on page 29.
9. Start the probe with the propsfile option to specify the Logstash properties file using the following command:

   $OMNIHOME/probes/nco_p_message_bus -propsfile $OMNIHOME/probes/linux2x86/message_bus_logstash.props
10. Verify that the probe is running.

    The Webhook URL will be logged in the probe log file for reference. This is the URL of the probe Webhook.
11. Re-configure Logstash in ICP to forward HTTP POST notifications to the probe host with the path specified in the **webhookURI** property.

    For example: http://<probehost>:80/probe/webhook/logstash

    See "Configuring Logstash in ICP from the command line" on page 57.
12. Optional. Verify the probe version info with the version command line option using the following command:

    $OMNIHOME/probes/nco_p_message_bus -version

Configuring Logstash in ICP from the command line

Procedure for configuring Logstash to point to the probe's webhook running on ICP. It can also be used with the on-premises version of the probe.

Modifying Logstash configuration on IBM Cloud Private 3.2.0

To modify the default Logstash configuration, use the following steps:

1. Determine the Logstash Pipeline ConfigMap in the same namespace. In this procedure, the ConfigMap in the kube-system namespace is logging-elk-logstash-pipeline-config. If a separate Logstash is deployed, determine the pipeline ConfigMap and add a new http output. Note: In ICP 3.1.2 or below, the Logstash Pipeline ConfigMap name is logging-elk-logstash-config.
2. Edit the Logstash pipeline ConfigMap to add a new http output. To do this via the command line, configure kubectl client and follow the steps below.
3. Load the config map into a file using the following command:

   kubectl get configmap logging-elk-logstash-pipeline-config --namespace=kube-system -o yaml > logging-elk-logstash-pipeline-config.yaml
4. Edit the logging-elk-logstash-pipeline-config.yaml file and modify the output object to add a new http output object as shown below (using the full webhook URL as shown in step 1 above in the http.url parameter):

```
output {
      elasticsearch {
        index => "logstash-%{+YYYY.MM.dd}"
        hosts => "elasticsearch:9200"
      }
       http {
          url => "http://<ip_address>:<port>/probe/webhook/logstash"
          format => "json"
```

```
        http_method => "post"
        pool_max_per_route => "5"
      }
    }
```

Note: (Optional) The `pool_max_per_route` is set to limit concurrent connections to the probe to 5 so that Logstash does not flood the probe which may cause event loss.

5. Save the changes in the file and replace the config map using the following command:

```
kubectl replace --namespace kube-system logging-elk-logstash-pipeline-config -f logging-elk-
logstash-pipeline-config.yaml
configmap "logging-elk-logstash-pipeline-config" replaced
```

Note: It usually takes a minute or so for Logstash to reload the new configmap.

6. Check the logs to make sure there are no errors sending HTTP POST notifications to the probe.

Probe integration for IBM Event Streams for IBM Cloud

The Message Bus Probe can be configured to integrate with IBM Event Streams for IBM Cloud.

Configuring the Message Bus Probe

The following configuration files are supplied with the probe for the integration with the Message Bus server:

- `message_bus.props`
- `message_bus_parser_config.json`
- `kafkaClient.properties`
- `kafkaConnectionProperties.json`
- `kafkaTransport.properties`

The integration also requires the following files which you must create manually:

- `kafka_client_jaas.conf`
- `kafkaClient_javaSys.properties`

To configure the Message Bus Probe to consume new events from IBM Event Streams, use the following steps:

1. Create and configure an IBM Event Streams on IBM Cloud. For details about how to do so, see the following web page: https://cloud.ibm.com/docs/services/EventStreams?topic=eventstreams-getting_started

   Note: The credentials needed by the probe to connect to the target can be found under **Service Credentials** in the Event Stream service created.

2. Install/update the Message Bus Probe using IBM Installation Manager, see “Installing probes” on page 4.
3. Configure the properties in the `message_bus.props` file.

   Example values:

```
PropsFile : 'C:\IBM\Tivoli\Netcool\omnibus\probes\win32\message_bus.props'
RulesFile : 'C:\IBM\Tivoli\Netcool\omnibus\probes\win32\message_bus_kafka.rules'
TransportType : 'KAFKA'
TransportFile : 'C:\IBM\Tivoli\Netcool\omnibus\java\conf\kafkaTransport.properties'
TransformerFile : 'C:\IBM\Tivoli\Netcool\omnibus\probes
\win32\message_bus_parser_config.json'
MessagePayload : 'JSON'

EnableSSL : 'true'
NHttpd.EnableHTTP : TRUE
NHttpd.ListeningPort : 8080
HeartbeatInterval : 10
```

Note: There is no need to configure **Username** and **Password** in probe properties file, instead the probe takes the values specified in the kafka_client_jaas.conf file.

4. Configure the parser properties in the message_bus_parser_config.json file.

   Example values:

```
{
 "eventSources" : [ {
   "endpoint" : "/notification",
   "name" : "NotificationAlarmParser",
   "config" : {
     "dataToRecord" : [ ],
     "messagePayload" : "json",
     "messageHeader" : "",
     "jsonNestedPayload" : "",
     "jsonNestedHeader" : "",
     "messageDepth" : 3
   }
 }, {
   "endpoint" : "/resync",
   "name" : "ResyncAlarmParser",
   "config" : {
     "dataToRecord" : [ ],
     "messagePayload" : "json",
     "messageHeader" : "",
     "jsonNestedPayload" : "",
     "jsonNestedHeader" : "",
     "messageDepth" : 3
   }
 }, {
   "name" : "OtherAlarmParser",
   "type" : "ANY",
   "config" : {
     "dataToRecord" : [ ],
     "messagePayload" : "json",
     "messageHeader" : "",
     "jsonNestedPayload" : "",
     "jsonNestedHeader" : "",
     "messageDepth" : 5
   }
 }]
}
```

5. Configure the Kafka client login details in the kafka_client_jaas.conf file.

   Example values:

```
KafkaClient {
    org.apache.kafka.common.security.plain.PlainLoginModule required
    serviceName="kafka"
    username="<Event Streams USER>"
    password="<Event Streams API_KEY>";
};
```

6. Specify the Java security authentication configuration to use in the kafkaClient_javaSys.properties file.

   Example value:

```
java.security.auth.login.config=C:\IBM\Tivoli\Netcool\omnibus\java\conf
\kafka_client_jaas.conf
```

7. Configure the Kafka client properties in the kafkaClient.properties file.

   Example values:

```
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer

key.serializer=org.apache.kafka.common.serialization.StringSerializer
value.serializer=org.apache.kafka.common.serialization.StringSerializer

acks=all
security.protocol=SASL_SSL
sasl.mechanism=PLAIN
ssl.protocol=TLSv1.2
```

```
ssl.enabled.protocols=TLSv1.2
ssl.keystore.location=C:\IBM\Tivoli\Netcool\platform\win32\jre_1.8.0\jre\lib\security
\cacerts
ssl.keystore.password=changeit
ssl.keystore.type=JKS
group.id=test-consumer-group
```

8. Configure the Kafka client connection properties in the `kafkaConnectionProperties.json` file.

   Example values:

```
{
    "zookeeper_client" :
        {
            "target" : "",
            "properties" : "",
            "java_sys_props" : "",
            "topic_watch": false,
            "broker_watch": false
        },
    "brokers" :"<KAFKA_BROKERS_SASL>,
    "topics": "<Event Streams TOPIC NAME>",
    "kafka_client" :
        {
            "properties" : "C:\\IBM\\Tivoli\\Netcool\\omnibus\\java\\conf\\kafkaClient.properties",
            "java_sys_props" : "C:\\IBM\\Tivoli\\Netcool\\omnibus\\java\\conf\
\kafkaClient_javaSys.properties"
        }
}
```

   Note: Event Streams service provides a list of brokers. If you wish to use multiple brokers, you can configure this file with multiple brokers, each separated with a comma.

9. Configure the Kafka transport properties in the `kafkaTransport.properties` file.

   Example values:

```
KafkaClientMode=CONSUMER
ConnectionPropertiesFile=C:\IBM\Tivoli\Netcool\omnibus\java\conf
\kafkaConnectionProperties.json
```

10. Start the probe with the `propsfile` option to specify the Kafka properties file using the following command:

    `$OMNIHOME\probes\nco_p_message_bus -propsfile $OMNIHOME\probes \win32\message_bus_message.props`

11. To test if your Event Stream service is configured correctly and has started successfully, send events to the target using the sample producer application provided in the IBM Event Streams.

Probe integration for iDirect Pulse

The Message Bus Probe can be configured to integrate with iDirect Pulse version 2.1/2.2.

The following configuration files are supplied with the probe for the integration with iDirect Pulse:

- `message_bus_iDirect_pulse.props`
- `message_bus_iDirect_pulse.rules`
- `iDirectPulseTransport.properties`

To integrate the Message Bus Probe with iDirect Pulse, use the following steps:

1. Install/update the Message Bus Probe using IBM Installation Manager, see “Installing probes” on page 4.
2. Edit the probe configuration in the following probe properties file:

   `$OMNIHOME/probes/java/<arch>/message_bus_iDirect_pulse.props`

   Where *<arch>* is the architecture directory, for example `linux2x86`
3. Update the following property values with the appropriate path:

```
MessageLog     : '$OMNIHOME/log/message_bus_iDirect_pulse.log'
PropsFile      : '$OMNIHOME/probes/linux2x86/message_bus_iDirect_pulse.props'
```

```
RulesFile      : '$OMNIHOME/probes/linux2x86/message_bus_iDirect_pulse.rules'
DataBackupFile : '$OMNIHOME/var/message_bus_pulse.backup'
TransportFile  : '$OMNIHOME/java/conf/iDirectPulseTransport.properties'
```

4. Uncomment and update the following properties with the appropriate host information and user credentials:

```
# Host : ''
# Port : 80
# Username : ''
# Password : ''
```

5. If you require secure connection, refer to "Connecting to WebSocket using SSL" on page 29.
6. Start the probe with the `propsfile` option to specify the iDirect Pulse properties file using the following command:

   `$OMNIHOME/probes/nco_p_message_bus -propsfile $OMNIHOME/probes/linux2x86/message_bus_iDirect_pulse.props`
7. Verify that the probe is running.
8. Optional. Verify the probe version info with the `version` command line option using the following command:

   `$OMNIHOME/probes/nco_p_message_bus -version`

Probe integration for Kafka

The Message Bus Probe can be configured to integrate with a Kafka server to consume events.

The Message Bus Probe connects to the Kafka server using the Kafka transport. This enables the probe support the Kafka Client version 2.3.1 and Zookeeper version 3.4.14.

Check the Apache Kafka compatibility matrix for the support of the target system with respect to the dependency.

Configuring the Message Bus Probe to consume new events from Kafka

The following configuration files are supplied with the probe for the integration with Kafka:

- `message_bus_kafka.props`
- `message_bus_kafka.rules`
- `kafkaTransport.properties`
- `kafkaConnectionProperties.json`
- `kafkaClient.properties`

To configure the Message Bus Probe to consume new events from Kafka, use the following steps:

1. Install/update the Message Bus Probe using IBM Installation Manager, see "Installing probes" on page 4.
2. Edit the probe configuration in the following probe properties file:

   `$OMNIHOME/probes/<arch>/message_bus_kafka.props`

   Where *<arch>* is the architecture directory, for example `linux2x86`
3. Update the following property values with the appropriate path:

```
Manager           : 'Kafka'
MessageLog        : '$OMNIHOME/log/message_bus_kafka.log'
PropsFile         : '$OMNIHOME/probes/linux2x86/message_bus_kafka.props'
RulesFile         : '$OMNIHOME/probes/linux2x86/message_bus_kafka.rules'
TransportType     : 'KAFKA'
TransportFile     : '$OMNIHOME/java/conf/kafkaTransport.properties'
TransformerFile   : '$OMNIHOME/probes/linux2x86/message_bus_parser_config.json'
MessagePayload    : 'JSON'
```

4. Configure the Kafka transport properties.
   a. Edit the Kafka transport configuration in the following transport properties file:

      `$OMNIHOME/probes/java/conf/kafkaTransport.properties`

   b. Update the following property value with the appropriate path:

```
kafkaClientMode=CONSUMER
connectionPropertiesFile=$OMNIHOME/java/conf/kafkaConnectionProperties.json
```

5. For descriptions of the Kafka transport properties, see the “Configuring the Kafka transport” on page 43.
6. Configure the Kafka connection properties.

   Kafka connection properties are defined in the `kafkaConnectionProperties.json` file. This file contains the following properties:

```
{
    "zookeeper_client" :
        {
            "target" : "",
            "properties" : "",
            "java_sys_props" : "",
            "topic_watch": true,
            "broker_watch": true
        },
    "brokers" : "",
    "topics": "",
    "kafka_client" :
        {
            "properties" : "",
            "java_sys_props" : ""
        }
     }
```

   The ZooKeeper event access is required.

   a. Within the sample configuration file supplied with the probe, update the path to the ZooKeeper client properties file.

```
    "zookeeper_client" :
        {
            "target" : "localhost:2181",
            "properties" : "<Path to zookeeper client properties file>",
            "java_sys_props" : "",
            "topic_watch": true,
            "broker_watch": true
        },
```

   b. Update the path to the Kafka client properties file:

```
    "brokers" : "PLAINTEXT://localhost:9092",
    "topics": "topicABC,topicXYZ",
    "kafka_client" :
        {
            "properties" : "<omnihome_path>/java/conf/kafkaClient.properties",
            "java_sys_props" : ""
        }
```

7. Specify additional configuration to use with SASL authentication, if SASL authentication is required.

   Kafka brokers supports client authentication using SASL. Additional configuration settings are required to authenticate with SASL.

   a. Enable the following properties in the Kafka connection properties file (`kafkaConnectionProperties.json`)

"java_sys_props" : "<Path to a text file containing java system properties configuration>"

Example:

"java_sys_props" : "java.security.auth.login.config=C:\\IBM\\Tivoli\\Netcool\\omnibus\\java\\conf\\java_sys_prop.conf"

b. Create and enable the following properties in the Java system properties file (java_sys_prop.conf):

Example:

```
KafkaClient {
      org.apache.kafka.common.security.plain.PlainLoginModule required
      serviceName="kafka"
      username="wfE7hGteeNl4i9JJ"
      password="xtPqsWiPXN4cSwi6h2BE4GbaJ3uheal2";
   };
```

c. Enable and set the following properties in the Kafka client properties file (kafkaClient.properties)

```
security.protocol
     ssl.enabled.protocols

     ssl.keystore.location
     ssl.keystore.password
     ssl.keystore.type

     Included when trust stores are in use.
     ssl.truststore.location
     ssl.truststore.password
     ssl.truststore.type
```

Example:

```
security.protocol=SASL_SSL
     ssl.enabled.protocols=TLSv1.2

     ssl.keystore.location=C:\\Programs\\ibm-java-sdk-80-win-x86_64\\sdk
\\jre\\lib\\security\\cacerts
     ssl.keystore.password=changeit
     ssl.keystore.type=JKS

     ssl.truststore.location=C:\\Programs\\ibm-java-sdk-80-win-x86_64\\sdk
\\jre\\lib\\security\\cacerts
     ssl.truststore.password=changeit
     ssl.truststore.type=JKS
```

8. For descriptions of the Kafka connection properties, see the "Configuring the Kafka transport" on page 43.
9. Start the probe with the propsfile option to specify the Kafka properties file using the following command:

   $OMNIHOME/probes/nco_p_message_bus -propsfile $OMNIHOME/probes/linux2x86/message_bus_kafka.props
10. Verify that the probe is running.

Probe integration for Microsoft Azure Monitoring

The Message Bus Probe can be configured to obtain events from Microsoft Azure Monitoring.

The following configuration files are supplied with the probe for the integration:

- message_bus_azure.props
- message_bus_azure.rules
- message_bus_azure_WebhookTransport.properties

- message_bus_azure_config.json

To enable the Message Bus Probe to host a Webhook server for listening to incoming data from Azure Monitoring, use the following steps:

1. Set the following properties in the message_bus_azure.props file:

```
.
.
.
    SETTING PROBE LOGS, PROPS, RULES:
    MessageLog : '$OMNIHOME/log/message_bus_azure.log'
    PropsFile : '$OMNIHOME/probes//message_bus_azure.props'
    RulesFile : '$OMNIHOME/probes//message_bus_azure.rules'

    SETTING TRANSPORT TYPE:
    TransportType : 'Webhook'
    TransportFile : '$OMNIHOME/java/conf/message_bus_azure_WebhookTransport.properties'

    SETTING PARSER CONFIGURATIONS. (SUPPORTS JSON)
    TransformerFile : '$OMNIHOME/probes//message_bus_azure_parser.json'
    MessagePayload : 'JSON'
.
.
.
```

2. Set the following properties in the message_bus_azure_parser.json file:

   Note: The probe package contains a default parser configuration for Microsoft Azure Monitoring.

```
    {
    "eventSources" : [ {
    "endpoint" : "/notification",
    "name" : "NotificationAlarmParser",
    "config" : {
    "dataToRecord" : [ ],
    "messagePayload" : "json",
    "messageHeader" : "",
    "jsonNestedPayload" : "",
    "jsonNestedHeader" : "",
    "messageDepth" : 5
    }
    }, {
    "endpoint" : "/resync",
    "name" : "ResyncAlarmParser",
    "config" : {
    "dataToRecord" : [ ],
    "messagePayload" : "json",
    "messageHeader" : "",
    "jsonNestedPayload" : "",
    "jsonNestedHeader" : "",
    "messageDepth" : 3
    }
    }, {
    "name" : "OtherAlarmParser",
    "type" : "ANY",
    "config" : {
    "dataToRecord" : [ ],
    "messagePayload" : "json.data.context",
    "messageHeader" : "json",
    "jsonNestedPayload" : "",
    "jsonNestedHeader" : "",
    "messageDepth" : 5
    }
    }]
    }

.
.
.
```

3. Update **webhookURI** property in the message_bus_WebhookTransport.properties file.

   webhookURI=http://:<Probe's Webhook Port>/probe/webhook

   Note: webHookURI property can be initialized in one of three ways:

Option 1: Assign host and port value in URI as described above. For example:

`webhookURI=http://:<User_Assigned_Host>:<User_Assigned_Port>/probe/webhook`

Option 2: Assign host and port value in probes properties file and assign **webhookURI** property with webhook path's value only. For example:

`webhookURI=/probe/webhook`

Option 3: When **Host** and **Port** are not assigned in the probe's property and transport configuration file, by default probe attempts to use hostname and port 80 to construct the webhook's path on local server. For example:

`webhookURI=http://:<Sys_Resolved_Host>:80/probe/webhook`

Probe integration for Nokia 1350 OMS

The Message Bus Probe can be configured to integrate with Nokia 1350 OMS.

Note: Nokia has integrated Nokia 1350 OMS into the Network Functions Manager - Transport (NFM-T) module within Nokia Network Services Platform. This probe is compatible with NFM-T R17.9 and R17.12.

The following configuration files are supplied with the probe for the integration with Nokia 1350 OMS:

- `message_bus_nokia_oms1350.props`
- `message_bus_nokia_oms1350.rules`
- `cometdTransport.properties`

To integrate the Message Bus Probe with Nokia 1350 OMS, use the following steps:

1. Install/update the Message Bus Probe using IBM Installation Manager, see “Installing probes” on page 4.
2. Edit the probe configuration in the following probe properties file:

   `$OMNIHOME/probes/java/<arch>/message_bus_nokia_oms1350.props`

   Where *<arch>* is the architecture directory, for example `linux2x86`
3. Update the following property values with the appropriate path:

```
MessageLog        : '$OMNIHOME/log/message_bus_nokia_oms1350.log'
PropsFile         : '$OMNIHOME/probes/linux2x86/message_bus_nokia_oms1350.props'
RulesFile         : '$OMNIHOME/probes/linux2x86/message_bus_nokia_oms1350.rules'
MessagePayload    : 'json'
TransportType     : 'Cometd'
TransportFile     : '$OMNIHOME/java/conf/nokiaOms1350CometdTransport.properties'
HeartbeatInterval : 60
```

4. Uncomment and update the following properties with the appropriate host information and user credentials:

```
# Host : ''
# Port :
# Username : ''
# Password : ''
```

5. If you require secure connection, refer to “Connecting to WebSocket using SSL” on page 29.
6. Start the probe with the `propsfile` option to specify the Nokia 1350 OMS properties file using the following command:

   `$OMNIHOME/probes/nco_p_message_bus -propsfile $OMNIHOME/probes/linux2x86/message_bus_nokia_om1350.props`
7. Verify that the probe is running.
8. Optional. Verify the probe version info with the `version` command line option using the following command:

   `$OMNIHOME/probes/nco_p_message_bus -version`

Probe integration for Nokia NSP

Details about the integration with Nokia NSP have been moved to the *Probe Integration for Nokia NSP Reference Guide*.

Using the transformer module

The transformer module reads the XML event stream and converts the event data into a format suitable for the destination application.

The endpoint from which the events are generated determines which transformation is required.

The probe uses the transformer module to transform the XML messages into a set of typed name-value pairs. For this conversion, the transformer module uses the `netcool2nvpairs.xsl` file to convert Netcool® XML events. For other XML events, the probe uses the XSLT file created for that type of XML event to generate the name-value pairs.

This section contains the following topics:

- "Using XSLT files to transform events" on page 66
- "Using the transformer testing tool" on page 68
- "Configuring the transformer definition file" on page 69
- "Using the XML validation tool" on page 71

Using XSLT files to transform events

Each XML event source generates events in a format that is specified by its own XML schema. You create an Extensible Stylesheet Language Translation (XSLT) file to transform events from that event source to another XML format, making it possible for applications to share XML events.

The following table shows the XSLT files that are supplied with the transformer module.

Table 21. XSLT files supplied with the transformer module

XSLT file	Description
`addnvpairs.xsl`	This XSLT file is a support template. For details see "Probe XSLT files" on page 67.
`cbe2nvpairs.xsl`	This XSLT file converts Common Base Event (CBE) events into name-value pairs for the probe to read.
`netcool2cbe.xsl`	This XSLT file converts a Netcool event into an event in CBE format.
`netcool2nvpairs.xsl`	This XSLT file converts Netcool events into name-value pairs that the probe can read.
`u20002nvpairs.xsl`	This XSLT file converts U2000 events into name-value pairs.
`netcool2wef.xsl`	This XSLT file converts Netcool events from the gateway into Web Services Distributed Management (WSDM) Event Format (WEF) events.
`wbe2nvpairs.xsl`	This XSLT file converts WebSphere Business Event (WBE) events into name-value pairs for the probe to read.

Table 21. XSLT files supplied with the transformer module (continued)	
XSLT file	**Description**
`wbepl2nvpairs.xsl`	This XSLT file includes `wbm2nvpairs.xsl` for handling WebSphere Business Monitoring (WBM) events that are contained within WBE events.
`wbm2nvpairs.xsl`	This example XSLT file contains support XSLT match functions for handling specific WBM trade array event elements.
`wef2nvpairs.xsl`	This XSLT file converts WEF events into name-value pairs for the probe to read.

If you require other types of XSLT files, you must create them. The following topics provide information that will help you to create XSLT files.

For details of the syntax required for XSLT files, see the XSL Transformations page on the W3C Web site:

http://www.w3.org/TR/xslt

Probe XSLT files

The input can be any XML message that is to be inserted into the ObjectServer. This type of XSLT file must generate a set of name-type-value elements in the following format:

name:*type*:"*value*"

where:

- *name* consists of alphanumeric characters and underscores.
- *type* can be `string`, `utc`, or `integer`.
- *value* is any arbitrary string that does not include a new-line character.

The module is supplied with an XSLT template, `addnvpairs.xsl`, that you must include in your XSLT file. You can call the template as a function, providing the function with the name, type, and value, to format the output correctly. To use the support template, include the XSLT file using the following XSLT include directive:

```
<xsl:include href="addnvpair.xsl"/>
```

Note: The `href` parameter for the file is relative to the location of the XSLT file that is including it. All XSLT files supplied with the module are installed in the same directory. If the XSLT file that you create is not in the same directory, you must specify the relative path to the support template within the `href` parameter. The module is supplied with a basic name-value pair XSLT file (`netcool2nvpairs.xsl`) which converts a Netcool XML event into a name-value pair for consumption by the probe.

The following example shows the content of the `netcool2nvpairs.xsl` with the include directive highlighted:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        version="1.0"
        xmlns:tns="http://item.tivoli.ibm.com/omnibus/netcool/nvpairs"
        xmlns:ens="http://item.tivoli.ibm.com/omnibus/netcool"
        exclude-result-prefixes="tns ens">
<xsl:output method="text"/>
<xsl:strip-space elements="*"/>

<xsl:include href="addnvpair.xsl"/>

<xsl:template match="/">
    <xsl:for-each select="ens:netcoolEvent">
        <xsl:call-template name="AddNVPair">
            <xsl:with-param name="name">
                <xsl:text>NetcoolEventAction</xsl:text>
```

```
            </xsl:with-param>
            <xsl:with-param name="type">
                <xsl:text>string</xsl:text>
            </xsl:with-param>
            <xsl:with-param name="value">
                <xsl:value-of select="@type"/>
            </xsl:with-param>
        </xsl:call-template>
        <xsl:apply-templates/>
    </xsl:for-each>
</xsl:template>

<xsl:template match="ens:netcoolEvent/ens:netcoolField">
    <xsl:call-template name="AddNVPair">
        <xsl:with-param name="name">
            <xsl:value-of select="@name"/>
        </xsl:with-param>
        <xsl:with-param name="type">
            <xsl:value-of select="@type"/>
        </xsl:with-param>
        <xsl:with-param name="value">
            <xsl:value-of select="."/>
        </xsl:with-param>
    </xsl:call-template>
</xsl:template>

</xsl:stylesheet>
```

Within this example there are two other directives that you should include in all probe XSLT files:

- `<xsl:output method="text"/>`

  This sets the XSLT to plain text, rather than the default of XML. As the output is required in name-type-value elements, it must be in plain text.

- `<xsl:strip-space elements="*"/>`

  This directive forces all unessential whitespace to be stripped from the output.

For additional guidance about creating XSLT files for use with the Message Bus Probe, see the following Tech Note: http://www-01.ibm.com/support/docview.wss?uid=swg21622274

Using the transformer testing tool

The transformer testing tool helps you verify the XSLT file created for an XML event source.

The transformer testing tool allows you to check that the XSLT file that you have created for an XML event source generates XML events in the expected format. This tool runs the source XML through the XSLT file and prints the result of the XSLT transformation.

To start the transformer testing tool, run the following command:

`java -cp $OMNIHOME/java/jars/Transformer.jar com.ibm.tivoli.netcool.integrations.transformer.XSLTTransformer` *`XSLT_file Source_file`*

where:

- *XSLT_file* is the name of the XSLT file that you are testing.
- *Source_file* is the name of the XML file conforming to the schema file of the event source.

Example output from the transformer testing tool

The following is sample output from the transformer testing tool:

```
java -cp $OMNIHOME/java/jars/Transformer.jar
com.ibm.tivoli.netcool.integrations.transformer.XSLTTransformer
netcool2nvpairs.xsl netcool.xml
Output from applying transformer 'netcool2nvpairs.xsl' to source file
'netcool.xml':-
NetcoolEventAction:string:update
Identifier:string:"GATEWAY:Gateway Reader@hostname.Mon Nov 10 14:37:55 2008"
NodeAlias:string:"hostname"
Manager:string:"ConnectionWatch"
```

```
Agent:string:""
AlertGroup:string:"Gateway"
AlertKey:string:"GATEWAY:Gateway Reader"
Severity:integer:"0"
Summary:string:"A GATEWAY process Gateway Reader running on
hostname has disconnected as username gateway"
StateChange:utc:"2008-11-10T14:38:33"
FirstOccurrence:utc:"2008-11-10T14:37:55"
LastOccurrence:utc:"2008-11-10T14:37:55"
InternalLast:utc:"2008-11-10T14:37:55"
Poll:integer:"0"
Type:integer:"1"
Tally:integer:"1"
Class:integer:"0"
Grade:integer:"0"
Location:string:""
OwnerUID:integer:"65534"
OwnerGID:integer:"0"
Acknowledged:integer:"0"
Flash:integer:"0"
EventId:string:""
ExpireTime:integer:"0"
ProcessReq:integer:"0"
SuppressEscl:integer:"0"
Customer:string:""
Service:string:""
PhysicalSlot:integer:"0"
PhysicalPort:integer:"0"
PhysicalCard:string:""
TaskList:integer:"0"
NmosSerial:string:""
NmosObjInst:integer:"0"
NmosCauseType:integer:"0"
LocalNodeAlias:string:""
LocalPriObj:string:""
LocalSecObj:string:""
LocalRootObj:string:""
RemoteNodeAlias:string:""
RemotePriObj:string:""
RemoteSecObj:string:""
RemoteRootObj:string:""
X733EventType:integer:"0"
X733ProbableCause:integer:"0"
X733SpecificProb:string:""
X733CorrNotif:string:""
URL:string:""
ExtendedAttr:string:""
ServerName:string:"NCOMS"
ServerSerial:integer:"1841"
```

Configuring the transformer definition file

The transformer definition file maps the XML event sources to their related XSLT files, and directs the probe and the gateway to use the XSLT file associated to the XML event source.

The `transformers.xml` transformer definition file, located in the `$OMNIHOME/java/conf` directory, defines how the messages that the probe reads, or that the gateway sends, are transformed. This file is divided into two logical sections, one for the probe (southbound) and one for the gateway (northbound).

After creating an XSLT file for each event source, you create a transformer entry in the transformer definition file. This enables the transformer module to use the specified XSLT file when transforming events for that event source.

By default, the probe section enables the following transformations:

- messages received on a topic name of `cbe` are transformed by the `cbe2nvpairs` XSLT file
- messages received on a topic name of `wef` are transformed by the `wef2nvpairs` XSLT file
- messages received on a topic name of `wbe` are transformed by the `wbe2nvpairs` XSLT file
- messages received on a topic name of `netcool` are transformed by the `netcool2nvpairs` XSLT file

By default, the gateway section enables the following transformations:

- Netcool events that have an identifier of `cbeEvents` are transformed into CBE events (using the netcool2cbe XSLT file) and published to the JMS using the topic `cbe`

- Netcool events that have an identifier of `wefEvents` are transformed into WEF events (using the netcool2wef XSLT file) and published to the JMS using the topic `wef`
- Netcool events that have an identifier of `netcoolEvents` are published to the JMS using the topic `netcool` (without using an XSLT transformer)

Each entry takes the following format:

```
<tns:transformer name="transformer_name" type="southbound | northbound"
endpoint="event_endpoint" className="class_name">

<tns:property name="property_name" type="property_type" value="property_value"
description="description"/>

</tns:transformer>
```

where:

- `transformer_name` specifies the name of the transformer definition for an event source.
- `type` specifies the type of transformer being defined.

  This is either `southbound` for transformers used by the probe, or `northbound` for transformers used by the gateway.
- `event_endpoint` maps the endpoint from which the event arrived to an XSLT transformer and determines the endpoint to which the event source sends events.
- `class_name` is the name of the class to be used.
- `property_name` is the name of a property to be set in the transformation.

  For XSLT transformers, this is the name of the XSLT file to use.
- `property_type` identifies the type of entry that is set by the `property_name` field.

  For an XSLT file name, this would be `string`.
- `property_value` is the field that specifies the path to the XSLT file created for the event source.
- `description` specifies the description for the XSLT file created for the event source.

Sample transformer file

The following is a sample transformer file with transformers for various event sources:

```
<?xml version="1.0" encoding="UTF-8" ?>
- <tns:transformers
    xmlns:tns="http://item.tivoli.ibm.com/omnibus/netcool/transformer"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

< !-- Southbound (probe) transformer definitions -->

< tns:transformer name="cbe2nvpairs" type="southbound" endpoint="cbe"
   className="com.ibm.tivoli.netcool.integrations.transformer.XSLTTransformer">

< tns:property name="xsltFilename" type="java.lang.String" value="${OMNIHOME}/java
  /conf/cbe2nvpairs.xsl" description="XSLT file for converting CBE events to
  name/value pairs"/>

< /tns:transformer>

< tns:transformer name="wef2nvpairs" type="southbound" endpoint="wef"
    className="com.ibm.tivoli.netcool.integrations.transformer.XSLTTransformer">

< tns:property name="xsltFilename" type="java.lang.String" value="${OMNIHOME}/
    java/conf/wef2nvpairs.xsl" description="XSLT file for converting WEF events
    to name/value pairs"/>

< /tns:transformer>

< tns:transformer name="netcool2nvpairs" type="southbound" endpoint="netcool"
   className="com.ibm.tivoli.netcool.integrations.transformer.XSLTTransformer">

< tns:property name="xsltFilename" type="java.lang.String" value="${OMNIHOME}/
    java/conf/netcool2nvpairs.xsl" description="XSLT file for converting Netcool
    events to name/value pairs"/>
```

```
< /tns:transformer>

< !-- Northbound (gateway) transformer definitions -->

< tns:transformer name="netcool2wef" type="northbound" endpoint="wef"
    className="com.ibm.tivoli.netcool.integrations.transformer.XSLTTransformer">

< tns:property name="xsltFilename" type="java.lang.String" value="${OMNIHOME}/
    java/conf/netcool2wef.xsl" description="XSLT file for converting Netcool
    events to WEF events"/>

< /tns:transformer>

< tns:transformer name="netcool2cbe" type="northbound"  endpoint="cbe"
    className="com.ibm.tivoli.netcool.integrations.transformer.XSLTTransformer">

< tns:property name="xsltFilename" type="java.lang.String" value="${OMNIHOME}/
    java/conf/netcool2cbe.xsl" description="XSLT file for converting Netcool
    events to CBE events"/>

< /tns:transformer>

< tns:transformer name="netcoolEvents" type="northbound" endpoint="netcool"
    className="com.ibm.tivoli.netcool.integrations.transformer.
    EmptyTransformer">

< /tns:transformer>

< /tns:transformers>
```

You can prevent events from an event source being transformed by specifying an empty transformer for that event source. The following example shows the entry for an event source with empty transformer details:

```
<tns:transformer name="empty" id="empty" className="com.micromuse.common.
     transformer.EmptyTransformer"/>
  </tns:transformers>
```

Using the XML validation tool

You can use the XML validation tool to verify transformed XML events.

The XML validation tool allows you to validate transformed XML events against the XML schema of an event source.

To validate the XML output, run the following command:

`java -cp $OMNIHOME/java/jars/Transformer.jar com.ibm.tivoli.netcool.integrations.transformer.XMLValidator` *XML_schema XML_file*

where:

- *XML_schema* is the XML schema against which you are validating the XML output.
- *XML_file* is the name of the XML output file whose format you are validating.

Running the probe

Probes can be run in a variety of ways. The way you chose depends on a number of factors, including your operating system, your environment, and the any high availability considerations that you may have.

For details about how to run the probe, visit the following page on the IBM Tivoli Knowledge Center:

http://www-01.ibm.com/support/knowledgecenter/SSSHTQ/omnibus/probes/all_probes/wip/concept/running_probe.html

Data acquisition

The probe acquires events from various sources using Java Message Service (JMS), Webhook, Message Queue Telemetry Transport (MQTT), Web Socket or data files.

Data acquisition is described in the following topics:

- "Peer-to-peer failover functionality" on page 72

Peer-to-peer failover functionality

The probe supports failover configurations where two probes run simultaneously. One probe acts as the `master` probe, sending events to the ObjectServer; the other acts as the `slave` probe on standby. If the master probe fails, the slave probe activates.

While the slave probe receives heartbeats from the master probe, it does not forward events to the ObjectServer. If the master probe shuts down, the slave probe stops receiving heartbeats from the master and any events it receives thereafter are forwarded to the ObjectServer on behalf of the master probe. When the master probe is running again, the slave probe continues to receive events, but no longer sends them to the ObjectServer.

Example property file settings for peer-to-peer failover

You set the peer-to-peer failover mode in the properties files of the master and slave probes. The settings differ for a master probe and slave probe.

Note: In the examples, make sure to use the full path for the property value. In other words replace $OMNIHOME with the full path. For example: `/opt/IBM/tivoli/netcool/omnibus`.

The following example shows the peer-to-peer settings from the properties file of a master probe:

```
Server      :    "NCOMS"
RulesFile   :    "master_rules_file"
MessageLog  :    "master_log_file"
PeerHost    :    "slave_hostname"
PeerPort    :    5555 # [communication port between master and slave probe]
Mode        :    "master"
PidFile     : "$OMNIHOME/var/message_bus"
```

The following example shows the peer-to-peer settings from the properties file of the corresponding slave probe:

```
Server      :    "NCOMS"
RulesFile   :    "slave_rules_file"
MessageLog  :    "slave_log_file"
PeerHost    :    "master_hostname"
PeerPort    :    5555 # [communication port between master and slave probe]
Mode        :    "slave"
PidFile     : "$OMNIHOME/var/message_bus2"
```

HTTP/HTTPS command interface

IBM Tivoli Netcool/OMNIbus Version 7.4.0 (and later) includes a facility for managing the probe over an HTTP/HTTPS connection. This facility uses the **nco_http** utility supplied with Tivoli Netcool/OMNIbus.

The HTTP/HTTPS command interface replaces the Telnet-based command line interface used in previous versions of IBM Tivoli Netcool/OMNIbus.

The following sections show:

- How to configure the command interface.
- The format of the **nco_http** command line.
- The format of the individual probe commands.
- The messages that appear in the log files.

- How to store frequently-used commands in a properties file.

For more information on the HTTP/HTTPS command interface and the utilities it uses, see the chapter on remotely administering probes in the *IBM Tivoli Netcool/OMNIbus Probe and Gateway Guide*.

Configuring the command interface

To configure the HTTP/HTTPS command interface, set the following properties in the probe's property file:

NHttpd.EnableHTTP: Set this property to `True`.
NHttpd.ListeningPort: Set this property to the number of the port that the probe uses to listen for HTTP commands, namely `4000`.

Optionally, set a value for the following property as required:

NHttpd.ExpireTimeout: Set this property to the maximum time (in seconds) that the HTTP connection remains idle before it is disconnected.

The *IBM Tivoli Netcool/OMNIbus Probe and Gateway Guide* contains a full description of these and all properties for the HTTP/HTTPS command interface.

Format of the nco_http command line

The format of the **nco_http** command line to send a command to the probe is:

```
$OMNIHOME/bin/nco_http -uri probeuri:probeport/probes/message_bus -datatype
application/json -method post -data '{"command":"command-name","params":
[command-parameters]}'
```

Where:

- *probeuri* is the URI of the probe.
- *probeport* is the port that the probe uses to listen for HTTP/HTTPS commands. Specify the same value as that set for the **NHttp.ListeningPort**.
- *command-name* is the name of the command to send to the probe. The following command names are available:

  disconnectProbe
  shutdownProbe

- *command-parameters* is a list of zero or more command parameters. For commands that have no parameters, this component is empty. The command descriptions in the following section define the parameters that each takes.

Probe commands

The following sections define the structure of the JavaScript Object Notation (JSON)-formatted commands that you can send to the probe. There is an example of each command.

All the examples use a probe URI of `http://localhost` and a HTTP listening port of `8080`.

disconnectProbe

Use the **disconnectProbe** command to disconnect the probe from the target system, and to return OK if successful.

The format of the `-data` option for the **disconnectProbe** command is:

```
-data '{"command":"disconnectProbe","params":[]}'
```

The following command performs a disconnection:

```
$OMNIHOME/bin/nco_http -uri http://localhost:8080/probes/message_bus -datatype
application/JSON -method POST -data '{"command":"disconnectProbe","params":[]}'
```

shutdownProbe

Use the **shutdownProbe** command to shutdown the probe, and to return OK if successful.

The format of the `-data` option for the **shutdownProbe** command is:

```
-data '{"command":"shutdownProbe","params":[]}'
```

The following command performs a disconnection:

```
$OMNIHOME/bin/nco_http -uri http://localhost:8080/probes/message_bus -datatype
application/JSON -method POST -data '{"command":"shutdownProbe","params":[]}'
```

Messages in the log file
The `nco_http` utility can make extensive entries in the probe's log file indicating the progress of each operation. These messages can help isolate problems with a request, such as a syntax problem in a command.

To obtain the detailed log information, set the probe's **MessageLevel** property to debug. This enables the logging of the additional information that tracks the progress of a command's execution. For example, the following shows the progress of a **disconnectProbe** command:

```
Information: I-JPR-000-000: DISCONNECT 'DisonnectProbe command received.
Disconnecting from target.
```

Storing commands in the nco_http properties file
You can use the **nco_http** utility's properties file (`$OMNIHOME/etc/nco_http.props`) to hold frequently used command characteristics.

If you have a particular command that you send to the probe regularly, you can store characteristics of that command in the **nco_http** properties file. Once you have done that, the format of the **nco_http** command line is simplified.

You can use one or more of the following **nco_http** properties to hold default values for the equivalent options on the **nco_http** command line:

Data
DataType
Method
URI

Specify the value of each property in the same way as you would on the command line. Once you have these values in place you do not need to specify the corresponding command line switch unless you want to override the value of the property.

The following is an example of the use of the properties file and the simplification of the **nco_http** command that results. In this example, the **nco_http** properties file contains the following values (note that line breaks appear for presentational purposes only; when editing the properties use one line for each property value):

```
Data : '{"command":"disconnectProbe","params":[]}'
DataType : 'application/JSON'
Method : 'POST'
```

Properties and command line options

You use properties to specify how the probe interacts with the device. You can override the default values by using the properties file or the command line options.

The following table describes the properties and command line options specific to this probe. For information about default properties and command line options, see the *IBM Tivoli Netcool/OMNIbus Probe and Gateway Guide*.

Table 22. Probe properties and command line options		
Property name	**Command line option**	**Description**
Cookie *string*	`-cookie` *string*	Use this property to specify the HTTP cookie name to be retrieved from the probe store. The probe uses the value retrieved from the cookie to replace ++*property_setting*++ in the `restWebSocketTransport.properties` file. You can specify multiple values for this property by separating each string with a comma (,). The default is `""`. The XML or JSON event source sends the cookie in response to the probe's login request. The default setting for this property instructs the probe to replace the ++*property_setting*++ token in the `restWebSocketTransport.properties` file with the cookie value.
EnableSSL *string*	`-noenablessl` (This is equivalent to **EnableSSL** with a value of `false`.) `-enablessl` (This is equivalent to **EnableSSL** with a value of `true`.)	Use this property to specify whether SSL connectivity between the probe and the EMS server is enabled or disabled. This property takes the following values: `false`: SSL connectivity between the probe and the EMS server is disabled. `true`: SSL connectivity between the probe and the EMS server is enabled. The default is `false`. **Note:** This property is only used by the probe if you are using the `WebSocket` **`TransportType`**.
Host *string*	`-host` *string*	Use this property to specify the host name or IP address of the instance of the XML or JSON event source to which the probe connects. This property is only used by the probe if you are using the `WebSocket`, `WebHook`, or `CometD` **`TransportType`**. The default is `""`. **Note:** The probe also uses this value to replace the ++Host++ token in the `restWebSocketTransport.properties` file.

Table 22. Probe properties and command line options (continued)

Property name	Command line option	Description
JsonMessageDepth *integer*	`-jsonmessagedepth` *integer*	Use this property to specify the number of levels in the message to traverse during parsing. This enables you to prevent the probe from having to traverse all sub-trees exhaustively. The default is 3.
JsonNestedHeader *string*	`-jsonnestedheader` *string*	Use this property to specify either XML or the JSON tree structure to the nested message header. **Note:** The message header is included in the events generated by the probe. The default is `""`.
JsonNestedPayload *string*	`-jsonnestedpayload` *string*	Use this property to specify whether nested parsing on JSON data is enabled. To enable, specify either XML or JSON tree structure to the nested message payload in the JSON string values in the JSON array as specified by the **MessagePayload** property. This property has the same semantics as **MessagePayload** except that the default value is blank (an empty string), which turns off nested parsing. The default is `""`.
JsonParserName *string*	`-jsonparsername` *string*	Use this property to specify the parser type. This property takes the following values: `DEFAULT`: Generic parser for all target systems. `AWS`: Specific parser for the AWS integration. The default is `"DEFAULT"`.
KeyStore *string*	`-keystore` *string*	Use this property to specify the location of the keystore file that contains the client certificate for the SSL and trusted authority certificate. The default is `""`.

Table 22. Probe properties and command line options (continued)		
Property name	**Command line option**	**Description**
KeyStorePassword *string*	`-keystorepassword` *string*	Use this property to specify the password required to access the certificate specified by the **Keystore** property. The default is `""`. **Note:** You can encrypt this password using the `nco_aes_crypt` utility within Netcool/OMNIbus.
MessageHeader *string*	`-messageheader` *string*	Use this property to specify either XML or the JSON tree structure to the message header. **Note:** The message header is included in the events generated by the probe. The default is `""`.
MessagePayload *string*	`-messagepayload` *string*	Use this property to specify either XML or the JSON tree structure to the message payload. The default is `xml`. If this property is set to `xml`, the **TransformerFile** property must be set to the XML data transformer configuration file. For JSON object parsing, consider migrating to use the new JSON parser configuration file. **Note:** If you specify a JSON tree structure, it must start with `json` to indicate that the message is a JSON object. A probe event is derived from a JSON object pointed by message payload. The message payload object consists of name-value data pairs. The probe processes the message payload object to generate probe name-value pair elements.
PartialResync *string*	`-partialresync` *string*	Use this property to specify that the probe performs a partial resync on startup. If this property is set to `true`, the probe performs a partial resync based on the last event received timestamp stored in a persistent file. The default is `false`. **Note:** This property is only for use with the Probe Integration for Nokia Network Services Platform.

Table 22. Probe properties and command line options (continued)		
Property name	**Command line option**	**Description**
Password *string*	-password *string*	Use this property to specify the password associated with the **Username** property for logging into the XML or JSON event source. The default is "". **Note:** The probe uses this value to replace the ++Password++ token (if it is specified) in the restWebSocketTransport.properties file or in the restWebHookTransport.properties file.
Port *integer*	-port *integer*	Use this property to specify the host port of the instance of the XML or JSON event source to which the probe connects. This property is only used by the probe if you are using the WebSocket, WebHook, or CometD **TransportType**. The default is 0. **Note:** The probe also uses this value to replace the ++Port++ token in the restWebSocketTransport.properties file.
RecordData *string*	-recorddata *string*	Use this property to specify a comma-separated list of attributes from the event to be recorded in the file specified by the **DataBackupFile** property. The data recorded can be used by the probe to resolve transport properties using tokens with the prefix "RecordData.". For example, if the event generated by the probe has a URL attribute that should be recorded, set the **RecordData** property to URL. To use this attribute to resolve a property in the probe's transport property file, set the property with the following token: WebSocketURL=++URL++
StreamCapture *string*	-streamcapture *string*	Use this property to specify whether or not the probe stores the XML or JSON event data in a stream capture file. The default is false.

Table 22. Probe properties and command line options (continued)		
Property name	**Command line option**	**Description**
StreamCaptureFile *string*	`-streamcapturefile` *string*	Use this property to specify the location of the stream capture file. On UNIX and Linux operating systems, the default is `$OMNIHOME/var/message_bus.stream`. On Windows operating systems, you must specify the full directory path to the file. For example: `C:\\IBM\\Tivoli\\Netcool\\omnibus\\var\\ message_bus.stream`
TransformerFile *string*	`-transformerfile` *string*	Use this property to specify the location of the transformer properties file. This property can be used to specify the transformer configuration file for XML event data transformation, or the JSON parser configuration file for parsing different JSON object structures. On UNIX and Linux operating systems, the default is `$OMNIHOME/java/conf/transformers.xml`. On Windows operating systems, you must specify the full directory path to the file. For example: `'C:\\IBM\\Tivoli\\Netcool\\omnibus\\java\\conf\\transformers.xml'`
TransportFile *string*	`-transportfile` *string*	Use this property to specify the location of the transport properties file. On UNIX and Linux operating systems, the default is `$OMNIHOME/java/conf/jmsTransport.properties`. On Windows operating systems, you must specify the full directory path to the file. For example: `'C:\\IBM\\Tivoli\\Netcool\\omnibus\\java\\conf\\jmsTransport.properties'`

Table 22. Probe properties and command line options (continued)		
Property name	**Command line option**	**Description**
TransportType *string*	-transporttype *string*	Use this property to either specify the transport method to be used or to define the name of the transport module class to use. This property takes the following values: • Cometd • EventSource • File • HTTP • JMS • KAFKA • MQTT • Socket • Webhook • WebSocket The default is JMS.
Username *string*	-username *string*	Use this property to specify the user account for logging into the XML or JSON event source. This property is only used by the probe if you are using the WebSocket **TransportType**. The default is "". **Note:** The probe uses this value to replace the ++Username++ token (if it is specified) in the restWebSocketTransport.properties file or in the restWebHookTransport.properties file.

Properties and command line options provided by the Java Probe Integration Library (probe-sdk-java) version 11.0

All probes can be configured by a combination of generic properties and properties specific to the probe.

The following table describes the properties and command line options that are provided by the Java Probe Integration Library (probe-sdk-java) version 11.0.

Note: Some of the properties listed may not be applicable to your probe.

Table 23. Properties and command line options

Property name	Command line option	Description
CommandPort *integer*	-commandport *integer*	Use this property to specify the port to which users can Telnet to communicate with the probe using the Command Line Interface (CLI) supplied. The default is 6970.
CommandPortLimit *integer*	-commandportlimit *integer*	Use this property to specify the maximum number of Telnet connections that can be made to the probe. The default is 10.
DataBackupFile *string*	-databackupfile *string*	Use this property to specify the path to the file that stores data between probe sessions. The default is "". **Note:** Specify the path relative to $OMNIHOME/var.
HeartbeatInterval *integer*	-heartbeatinterval *integer*	Use this property to specify the frequency (in seconds) with which the probe checks the status of the host server. The default is 1.
Inactivity *integer*	-inactivity *integer*	Use this property to specify the length of time (in seconds) that the probe allows the port to receive no incoming data before disconnecting. The default is 0 (which instructs the probe to not disconnect during periods of inactivity).
InactivityAction *string*	-inactivityaction *string*	Use this property to specify the action the probe takes when the inactivity timeout is reached: SHUTDOWN: The probe sends a ProbeWatch message to notify the user and then shuts down. CONTINUE: The probe sends a ProbeWatch message to notify the user, but does not shut down. The default is SHUTDOWN.

Table 23. Properties and command line options (continued)

Property name	Command line option	Description
InitialResync *string*	`-initialresync` *string*	Use this property to specify whether the probe performs resynchronization on startup. This property takes the following values: `false`: The probe does not request resynchronization on startup. `true`: The probe requests resynchronization on startup. For most probes, the default value for this property is `false`. If you are running the JDBC Probe, the default value for the **InitialResync** property is `true`. This is because the JDBC Probe only acquires data using the resynchronization process.
MaxEventQueueSize *integer*	`-maxeventqueuesize` *integer*	Use this property to specify the maximum number of events that can be queued between the non native process and the ObjectServer. The default is `0`. **Note:** You can increase this number to increase the event throughput when a large number of events is generated.
ResyncInterval *integer*	`-resyncinterval` *integer*	Use this property to specify the interval (in seconds) at which the probe makes successive resynchronization requests. For most probes, the default value for this property is `0` (which instructs the probe to not make successive resynchronization requests). If you are running the JDBC Probe, the default value for the **ResyncInterval** property is `60`. This is because the JDBC Probe only acquires data using the resynchronization process.
RetryCount *integer*	`-retrycount` *integer*	Use this property to specify how many times the probe attempts to retry a connection before shutting down. The default is `0` (which instructs the probe to not retry the connection).

Table 23. Properties and command line options (continued)		
Property name	**Command line option**	**Description**
RetryInterval *integer*	`-retryinterval` *integer*	Use this property to specify the length of time (in seconds) that the probe waits between successive connection attempts to the target system. The default is `0` (which instructs the probe to use an exponentially increasing period between successive connection attempts, for example, the probe will wait for 1 second, then 2 seconds, then 4 seconds, and so forth).
RotateEndpoint *string*	`-rotateendpoint` *string*	Use this property to specify whether the probe attempts to connect to another endpoint if the connection to the first endpoint fails. This property takes the following values: `false`: The probe does not attempt to connect to another endpoint if the connection to the first endpoint fails. `true`: The probe attempts to connect to another endpoint if the connection to the first endpoint fails. The default is `false`.

Elements

The probe breaks event data down into tokens and parses them into elements. Elements are used to assign values to ObjectServer fields; the field values contain the event details in a form that the ObjectServer understands.

During installation of the probe, several rules files are installed in addition to the main `message_bus.rules` file. These files contain default rules for specific event sources that can be included in the main rules file. The following table lists these files and the format of their event sources.

Table 24. Additional rules files	
Rules file	**Event source format**
`message_bus_cbe.rules`	Common Base Event (CBE)
`message_bus_netcool.rules`	Netcool
`message_bus_wbe.rules`	WebSphere Business Event (WBE)
`message_bus_wef.rules`	WSDM Event Format (WEF)

The probe can create different elements based on the XSLT file of an event source. The following table describes the elements that the probe generates. Not all the elements described are generated for each event. The elements that the probe generates depend on the event type.

Table 25. Elements	
Element name	**Element description**
$Acknowledged	This element indicates whether the alert has been acknowledged. Alerts can be acknowledged manually by a network operator or automatically by a correlation or workflow process.
$Agent	This element displays the agent information.
$AlertGroup	This element shows the descriptive name of the type of failure indicated by the alert.
$AlertKey	This element displays the descriptive key that indicates the managed object instance referenced by the alert.
$Class	This element identifies the class of the XML event source from which the alert was generated. The $Class controls the applicability of context-sensitive event list tools.
$Customer	This element displays the name of the customer affected by this alert.
$EventId	This element displays the ID of the event.
$ExpireTime	This element contains the number of seconds from the time an alert was last received by the ObjectServer (stored by the LastOccurence field) until it is cleared automatically. This element is used by the Netcool/OMNIbus Expire automation.
$ExtendedAttr	This element displays the extended attribute type of the managed entity. The probe groups the respective elements based on their parent-child relation; and then sorts all the elements in the parent and child sections by their associated values.
$FirstOccurrence	This element contains the time in seconds (from midnight Jan 1, 1970) when this alert was created or when polling started at the probe.
$Flash	This element indicates whether the option to make the event list flash is enabled.
$Grade	This element indicates the escalation status for the alert.
$Identifier	This element contains the identifier information of the alert.
$InternalLast	This element displays the time when the alert was at the ObjectServer.
$LastOccurrence	This element contains the time when this alert was last updated at the probe.

Table 25. Elements (continued)

Element name	Element description
$LocalNodeAlias	This element displays 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 XML event source. Use it in managed object instance identification.
$LocalPriObj	This element displays the primary object referenced by the alert.
$LocalRootObj	This element displays an object that is equivalent to the primary object referenced in the alarm. Use it in managed object instance identification.
$LocalSecObj	This element displays the secondary object referenced by the alert.
$NmosCauseType	This element displays the type of the cause that triggered the alert.
$NmosObjInst	This element shows the populated details of the alert.
$NmosSerial	This element displays the serial number of a suppressed alert.
$Node	This element identifies the managed entity from which the alert originated.
$NodeAlias	This element displays the alias of the node. For network devices or hosts, this should be the logical (layer-3) address of the entity. For IP devices or hosts, this should be the IP address.
$OwnerGID	This element displays 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.
$OwnerUID	This element shows the identifier of the user who is assigned to handle this alert. The default is 65534, which is the identifier for the nobody user.
$PhysicalCard	This element displays the card name or description indicated by the alert.
$PhysicalPort	This element displays the port number indicated by the alert.
$PhysicalSlot	This element displays the slot number indicated by the alert.
$Poll	This element displays the time (in seconds) the probe has polled for the alert.

Table 25. Elements (continued)	
Element name	**Element description**
$ProcessReq	This element indicates whether the alert should be processed by Netcool/OMNIbus.
$RemoteNodeAlias	This element displays the network address of the remote network entity. Use it in managed object instance identification.
$RemotePriObj	This element displays the primary object of a remote network entity referenced by an alarm. Use it in managed object instance identification.
$RemoteRootObj	This element displays an object that is equivalent to the remote entity's primary object referenced in the alarm. Use it in managed object instance identification.
$RemoteSecObj	This element displays the secondary object of a remote network entity referenced by an alarm. Use it in managed object instance identification.
$resync_event	This element indicates whether the current event was received by the probe during resynchronization of active alarms with the target system.
$ServerName	This element displays the name of the originating ObjectServer. The Gateway for Message Bus uses it to control propagation of alerts between ObjectServers..
$ServerSerial	This element displays the serial number of the alert on the originating ObjectServer.
$Service	This element displays the name of the service affected by this alert.
$Severity	This element indicates the severity level of the alert. It provides an indication of 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.
$StateChange	This element indicates the state change of the alert. It is an automatically maintained ObjectServer timestamp of the last insert or update of the alert from any XML event source.
$Summary	This element contains the summary information on the cause of the alert.
$SuppressEscl	This element displays the suppression level manually selected by operators from the event list.
$Tally	This element shows the number of times that the alert has occurred.

Table 25. Elements (continued)	
Element name	**Element description**
$TaskList	This element indicates whether an operator has added the alert to the Task List.
$Type	This element identifies the alert type.
$URL	This element displays an optional URL, which provides a link to additional information in the XML event source.
$X733CorrNotif	This element displays a listing of all notifications with which this notification is correlated.
$X733EventType	This element indicates the alert type.
$X733ProbableCause	This element the indicates probable cause of the alert.
$X733SpecificProb	This element indicates the probable cause of the alert.

Error messages

Error messages provide information about problems that occur while running the probe. You can use the information that they contain to resolve such problems.

The following table describes the error messages specific to this probe. For information about generic error messages, see the *IBM Tivoli Netcool/OMNIbus Probe and Gateway Guide*.

Table 26. Error messages		
Error	**Description**	**Action**
Failed to startup probe	The probe failed to start, probably due to an invalid combination of properties set in the message_bus.props file.	Check the values set for the **Host**, **Port Username**, and **Password** properties.
Failed to transform	The probe is unable to transform the XML into name-value pairs.	Check the entries in the transformer file. Then test the XSLT file created for the event source.
Failed to parse message	The probe could not parse the event data.	Check the format of the event generated by the XML event source.
Failed to record data into backup file	The probe could not write record data into the backup file.	Check that the backup file is specified correctly by the **DataBackupFile** property in the message_bus.props file and that the file has the appropriate permissions set.

Table 26. Error messages (continued)		
Error	**Description**	**Action**
`Failed to start Transport module for connection`	The transport module failed to start.	Check the value set for the **`TransportType`** property and the details specified in the **`TransportFile`**.
`Failed to subscribe Transport module to the interface`	The transport module failed to subscribe to the event source.	Check the details specified in the **`TransportFile`**.
`Failed to get active alarms during resync`	The probe failed to received active alarms during resynchronization with the event source.	Check the details specified in the **`TransportFile`**.
`Exception caught in WebSocketClientHandler: Queue full` **Note:** This error message only applies when the Probe for Message Bus runs with the WebSocket transport.	The WebSocket transport event queue has reached its limit and has started to discard events. This usually occurs in a flooding scenario or if the event processing is slow or blocked.	Verify that no other error occurred in the probe log or ObjectServer logs that could potentially slow down or block the probe event processing. Verify that the probe is not under a flood or denial-of-service attack.

Common error messages

The transporter and transformer modules generate error messages that relate to both the Gateway for Message Bus and the Probe for Message Bus.

The following table describes the error messages that are generated by the transporter and transformer modules.

Table 27. Common error messages		
Error	**Description**	**Action**
`Failed to get message text`	The JMS transport module failed to get the text from the JMS message that it received.	There was a problem with the format of the message received from the JMS.
`Unsupported message type`	The JMS has received a message that is not a text message.	Only messages in text format are supported.
`No transformer defined for name` *`name`*	The gateway is trying to find the correct transformer to use to transform an event, but the message ID for that transformer is not present in the `transformers.xml` file.	Add a transformer to the `transformers.xml` file that corresponds to the message ID.
`No endpoint defined for name` *`name`*	The gateway is trying to find the correct endpoint to send a message to, but the transformer in the `transformers.xml` file for this message does not specify an endpoint.	Specify an **`endpoint`** for the transformer that corresponds to this message.

Table 27. Common error messages (continued)

Error	Description	Action
`No transformer defined for endpoint endpoint`	The probe has received an event from an endpoint for which there is no transformer in the `transformers.xml` file.	Add a transformer to the `transformers.xml` file for this endpoint.
`Invalid entry in transformer definition file - duplicate endpoint endpoint in southbound transformer entry`	Two southbound transformers in the `transformers.xml` file have been specified with the same endpoint.	You can only create one southbound transformer for each endpoint. Remove one of the transformers from the `transformers.xml` file.
`Invalid entry in transformer definition file - duplicate name name in northbound transformer entry`	Two northbound transformers in the `transformers.xml` file have been specified with the same name.	You cannot create two northbound transformers with the same name. Rename one of the transformers.
`Invalid transformer type type specified in transformers file`	A transformer in the `transformers.xml` file specifies a transformer type that is not supported.	The message type should either be northbound or southbound. Update the transformer in the `transformers.xml` file.
`Property type type does not exist`	The type specified for a property definition with a transformer is not a Java object.	Correct the transformer in the `transformers.xml` file by specifying a valid Java object name, for example, java.lang.String.
`Invalid property name propName in transformer name`	The name specified for a property definition with a transformer is not valid.	Correct the transformer in the `transformers.xml` file by specifying a valid name. This should be XSLT file name.
`Method method_name in class transformerClass has thrown an exception`	The method specified in the transformer definition file has failed.	Use the information in the message to diagnose the problem.
`Failed to read transformer definition file`	The transformer module failed to read the transformer definition file (`transformers.xml`).	Check that the file is specified correctly and that the permissions on the `transformers.xml` file are set correctly.
`Unknown property`	A transformer in the `transformers.xml` file contained a property that is not recognized.	Update the transformer in the `transformers.xml` file.
`Failed to transform message`	The transformer module could not transform an XML event.	There may be a problem with the XLST file. Try testing it using the transformer validation tool.

Table 27. Common error messages (continued)

Error	Description	Action
`Failed to read XSLT file`	The transformer module failed to read an XSLT file specified within the transformer definition file.	Check that the file is specified correctly and that the permissions on the .xsl file are set correctly.
`Invalid transformer class name found`	A transformer in the `transformers.xml` file contains an invalid transformer class name.	Check that you have specified the transformer class correctly in the `transformers.xml` file.
`Failed to close file` `Failed to find file`	The named file was not found.	Check that the name of the file has been specified correctly in the properties file.
`Stream file` *`filename`* `is not a stream capture file`	The file identified is not a stream capture file and cannot be used by the transport module.	Specify an alternative file name in the transport properties file.
`I/O exception whilst opening file`	The transport module could not open the file specified.	Check the permissions on the file indicated.
`Failed to read object from file`	The probe failed to read the stream capture file.	Try recreating the stream capture file and running it through the probe again.
`Error in stream capture file`	The stream capture file identified contains an error.	Try recreating the stream capture file and running it through the probe again.
`Failed to find initial context`	The class name in the transport properties file is incorrect.	Check the setting of the **`initialContextFactory`** property.
`Failed to create subscriber for topic:` *`topicName`*	The transport module failed to subscribe to a topic with the JMS.	Use the information in the message to diagnose the problem.
`JNDI lookup for topic:` *`topicName`* `Failed to find` *`topicName`* `in JNDI`	The transport module could not find one of the topics specified in the JNDI.	Check that the topic has been specified correctly in the transport file.
`Failed to start JMS subscriber`	The transport module failed to start the JMS subscriber.	Use the information in the message to diagnose the problem.
`Failed to send message`	The transport module failed to send a message to the endpoint specified in the transformer.	Use the information in the message to diagnose the problem.
`Failed to find` *`topicName`* `in JNDI`	The transport module cannot find one of the topics in the JNDI.	Check that the topic has been specified correctly in the transport file.

Table 27. Common error messages (continued)

Error	Description	Action
`Failed to close topic connection`	The transport module failed to close a topic connection within JMS.	Use the information in the message to diagnose the problem.
`Invalid property` *`propName`*	The transport properties file contains a property that is not supported.	Update the transport properties file.
`Invalid transport class name found`	The **`transportModule`** property of the transport file contains an invalid class name.	Change the value of the **`transportModule`** property to a class that exists in the jar file.
`Failed to load properties file`	The transport module could not read the transport properties file.	Check that the properties file exists and that the permissions are set correctly.
`Invalid property` *`propName`* `in file` *`transportFile`* `does not exist in transport class` *`transportClass`*	The transport file contained a property that is not supported by the transport module.	Update the transport properties file. For a description of the properties that the transport files supports, see “Configuring the transport properties files” on page 18.
`Method` *`methodName`* `in class` *`transportClass`* `has thrown an exception`	The method specified in the transport file has failed. It depends on the method but the method would probably tell you what's wrong.	Use the information in the message to diagnose the problem.
`key + " contains non-existent environment variable` *`value`*	A value in the transport properties file or in the transformer file contains an environment variable which does not exist.	Set the environment variable specified to an appropriate value.
`Failed to create JMS connection`	The transport module failed to create a connection to the JMS.	Check that you have specified the parameters correctly in the transport properties file.
`Failed to parse message`	The probe failed to parse the message. This could be caused by an invalid character such as a non-printable character in the raw data.	Verify that the inbound data is in the correct format and structure.
`Timed out waiting for response`	The probe sent an HTTP request but the response did not arrive within the timeout period. This applies to the REST/Websocket transport.	Check the URI used for each request. Check that the target received the HTTP request from the probe and responded within the specified timeout period.

ProbeWatch messages

During normal operations, the probe generates ProbeWatch messages and sends them to the ObjectServer. These messages tell the ObjectServer how the probe is running.

The following table describes the raw ProbeWatch error messages that the probe generates. For information about generic ProbeWatch messages, see the *IBM Tivoli Netcool/OMNIbus Probe and Gateway Guide*.

Table 28. ProbeWatch messages

ProbeWatch message	Description	Triggers/causes
`Connection to source lost`	The connection to the XML source has been lost.	The target system might have disconnected or gone down.

`Failed to open stream capture file`
`Failed to write to stream capture file`
`Failed to close stream capture file`
	The probe is unable to use the specified stream capture file.	Check the permissions set for the file and the directory in which it is being written. Then check the value specified for the **`StreamCaptureFile`** property.

`Start resynchronization`	The resynchronization process started.	The probe started with the **`InitialResync`** property set to `true`.
`Finish resynchronization`	The resynchronization process ended.	The probe completed the resynchronization process.

Using the probe with the Gateway for Message Bus

You can use the Probe for Message Bus with the Gateway for Message Bus to process XML messages stored in a file or transmitted using the JMS, Webhook, or MQTT transport protocols.

You can use the probe and gateway together as a single implementation. In this scenario, the probe uses the transport module to acquire XML events, uses the transformer module to convert them into name-value pairs, tokenizes them, and then sends them as Netcool events to the ObjectServer. The gateway reads Netcool XML events from the ObjectServer, uses the transformer module to convert them into a format appropriate for the destination application, and uses the transport module to send the transformed events to their destination application.

The following sections describe how to use the probe and gateway together as a single implementation:

- "Requirements" on page 92
- "Sample implementation using JMS" on page 93

Requirements

Several software packages are required to operate the Probe for Message Bus and the Gateway for Message Bus together.

You can download the probe, the gateway, and all required installation packages from the IBM Passport Advantage® Online website:

http://www-306.ibm.com/software/howtobuy/passportadvantage/

To use the probe and the gateway together, you will require the following packages:

- `omnibus-`*`arch`*`-common-transformer-`*`version`*

- `omnibus-arch-common-transportmodule-version`
- `omnibus-arch-gateway-libngjava-version`
- `omnibus-arch-probe-nonnative-base-version`
- `omnibus-arch-gateway-nco-g-xml-version`
- `omnibus-arch-probe-nco-p-xml-version`

where *arch* is the operating system you are installing the components on and *version* is the package version.

Sample implementation using JMS

This section describes an implementation of the probe and gateway working together.

The following diagram shows an example of the probe and gateway working together, with an Enterprise Service Bus (ESB) using JMS:

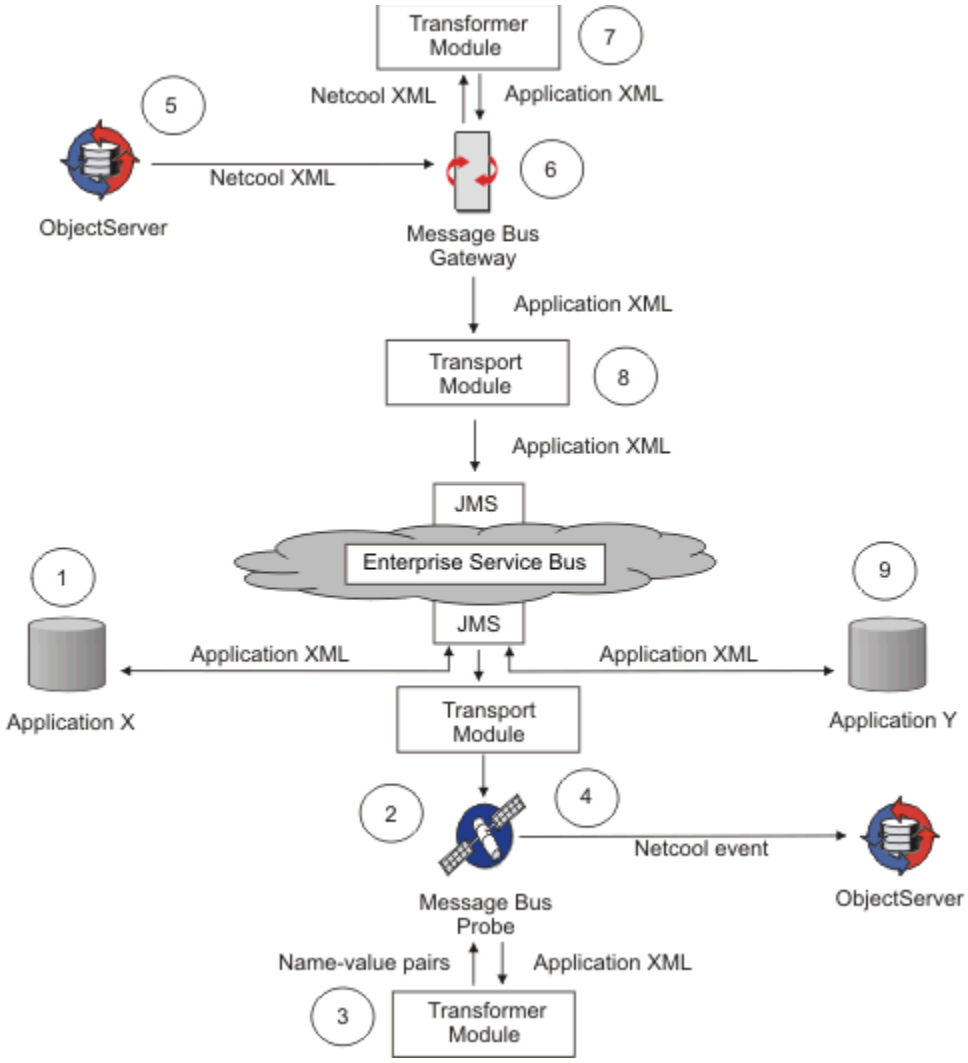

Figure 1. Probe and gateway working with ESB using JMS

The flow of data between XML event sources and consumers, and Netcool/OMNIbus is as follows:

1. Applications generate XML events and publish them to topics in the ESB.
2. The probe uses the transport module to subscribe to the topics in the ESB and receives the XML events published to those topics.
3. The probe uses the transformer module to convert the XML events in name-value pairs using the transformation defined by the `transformers.xml` file.

For each event source, the `transformers.xml` file contains a transformer entry. This entry identifies the source of the XML event and determines which .xsl file to use to convert the XML event.

4. The probe parses the name-value pairs in Netcool events and sends them to the ObjectServer.
5. The ObjectServer generates events that need to be written in XML format to various applications.
6. The gateway reads the Netcool events generated by the ObjectServer.
7. The gateway uses the transformer module to convert the Netcool events into XML events using the transformation defined by the `transformers.xml` file.

   For each event source, the `transformers.xml` file contains a transformer entry. Within this entry, the transformation name (which equates to the message ID of the Netcool event) determines which .xsl file to use to convert the event to XML format and to which endpoint to publish the XML event.
8. The gateway uses the transport module to publish the transformed XML events to topics in the ESB.
9. The applications subscribe to the topics and receive the XML events published by the gateway.

Note: XML events are not always transformed. For example, if the event source is a Gateway for Message Bus and the event consumer is a Probe for Message Bus, the XML events are not transformed by the transformer module.

Frequently asked questions

Various questions arise as users work with the probe. Answers to these questions are provided for your reference.

The probe is running but not parsing the .xml file. What is wrong?

The probe is running correctly and there are no errors in the probe debug log, but the probe is not parsing anything in the `.xml` file.

The `.xml` file is not correctly specified. Ensure that the `.xml` file conforms to the standard specified by W3C. The standard is documented at:

http://www.w3.org/TR/xslt

Use the transformer testing tool to ensure that the `.xml` file conforms to the W3C standard. For information about using the tool, see “Using the transformer testing tool” on page 68.

Troubleshooting

This topic provides troubleshooting assistance for running the Message Bus Probe.

Performing an access token request when running the probe with either the Kafka or EventSource transport

The Message Bus Probe running with either the Kafka or EventSource transport can be configured with one or more REST API request such as to perform an access token request and to create a subscription request with the target system after initiating a connection (`login`). If an error occurs before the connection stage completes (after the probe has sent out the `login` HTTP request(s)), the probe may not send the configured `logout` requests to revoke the access token and to remove the subscription. For some systems that limits the number of client access token or subscription, it may reject subsequent `login` requests from the probe.

Solution

After several connection attempts, if the probe is still unable to `login` to a target system although it has been configured to construct a valid HTTP request for the REST API request, it could be that the target system has reached its limit and is rejecting the probe `login` request until the previous logins are cleaned up on the target system.

Check whether there are any invalid tokens or subscriptions on the target system and clear them if necessary.

Known issues with the Probe for Message Bus

This section explains some known issues with the Probe for Message Bus.

Recommendation when using SSL

If the probe is running on Java 6 and encounters an SSLHandshakeException with a fatal alert handshake failure message using TLSv1 , TLSv1.1, or TLSv1.2 protocol, you should upgrade to Java 7. To use TLSv1.2 SecurityProtocol, Java 7 is required.

You have the following options:

- Either install OMNIbus V8.1 (because it comes with JRE 7)
- or, install JRE 7 separately

Upgrading the common TransportModule library

After upgrading the TransportModule, the probe can fail to start due to a ClassNotFoundException being thrown during startup. This is possibly due to the probe missing a dependency libraries in its CLASSPATH.

TransportModule version 12 contains the following new files to set the CLASSPATH used:

- TransportModule.env (for UNIX operating systems)
- TransportModule.bat (for Windows operating systems)

You can configure the probe to call these files and include the TRANSPORT_CLASSPATH variable in the probe's CLASSPATH.

Update the probe's environment file or batch file with the following lines to replace old variables which defines the path to the libraries used by the TransportModule and make sure the TRANSPORT_CLASSPATH is used.

On UNIX operating systems:

```
# Load TransportModule and dependency JARS
. $OMNIHOME/java/jars/TransportModule.env
CLASSPATH_SETTING=${TRANSFORMER_JAR}:${TRANSPORT_CLASSPATH}:${JACKSON_JAR};
```

On Windows operating systems:

```
REM Set TRANSPORT_CLASSPATH as specified in TransportModule
call %OMNIHOME%\java\jars\TransportModule.bat
set MESSAGEBUS_CLASSPATH=%TRANSFORMER_CLASSPATH%;%TRANSPORT_CLASSPATH%;
%JACKSON_CLASSPATH%
```

Incorrect setting for the keyStorePassword property in the HTTP transport properties

If an incorrect password is set using the **keyStorePassword** property in the HTTP transport properties file, the probe will fail to load the Keystore during initialization. This will make the probe hang, instead of shutting down. To shutdown the probe, kill the probe process manually. The following log message is printed when this error occurs:

```
2017-02-24T05:06:56: Debug: D-JPR-000-000: com.ibm.tivoli.oidk.ProbeImpl.connect
     EXITING
2017-02-24T05:06:56: Information: I-JPR-000-000: [HttpParser]:
     Max http payload size: 2097152
2017-02-24T05:06:56: Error: E-JPR-000-000: Failed to create server listening socket
     'null:5490'. [java.net.SocketException: java.security.NoSuchAlgorithmException:
     Error constructing implementation (algorithm: Default, provider:
     IBMJSSE2, class: com.ibm.jsse2.ec)]
2017-02-24T05:06:56: Error: E-JPR-000-000: Fail to subscribe Transport module
     to the interface
```

```
2017-02-24T05:06:56: Error: E-JPR-000-000: Failed to connect; ProbeException:
     Fail to subscribe Transport module to the interface;
     TransportSubscribeException:
     Failed to start all HTTP Server ports.
2017-02-24T05:06:56: Debug: D-JPR-000-000:
     com.ibm.tivoli.netcool.omnibus.probe.ProbeException:
     Fail to subscribe Transport module to the interface
at com.ibm.tivoli.oidk.ProbeImpl.subscribe(Unknown Source)
at com.ibm.tivoli.netcool.omnibus.probe.framework.ProbeRunner.
     connect(Unknown Source)
at com.ibm.tivoli.netcool.omnibus.probe.framework.ProbeRunner.
     connectAndRun(Unknown Source)
at com.ibm.tivoli.netcool.omnibus.probe.framework.ProbeRunner.run(Unknown Source)
at com.ibm.tivoli.netcool.omnibus.oidk.Probe.start(Unknown Source)
at com.ibm.tivoli.netcool.omnibus.oidk.Probe.main(Unknown Source)
Caused by: com.ibm.tivoli.netcool.integrations.transportmodule.
     TransportSubscribeException:
     Failed to start all HTTP Server ports.
at com.ibm.tivoli.netcool.integrations.transportmodule.HttpTransport.
     subscribe(Unknown Source)
... 6 more
2017-02-24T05:06:56: Information: I-UNK-000-000: Probewatch: Unable to get events.
     Failed to connect; ProbeException:
     Fail to subscribe Transport module to the interface;
     TransportSubscribeException: Failed to start all HTTP Server ports.
2017-02-24T05:06:56: Debug: D-UNK-000-000: Rules file processing took 18 usec.
2017-02-24T05:06:56: Debug: D-UNK-000-000: Flushing events to object servers
2017-02-24T05:06:56: Debug: D-UNK-000-000: 1 buffered alerts
2017-02-24T05:06:56: Debug: D-UNK-000-000: Flushing events to object servers
2017-02-24T05:06:56: Debug: D-UNK-000-000: 0 buffered alerts
2017-02-24T05:06:56: Debug: D-JPR-000-000:
     com.ibm.tivoli.netcool.omnibus.probe.framework.
     ProbeRunner.connect EXITING
2017-02-24T05:06:56: Information: I-JPR-000-000: DISCONNECT 'Unable to connect'
2017-02-24T05:06:56: Debug: D-JPR-000-000:
     com.ibm.tivoli.netcool.omnibus.probe.framework.
     ProbeRunner.resetForRetry ENTERING
2017-02-24T05:06:56: Debug: D-JPR-000-000:
     com.ibm.tivoli.netcool.omnibus.probe.framework.
     ProbeRunner.haltScheduledTasks ENTERING
2017-02-24T05:06:56: Debug: D-JPR-000-000:
     com.ibm.tivoli.netcool.omnibus.probe.framework.
     ProbeRunner.haltScheduledTasks EXITING
2017-02-24T05:06:56: Debug: D-JPR-000-000:
     com.ibm.tivoli.netcool.omnibus.probe.framework.
     ProbeRunner.disconnectProbe ENTERING
2017-02-24T05:06:56: Debug: D-JPR-000-000:
     com.ibm.tivoli.oidk.ProbeImpl.disconnect
     ENTERING
     Exception in thread "MessageSenderThread" java.lang.NullPointerException
at com.ibm.tivoli.netcool.integrations.transportmodule.http.HttpServer.
     isRunning(Unknown Source)
at com.ibm.tivoli.netcool.integrations.transportmodule.HttpTransport.
     httpServerIsAlive(Unknown Source)
at com.ibm.tivoli.netcool.integrations.transportmodule.HttpTransport.
     isConnected(Unknown Source)
at com.ibm.tivoli.oidk.ProbeImpl.disconnect(Unknown Source)
at com.ibm.tivoli.netcool.omnibus.probe.framework.ProbeRunner.
     disconnectProbe(Unknown Source)
at com.ibm.tivoli.netcool.omnibus.probe.framework.ProbeRunner.
     resetForRetry(Unknown Source)
at com.ibm.tivoli.netcool.omnibus.probe.framework.ProbeRunner.
     messageReceived(Unknown Source)
at com.ibm.tivoli.netcool.omnibus.probe.services.impl.
     SimpleMessageService$SenderThread.
     send(Unknown Source)
at com.ibm.tivoli.netcool.omnibus.probe.services.impl.
     SimpleMessageService$SenderThread.
     run(Unknown Source)
```

Recommended Action: Ensure the correct keystore password is set when securing the HTTP transport.

Probe fails to start when using a JSON parser configuration file as a TransformerFile due to an incompatible Java version

Java 7 is required by the supporting libraries used to load the JSON parser configuration file. If a lower Java version is used, the probe will fail to initialize when loading the parser configuration file and will print the following error message:

```
Exception in thread "ProbeRunner" java.lang.UnsupportedClassVersionError:
JVMCFRE003 bad major version; class=com/fasterxml/jackson/databind/
ObjectMapper, offset=6
```

Consider upgrading to Netcool/OMNIbus V8.1 which includes JRE 7, or install JRE 7 separately.

When integrating with iDirect Pulse

Disconnection during a clean start

In the first connection to iDirect (clean start), the probe creates a WebSocket connection with the `/api/1.0/dde/alarm?start_time__gte=0` URI to query historical alarms and continue listening for new alarms or updates to existing alarms. However, the URI time window maybe too large, which may cause the server to disconnect from the probe.

Recommended action: Use a smaller time window, or configure the probe to enable **`Retry`** so that it reconnects using the WebSocket Persistent URI `/api/1.0/dde/alarm?start_time__gte=+ +ProbeDisconnectTime++`. You must set the **`DataBackupFile`**' property for the probe to record the last disconnection time in the backup file.

During beta testing, the following query windows were tested and the results are as below:

Table 29. Beta testing results

Query window	Connection status
Past 1 day	OK
Past 1 week	OK
Past 1 month	OK
Past 1 year	Probe disconnected

No events received through WebSocket while the Pulse server still initializing after a restart

If the Pulse server is restarted and is still initializing when the probe connects, the server might accept the WebSocket connection but then send an event with error code `500` (Internal Server Error) and then send no further alarms or a close connection request to the probe. The probe continues listening but is unaware that the server is not sending any alarms.

Recommended action: Increase the value set by the **`RetryInterval`** property to give sufficient time for the server to be ready before attempting to connect. The probe only retries the connection if it has successfully established a WebSocket connection before the server was restarted.

Configure the probe to disconnect and shutdown due to a period of inactivity by setting the **`Inactivity`** property to a time (in seconds) greater than zero to shut down the probe. The probe will then need to be restarted. You can configure a Process Agent to manage the probe process and restart it automatically if it is down.

No new updates received after a period upon successful WebSocket connection

If the server stops sending updates but does not disconnect the probe, the probe session may have ended on the server but the server did not request to close the connection.

Recommended Action: Enable the following set of properties in the WebSocket transport to send a periodic HTTP request as a keep-alive mechanism and inform the server that the probe is still listening. The following configuration is a suggestion and should be changed to use the correct URI if necessary. The **`subscribeRefreshInterval`** must be configured to a period before the probe session ends.

```
subscribeRefreshURI=/api/1.0/config/element/user?limit=1
subscribeRefreshMethod=GET
```

```
subscribeRefreshContent=
subscribeRefreshInterval=30
```

Recommendation when using WebSocketTransport or WebhookTransport with `autoReconnect=ON`

If **`HeartbeatInterval`** is set to too a low number, **`AutoReconnect`** may not arrive at the maximum count.

For example:

The attempts of autoReconnect are run by an exponentially increased interval. For a count of 5 attempts (1s, 2s, 4s, 8s, 16s) a total of 31s is required to complete 5 attempts.

If the **`HeartbeatInterval`** probe property is configured to 10s and the probe detects that the transport is not in an active connection state, it may shutdown earlier, after the 3rd try, leaving the 4th and 5th tries un-attempted.

Recommend Action: When using WebSocketTransport or WebhookTransport with `autoReconnect=ON`, set the **`HeatbeatInterval`** to a value greater than 31s (one minute is recommended).

Recommendation when using WebSocketTransport and WebhookTransport with `autoReconnect=ON` and the `httpHeaders` transport property

The **`httpHeaders`** transport property is used for all outgoing HTTP messages and is capable of accepting the HTTP header token substituted from the probe properties file or from tokens retrieved from an EMS at runtime.

However, `httpHeaders` tokens may be set to a `null`value during the early phase of probe initialization before tokens from the EMS are retrieved, which may risk request failure.

Recommend Action: When in use, set the probe properties as tokens using **`httpHeaders`**.

When in use, set dynamically retrieved tokens from EMS with the header properties specifically designed for its use case to delay the use of the following external token names:

- `loginRequestHeaders`
- `loginRefreshHeaders`
- `logoutRequestHeaders`
- `resyncRequestHeaders`
- `subscribeRequestHeaders`
- `subscribeRefreshHeaders`

Monitoring updates from ZooKeeper

Updates to topics or brokers are monitored by the ZooKeeper's topic/broker watch function when used.

Updates to topics or brokers are not displayed via ProbeWatch messages. They can be found in the probe log.

TLS handshake issue in Message Bus 8.0 when connecting to a server which only accepts TLSv1.2 Security Protocol using Webhook or Websocket transport

The Message Bus Probe Webhook and WebSocket transports have an HTTP client component which is used to make REST API calls to a remote target system. This component has an OAuth2.0 Module to request an access token from servers using the OAuth2.0 standard.

Known Issue Symptom

When configured to request an access token from a remote server which only accepts the TLS v1.2 protocol, the probe will throw a `TransportAuthorizeException` error due to a TLS handshake failure.

This is due to that the OAuth2.0 module in the HTTP client component of the transport starts the SSL handshake with a lower TLS version which is rejected by the server.

Resolution

Configure the probe's Webhook or Websocket transport to use **loginRequest** properties, instead of **tokenEndpointURI** to create a HTTP request to request an access token. For example, to create the HTTP request below, use the settings:

```
POST /token HTTP/1.1
Host: server.example.com
Authorization:Basic VXNlckZvckJhc2ljQXV0aGVudGljYXRpb246UGFzc3dvcmRGb3JCYXNpY0F1dGhlbnRpY2F0aW9u
Content-Type: application/x-www-form-urlencoded

grant_type=password&username=johndoe&password=A3ddj3w
```

Configure the probe properties file:

```
Host: 'server.example.com'
Port: 443
Username : 'UserForBasicAuthentication'
Password: 'PasswordForBasicAuthentication'
EnableSSL: 'true'
## Keystore file should contain the target server certificate imported.
KeyStore : '/home/keystore.jks'
KeyStorePassword: 'TheKeystorePassword'
```

Configure the transport properties file:

```
httpVersion=1.1
loginRequestURI=/token
loginRequestMethod=POST
loginRequestHeaders=
    Authorization=Basic ++Username++:++Password++,Content-Type=application/x-www-form-urlencoded
loginRequestContent=grant_type=password&username=johndoe&password=A3ddj3w

# For TLSv1.2 enabled server
securityProtocol=TLSv1.2
```

Additional troubleshooting topics

For additional troubleshooting topics for issues on common libraries such as Non-native, see https://www.ibm.com/support/knowledgecenter/SSSHTQ/omnibus/probes/all_probes/wip/reference/troubleshoot_probe_mtupacketissue.html.

Appendix A. Notices and Trademarks

This appendix contains the following sections:

- Notices
- Trademarks

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 want to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department 49XA

© Copyright IBM Corp. 2015, 2020

3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

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.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

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

Trademarks

IBM, the IBM logo, ibm.com, AIX, Tivoli, zSeries, and Netcool are 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.

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation 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.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Part Number:

(1P) P/N:

SC27-8701-14