Red Hat JBoss A-MQ 6.2

Connection Reference

A reference for all of the options for creating connections to a broker

Last Updated: 2018-06-15

Red Hat JBoss A-MQ 6.2 Connection Reference

A reference for all of the options for creating connections to a broker

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Copyright © 2015 Red Hat.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

Red Hat JBoss A-MQ supports a number of different wire protocols and message formats. In addition, it overlays reconnection logic and discovery logic over these options. This guide provides a quick reference for understanding how to configure connections between brokers, clients, and other brokers.

Table of Contents

CHAPTER 1. OPENWIRE OVER TCP **4**
- URI SYNTAX 4
- SETTING TRANSPORT OPTIONS 4
- TRANSPORT OPTIONS 5

CHAPTER 2. OPENWIRE OVER SSL **8**
- URI SYNTAX 8
- SETTING TRANSPORT OPTIONS 8
- SSL TRANSPORT OPTIONS 9
- CONFIGURING BROKER SSL OPTIONS 10
- CONFIGURING CLIENT SSL OPTIONS 10

CHAPTER 3. OPENWIRE OVER HTTP(S) **11**
- URI SYNTAX 11
- DEPENDENCIES 11

CHAPTER 4. OPENWIRE OVER UDP/IP **12**
- URI SYNTAX 12
- SETTING TRANSPORT OPTIONS 12
- TRANSPORT OPTIONS 13

CHAPTER 5. STOMP PROTOCOL **14**
- OVERVIEW 14
- URI SYNTAX 14
- TRANSPORT OPTIONS 15
- SSL TRANSPORT OPTIONS 15
- CONFIGURING BROKER SSL OPTIONS 16
- CONFIGURING CLIENT SSL OPTIONS 16

CHAPTER 6. MULTICAST PROTOCOL **17**
- URI SYNTAX 17
- TRANSPORT OPTIONS 17

CHAPTER 7. MQ TELEMETRY TRANSPORT(MQTT) PROTOCOL **19**
- URI SYNTAX 19
- TRANSPORT OPTIONS 19
- SSL TRANSPORT OPTIONS 20
- CONFIGURING BROKER SSL OPTIONS 20
- CONFIGURING CLIENT SSL OPTIONS 20

CHAPTER 8. ADVANCED MESSAGE QUEUING PROTOCOL (AMQP) **22**
- URI SYNTAX 22
- IDLETIMEOUT 22
- SECURITY 22
- SSL TRANSPORT OPTIONS 23
- CONFIGURING BROKER SSL OPTIONS 23
- CONFIGURING CLIENT SSL OPTIONS 23
- MAPPING FROM AMQP TO JMS 23
- AMQP-TO-JMS TRANSFORMERS 24
- HEADER MAPPING FOR ALL TRANSFORMERS 24
- HEADER MAPPING FOR NATIVE OR JMS TRANSFORMERS 24
- DEFAULT HEADER VALUES 26
- PROPERTY TYPE MAPPING 26

MESSAGE BODY MAPPING 27

CHAPTER 9. VM TRANSPORT 29
9.1. SIMPLE VM URI SYNTAX 29
9.2. ADVANCED VM URI SYNTAX 31

CHAPTER 10. DISCOVERING BROKERS 33
10.1. DISCOVERY AGENTS 33
10.2. DYNAMIC DISCOVERY PROTOCOL 38
10.3. FANOUT PROTOCOL 40

CHAPTER 11. PEER PROTOCOL 44
URI SYNTAX 44
BROKER OPTIONS 44
DEPENDENCIES 44

APPENDIX A. OPENWIRE FORMAT OPTIONS 46
FORMAT OPTIONS TABLE 46

APPENDIX B. CLIENT CONNECTION OPTIONS 48
OVERVIEW 48
OPTIONS 48
BLOB HANDLING 51
PREFETCH LIMITS 51
REDELIVERY POLICY 52

APPENDIX C. SERVER OPTIONS 54
SERVER OPTIONS TABLE 54

INDEX 55

CHAPTER 1. OPENWIRE OVER TCP

URI SYNTAX

A vanilla TCP URI has the syntax shown in Example 1.1, “Syntax for a vanilla TCP Connection” .

Example 1.1. Syntax for a vanilla TCP Connection

```
tcp://Host[:Port]?transportOptions
```

An NIO URI has the syntax shown in Example 1.2, “Syntax for NIO Connection”.

Example 1.2. Syntax for NIO Connection

```
nio://Host[:Port]?transportOptions
```

SETTING TRANSPORT OPTIONS

OpenWire transport options, *transportOptions*, are specified as a list of matrix parameters. How you specify the options to use differs between a client-side URI and a broker-side URI:

- When using a URI to open a connection between a client and a broker, you just specify the name of the option as shown.

  Example 1.3. Setting an Option on a Client-Side TCP URI

  ```
  tcp://fusesource.com:61616?trace=true
  ```

- When using a URI to open a broker listener socket, you prefix the option name with `transport.` as shown.

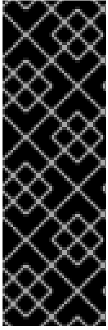
  Example 1.4. Specifying Transport Options for a Listener Socket

  ```
  tcp://fusesource.com:61616?transport.trace=true
  ```

- When using a URI to open a broker connection socket, you just specify the name of the option as shown.

  Example 1.5. Setting an Option on a Client-Side TCP URI

  ```
  tcp://fusesource.com:61616?trace=true
  ```



IMPORTANT

In XML configuration, you must escape the & symbol, replacing it with & as shown.

Example 1.6. Transport Options in XML

```
?option=value&amp;option=value&amp;...
```

TRANSPORT OPTIONS

Table 1.1, “TCP and NIO Transport Options” shows the options supported by the TCP and the NIO URIs.

Table 1.1. TCP and NIO Transport Options

Option	Default	Description
`minmumWireFormatVersion`	`0`	Specifies the minimum wire format version that is allowed.
`trace`	`false`	Causes all commands sent over the transport to be logged.
`daemon`	`false`	Specifies whether the transport thread runs as a daemon or not. Useful to enable when embedding in a Spring container or in a web container, to allow the container to shut down properly.
`useLocalHost`	`true`	When `true`, causes the local machine's name to resolve to `localhost`.
`socketBufferSize`	`64*1024`	Sets the socket buffer size in bytes.
`keepAlive`	`false`	When `true`, enables TCP KeepAlive on the broker connection. Useful to ensure that inactive consumers do not time out.
`soTimeout`	`0`	Specifies, in milliseconds, the socket timeout.
`soWriteTimeout`	`0`	Specifies, in milliseconds, the timeout for socket write operations.

Option	Default	Description
connectionTimeout	30000	Specifies, in milliseconds, the connection timeout. Zero means wait forever for the connection to be established.
closeAsync	true	The false value causes all sockets to be closed synchronously.
soLinger	MIN_INTEGER	When > -1, enables the SoLinger socket option with this value. When equal to -1, disables SoLinger.
maximumConnections	MAX_VALUE	The maximum number of sockets the broker is allowed to create.
diffServ	0	*(Client only)* The preferred Differentiated Services traffic class to be set on outgoing packets, as described in RFC 2475. Valid integer values are [0,64). Valid string values are EF, AF[1-3][1-4] or CS[0-7]. With JDK 6, only works when the Java Runtime uses the IPv4 stack, which can be done by setting the java.net.preferIPv4Stack system property to true. Cannot be used at the same time as the typeOfService option.
typeOfService	0	*(Client only)* The preferred *type of service* value to be set on outgoing packets. Valid integer values are [0,256). With JDK 6, only works when the Java Runtime uses the IPv4 stack, which can be done by setting the java.net.preferIPv4Stack system property to true. Cannot be used at the same time as the diffServ option.
wireFormat		The name of the wire format to use.

Option	Default	Description
`wireFormat.*`		All the properties with this prefix are used to configure the wireFormat. See Table A.1, “Wire Format Options Supported by OpenWire Protocol” for more information.
`jms.*`		All the properties with this prefix are used to configure client connections to a broker. See Appendix B, *Client Connection Options* for more information.

CHAPTER 2. OPENWIRE OVER SSL

URI SYNTAX

A vanilla SSL URI has the syntax shown in Example 2.1, “Syntax for a vanilla SSL Connection” .

Example 2.1. Syntax for a vanilla SSL Connection

```
ssl://Host[:Port]?transportOptions
```

An SSL URI for using NIO has the syntax shown in Example 2.2, “Syntax for NIO Connection”.

Example 2.2. Syntax for NIO Connection

```
nio+ssl://Host[:Port]?transportOptions
```

SETTING TRANSPORT OPTIONS

OpenWire transport options, *transportOptions*, are specified as a list of matrix parameters. How you specify the options to use differs between a client-side URI and a broker-side URI:

- When using a URI to open a connection between a client and a broker, you just specify the name of the option as shown.

  Example 2.3. Setting an Option on a Client-Side TCP URI

  ```
  tcp://fusesource.com:61616?trace=true
  ```

- When using a URI to open a broker listener socket, you prefix the option name with **`transport.`** as shown.

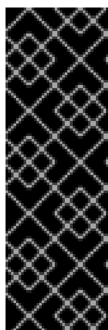
  Example 2.4. Specifying Transport Options for a Listener Socket

  ```
  tcp://fusesource.com:61616?transport.trace=true
  ```

- When using a URI to open a broker connection socket, you just specify the name of the option as shown.

  Example 2.5. Setting an Option on a Client-Side TCP URI

  ```
  tcp://fusesource.com:61616?trace=true
  ```



IMPORTANT

In XML configuration, you must escape the & symbol, replacing it with & as shown.

Example 2.6. Transport Options in XML

```
?option=value&amp;option=value&amp;...
```

SSL TRANSPORT OPTIONS

In addition to the options supported by the non-secure TCP/NIO transport listed in Table 1.1, “TCP and NIO Transport Options”, the SSL transport also supports the options for configuring the **SSLServerSocket** created for the connection. These options are listed in Table 2.1, “SSL Transport Options”.

Table 2.1. SSL Transport Options

Option	Default	Description
enabledCipherSuites		Specifies the cipher suites accepted by this endpoint, in the form of a comma-separated list.
enabledProtocols		Specifies the secure socket protocols accepted by this endpoint, in the form of a comma-separated list. If using Oracle's JSSE provider, possible values are: **TLSv1**, **TLSv1.1**, or **TLSv1.2** (do *not* use **SSLv2Hello** or **SSLv3**, because of the POODLE security vulnerability, which affects SSLv3).
wantClientAuth		*(broker only)* If **true**, the server requests (but does not require) the client to send a certificate.
needClientAuth	**false**	*(broker only)* If **true**, the server *requires* the client to send its certificate. If the client fails to send a certificate, the server will throw an error and close the session.

Option	Default	Description
`enableSessionCreation`	`true`	*(broker only)* If `true`, the server socket creates a new SSL session every time it accepts a connection and spawns a new socket. If `false`, an existing SSL session must be resumed when the server socket accepts a connection.

WARNING

If you are planning to enable SSL/TLS security, you must ensure that you explicitly disable the SSLv3 protocol, in order to safeguard against the Poodle vulnerability (CVE-2014-3566). For more details, see Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x.

CONFIGURING BROKER SSL OPTIONS

On the broker side, you must specify an SSL transport option using the syntax **transport.*OptionName***. For example, to enable an OpenWire SSL port on a broker, you would add the following transport element:

```
<transportConnector name="ssl" uri="ssl:localhost:61617?
transport.enabledProtocols=TLSv1,TLSv1.1,TLSv1.2" />
```

TIP

Remember, if you are specifying more than one option in the context of XML, you need to escape the ampersand, &, between options as `&`.

CONFIGURING CLIENT SSL OPTIONS

On the client side, you must specify an SSL transport option using the syntax **socket.*OptionName***. For example, to connect to an OpenWire SSL port, you would use a URL like the following:

```
ssl:localhost:61617?socket.enabledProtocols=TLSv1,TLSv1.1,TLSv1.2
```

CHAPTER 3. OPENWIRE OVER HTTP(S)

URI SYNTAX

An HTTP URI has the syntax shown in Example 3.1, “Syntax for an HTTP Connection”.

Example 3.1. Syntax for an HTTP Connection

```
tcp://Host[:Port]
```

An HTTPS URI has the syntax shown in Example 3.2, “Syntax for an HTTPS Connection”.

Example 3.2. Syntax for an HTTPS Connection

```
https://Host[:Port]
```

DEPENDENCIES

To use the HTTP(S) transport requires that the following JARs from the `lib/optional` folder are included on the classpath:

- `activemq-http-x.x.x.jar`
- `xstream-x.x.x.jar`
- `commons-logging-x.x.x.jar`
- `commons-codec-x.x.x.jar`
- `httpcore-x.x.x.jar`
- `httpclient-x.x.x.jar`

CHAPTER 4. OPENWIRE OVER UDP/IP

URI SYNTAX

A UDP URI has the syntax shown in Example 4.1, “Syntax for a UDP Connection” .

Example 4.1. Syntax for a UDP Connection

```
udp://Host[:Port]?transportOptions
```

SETTING TRANSPORT OPTIONS

OpenWire transport options, *transportOptions*, are specified as a list of matrix parameters. How you specify the options to use differs between a client-side URI and a broker-side URI:

- When using a URI to open a connection between a client and a broker, you just specify the name of the option as shown.

  Example 4.2. Setting an Option on a Client-Side TCP URI

  ```
  tcp://fusesource.com:61616?trace=true
  ```

- When using a URI to open a broker listener socket, you prefix the option name with `transport.` as shown.

  Example 4.3. Specifying Transport Options for a Listener Socket

  ```
  tcp://fusesource.com:61616?transport.trace=true
  ```

- When using a URI to open a broker connection socket, you just specify the name of the option as shown.

  Example 4.4. Setting an Option on a Client-Side TCP URI

  ```
  tcp://fusesource.com:61616?trace=true
  ```

IMPORTANT

In XML configuration, you must escape the & symbol, replacing it with & as shown.

Example 4.5. Transport Options in XML

```
?option=value&amp;option=value&amp;...
```

TRANSPORT OPTIONS

The UDP transport supports the options listed in Table 4.1, “UDP Transport Options”.

Table 4.1. UDP Transport Options

Option	Default	Description
`minmumWireFormatVersion`	`0`	The minimum version wire format that is allowed.
`trace`	`false`	Causes all commands sent over the transport to be logged.
`useLocalHost`	`true`	When `true`, causes the local machine's name to resolve to `localhost`.
`datagramSize`	`4*1024`	Specifies the size of a datagram.
`wireFormat`		The name of the wire format to use.
`wireFormat.*`		All options with this prefix are used to configure the wire format. See Table A.1, “Wire Format Options Supported by OpenWire Protocol” for more information.
`jms.*`		All the properties with this prefix are used to configure client connections to a broker. See Appendix B, *Client Connection Options* for more information.

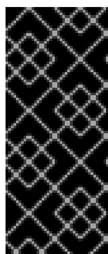
CHAPTER 5. STOMP PROTOCOL

Abstract

The Stomp protocol is a simplified messaging protocol that is specially designed for implementing clients using scripting languages. This chapter provides a brief introduction to the protocol.

OVERVIEW

The Stomp protocol is a simplified messaging protocol that is being developed as an open source project (http://stomp.codehaus.org/). The advantage of the stomp protocol is that you can easily improvise a messaging client—even when a specific client API is not available—because the protocol is so simple.



IMPORTANT

Apache ActiveMQ implements the Stomp v1.2 specification, except for the treatment of spaces that appear at the beginning or end of message header keys. The ActiveMQ implementation of Stomp trims leading and trailing spaces in message header keys (but preserves leading and trailing spaces in the header values). This behaviour is liable to change in a future release.

URI SYNTAX

Example 5.1, “Vanilla Stop URI” shows the syntax for a vanilla Stomp connection.

Example 5.1. Vanilla Stop URI

```
stomp://Host:[Port]?transportOptions
```

An NIO URI has the syntax shown in Example 5.2, “Syntax for Stomp+NIO Connection” .

Example 5.2. Syntax for Stomp+NIO Connection

```
stomp+nio://Host[:Port]?transportOptions
```

A secure Stomp URI has the syntax shown in Example 5.3, “Syntax for a Stomp SSL Connection” .

Example 5.3. Syntax for a Stomp SSL Connection

```
stomp+ssl://Host[:Port]?transportOptions
```

A secure Stomp+NIO URI has the syntax shown in Example 5.4, “Syntax for a Stomp+NIO SSL Connection”.

Example 5.4. Syntax for a Stomp+NIO SSL Connection

```
stomp+nio+ssl://Host[:Port]?transportOptions
```

TRANSPORT OPTIONS

The Stomp protocol supports the following transport options:

Table 5.1. Transport Options Supported by Stomp Protocol

Property	Default	Description
`transport.defaultHeartBeat`	`0,0`	Specifies how the broker simulates the heartbeat policy when working with legacy Stomp 1.0 clients. The first value in the pair specifies, in milliseconds, the server will wait between messages before timing out the connection. The second value specifies, in milliseconds, the the client will wait between messages received from the server. Because Stomp 1.0 clients do not understand heartbeat messages, the second value should always be 0. This option is set in the `uri` attribute of a broker's `transportConnector` element to enable backward compatibility with Stomp 1.0 clients.
`jms.*`		All the properties with this prefix are used to configure client connections to a broker. See Appendix B, *Client Connection Options* for more information.

SSL TRANSPORT OPTIONS

In addition to the options supported by the non-secure Stomp transports, the SSL transport also supports the options for configuring the `SSLServerSocket` created for the connection. These options are listed in Table 2.1, "SSL Transport Options".

WARNING

If you are planning to enable SSL/TLS security, you must ensure that you explicitly disable the SSLv3 protocol, in order to safeguard against the Poodle vulnerability (CVE-2014-3566). For more details, see Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x.

CONFIGURING BROKER SSL OPTIONS

On the broker side, you must specify an SSL transport option using the syntax **transport.*OptionName***. For example, to enable a Stomp SSL port on a broker, you would add the following transport element:

```
<transportConnector name="stompssl" uri="stomp+ssl://localhost:61617?
transport.enabledProtocols=TLSv1,TLSv1.1,TLSv1.2" />
```

TIP

Remember, if you are specifying more than one option in the context of XML, you need to escape the ampersand, &, between options as `&`.

CONFIGURING CLIENT SSL OPTIONS

On the client side, you must specify an SSL transport option using the syntax **socket.*OptionName***. For example, to connect to a Stomp SSL port, you would use a URL like the following:

```
stomp+ssl://localhost:61617?socket.enabledProtocols=TLSv1,TLSv1.1,TLSv1.2
```

CHAPTER 6. MULTICAST PROTOCOL

Abstract

Multicast is an unreliable protocol that allows clients to connect to brokers using IP multicast.

URI SYNTAX

Example 6.1, “Multicast URI” shows the syntax for a Multicast connection.

Example 6.1. Multicast URI

```
multicast://Host:[Port]?transportOptions
```

TRANSPORT OPTIONS

The Multicast protocol supports the following transport options:

Table 6.1. Transport Options Supported by Multicast Protocol

Property	Default	Description
`group`	`default`	Specifies a unique group name that can segregate multicast traffic.
`minmumWireFormatVersion`	`0`	Specifies the minimum wire format version that is allowed.
`trace`	`false`	Causes all commands sent over the transport to be logged.
`useLocalHost`	`true`	When `true`, causes the local machine's name to resolve to `localhost`.
`datagramSize`	`4 * 1024`	Specifies the size of a datagram.
`timeToLive`	`-1`	Specifies the time to live of datagrams. Set greater than 1 to send packets beyond the local network. [a]
`loopBackMode`	`false`	Specifies whether loopback mode is used.

Property	Default	Description
`wireFormat`		The name of the wire format to use.
`wireFormat.*`		All the properties with this prefix are used to configure the wireFormat. See Table A.1, “Wire Format Options Supported by OpenWire Protocol” for more information.
`jms.*`		All the properties with this prefix are used to configure client connections to a broker. See Appendix B, *Client Connection Options* for more information.

[a] This won't work for IPv4 addresses without setting the property `java.net.preferIPv4Stack=true`.

CHAPTER 7. MQ TELEMETRY TRANSPORT(MQTT) PROTOCOL

Abstract

MQTT is a machine-to-machine (M2M)/"Internet of Things" connectivity protocol. It was designed as a lightweight publish/subscribe messaging transport.

URI SYNTAX

Example 7.1, "MQTT URI" shows the syntax for an MQTT connection.

Example 7.1. MQTT URI

```
mqtt://Host:[Port]?transportOptions
```

An NIO URI has the syntax shown in Example 7.2, "Syntax for MQTT+NIO Connection".

Example 7.2. Syntax for MQTT+NIO Connection

```
mqtt+nio://Host[:Port]?transportOptions
```

A secure MQTT URI has the syntax shown in Example 7.3, "Syntax for an MQTT SSL Connection" .

Example 7.3. Syntax for an MQTT SSL Connection

```
mqtt+ssl://Host[:Port]?transportOptions
```

A secure MQTT+NIO URI has the syntax shown in Example 7.4, "Syntax for a MQTT+NIO SSL Connection".

Example 7.4. Syntax for a MQTT+NIO SSL Connection

```
mqtt+nio+ssl://Host[:Port]?transportOptions
```

TRANSPORT OPTIONS

The MQTT protocol supports the following transport options:

Table 7.1. MQTT Transport Options

Property	Default	Description

Property	Default	Description
`transport.defaultKeepAlive`	`0`	Specifies, in milliseconds, the broker will allow a connection to be silent before it is closed. If a client specifies a keep-alive duration, this setting is ignored. This option is set in the `uri` attribute of a broker's `transportConnector` element.
`jms.*`		All the properties with this prefix are used to configure client connections to a broker. See Appendix B, *Client Connection Options* for more information.

SSL TRANSPORT OPTIONS

In addition to the options supported by the non-secure MQTT transports, the SSL transport also supports the options for configuring the `SSLServerSocket` created for the connection. These options are listed in Table 2.1, "SSL Transport Options".

WARNING

If you are planning to enable SSL/TLS security, you must ensure that you explicitly disable the SSLv3 protocol, in order to safeguard against the Poodle vulnerability (CVE-2014-3566). For more details, see Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x.

CONFIGURING BROKER SSL OPTIONS

On the broker side, you must specify an SSL transport option using the syntax `transport.`*`OptionName`*. For example, to enable an MQTT SSL port on a broker, you would add the following transport element:

```
<transportConnector name="mqttssl" uri="mqtt+ssl://localhost:61617?
transport.enabledProtocols=TLSv1,TLSv1.1,TLSv1.2" />
```

TIP

Remember, if you are specifying more than one option in the context of XML, you need to escape the ampersand, &, between options as `&`.

CONFIGURING CLIENT SSL OPTIONS

On the client side, you must specify an SSL transport option using the syntax **`socket.`*`OptionName`***. For example, to connect to a MQTT SSL port, you would use a URL like the following:

```
mqtt+ssl://localhost:61617?socket.enabledProtocols=TLSv1,TLSv1.1,TLSv1.2
```

CHAPTER 8. ADVANCED MESSAGE QUEUING PROTOCOL (AMQP)

Abstract

Oasis AMQP is an open standard application protocol for messaging. In contrast to JMS, AMQP standardizes the wire protocol, not the programming API, thus facilitating interoperability at the transport level.

URI SYNTAX

The URI syntax given here is valid only for specifying the endpoint in a transport connector element (broker endpoint).

A basic AMQP endpoint has the following URI syntax:

```
amqp://Host:[Port]?transportOptions
```

An AMQP endpoint with NIO support has the following syntax:

```
amqp+nio://Host:[Port]?transportOptions
```

A secure AMQP endpoint has the following URI syntax:

```
amqp+ssl://Host:[Port]?transportOptions
```

IDLETIMEOUT

Connections are subject to a configurable idle timeout threshold. The idle timeout is measured in milliseconds.

To configure the timeout threshold, use the following URI option:

```
transport.wireFormat.idleTimeout=10000
```

It can be used in the following way:

```
<transportConnector name="amqp"
uri="amqp://0.0.0.0:5672?transport.wireFormat.idleTimeout=10000&...."/>
```

Replace **`10000`** in the example with the number of milliseconds after which the connection will timeout due to inactivity.

SECURITY

The AMQP adapter is fully integrated with Apache ActiveMQ security. This means that the broker accepts SASL (Simple Authentication and Security Layer) authentication and any authorization settings configured on the broker will be applied.

SSL security can also be enabled for AMQP. To enable SSL, configure the broker's **`sslContext`**

element in the XML configuration and use the secure AMQP scheme, **amqp+ssl**, to define the AMQP URI in the broker's **transportConnector** element. For more details about SSL security, see the "Security Guide".

SSL TRANSPORT OPTIONS

In addition to the options supported by the non-secure AMQP transports, the SSL transport also supports the options for configuring the **SSLServerSocket** created for the connection. These options are listed in Table 2.1, “SSL Transport Options”.

WARNING

If you are planning to enable SSL/TLS security, you must ensure that you explicitly disable the SSLv3 protocol, in order to safeguard against the Poodle vulnerability (CVE-2014-3566). For more details, see Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x.

CONFIGURING BROKER SSL OPTIONS

On the broker side, you must specify an SSL transport option using the syntax **transport.*OptionName***. For example, to enable an AMQP SSL port on a broker, you would add the following transport element:

```
<transportConnector name="amqpssl" uri="amqp+ssl://localhost:61617?
transport.enabledProtocols=TLSv1,TLSv1.1,TLSv1.2" />
```

TIP

Remember, if you are specifying more than one option in the context of XML, you need to escape the ampersand, &, between options as **&**.

CONFIGURING CLIENT SSL OPTIONS

On the client side, you must specify an SSL transport option using the syntax **socket.*OptionName***. For example, to connect to an AMQP SSL port, you would use a URL like the following:

```
amqp+ssl://localhost:61617?socket.enabledProtocols=TLSv1,TLSv1.1,TLSv1.2
```

MAPPING FROM AMQP TO JMS

Because AMQP is not a JMS compliant protocol, the AMQP messages and their headers are defined in a different format from JMS. It is therefore necessary to map AMQP messages to JMS format. The mapping is implemented by a transformer and the transformer type can be selected by specifying the **transport.transformer** option on the AMQP endpoint.

For example, the following transport connector selects the **jms** transformer type:

```
<transportConnector name="amqp" uri="amqp://localhost:5672?
transport.transformer=jms"/>
```

AMQP-TO-JMS TRANSFORMERS

Table 8.1, "AMQP-to-JMS Transformer Types" lists the available transformer types and describes the basic characteristics of each mapping.

Table 8.1. AMQP-to-JMS Transformer Types

Transformer	Description
`native`	*(Default)* Wraps the bytes of the AMQP message into a JMS `BytesMessage`, and maps the AMQP message headers to JMS message headers.
`raw`	Wraps the bytes of the AMQP message into a JMS `BytesMessage`.
`jms`	Maps the body of the AMQP message to JMS body, and maps the AMQP message headers to JMS message headers.

HEADER MAPPING FOR ALL TRANSFORMERS

The JMS headers shown in the following table are always created, no matter which transformer type is selected.

AMQP Header	JMS Header
	`JMS_AMQP_NATIVE`
`message-format`	`JMS_AMQP_MESSAGE_FORMAT`

The `JMS_AMQP_NATIVE` header is a `boolean` type, which indicates whether or not the JMS message body is a direct copy of the raw AMQP message body. It is set to `true` for the `native` and `jms` transformer types and `false` for the `jms` transformer type.

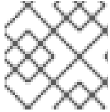
HEADER MAPPING FOR NATIVE OR JMS TRANSFORMERS

The JMS headers shown in the following table are mapped from AMQP headers, if the `native` or `jms` transformer type is selected.

AMQP Header	JMS Header
`header.durable`	`JMSDeliveryMode`

AMQP Header	JMS Header
`header.priority`	`JMSPriority`
`header.ttl`	`JMSExpiration`
`header.first-acquirer`	`JMS_AMQP_FirstAcquirer`
`header.deliveryCount`	`JMSXDeliveryCount`
`delivery-annotations.`*`name`*	`JMS_AMQP_DA_`*`name`*
`message-annotations.x-opt-jms-type`	`JMSType`
`message-annotations.x-opt-to-type`	Type of the `JMSDestination`
`message-annotations.x-opt-reply-type`	Type of the `JMSReplyTo`
`message-annotations.`*`name`*	`JMS_AMQP_MA_`*`name`*
`application-properties.JMSXGroupID`	`JMSXGroupID`
`application-properties.JMSXGroupSequence`	`JMSXGroupSequence`
`application-properties.JMSXUserID`	`JMSXUserID`
`application-properties.`*`name`*	*`name`*
`properties.message-id`	`JMSMessageID`
`properties.user-id`	`JMSXUserID`
`properties.to`	`JMSDestination`
`properties.subject`	`JMS_AMQP_Subject`
`properties.reply-to`	`JMSReplyTo`
`properties.correlation-id`	`JMSCorrelationID`
`properties.content-type`	`JMS_AMQP_ContentType`
`properties.content-encoding`	`JMS_AMQP_ContentEncoding`
`properties.creation-time`	`JMSTimestamp`

AMQP Header	JMS Header
`properties.group-sequence`	`JMSXGroupSequence`
`properties.reply-to-group-id`	`JMS_AMQP_ReplyToGroupID`
`footer.`*`name`*	`JMS_AMQP_FT_`*`name`*



NOTE

The `properties.user-id` property is decoded as a UTF-8 `String`.

DEFAULT HEADER VALUES

When mapping AMQP message properties to JMS header values, the following default JMS header values are used:

`JMS_AMQP_NATIVE`
: Defaults to `true`, if the transformer is `native` or `raw`, otherwise `false`.

`JMSDeliveryMode`
: Defaults to `javax.jms.Message.DEFAULT_DELIVERY_MODE`.

`JMSPriority`
: Defaults to `javax.jms.Message.DEFAULT_PRIORITY`.

`JMSExpiration`
: Defaults to `javax.jms.Message.DEFAULT_TIME_TO_LIVE`.

`JMSDestination` type
: Defaults to **queue**.

`JMSReplyTo` type
: Defaults to **queue**.

`JMSMessageID`
: Auto-generated, if not set.

PROPERTY TYPE MAPPING

AMQP property types are converted to Java types as shown in the following table:

AMQP Type	Java Type	Notes
`bool`	`Boolean`	

AMQP Type	Java Type	Notes
byte	Byte	
short	Short	
int	Integer	
long	Long	
ubyte	Byte or Short	Short is used, if value > Byte.MAX_VALUE
ushort	Short or Integer	Integer is used if value > Short.MAX_VALUE
uint	Integer or Long	Long is used, if value > Integer.MAX_VALUE
ulong	Long	
double	Double	
float	Float	
symbol	String	
binary	String	Hex encoding of the binary value

MESSAGE BODY MAPPING

When the jms transformer type is selected, the AMQP message body is mapped to a JMS message type, as shown in the following table:

AMQP Body Type	JMS Message Type
null	Message
Data	BytesMessage
AmqpSequence	StreamMessage
AmqpValue holding a null	Message

AMQP Body Type	JMS Message Type
`AmqpValue` holding a`String`	`TextMessage`
`AmqpValue` holding a`binary`	`BytesMessage`
`AmqpValue` holding a`list`	`StreamMessage`
`AmqpValue`	`ObjectMessage`

CHAPTER 9. VM TRANSPORT

Abstract

The VM transport allows clients to connect to each other inside the Java Virtual Machine (JVM) without the overhead of network communication.

The URI used to specify the VM transport comes in two flavors to provide maximum control over how the embedded broker is configured:

- simple—specifies the name of the embedded broker to which the client connects and allows for some basic broker configuration
- advanced—uses a broker URI to configure the embedded broker

9.1. SIMPLE VM URI SYNTAX

URI syntax

The simple VM URI is used in most situations. It allows you to specify the name of the embedded broker to which the client will connect. It also allows for some basic broker configuration.

Example 9.1, "Simple VM URI Syntax" shows the syntax for a simple VM URI.

Example 9.1. Simple VM URI Syntax

```
vm://BrokerName?TransportOptions
```

- *BrokerName* specifies the name of the embedded broker to which the client connects.
- *TransportOptions* specifies the configuration for the transport. They are specified in the form of a query list. Table 9.2, "VM Transport Options" lists the available options.

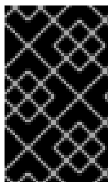
Broker options

In addition to the transport options listed in Table 9.2, "VM Transport Options", the simple VM URI can use the options described in Table 9.1, "VM Transport Broker Configuration Options" to configure the embedded broker.

Table 9.1. VM Transport Broker Configuration Options

Option	Description
`broker.useJmx`	Specifies if JMX is enabled. Default is`true`.
`broker.persistent`	Specifies if the broker uses persistent storage. Default is `true`.

Option	Description
`broker.populateJMSXUserID`	Specifies if the broker populates the `JMSXUserID` message property with the sender's authenticated username. Default is `false`.
`broker.useShutdownHook`	Specifies if the broker installs a shutdown hook, so that it can shut down properly when it receives a JVM kill. Default is `true`.
`broker.brokerName`	Specifies the broker name. Default is `localhost`.
`broker.deleteAllMessagesOnStartup`	Specifies if all the messages in the persistent store are deleted when the broker starts up. Default is `false`.
`broker.enableStatistics`	Specifies if statistics gathering is enabled in the broker. Default is `true`.
`brokerConfig`	Specifies an external broker configuration file. For example, to pick up the broker configuration file, `activemq.xml`, you would set `brokerConfig` as follows: `brokerConfig=xbean:activemq.xml`.



IMPORTANT

The broker configuration options specified on the VM URI are only meaningful if the client is responsible for instantiating the embedded broker. If the embedded broker is already started, the transport will ignore the broker configuration properties.

Example

Example 9.2, "Basic VM URI" shows a basic VM URI that connects to an embedded broker named `broker1`.

Example 9.2. Basic VM URI

```
vm://broker1
```

Example 9.3, "Simple URI with broker options" creates and connects to an embedded broker that uses a non-persistent message store.

Example 9.3. Simple URI with broker options

```
vm://broker1?broker.persistent=false
```

9.2. ADVANCED VM URI SYNTAX

URI syntax

The advanced VM URI provides you full control over how the embedded broker is configured. It uses a broker configuration URI similar to the one used by the administration tool to configure the embedded broker.

Example 9.4, "Advanced VM URI Syntax" shows the syntax for an advanced VM URI.

Example 9.4. Advanced VM URI Syntax

```
vm://(BrokerConfigURI)?TransportOptions
```

- *BrokerConfigURI* is a broker configuration URI.
- *TransportOptions* specifies the configuration for the transport. They are specified in the form of a query list. Table 9.2, "VM Transport Options" lists the available options.

Transport options

Table 9.2, "VM Transport Options" shows options for configuring the VM transport.

Table 9.2. VM Transport Options

Option	Description
`marshal`	If `true`, forces each command sent over the transport to be marshalled and unmarshalled using the specified wire format. Default is `false`.
`wireFormat`	The name of the wire format to use.
`wireFormat.*`	All options with this prefix are used to configure the wire format. See Table A.1, "Wire Format Options Supported by OpenWire Protocol" for more information.
`jms.*`	All the properties with this prefix are used to configure client connections to a broker. See Appendix B, *Client Connection Options* for more information.
`create`	Specifies if the VM transport will create an embedded broker if one does not exist. The default is `true`.

Option	Description
`waitForStart`	Specifies the time, in milliseconds, the VM transport will wait for an embedded broker to start before creating one. The default is `-1` which specifies that the transport will not wait.

Example

Example 9.5, "Advanced VM URI" creates and connects to an embedded broker configured using a broker configuration URI.

Example 9.5. Advanced VM URI

```
vm:(broker:(tcp://localhost:6000)?persistent=false)?marshal=false
```

CHAPTER 10. DISCOVERING BROKERS

Abstract

One of the main strengths of Red Hat JBoss A-MQ is that brokers can be located dynamically through out your infrastructure. In order for clients and other brokers to be able to interact with a broker, they need some way of discovering that the broker exists. JBoss A-MQ does this using a combination of discovery agents and special URI schemes. In order for location transparency to work, the members of a messaging application need a way for discovering each other. In Red Hat JBoss A-MQ this is accomplished using two pieces: *discovery agents*, components that advertise the brokers available to other members of a messaging application; and *discovery URI*, a URI that looks up all of the discoverable brokers and presents them as a list of actual URIs for use by the client or network connector.

10.1. DISCOVERY AGENTS

Abstract

A discovery agent is a mechanism that advertises available brokers to clients and other brokers.

10.1.1. Introduction to Discovery Agents

What is a discovery agent?

A discovery agent is a mechanism that advertises available brokers to clients and other brokers. When a client, or broker, using a discovery URI starts up it will look for any brokers that are available using the specified discovery agent. The clients will update their lists periodically using the same mechanism.

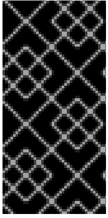
Discovery mechanisms

How a discovery agent learns about the available brokers varies between agents. Some agents use a static list, some use a third party registry, and some rely on the brokers to provide the information. For discovery agents that rely on the brokers for information, it is necessary to enable the discovery agent in the message broker configuration. For example, to enable the multicast discovery agent on an Openwire endpoint, you edit the relevant **`transportConnector`** element as shown in Example 10.1, “Enabling a Discovery Agent on a Broker”.

Example 10.1. Enabling a Discovery Agent on a Broker

```
<transportConnectors>
  <transportConnector name="openwire"
    uri="tcp://localhost:61716"
    discoveryUri="multicast://default" />
</transportConnectors>
```

Where the **`discoveryUri`** attribute on the **`transportConnector`** element is initialized to **`multicast://default`**.



IMPORTANT

If a broker uses multiple transport connectors, you need to configure each transport connector to use a discovery agent individually. This means that different connectors can use different discovery mechanisms or that one or more of the connectors can be undiscoverable.

Discovery agent types

Red Hat JBoss A-MQ currently supports the following discovery agents:

- Fuse Fabric Discovery Agent
- Static Discovery Agent
- Multicast Discovery Agent
- Zeroconf Discovery Agent

10.1.2. Fuse Fabric Discovery Agent

Abstract

The Fuse Fabric discovery agent uses Fuse Fabric to discovery brokers that are deployed into a fabric.

Overview

The *Fuse Fabric discovery agent*uses Fuse Fabric to discover the brokers in a specified group. The discovery agent requires that all of the discoverable brokers be deployed into a single fabric. When the client attempts to connect to a broker the agent looks up all of the available brokers in the fabric's registry and returns the ones in the specified group.

URI

The Fuse Fabric discovery agent URI conforms to the syntax in Example 10.2, “Fuse Fabric Discovery Agent URI Format”.

Example 10.2. Fuse Fabric Discovery Agent URI Format

```
fabric://GID
```

Where *GID* is the ID of the broker group from which the client discovers the available brokers.

Configuring a broker

The Fuse Fabric discovery agent requires that the discoverable brokers are deployed into a single fabric.

The best way to deploy brokers into a fabric is using the management console. For information on using the management console see "Management Console User Guide".

You can also use the console to deploy brokers into a fabric. See chapter "Fabric Console Commands" in "Console Reference".

Configuring a client

To use the agent a client must be configured to connect to a broker using a discovery protocol that uses a Fuse Fabric agent URI as shown in Example 10.3, “Client Connection URL using Fuse Fabric Discovery”.

Example 10.3. Client Connection URL using Fuse Fabric Discovery

```
discovery:(fabric://nwBrokers)
```

A client using the URL in Example 10.3, “Client Connection URL using Fuse Fabric Discovery” will discover all the brokers in the **nwBrokers** broker group and generate a list of brokers to which it can connect.

10.1.3. Static Discovery Agent

Abstract

The static discovery agent uses an explicit list of broker URLs to specify the available brokers.

Overview

The *static discovery agent* does not truly discover the available brokers. It uses an explicit list of broker URLs to specify the available brokers. Brokers are not involved with the static discovery agent. The client only knows about the brokers that are hard coded into the agent's URI.

Using the agent

The static discovery agent is a client-side only agent. It does not require any configuration on the brokers that will be discovered.

To use the agent, you simply configure the client to connect to a broker using a discovery protocol that uses a static agent URI.

The static discovery agent URI conforms to the syntax in Example 10.4, “Static Discovery Agent URI Format”.

Example 10.4. Static Discovery Agent URI Format

```
static://(URI1,URI2,URI3,...)
```

Example

Example 10.5, “Discovery URI using the Static Discovery Agent” shows a discovery URI that configures a client to use the static discovery agent to connect to one member of a broker pair.

Example 10.5. Discovery URI using the Static Discovery Agent

```
discovery:(static://(tcp://localhost:61716,tcp://localhost:61816))
```

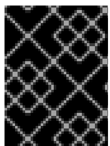
10.1.4. Multicast Discovery Agent

Abstract

The multicast discovery agent uses the IP multicast protocol to find any message brokers currently active on the local network.

Overview

The *multicast discovery agent* uses the IP multicast protocol to find any message brokers currently active on the local network. The agent requires that *each* broker you want to advertise is configured to use the multicast agent to publish its details to a multicast group. Clients using the multicast agent as part of the discovery URI they use for connecting to a broker will use the agent to receive the list of available brokers advertising in the specified multicast group.



IMPORTANT

Your local network (LAN) must be configured appropriately for the IP/multicast protocol to work.

URI

The multicast discovery agent URI conforms to the syntax in Example 10.6, “Multicast Discovery Agent URI Format”.

Example 10.6. Multicast Discovery Agent URI Format

```
multicast://GroupID
```

Where *GroupID* is an alphanumeric identifier. All participants in the same discovery group must use the same *GroupID*.

Configuring a broker

For a broker to be discoverable using the multicast discovery agent, you must enable the discovery agent in the broker's configuration. To enable the multicast discovery agent you set the **`transportConnector`** element's **`discoveryUri`** attribute to a mulitcast discovery agent URI as shown in Example 10.7, “Enabling a Multicast Discovery Agent on a Broker” .

Example 10.7. Enabling a Multicast Discovery Agent on a Broker

```
<transportConnectors>
  <transportConnector name="openwire"
    uri="tcp://localhost:61716"
```

```
    discoveryUri="multicast://default" />
</transportConnectors>
```

The broker configured in Example 10.7, “Enabling a Multicast Discovery Agent on a Broker” is discoverable as part of the multicast group `default`.

Configuring a client

To use the multicast agent a client must be configured to connect to a broker using a discovery URI that uses a multicast agent URI as shown in Example 10.8, “Client Connection URL using Multicast Discovery”.

Example 10.8. Client Connection URL using Multicast Discovery

```
discovery:(multicast://default)
```

A client using the URI in Example 10.8, “Client Connection URL using Multicast Discovery” will discover all the brokers advertised in the `default` multicast group and generate a list of brokers to which it can connect.

10.1.5. Zeroconf Discovery Agent

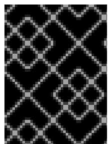
Abstract

The zeroconf discovery agent uses an open source implementation of Apple's Bonjour networking technology to find any brokers currently active on the local network.

Overview

The *zeroconf discovery agent* is derived from Apple’s Bonjour Networking technology, which defines the zeroconf protocol as a mechanism for discovering services on a network. Red Hat JBoss A-MQ bases its implementation of the zeroconf discovery agent on JmDSN, which is a service discovery protocol that is layered over IP/multicast and is compatible with Apple Bonjour.

The agent requires that *each* broker you want to advertise is configured to use a multicast discovery agent to publish its details to a multicast group. Clients using the zeroconf agent as part of the discovery URI they use for connecting to a broker will use the agent to receive the list of available brokers advertising in the specified multicast group.



IMPORTANT

Your local network (LAN) must be configured to use IP/multicast for the zeroconf agent to work.

URI

The zeroconf discovery agent URI conforms to the syntax in Example 10.9, “Zeroconf Discovery Agent URI Format”.

Example 10.9. Zeroconf Discovery Agent URI Format

```
zeroconf://GroupID
```

Where the *GroupID* is an alphanumeric identifier. All participants in the same discovery group must use the same *GroupID*.

Configuring a broker

For a broker to be discoverable using the zeroconf discovery agent, you must enable a multicast discovery agent in the broker's configuration. To enable the multicast discovery agent you set the **transportConnector** element's **discoveryUri** attribute to a mulitcast discovery agent URI as shown in Example 10.10, "Enabling a Multicast Discovery Agent on a Broker" .

Example 10.10. Enabling a Multicast Discovery Agent on a Broker

```
<transportConnectors>
  <transportConnector name="openwire"
    uri="tcp://localhost:61716"
    discoveryUri="multicast://NEGroup" />
</transportConnectors>
```

The broker configured in Example 10.10, "Enabling a Multicast Discovery Agent on a Broker" is discoverable as part of the multicast group **NEGroup**.

Configuring a client

To use the agent a client must be configured to connect to a broker using a discovery protocol that uses a zeroconf agent URI as shown in Example 10.11, "Client Connection URL using Zeroconf Discovery".

Example 10.11. Client Connection URL using Zeroconf Discovery

```
discovery:(zeroconf://NEGroup)
```

A client using the URL in Example 10.11, "Client Connection URL using Zeroconf Discovery" will discover all the brokers advertised in the **NEGroup** multicast group and generate a list of brokers to which it can connect.

10.2. DYNAMIC DISCOVERY PROTOCOL

Abstract

The dynamic discovery protocol combines reconnect logic with a discovery agent to dynamically create a list of brokers to which the client can connect.

Overview

The *dynamic discovery protocol* combines reconnect logic with a discovery agent to dynamically create a list of brokers to which the client can connect. The discovery protocol invokes a discovery agent in order to build up a list of broker URIs. The protocol then randomly chooses a URI from the list and attempts to establish a connection to it. If it does not succeed, or if the connection subsequently fails, a new connection is established to one of the other URIs in the list.

URI syntax

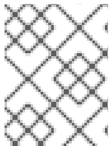
Example 10.12, “Dynamic Discovery URI” shows the syntax for a discovery URI.

Example 10.12. Dynamic Discovery URI

```
discovery:(DiscoveryAgentUri)?Options
```

DiscoveryAgentUri is URI for the discovery agent used to build up the list of available brokers. Discovery agents are described in Section 10.1, “Discovery Agents”.

The options, ***?Options***, are specified in the form of a query list. The discovery options are described in Table 10.1, “Dynamic Discovery Protocol Options” . You can also inject transport options as described in the section called “Setting options on the discovered transports” .



NOTE

If no options are required, you can drop the parentheses from the URI. The resulting URI would take the form discovery:*DiscoveryAgentUri*

Transport options

The discovery protocol supports the options described in Table 10.1, “Dynamic Discovery Protocol Options”.

Table 10.1. Dynamic Discovery Protocol Options

Option	Default	Description
`initialReconnectDelay`	`10`	Specifies, in milliseconds, how long to wait before the first reconnect attempt.
`maxReconnectDelay`	`30000`	Specifies, in milliseconds, the maximum amount of time to wait between reconnect attempts.
`useExponentialBackOff`	`true`	Specifies if an exponential back-off is used between reconnect attempts.
`backOffMultiplier`	`2`	Specifies the exponent used in the exponential back-off algorithm.

Option	Default	Description
`maxReconnectAttempts`	`0`	Specifies the maximum number of reconnect attempts before an error is sent back to the client. `0` specifies unlimited attempts.

Sample URI

Example 10.13, "Discovery Protocol URI" shows a discovery URI that uses a multicast discovery agent.

Example 10.13. Discovery Protocol URI

```
discovery:(multicast://default)?initialReconnectDelay=100
```

Setting options on the discovered transports

The list of transport options, *Options*, in the discovery URI can also be used to set options on the *discovered* transports. If you set an option *not* listed in the section called "Setting options on the discovered transports", the URI parser attempts to inject the option setting into every one of the discovered endpoints.

Example 10.14, "Injecting Transport Options into a Discovered Transport" shows a discovery URI that sets the TCP **connectionTimeout** option to 10 seconds.

Example 10.14. Injecting Transport Options into a Discovered Transport

```
discovery:(multicast://default)?connectionTimeout=10000
```

The 10 second timeout setting is injected into every discovered TCP endpoint.

10.3. FANOUT PROTOCOL

Abstract

The fanout protocol allows clients to connect to multiple brokers at once and broadcast messages to consumers connected to all of the brokers at once.

Overview

The *fanout protocol* enables a producer to auto-discover broker endpoints and broadcast topic messages to *all* of the discovered brokers. The fanout protocol gives producers a convenient mechanism for broadcasting messages to multiple brokers that are not part of a network of brokers.

The fanout protocol relies on a discovery agent to build up the list of broker URIs to which it connects.

URI syntax

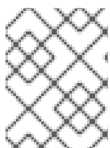
Example 10.15, “Fanout URI Syntax” shows the syntax for a fanout URI.

Example 10.15. Fanout URI Syntax

```
fanout://(DiscoveryAgentUri)?Options
```

DiscoveryAgentUri is URI for the discovery agent used to build up the list of available brokers. Discovery agents are described in Section 10.1, “Discovery Agents”.

The options, ***?Options***, are specified in the form of a query list. The discovery options are described in Table 10.2, “Fanout Protocol Options”. You can also inject transport options as described in the section called “Setting options on the discovered transports”.



NOTE

If no options are required, you can drop the parentheses from the URI. The resulting URI would take the form fanout://*DiscoveryAgentUri*

Transport options

The fanout protocol supports the transport options described in Table 10.2, “Fanout Protocol Options”.

Table 10.2. Fanout Protocol Options

Option Name	Default	Description
`initialReconnectDelay`	`10`	Specifies, in milliseconds, how long the transport will wait before the first reconnect attempt.
`maxReconnectDelay`	`30000`	Specifies, in milliseconds, the maximum amount of time to wait between reconnect attempts.
`useExponentialBackOff`	`true`	Specifies if an exponential back-off is used between reconnect attempts.
`backOffMultiplier`	`2`	Specifies the exponent used in the exponential back-off algorithm.
`maxReconnectAttempts`	`0`	Specifies the maximum number of reconnect attempts before an error is sent back to the client. `0` specifies unlimited attempts.

Option Name	Default	Description
`fanOutQueues`	`false`	Specifies whether queue messages are replicated to every connected broker. For more information see the section called "Applying fanout to queue messages".
`minAckCount`	`2`	Specifies the minimum number of brokers to which the client must connect before it sends out messages. For more informaiton see the section called "Minimum number of brokers".

Sample URI

Example 10.16, "Fanout Protocol URI" shows a discovery URI that uses a multicast discovery agent.

Example 10.16. Fanout Protocol URI

```
fanout://(multicast://default)?initialReconnectDelay=100
```

Applying fanout to queue messages

The fanout protocol replicates topic messages by sending each topic message to all of the connected brokers. By default, however, the fanout protocol does *not* replicate queue messages.

For queue messages, the fanout protocol picks one of the brokers at random and sends all of the queue messages to that broker. This is a sensible default, because under normal circumstances, you would not want to create more than one copy of a queue message.

It is possible to change the default behavior by setting the `fanOutQueues` option to `true`. This configures the protocol so that it also replicates queue messages.

Minimum number of brokers

By default, the fanout protocol does not start sending messages until the producer has connected to a *minimum of two brokers*. You can customize this minimum value using the `minAckCount` option.

Setting minimum number of brokers equal to the expected number of discovered brokers ensures that all of the available brokers start receiving messages at the same time. This ensures that no messages are missed if a broker starts up after the producer has started sending messages.

Using fanout with a broker network

You have to be careful when using the fanout protocol with brokers that are joined in a network of brokers.

The combination of the fanout protocol's broadcasting behavior and the nature of how messages are

propagated through a network of brokers makes it likely that consumers will receive duplicate messages. If, for example, you joined four brokers into a network of brokers and connected a consumer listening for messages on topic `hello.jason` to broker A and connected a producer to broker B to send messages to topic `hello.jason`, the consumer would get one copy of the messages. If, on the other hand, the producer connects to the network using the fanout protocol, the producer will connect to every broker in the network simultaneously and start sending messages. Each of the four brokers will receive a copy of every message and deliver its copy to the consumer. So, for each message, the consumer will get four copies.

CHAPTER 11. PEER PROTOCOL

Abstract

The peer protocol uses embedded brokers to enable messaging clients to communicate with each other directly.

URI SYNTAX

A **peer** URI must conform to the following syntax:

```
peer://PeerGroup/BrokerName?BrokerOptions
```

Where the group name, *PeerGroup*, identifies the set of peers that can communicate with each other. That is, a given peer can connect only to the set of peers that specify the *same PeerGroup* name in their URLs. The *BrokerName* specifies the broker name for the embedded broker. The broker options, *BrokerOptions*, are specified in the form of a query list (for example, **?persistent=true**).

BROKER OPTIONS

The peer URL supports the broker options described in Table 11.1, "Broker Options".

Table 11.1. Broker Options

Option	Description
useJmx	If **true**, enables JMX. Default is **true**.
persistent	If **true**, the broker uses persistent storage. Default is **true**.
populateJMSXUserID	If **true**, the broker populates the **JMSXUserID** message property with the sender's authenticated username. Default is **false**.
useShutdownHook	If **true**, the broker installs a shutdown hook, so that it can shut down properly when it receives a JVM kill. Default is **true**.
brokerName	Specifies the broker name. Default is **localhost**.
deleteAllMessagesOnStartup	If **true**, deletes all the messages in the persistent store as the broker starts up. Default is **false**.
enableStatistics	If **true**, enables statistics gathering in the broker. Default is **true**.

DEPENDENCIES

The peer protocol uses multicast discovery to locate active peers on the network. In order for this to work, you must ensure that the IP multicast protocol is enabled on your operating system.

APPENDIX A. OPENWIRE FORMAT OPTIONS

FORMAT OPTIONS TABLE

Table A.1, "Wire Format Options Supported by OpenWire Protocol" shows the wire format options supported by the OpenWire protocol.

Table A.1. Wire Format Options Supported by OpenWire Protocol

Option	Default	Description	Negotiation Policy
`wireformat.stackTraceEnabled`	`true`	Specifies if the stack trace of an exception occurring on the broker is sent to the client.	`false` if either side is `false`.
`wireformat.tcpNoDelayEnabled`	`false`	Specifies if a hint is provided to the peer that TCP `nodelay` should be enabled on the communications socket.	`false` if either side is `false`.
`wireformat.cacheEnabled`	`true`	Specifies that commonly repeated values are cached so that less marshalling occurs.	`false` if either side is `false`.
`wireformat.cacheSize`	`1024`	Specifies the maximum number of values to cache.	Use the smaller of the two values.
`wireformat.tightEncodingEnabled`	`true`	Specifies if wire size be optimized over CPU usage.	`false` if either side is `false`.
`wireformat.prefixPacketSize`	`true`	Specifies if the size of the packet be prefixed before each packet is marshalled.	`true` if both sides are `true`.
`wireformat.maxInactivityDuration`	`30000`	Specifies the maximum inactivity duration, in milliseconds, before the broker considers the connection dead and kills it. `<= 0` disables inactivity monitoring.	Use the smaller of the two values.

Option	Default	Description	Negotiation Policy
`wireformat.maxInactivityDurationInitalDelay`	`10000`	Specifies the initial delay in starting inactivity checks.	

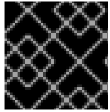
APPENDIX B. CLIENT CONNECTION OPTIONS

OVERVIEW

When creating a connection to a broker, a client can use the connection URI to configure a number of the connection properties. The properties are added to the connection URI as matrix parameters on the URI as shown in Example B.1, “Client Connection Options Syntax”.

Example B.1. Client Connection Options Syntax

```
URI?jms.option?jms.option...
```



IMPORTANT

All of the client connection options are prefixed with `jms.`

OPTIONS

Table B.1, “Client Connection Options” shows the client connection options.

Table B.1. Client Connection Options

Option	Default	Description
`alwaysSessionAsync`	`true`	Specifies if a separate thread is used for dispatching messages for each `Session` in the `Connection`. However, a separate thread is always used if there is more than one session, or the session isn't in auto acknowledge or dups ok mode.
`clientID`		Specifies the JMS `clientID` to use for the connection.
`closeTimeout`	`15000`	Specifies the timeout, in milliseconds, before a connection close is considered complete. Normally a `close()` on a connection waits for confirmation from the broker; this allows that operation to timeout and save the client from hanging if there is no broker.

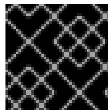
Option	Default	Description
`copyMessageOnSend`	`true`	Specifies if a JMS message should be copied to a new JMS `Message` object as part of the `send()` method in JMS. This is enabled by default to be compliant with the JMS specification. Disabling this can give you a performance, however you must not mutate JMS messages after they are sent.
`disableTimeStampsByDefault`	`false`	Specifies whether or not timestamps on messages should be disabled or not. Disabling them it adds a small performance boost.
`dispatchAsync`	`false`	Specifies if the broker dispatches messages to the consumer asynchronously.
`nestedMapAndListEnabled`	`true`	Enables/disables whether or not structured message properties and `MapMessages` are supported so that `Message` properties and `MapMessage` entries can contain nested `Map` and `List` objects.
`objectMessageSerializationDefered`	`false`	Specifies that the serialization of objects when they are set on an `ObjectMessage` is deferred. The object may subsequently get serialized if the message needs to be sent over a socket or stored to disk.
`optimizeAcknowledge`	`false`	Specifies if messages are acknowledged in batches rather than individually. Enabling this could cause some issues with auto-acknowledgement on reconnection.
`optimizeAcknowledgeTimeOut`	`300`	Specifies the maximum time, in milliseconds, between batch acknowledgements when `optimizeAcknowledge` is enabled.

Option	Default	Description
`optimizedMessageDispatch`	`true`	Specifies if a larger prefetch limit is used for durable topic subscribers.
`useAsyncSend`	`false`	Specifies in sends are performed asynchronously. Asynchronous sends provide a significant performance boost. The tradeoff is that the `send()` method will return immediately whether the message has been sent or not which could lead to message loss.
`useCompression`	`false`	Specifies if message bodies are compressed.
`useRetroactiveConsumer`	`false`	Specifies whether or not retroactive consumers are enabled. Retroactive consumers allow non-durable topic subscribers to receive messages that were published before the non-durable subscriber started.
`warnAboutUnstartedConnectionTimeout`	`500`	Specifies the timeout, in milliseconds, from connection creation to when a warning is generated if the connection is not properly started and a message is received by a consumer. -1 disables the warnings.
`auditDepth`	`2048`	Specifies the size of the message window that will be audited for duplicates and out of order messages.
`auditMaximumProducerNumber`	`64`	Specifies the maximum number of producers that will be audited.
`alwaysSyncSend`	`false`	Specifies if a message producer will always use synchronous sends when sending a message.
`blobTransferPolicy.*`		Used to configure how the client handles blob messages. See the section called “Blob handling”.

Option	Default	Description
prefetchPolicy.*		Used to configure the prefect limits. See the section called “Prefetch limits”.
redeliveryPolicy.*		Used to configure the redelivery policy. See the section called “Redelivery policy”.

BLOB HANDLING

Blob messages allow the broker to use an out of band transport to pass large files between clients. Table B.2, “Blob Message Properties” describes the connection URI options used to configure how a client handles blob messages.



IMPORTANT

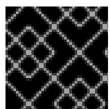
All of the prefetch options are prefixed with `jms.blobTransferPolicy.`

Table B.2. Blob Message Properties

Option	Description
bufferSize	Specifies the size of the buffer used when uploading or downloading blobs.
uploadUrl	Specifies the URL to which blob messages are stored for transfer. This value overrides the upload URI configured by the broker.

PREFETCH LIMITS

The prefetch limits control how many messages can be dispatched to a consumer and waiting to be acknowledged. Table B.3, “Connection URI Prefect Limit Options” describes the options used to configure the prefetch limits of consumers using a connection.



IMPORTANT

All of the prefetch options are prefixed with `jms.prefetchPolicy.`

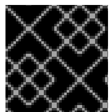
Table B.3. Connection URI Prefect Limit Options

Option	Description
queuePrefetch	Specifies the prefect limit for all consumers using queues.

Option	Description
`queueBrowserPrefetch`	Specifies the prefect limit for all queue browsers.
`topicPrefetch`	Specifies the prefect limit for non-durable topic consumers.
`durableTopicPrefetch`	Specifies the prefect limit for durable topic consumers.
`all`	Specifies the prefect limit for all types of message consumers.

REDELIVERY POLICY

The redelivery policy controls the redelivery of messages in the event of connectivity issues. Table B.4, “Redelivery Policy Options” describes the options used to configure the redelivery policy of consumers using a connection.



IMPORTANT

All of the prefetch options are prefixed with `jms.redeliveryPolicy.`

Table B.4. Redelivery Policy Options

Option	Default	Description
`collisionAvoidanceFactor`	`0.15`	Specifies the percentage of range of collision avoidance.
`maximumRedeliveries`	`6`	Specifies the maximum number of times a message will be redelivered before it is considered a poisoned pill and returned to the broker so it can go to a dead letter queue. `-1` specifies an infinite number of redeliveries.
`maximumRedeliveryDelay`	`-1`	Specifies the maximum delivery delay that will be applied if the `useExponentialBackOff` option is set. `-1` specifies that no maximum be applied.
`initialRedeliveryDelay`	`1000`	Specifies the initial redelivery delay in milliseconds.

Option	Default	Description
`redeliveryDelay`	`1000`	Specifies the delivery delay, in milliseconds, if `initialRedeliveryDelay` is 0.
`useCollisionAvoidance`	`false`	Specifies if the redelivery policy uses collision avoidance.
`useExponentialBackOff`	`false`	Specifies if the redelivery time out should be increased exponentially.
`backOffMultiplier`	`5`	Specifies the back-off multiplier.

APPENDIX C. SERVER OPTIONS

SERVER OPTIONS TABLE

Table C.1, "ActiveMQ TransportConnector Server Options" shows the options to change the behavior of TransportConnector in ActiveMQ broker configuration on the server.

Table C.1. ActiveMQ TransportConnector Server Options

Option	Default	Description
`uri`	`null`	Specifies the bind address for the transport connector.
`name`	`null`	Specifies the name of the transport connector instance.
`discoveryURI`	`null`	Specifies the multicast discovery address for client connection to find the broker.
`enableStatusMonitor`	`false`	Monitors the state of the connections and determines whether the connections are blocked.
`updateClusterClients`	`false`	Updates the client connections about the changes in the broker status.
`rebalanceClusterClients`	`false`	Rebalances clients automatically across the cluster on changes to the topology.
`updateClusterClientsOnRemove`	`false`	Updates clients if a broker is removed from the cluster.
`updateClusterFilter`	`null`	A comma separated list of regular expressions that specifies the list of brokers included for client updates.
`allowLinkStealing`	`false`	Specifies that if the last two or more connections have the same id, for example clientID for JMS then the last connection is deemed as a valid connection and the older connections are closed by the broker. This property is enable for default for MQTT transport.

Example C.1, “Server Options Configuration” shows the configuration of **enableStatusMonitor** server option.

Example C.1. Server Options Configuration

```
<broker >
...
  <transportConnectors >
  <transportConnector name="openwire" uri="tcp://0.0.0.0:61616"
enableStatusMonitor="true" >
  <transportConnectors >
...
  <broker >
```

INDEX

C

connection socket, Setting transport options, Setting transport options, Setting transport options

D

discovery agent

- Fuse Fabric, Fuse Fabric Discovery Agent
- multicast, Multicast Discovery Agent
- static, Static Discovery Agent
- zeroconf, Zeroconf Discovery Agent

discovery protocol

- backOffMultiplier, Transport options
- initialReconnectDelay, Transport options
- maxReconnectAttempts, Transport options
- maxReconnectDelay, Transport options
- URI, URI syntax
- useExponentialBackOff, Transport options

discovery URI, URI syntax

discovery:, URI syntax

discoveryUri, Configuring a broker, Configuring a broker

E

embedded broker

- brokerName, Broker options

deleteAllMessagesOnStartup, Broker options

enableStatistics, Broker options

persistent, Broker options

populateJMSXUserID, Broker options

useJmx, Broker options

useShutdownHook, Broker options

F

fabric://, URI

fanout protocol
backOffMultiplier, Transport options

fanOutQueues, Transport options

initialReconnectDelay, Transport options

maxReconnectAttempts, Transport options

maxReconnectDelay, Transport options

minAckCount, Transport options

URI, URI syntax

useExponentialBackOff, Transport options

fanout URI, URI syntax

fanout://, URI syntax

Fuse Fabric discovery agent
URI, URI

H

HTTP
URI, URI syntax

HTTPS
URI, URI syntax

L

listener socket, Setting transport options, Setting transport options, Setting transport options

M

MQTT, URI syntax

MQTT+NIO, URI syntax

MQTT+SSL, URI syntax

Multicast, URI syntax

multicast discovery agent

broker configuration, Configuring a broker

URI, URI

multicast://, URI

N

NIO

URI, URI syntax

NIO+SSL

URI, URI syntax

O

OpenWire

HTTP, URI syntax

HTTPS, URI syntax

NIO, URI syntax

NIO+SSL, URI syntax

SSL, URI syntax

TCP, URI syntax

transport options, Setting transport options, Setting transport options, Setting transport options

UDP, URI syntax

S

SSL

URI, URI syntax

static discovery agent

URI, Using the agent

static://, Using the agent

STOMP, URI syntax

STOMP+NIO, URI syntax

STOMP+SSL, URI syntax

T

TCP

URI, URI syntax

transport connector, Setting transport options, Setting transport options, Setting transport options

transportConnector

discoveryUri, Configuring a broker, Configuring a broker

U

UDP

URI, URI syntax

URI

HTTP, URI syntax

HTTPS, URI syntax

MQTT, URI syntax

MQTT+NIO, URI syntax

MQTT+SSL, URI syntax

Multicast, URI syntax

NIO, URI syntax

NIO+SSL, URI syntax

SSL, URI syntax

STOMP, URI syntax

STOMP+NIO, URI syntax

STOMP+SSL, URI syntax

TCP, URI syntax

UDP, URI syntax

V

VM

advanced URI, URI syntax

broker configuration, Broker options

broker name, URI syntax

brokerConfig, Broker options

create, Transport options

marshal, Transport options

simple URI, Simple VM URI Syntax

waitForStart, Transport options

wireFormat, Transport options

VM URI

advanced, URI syntax

simple, Simple VM URI Syntax

Z

zeroconf discovery agent

broker configuration, Configuring a broker

URI, URI

zeroconf://, URI