Red Hat JBoss A-MQ 6.0

Connection Reference

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

Last Updated: 2017-10-13

Red Hat JBoss A-MQ 6.0 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

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

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.
`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.

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
`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. 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

8.1. 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 8.1, “Simple VM URI Syntax” shows the syntax for a simple VM URI.

Example 8.1. 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 8.2, “VM Transport Options” lists the available options.

Broker options

In addition to the transport options listed in Table 8.2, “VM Transport Options”, the simple VM URI can use the options described in Table 8.1, “VM Transport Broker Configuration Options” to configure the embedded broker.

Table 8.1. 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 8.2, "Basic VM URI" shows a basic VM URI that connects to an embedded broker named **broker1**.

Example 8.2. Basic VM URI

```
vm://broker1
```

Example 8.3, "Simple URI with broker options" creates and connects to an embedded broker that uses a non-persistent message store.

Example 8.3. Simple URI with broker options

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

8.2. 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 8.4, “Advanced VM URI Syntax” shows the syntax for an advanced VM URI.

Example 8.4. 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 8.2, “VM Transport Options” lists the available options.

Transport options

Table 8.2, “VM Transport Options” shows options for configuring the VM transport.

Table 8.2. 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**.
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 8.5, “Advanced VM URI” creates and connects to an embedded broker configured using a broker configuration URI.

Example 8.5. Advanced VM URI

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

CHAPTER 9. 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.

URI SYNTAX

Example 9.1, “Dynamic Discovery URI” shows the syntax for a discovery URI.

Example 9.1. 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 Chapter 11, *Discovery Agents*.

The options, ***?Options***, are specified in the form of a query list. The discovery options are described in Table 9.1, “Dynamic Discovery Protocol Options”. You can also inject transport options into the discovered transports by adding their properties to the list.

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 9.1, “Dynamic Discovery Protocol Options”.

Table 9.1. 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.

Option	Default	Description
`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.

CHAPTER 10. 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.

URI SYNTAX

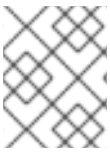
Example 10.1, “Fanout URI Syntax” shows the syntax for a fanout URI.

Example 10.1. Fanout URI Syntax

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

DiscoveryAgentUri is URI for the discovery agent used to build up the list of available brokers. The available discovery agents are listed in Chapter 11, *Discovery Agents*.

The options, **`?Options`**, are specified in the form of a query list. The discovery options are described in Table 10.1, “Fanout Protocol Options”. You can also inject transport options into the discovered transports by adding their properties to the list.



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 10.1, “Fanout Protocol Options”.

Table 10.1. 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.

Option Name	Default	Description
`maxReconnectAttempts`	`0`	Specifies the maximum number of reconnect attempts before an error is sent back to the client. `0` specifies unlimited attempts.
`fanOutQueues`	`false`	Specifies whether queue messages are replicated to every connected broker.
`minAckCount`	`2`	Specifies the minimum number of brokers to which the client must connect before it sends out messages.

CHAPTER 11. DISCOVERY AGENTS

Abstract

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

FABRIC AGENT

The Fuse Fabric discovery agent URI conforms to the syntax in Example 11.1, “Fuse Fabric Discovery Agent URI Format”.

Example 11.1. 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.

STATIC AGENT

The static discovery agent URI conforms to the syntax in Example 11.2, “Static Discovery Agent URI Format”.

Example 11.2. Static Discovery Agent URI Format

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

MULTICAST AGENT

The multicast discovery agent URI conforms to the syntax in Example 11.3, “Multicast Discovery Agent URI Format”.

Example 11.3. 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*.

ZEROCONF AGENT

The zeroconf discovery agent URI conforms to the syntax in Example 11.4, “Zeroconf Discovery Agent URI Format”.

Example 11.4. 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*.

CHAPTER 12. 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 12.1, “Broker Options”.

Table 12.1. 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

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.
`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.

INDEX

C

D

E

T

U

V

Z