Red Hat AMQ Broker 7.12

Configuring AMQ Broker

OVERVIEW

AMQ Broker configuration files define important settings for a broker instance. By editing a broker's configuration files, you can control how the broker operates in your environment.

AMQ BROKER CONFIGURATION FILES AND LOCATIONS

All of a broker's configuration files are stored in ***<broker_instance_dir>*/etc**. You can configure a broker by editing the settings in these configuration files.

Each broker instance uses the following configuration files:

broker.xml

The main configuration file. You use this file to configure most aspects of the broker, such as network connections, security settings, message addresses, and so on.

bootstrap.xml

The file that AMQ Broker uses to start a broker instance. You use it to change the location of **broker.xml**, configure the web server, and set some security settings.

logging.properties

You use this file to set logging properties for the broker instance.

artemis.profile

You use this file to set environment variables used while the broker instance is running.

login.config, **artemis-users.properties**, **artemis-roles.properties**

Security-related files. You use these files to set up authentication for user access to the broker instance.

UNDERSTANDING THE DEFAULT BROKER CONFIGURATION

You configure most of a broker's functionality by editing the **broker.xml** configuration file. This file contains default settings, which are sufficient to start and operate a broker. However, you will likely need to change some of the default settings and add new settings to configure the broker for your environment.

By default, **broker.xml** contains default settings for the following functionality:

- Message persistence
- Acceptors
- Security
- Message addresses

Default message persistence settings

By default, AMQ Broker persistence uses an append-only file journal that consists of a set of files on disk. The journal saves messages, transactions, and other information.

```
<configuration ...>

  <core ...>

    ...
    <persistence-enabled>true</persistence-enabled>

    <!-- this could be ASYNCIO, MAPPED, NIO
         ASYNCIO: Linux Libaio
         MAPPED: mmap files
         NIO: Plain Java Files
     -->
    <journal-type>ASYNCIO</journal-type>

    <paging-directory>data/paging</paging-directory>

    <bindings-directory>data/bindings</bindings-directory>

    <journal-directory>data/journal</journal-directory>

    <large-messages-directory>data/large-messages</large-messages-directory>

    <journal-datasync>true</journal-datasync>

    <journal-min-files>2</journal-min-files>

    <journal-pool-files>10</journal-pool-files>

    <journal-file-size>10M</journal-file-size>

    <!--
     This value was determined through a calculation.
     Your system could perform 8.62 writes per millisecond
     on the current journal configuration.
     That translates as a sync write every 115999 nanoseconds.

     Note: If you specify 0 the system will perform writes directly to the disk.
           We recommend this to be 0 if you are using journalType=MAPPED and journal-
datasync=false.
    -->
    <journal-buffer-timeout>115999</journal-buffer-timeout>

    <!--
      When using ASYNCIO, this will determine the writing queue depth for libaio.
     -->
    <journal-max-io>4096</journal-max-io>

    <!-- how often we are looking for how many bytes are being used on the disk in ms -->
    <disk-scan-period>5000</disk-scan-period>

    <!-- once the disk hits this limit the system will block, or close the connection in certain protocols
         that won't support flow control. -->
    <max-disk-usage>90</max-disk-usage>

    <!-- should the broker detect dead locks and other issues -->
    <critical-analyzer>true</critical-analyzer>

    <critical-analyzer-timeout>120000</critical-analyzer-timeout>

    <critical-analyzer-check-period>60000</critical-analyzer-check-period>
    <critical-analyzer-policy>HALT</critical-analyzer-policy>

    ...

  </core>

</configuration>
```

Default acceptor settings

Brokers listen for incoming client connections by using an **acceptor** configuration element to define the port and protocols a client can use to make connections. By default, AMQ Broker includes an acceptor for each supported messaging protocol, as shown below.

```
<configuration ...>

  <core ...>

    ...

    <acceptors>

      <!-- Acceptor for every supported protocol -->
      <acceptor name="artemis">tcp://0.0.0.0:61616?
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=CORE,AMQP,STOMP,HORNE
TQ,MQTT,OPENWIRE;useEpoll=true;amqpCredits=1000;amqpLowCredits=300</acceptor>

      <!-- AMQP Acceptor. Listens on default AMQP port for AMQP traffic -->
      <acceptor name="amqp">tcp://0.0.0.0:5672?
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=AMQP;useEpoll=true;amqpCre
dits=1000;amqpLowCredits=300</acceptor>

      <!-- STOMP Acceptor -->
      <acceptor name="stomp">tcp://0.0.0.0:61613?
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=STOMP;useEpoll=true</acce
ptor>

      <!-- HornetQ Compatibility Acceptor. Enables HornetQ Core and STOMP for legacy HornetQ
clients. -->
      <acceptor name="hornetq">tcp://0.0.0.0:5445?
anycastPrefix=jms.queue.;multicastPrefix=jms.topic.;protocols=HORNETQ,STOMP;useEpoll=true</a
cceptor>

      <!-- MQTT Acceptor -->
      <acceptor name="mqtt">tcp://0.0.0.0:1883?
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=MQTT;useEpoll=true</accept
or>

    </acceptors>

    ...

  </core>

</configuration>
```

Default security settings

AMQ Broker contains a flexible role-based security model for applying security to queues, based on their addresses. The default configuration uses wildcards to apply the **amq** role to all addresses (represented by the number sign, #).

```
<configuration ...>

  <core ...>

    ...

    <security-settings>
      <security-setting match="#">
        <permission  type="createNonDurableQueue"  roles="amq"/>
        <permission  type="deleteNonDurableQueue"  roles="amq"/>
        <permission  type="createDurableQueue"  roles="amq"/>
        <permission  type="deleteDurableQueue"  roles="amq"/>
        <permission type="createAddress"  roles="amq"/>
        <permission type="deleteAddress"  roles="amq"/>
        <permission type="consume" roles="amq"/>
        <permission type="browse" roles="amq"/>
        <permission type="send"  roles="amq"/>
        <!-- we need this otherwise ./artemis data imp wouldn't work -->
        <permission type="manage" roles="amq"/>
    </security-setting>
  </security-settings>

    ...

 </core>

</configuration>
```

Default message address settings

AMQ Broker includes a default address that establishes a default set of configuration settings to be applied to any created queue or topic.

Additionally, the default configuration defines two queues: **DLQ** (Dead Letter Queue) handles messages that arrive with no known destination, and **Expiry Queue** holds messages that have lived past their expiration and therefore should not be routed to their original destination.

```
<configuration ...>

  <core ...>

    ...

    <address-settings>
      ...
      <!--default for catch all-->
      <address-setting match="#">
        <dead-letter-address>DLQ</dead-letter-address>
        <expiry-address>ExpiryQueue</expiry-address>
        <redelivery-delay>0</redelivery-delay>
        <!-- with -1 only the global-max-size is in use for limiting -->
        <max-size-bytes>-1</max-size-bytes>
        <message-counter-history-day-limit>10</message-counter-history-day-limit>
        <address-full-policy>PAGE</address-full-policy>
        <auto-create-queues>true</auto-create-queues>
        <auto-create-addresses>true</auto-create-addresses>
        <auto-create-jms-queues>true</auto-create-jms-queues>
        <auto-create-jms-topics>true</auto-create-jms-topics>
      </address-setting>
    </address-settings>

    <addresses>
      <address name="DLQ">
        <anycast>
          <queue name="DLQ" />
        </anycast>
      </address>
      <address name="ExpiryQueue">
        <anycast>
          <queue name="ExpiryQueue" />
        </anycast>
     </address>
    </addresses>

  </core>

</configuration>
```

RELOADING CONFIGURATION UPDATES

By default, a broker checks for changes in the configuration files every 5000 milliseconds. If the broker detects a change in the "last modified" time stamp of the configuration file, the broker determines that a configuration change took place. In this case, the broker reloads the configuration file to activate the changes.

When the broker reloads the **broker.xml** configuration file, it reloads the following modules:

- Address settings and queues
  When the configuration file is reloaded, the address settings determine how to handle addresses and queues that have been deleted from the configuration file. You can set this with the **config-delete-addresses** and **config-delete-queues** properties.

- Security settings
  SSL/TLS keystores and truststores on an existing acceptor can be reloaded to establish new certificates without any impact to existing clients. Connected clients, even those with older or differing certificates, can continue to send and receive messages.

The certificate revocation list file, which is configured by using the **crlPath** parameter, can also be reloaded.

- Diverts
  A configuration reload deploys any new divert that you have added. However, removal of a divert from the configuration or a change to a sub-element within a **<divert>** element do not take effect until you restart the broker.

The following procedure shows how to change the interval at which the broker checks for changes to the **broker.xml** configuration file.

Procedure

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.
2. Within the **<core>** element, add the **<configuration-file-refresh-period>** element and set the refresh period (in milliseconds).
   This example sets the configuration refresh period to be 60000 milliseconds:

   ```
   <configuration>
       <core>
           ...
           <configuration-file-refresh-period>60000</configuration-file-refresh-period>
           ...
       </core>
    </configuration>
   ```

It is also possible to force the reloading of the configuration file using the Management API or the console if for some reason access to the configuration file is not possible. Configuration files can be reloaded using the management operation **reloadConfigurationFile()** on the **ActiveMQServerControl** (with the **ObjectName org.apache.activemq.artemis:broker="*BROKER_NAME*"** or the resource name **server**)

MODULARIZING THE BROKER CONFIGURATION FILE

If you have multiple brokers that share common configuration settings, you can define the common configuration in separate files, and then include these files in each broker's **broker.xml** configuration file.

The most common configuration settings that you might share between brokers include:

- Addresses
- Address settings
- Security settings

Procedure

3. Create a separate XML file for each **broker.xml** section that you want to share.
   Each XML file can only include a single section from **broker.xml** (for example, either addresses or address settings, but not both). The top-level element must also define the element namespace (**xmlns="urn:activemq:core"**).

   This example shows a security settings configuration defined in **my-security-settings.xml**:

   my-security-settings.xml

```
  <security-settings xmlns="urn:activemq:core">
    <security-setting match="a1">
      <permission type="createNonDurableQueue" roles="a1.1"/>
    </security-setting>
    <security-setting match="a2">
      <permission type="deleteNonDurableQueue" roles="a2.1"/>
   </security-setting>
</security-settings>
```

4. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file for each broker that should use the common configuration settings.

5. For each **broker.xml** file that you opened, do the following:

    a. In the **<configuration>** element at the beginning of **broker.xml**, verify that the following line appears:

    ```
    xmlns:xi="http://www.w3.org/2001/XInclude"
    ```

    b. Add an XML inclusion for each XML file that contains shared configuration settings. This example includes the **my-security-settings.xml** file.

    broker.xml

    ```
    <configuration ...>
        <core ...>
            ...
            <xi:include href="/opt/my-broker-config/my-security-settings.xml"/>
            ...
        </core>
    </configuration>
    ```

    c. If desired, validate **broker.xml** to verify that the XML is valid against the schema. You can use any XML validator program. This example uses **xmllint** to validate **broker.xml** against the **artemis-server.xsl** schema.

    ```
    $ xmllint --noout --xinclude --schema /opt/redhat/amq-broker/amq-broker-
    7.2.0/schema/artemis-server.xsd /var/opt/amq-broker/mybroker/etc/broker.xml
    /var/opt/amq-broker/mybroker/etc/broker.xml validates
    ```

Reloading modular configuration files

When the broker periodically checks for configuration changes (according to the frequency specified by **configuration-file-refresh-period**), it does not automatically detect changes made to configuration files that are included in the **broker.xml** configuration file via **xi:include**. For example, if **broker.xml** includes **my-address-settings.xml** and you make configuration changes to **my-address-settings.xml**, the broker does not automatically detect the changes in **my-address-settings.xml** and reload the configuration.

To *force* a reload of the **broker.xml** configuration file and any modified configuration files included within it, you must ensure that the "last modified" time stamp of the **broker.xml** configuration file has changed. You can use a standard Linux **touch** command to update the last-modified time stamp of **broker.xml** without making any other changes. For example:

```
$ touch -m <broker_instance_dir>/etc/broker.xml
```

Alternatively you can use the management API to force a reload of the Broker. Configuration files can be reloaded using the management operation **reloadConfigurationFile()** on the **ActiveMQServerControl** (with the **ObjectName org.apache.activemq.artemis:broker="*BROKER_NAME*"** or the resource name **server**)

Disabling External XML Entity (XXE) processing

If you don’t want to modularize your broker configuration in separate files that are included in the **broker.xml** file, you can disable XXE processing to protect AMQ Broker against XXE security vulnerabilities. If you don’t have a modular broker configuration, Red Hat recommends that you disable XXE processing.

Procedure

1. Open the **<broker_instance_dir>/etc/artemis.profile** file.

2. Add a new argument, **-Dartemis.disableXxe**, to the **JAVA_ARGS** list of Java system arguments.

   ```
   -Dartemis.disableXxe=true
   ```

3. Save the **artemis.profile** file.

EXTENDING THE JAVA CLASSPATH

By default, JAR files in the ***<broker_instance_dir>*/lib** directory are loaded at runtime because the directory is part of the Java classpath. If you want AMQ Broker to load JAR files from a directory other than ***<broker_instance_dir>*/lib**, you must add that directory to the Java classpath.

To add a directory to the Java class path, you can use either of the following methods:

- In the **<broker_instance_dir>/etc/artemis.profile** file, add a new property, **artemis.extra.libs** to the **JAVA_ARGS** list of system properties.

- Set the **ARTEMIS_EXTRA_LIBS** environment variable.

The following are examples of comma-separated lists of directories that are added to the Java Classpath by using both methods:

```
-Dartemis.extra.libs=/usr/local/share/java/lib1,/usr/local/share/java/lib2
```

```
export ARTEMIS_EXTRA_LIBS=/usr/local/share/java/lib1,/usr/local/share/java/lib2
```

NOTE

The **ARTEMIS_EXTRA_LIBS** environment variable is ignored if the **artemis.extra.libs** Java system property is configured in the **<broker_instance_dir>/etc/artemis.profile** file.

CONFIGURING ACCEPTORS AND CONNECTORSIN NETWORK CONNECTIONS

There are two types of connections used in AMQ Broker: network connections and *in-VM* connections. Network connections are used when the two parties are located in different virtual machines, whether on the same server or physically remote. An in-VM connection is used when the client, whether an application or a server, resides on the same virtual machine as the broker.

Network connections use Netty. Netty is a high-performance, low-level network library that enables network connections to be configured in several different ways; using Java IO or NIO, TCP sockets, SSL/TLS, or tunneling over HTTP or HTTPS. Netty also allows for a single port to be used for all messaging protocols. A broker will automatically detect which protocol is being used and direct the incoming message to the appropriate handler for further processing.

The URI of a network connection determines its type. For example, specifying **vm** in the URI creates an in-VM connection:

```
<acceptor name="in-vm-example">vm://0</acceptor>
```

Alternatively, specifying **tcp** in the URI creates a network connection. For example:

```
<acceptor name="network-example">tcp://localhost:61617</acceptor>
```

ABOUT ACCEPTORS

Acceptors define how connections are made to the broker. Each acceptor defines the port and protocols that a client can use to make a connection. A simple acceptor configuration is shown below.

```
<acceptors>
  <acceptor name="example-acceptor">tcp://localhost:61617</acceptor>
</acceptors>
```

Each **acceptor** element that you define in the broker configuration is contained within a single **acceptors** element. There is no upper limit to the number of acceptors that you can define for a broker. By default, AMQ Broker includes an acceptor for each supported messaging protocol, as shown below:

```
<configuration ...>
  <core ...>
    ...
    <acceptors>
      ...
      <!-- Acceptor for every supported protocol -->
      <acceptor name="artemis">tcp://0.0.0.0:61616?
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=CORE,AMQP,STOMP,HORNE
TQ,MQTT,OPENWIRE;useEpoll=true;amqpCredits=1000;amqpLowCredits=300</acceptor>

      <!-- AMQP Acceptor. Listens on default AMQP port for AMQP traffic -->
      <acceptor name="amqp">tcp://0.0.0.0:5672?
```

```
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=AMQP;useEpoll=true;amqpCre
dits=1000;amqpLowCredits=300</acceptor>

        <!-- STOMP Acceptor -->
        <acceptor name="stomp">tcp://0.0.0.0:61613?
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=STOMP;useEpoll=true</acce
ptor>

        <!-- HornetQ Compatibility Acceptor. Enables HornetQ Core and STOMP for legacy HornetQ
clients. -->
        <acceptor name="hornetq">tcp://0.0.0.0:5445?
anycastPrefix=jms.queue.;multicastPrefix=jms.topic.;protocols=HORNETQ,STOMP;useEpoll=true</a
cceptor>

        <!-- MQTT Acceptor -->
        <acceptor name="mqtt">tcp://0.0.0.0:1883?
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=MQTT;useEpoll=true</accept
or>
      </acceptors>
      ...
  </core>
</configuration>
```

CONFIGURING ACCEPTORS

The following example shows how to configure an acceptor.

Procedure

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

2. In the **acceptors** element, add a new **acceptor** element. Specify a protocol, and port on the broker. For example:

   ```
   <acceptors>
     <acceptor name="example-acceptor">tcp://localhost:61617</acceptor>
   </acceptors>
   ```

   The preceding example defines an acceptor for the TCP protocol. The broker listens on port 61617 for client connections that are using TCP.

3. Append key-value pairs to the URI defined for the acceptor. Use a semicolon (**;**) to separate multiple key-value pairs. For example:

   ```
   <acceptor name="example-acceptor">tcp://localhost:61617?sslEnabled=true;key-store-
   path=</path/to/key_store></acceptor>
   ```

   The configuration now defines an acceptor that uses TLS/SSL and defines the path to the required key store.

ABOUT CONNECTORS

While acceptors define how a broker *accepts* connections, connectors are used by clients to define how they can *connect* to a broker.

A connector is configured on a broker when the broker itself acts as a client. For example:

- When the broker is bridged to another broker
- When the broker takes part in a cluster

A simple connector configuration is shown below.

```
<connectors>
  <connector name="example-connector">tcp://localhost:61617</connector>
</connectors>
```

CONFIGURING CONNECTORS

The following example shows how to configure a connector.

Procedure

4. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

5. In the **connectors** element, add a new **connector** element. Specify a protocol, and port on the broker. For example:

   ```
   <connectors>
     <connector name="example-connector">tcp://localhost:61617</connector>
   </connectors>
   ```

   The preceding example defines a connector for the TCP protocol. Clients can use the connector configuration to connect to the broker on port 61617 using the TCP protocol. The broker itself can also use this connector for outgoing connections.

6. Append key-value pairs to the URI defined for the connector. Use a semicolon (**;**) to separate multiple key-value pairs. For example:

   ```
   <connector name="example-connector">tcp://localhost:61616?tcpNoDelay=true</connector>
   ```

   The configuration now defines a connector that sets the value of the **tcpNoDelay** property to **true**. Setting the value of this property to **true** turns off Nagle’s algorithm for the connection. Nagle’s algorithm is an algorithm used to improve the efficiency of TCP connections by delaying transmission of small data packets and consolidating these into large packets.

ONFIGURING A TCP CONNECTION

AMQ Broker uses Netty to provide basic, unencrypted, TCP-based connectivity that can be configured to use blocking Java IO or the newer, non-blocking Java NIO. Java NIO is preferred for better scalability with many concurrent connections. However, using the old IO can sometimes give you better latency than NIO when you are less worried about supporting many thousands of concurrent connections.

If you are running connections across an untrusted network, you should be aware that a TCP network connection is unencrypted. You might want to consider using an SSL or HTTPS configuration to encrypt messages sent over this connection if security is a priority.

When using a TCP connection, all connections are initiated by the client. The broker does not initiate any connections to the client. This works well with firewall policies that force connections to be initiated from one direction.

For TCP connections, the host and the port of the connector URI define the address used for the connection.

The following example shows how to configure a TCP connection.

Prerequisites

- You should be familiar with configuring acceptors and connectors.

Procedure

7. Open the ***<broker_instance_dir>*****/etc/broker.xml** configuration file.

8. Add a new acceptor or modify an existing one. In the connection URI, specify **tcp** as the protocol. Include both an IP address or host name and a port on the broker. For example:

   ```
   <acceptors>
     <acceptor name="tcp-acceptor">tcp://10.10.10.1:61617</acceptor>
     ...
   </acceptors>
   ```

   Based on the preceding example, the broker accepts TCP communications from clients connecting to port **61617** at the IP address **10.10.10.1**.

9. (Optional) You can configure a connector in a similar way. For example:

   ```
   <connectors>
     <connector   name="tcp-connector">tcp://10.10.10.2:61617</connector>
     ...
   </connectors>
   ```

   The connector in the preceding example is referenced by a client, or even the broker itself, when making a TCP connection to the specified IP and port, **10.10.10.2:61617**.

CONFIGURING AN HTTP CONNECTION

HTTP connections tunnel packets over the HTTP protocol and are useful in scenarios where firewalls allow only HTTP traffic. AMQ Broker automatically detects if HTTP is being used, so configuring a network connection for HTTP is the same as configuring a connection for TCP.

Prerequisites

- You should be familiar with configuring acceptors and connectors

Procedure

10. Open the ***<broker_instance_dir>*****/etc/broker.xml** configuration file.

11. Add a new acceptor or modify an existing one. In the connection URI, specify **tcp** as the protocol. Include both an IP address or host name and a port on the broker. For example:

    ```
    <acceptors>
      <acceptor name="http-acceptor">tcp://10.10.10.1:80</acceptor>
      ...
    </acceptors>
    ```

    Based on the preceding example, the broker accepts HTTP communications from clients connecting to port **80** at the IP address **10.10.10.1**. The broker automatically detects that the HTTP protocol is in use and communicates with the client accordingly.

12. (Optional) You can configure a connector in a similar way. For example:

```
<connectors>
  <connector name="http-connector">tcp://10.10.10.2:80</connector>
  ...
</connectors>
```

Using the connector shown in the preceding example, a broker creates an outbound HTTP connection on port **80** at the IP address **10.10.10.2**.

CONFIGURING SECURE NETWORK CONNECTIONS

You can secure network connections using TLS/SSL.

CONFIGURING AN IN-VM CONNECTION

You can use an in-VM connection when multiple brokers are co-located on the same virtual machine, for example, as part of a high availability (HA) configuration. In-VM connections can also be used by local clients running in the same JVM as the broker.

Prerequisites

- You should be familiar with configuring acceptors and connectors.

Procedure

13. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

14. Add a new acceptor or modify an existing one. In the connection URI, specify **vm** as the protocol. For example:

```
<acceptors>
  <acceptor name="in-vm-acceptor">vm://0</acceptor>
  ...
</acceptors>
```

Based on the acceptor in the preceding example, the broker accepts connections from a broker with an ID of **0**. The other broker must be running on the same virtual machine.

15. (Optional) You can configure a connector in a similar way. For example:

```
<connectors>
  <connector name="in-vm-connector">vm://0</connector>
  ...
</connectors>
```

The connector in the preceding example defines how a client can establish an in-VM connection to a broker with an ID of **0** that is running on the same virtual machine as the client. The client can be an application or another broker.

CONFIGURING MESSAGING PROTOCOLS INNETWORK CONNECTIONS

AMQ Broker has a pluggable protocol architecture, so that you can easily enable one or more protocols for a network connection.

The broker supports the following protocols:

- AMQP
- MQTT
- OpenWire
- STOMP

NOTE

In addition to the protocols above, the broker also supports its own native protocol known as "Core". Past versions of this protocol were known as "HornetQ" and used by Red Hat JBoss Enterprise Application Platform.

CONFIGURING A NETWORK CONNECTION TO USE A MESSAGING PROTOCOL

You must associate a protocol with a network connection before you can use it. The default configuration, located in the file ***<broker_instance_dir>*/etc/broker.xml**, includes several connections already defined. For convenience,AMQ Broker includes an acceptor for each supported protocol, plus a default acceptor that supports all protocols.

Overview of default acceptors

Shown below are the acceptors included by default in the **broker.xml** configuration file.

```
<configuration>
  <core>
    ...
    <acceptors>

      <!-- All-protocols acceptor -->
      <acceptor name="artemis">tcp://0.0.0.0:61616?
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=CORE,AMQP,STOMP,HORNE
TQ,MQTT,OPENWIRE;useEpoll=true;amqpCredits=1000;amqpLowCredits=300</acceptor>

      <!-- AMQP Acceptor. Listens on default AMQP port for AMQP traffic -->
      <acceptor name="amqp">tcp://0.0.0.0:5672?
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=AMQP;useEpoll=true;amqpCre
dits=1000;amqpLowCredits=300</acceptor>

      <!-- STOMP Acceptor -->
      <acceptor name="stomp">tcp://0.0.0.0:61613?
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=STOMP;useEpoll=true</acce
ptor>
```

```
    <!-- HornetQ Compatibility Acceptor. Enables HornetQ Core and STOMP for legacy HornetQ clients. -->
    <acceptor name="hornetq">tcp://0.0.0.0:5445?anycastPrefix=jms.queue.;multicastPrefix=jms.topic.;protocols=HORNETQ,STOMP;useEpoll=true</acceptor>

    <!-- MQTT Acceptor -->
    <acceptor name="mqtt">tcp://0.0.0.0:1883?tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=MQTT;useEpoll=true</acceptor>

  </acceptors>
  ...
 </core>
</configuration>
```

The only requirement to enable a protocol on a given network connnection is to add the **protocols** parameter to the URI for the acceptor. The value of the parameter must be a comma separated list of protocol names. If the protocol parameter is omitted from the URI, all protocols are enabled.

For example, to create an acceptor for receiving messages on port 3232 using the AMQP protocol, follow these steps:

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.
2. Add the following line to the **<acceptors>** stanza:

```
<acceptor name="ampq">tcp://0.0.0.0:3232?protocols=AMQP</acceptor>
```

Additional parameters in default acceptors

In a minimal acceptor configuration, you specify a protocol as part of the connection URI. However, the default acceptors in the **broker.xml** configuration file have some additional parameters configured. The following table details the additional parameters configured for the default acceptors.

Acceptor(s)	Parameter	Description
All-protocols acceptor AMQP STOMP	tcpSendBufferSize	Size of the TCP send buffer in bytes. The default value is **32768**.

Acceptor(s)	Parameter	Description
	tcpReceiveBufferSize	Size of the TCP receive buffer in bytes. The default value is **32768**. TCP buffer sizes should be tuned according to the bandwidth and latency of your network. In summary TCP send/receive buffer sizes should be calculated as: buffer_size = bandwidth * RTT. Where bandwidth is in bytes per second and network round trip time (RTT) is in seconds. RTT can be easily measured using the **ping** utility. For fast networks you may want to increase the buffer sizes from the defaults.
All-protocols acceptor AMQP STOMP HornetQ MQTT	useEpoll	Use Netty epoll if using a system (Linux) that supports it. The Netty native transport offers better performance than the NIO transport. The default value of this option is **true**. If you set the option to **false**, NIO is used.
All-protocols acceptor AMQP	amqpCredits	Maximum number of messages that an AMQP producer can send, regardless of the total message size. The default value is **1000**.
All-protocols acceptor AMQP	amqpLowCredits	Lower threshold at which the broker replenishes producer credits. The default value is **300**. When the producer reaches this threshold, the broker sends the producer sufficient credits to restore the **amqpCredits** value.

Acceptor(s)	Parameter	Description
HornetQ compatibility acceptor	anycastPrefix	Prefix that clients use to specify the **anycast** routing type when connecting to an address that uses both **anycast** and **multicast**. The default value is **jms.queue**.
	multicastPrefix	Prefix that clients use to specify the **multicast** routing type when connecting to an address that uses both **anycast** and **multicast**. The default value is **jms.topic**.

USING AMQP WITH A NETWORK CONNECTION

The broker supports the AMQP 1.0 specification. An AMQP link is a uni-directional protocol for messages between a source and a target, that is, a client and the broker.

Procedure

3. Open the **<*broker_instance_dir*>/etc/broker.xml** configuration file.

4. Add or configure an **acceptor** to receive AMQP clients by including the **protocols** parameter with a value of **AMQP** as part of the URI, as shown in the following example:

```
<acceptors>
  <acceptor name="amqp-acceptor">tcp://localhost:5672?protocols=AMQP</acceptor>
  ...
</acceptors>
```

In the preceding example, the broker accepts AMQP 1.0 clients on port 5672, which is the default AMQP port.

An AMQP link has two endpoints, a sender and a receiver. When senders transmit a message, the broker converts it into an internal format, so it can be forwarded to its destination on the broker. Receivers connect to the destination at the broker and convert the messages back into AMQP before they are delivered.

If an AMQP link is dynamic, a temporary queue is created and either the remote source or the remote target address is set to the name of the temporary queue. If the link is not dynamic, the address of the remote target or source is used for the queue. If the remote target or source does not exist, an exception is sent.

A link target can also be a Coordinator, which is used to handle the underlying session as a transaction, either rolling it back or committing it.

NOTE

AMQP allows the use of multiple transactions per session, **amqp:multi-txns-per-ssn**, however the current version of AMQ Broker will support only single transactions per session.

NOTE

The details of distributed transactions (XA) within AMQP are not provided in the 1.0 version of the specification. If your environment requires support for distributed transactions, it is recommended that you use the AMQ Core Protocol JMS.

Using an AMQP Link as a Topic

Unlike JMS, the AMQP protocol does not include topics. However, it is still possible to treat AMQP consumers or receivers as subscriptions rather than just consumers on a queue. By default, any receiving link that attaches to an address with the prefix **jms.topic.** is treated as a subscription, and a subscription queue is created. The subscription queue is made durable or volatile, depending on how the Terminus Durability is configured, as captured in the following table:

To create this kind of subscription for a multicast-only queue…	Set Terminus Durability to this…
Durable	**UNSETTLED_STATE** or **CONFIGURATION**
Non-durable	**NONE**

NOTE

The name of a durable queue is composed of the container ID and the link name, for example **my-container-id:my-link-name**.

AMQ Broker also supports the qpid-jms client and will respect its use of topics regardless of the prefix used for the address.

Configuring AMQP security

The broker supports AMQP SASL Authentication.

USING MQTT WITH A NETWORK CONNECTION

The broker supports MQTT v3.1.1 and v5.0 (and also the older v3.1 code message format). MQTT is a lightweight, client to server, publish/subscribe messaging protocol. MQTT reduces messaging overhead and network traffic, as well as a client’s code footprint. For these reasons, MQTT is ideally suited to constrained devices such as sensors and actuators and is quickly becoming the de facto standard communication protocol for Internet of Things(IoT).

Procedure

5. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

6. Add an acceptor with the MQTT protocol enabled. For example:

```
<acceptors>
  <acceptor name="mqtt">tcp://localhost:1883?protocols=MQTT</acceptor>
  ...
</acceptors>
```

MQTT comes with a number of useful features including:

Quality of Service

Each message can define a quality of service that is associated with it. The broker will attempt to deliver messages to subscribers at the highest quality of service level defined.

Retained Messages

Messages can be retained for a particular address. New subscribers to that address receive the last-sent retained message before any other messages, even if the retained message was sent before the client connected.
Retained messages are stored in a queue named **sys.mqtt.<*topic name*>** and remain in the queue until a client deletes the retained message or, if an expiry is configured, until the message expires. When a queue is empty, the queue is not removed until you explicitly delete it. For example, the following configuration deletes a queue:

```
<address-setting match="$sys.mqtt.retain.#">
   <auto-delete-queues>true</auto-delete-queues>
   <auto-delete-addresses>true</auto-delete-addresses>
</address-setting>
```

Wild card subscriptions

MQTT addresses are hierarchical, similar to the hierarchy of a file system. Clients are able to subscribe to specific topics or to whole branches of a hierarchy.

Will Messages

Clients are able to set a "will message" as part of their connect packet. If the client abnormally disconnects, the broker will publish the will message to the specified address. Other subscribers receive the will message and can react accordingly.

Configuring MQTT properties

You can append key-value pairs to the MQTT acceptor to configure connection properties. For example:

–

```
<acceptors>
  <acceptor name="mqtt">tcp://localhost:1883?
protocols=MQTT;receiveMaximum=50000;topicAliasMaximum=50000;maximumPacketSize;134217728

serverKeepAlive=30;closeMqttConnectionOnPublishAuthorizationFailure=false</acceptor>
  ...
</acceptors>
```

receiveMaximum

Enables flow-control by specifying the maximum number of QoS 1 and 2 messages that the broker can receive from a client before an acknowledgment is required. The default value is **65535**. A value of **-1** disables flow-control from clients to the broker. This has the same effect as setting the value to 0 but reduces the size of the CONNACK packet.

topicAliasMaximum

Specifies for clients the maximum number of aliases that the broker supports. The default value is **65535**. A value of -1 prevents the broker from informing the client of a topic alias limit. This has the same effect as setting the value to 0, but reduces the size of the CONNACK packet.

maximumPacketSize

Specifies the maximum packet size that the broker can accept from clients. The default value is **268435455**. A value of -1 prevents the broker from informing the client of a maximum packet size, which means that no limit is enforced on the size of incoming packets.

serverKeepAlive

Specifies the duration the broker keeps an inactive client connection open. The configured value is applied to the connection only if it is less than the keep-alive value configured for the client or if the value configured for the client is 0. The default value is **60** seconds. A value of **-1** means that the broker always accepts the client's keep alive value (even if that value is 0).

closeMqttConnectionOnPublishAuthorizationFailure

By default, if a PUBLISH packet fails due to a lack of authorization, the broker closes the network connection. If you want the broker to sent a positive acknowledgment instead of closing the network connection, set **closeMqttConnectionOnPublishAuthorizationFailure** to **false**.

USING OPENWIRE WITH A NETWORK CONNECTION

The broker supports the OpenWire protocol, which allows a JMS client to talk directly to a broker. Use this protocol to communicate with older versions of AMQ Broker.

Currently AMQ Broker supports OpenWire clients that use standard JMS APIs only.

Procedure

7. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

8. Add or modify an **acceptor** so that it includes **OPENWIRE** as part of the **protocol** parameter, as shown in the following example:

```
<acceptors>
  <acceptor name="openwire-acceptor">tcp://localhost:61616?
protocols=OPENWIRE</acceptor>
  ...
</acceptors>
```

In the preceding example, the broker will listen on port 61616 for incoming OpenWire commands.

USING STOMP WITH A NETWORK CONNECTION

STOMP is a text-orientated wire protocol that allows STOMP clients to communicate with STOMP Brokers. The broker supports STOMP 1.0, 1.1 and 1.2. STOMP clients are available for several languages and platforms making it a good choice for interoperability.

Procedure

9. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

10. Configure an existing **acceptor** or create a new one and include a **protocols** parameter with a value of **STOMP**, as below.

```
<acceptors>
  <acceptor   name="stomp-acceptor">tcp://localhost:61613?protocols=STOMP</acceptor>
  ...
</acceptors>
```

In the preceding example, the broker accepts STOMP connections on the port **61613**, which is the default.

STOMP limitations

When using STOMP, the following limitations apply:

1. The broker currently does not support virtual hosting, which means the **host** header in **CONNECT** frames are ignored.

2. Message acknowledgments are not transactional. The **ACK** frame cannot be part of a transaction, and it is ignored if its **transaction** header is set).

Providing IDs for STOMP Messages

When receiving STOMP messages through a JMS consumer or a QueueBrowser, the messages do not contain any JMS properties, for example **JMSMessageID**, by default. However, you can set a message ID on each incoming STOMP message by using a broker paramater.

Procedure

3. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

4. Set the **stompEnableMessageId** parameter to **true** for the **acceptor** used for STOMP connections, as shown in the following example:

```
<acceptors>
  <acceptor name="stomp-acceptor">tcp://localhost:61613?
protocols=STOMP;stompEnableMessageId=true</acceptor>
```

```
  ...
</acceptors>
```

By using the **stompEnableMessageId** parameter, each stomp message sent using this acceptor has an extra property added. The property key is **amq-message-id** and the value is a String representation of an internal message id prefixed with “STOMP”, as shown in the following example:

```
amq-message-id : STOMP12345
```

If **stompEnableMessageId** is not specified in the configuration, the default value is **false**.

Setting a connection time to live

STOMP clients must send a **DISCONNECT** frame before closing their connections. This allows the broker to close any server-side resources, such as sessions and consumers. However, if STOMP clients exit without sending a DISCONNECT frame, or if they fail, the broker will have no way of knowing immediately whether the client is still alive. STOMP connections therefore are configured to have a "Time to Live" (TTL) of 1 minute. The means that the broker stops the connection to the STOMP client if it has been idle for more than one minute.

Procedure

5. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

6. Add the **connectionTTL** parameter to URI of the **acceptor** used for STOMP connections, as shown in the following example:

```
<acceptors>
  <acceptor name="stomp-acceptor">tcp://localhost:61613?
protocols=STOMP;connectionTTL=20000</acceptor>
  ...
</acceptors>
```

In the preceding example, any stomp connection that using the **stomp-acceptor** will have its TTL set to 20 seconds.

NOTE

Version 1.0 of the STOMP protocol does not contain any heartbeat frame. It is therefore the user’s responsibility to make sure data is sent within connection-ttl or the broker will assume the client is dead and clean up server-side resources. With version 1.1, you can use heart-beats to maintain the life cycle of stomp connections.

Overriding the broker default time to live

As noted, the default TTL for a STOMP connection is one minute. You can override this value by adding the **connection-ttl-override** attribute to the broker configuration.

Procedure

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

2. Add the **connection-ttl-override** attribute and provide a value in milliseconds for the new default. It belongs inside the **<core>** stanza, as below.

```
<configuration ...>
 ...
 <core ...>
  ...
  <connection-ttl-override>30000</connection-ttl-override>
  ...
 </core>
<configuration>
```

In the preceding example, the default Time to Live (TTL) for a STOMP connection is set to 30 seconds, **30000** milliseconds.

Sending and consuming STOMP messages from JMS

STOMP is mainly a text-orientated protocol. To make it simpler to interoperate with JMS, the STOMP implementation checks for presence of the **content-length** header to decide how to map a STOMP message to JMS.

If you want a STOMP message to map to a ...	The message should....
JMS TextMessage	Not include a **content-length** header.
JMS BytesMessage	Include a **content-length** header.

The same logic applies when mapping a JMS message to STOMP. A STOMP client can confirm the presence of the **content-length** header to determine the type of the message body (string or bytes).

Mapping STOMP destinations to AMQ Broker addresses and queues

When sending messages and subscribing, STOMP clients typically include a **destination** header. Destination names are string values, which are mapped to a destination on the broker. In AMQ Broker, these destinations are mapped to addresses and queues.

Take for example a STOMP client that sends the following message (headers and body included):

```
SEND
destination:/my/stomp/queue

hello queue a
^@
```

In this case, the broker will forward the message to any queues associated with the address **/my/stomp/queue**.

For example, when a STOMP client sends a message (by using a **SEND** frame), the specified destination is mapped to an address.

It works the same way when the client sends a **SUBSCRIBE** or **UNSUBSCRIBE** frame, but in this case AMQ Broker maps the **destination** to a queue.

```
SUBSCRIBE
destination: /other/stomp/queue
ack: client

^@
```

In the preceding example, the broker will map the **destination** to the queue **/other/stomp/queue**.

Mapping STOMP destinations to JMS destinations
JMS destinations are also mapped to broker addresses and queues. If you want to use STOMP to send messages to JMS destinations, the STOMP destinations must follow the same convention:

- Send or subscribe to a JMS Queue by prepending the queue name by **jms.queue.**. For example, to send a message to the **orders** JMS Queue, the STOMP client must send the frame:

  ```
  SEND
  destination:jms.queue.orders
  hello queue orders
  ^@
  ```

- Send or subscribe to a JMS Topic by prepending the topic name by **jms.topic.**. For example, to subscribe to the **stocks** JMS Topic, the STOMP client must send a frame similar to the following:

  ```
  SUBSCRIBE
  destination:jms.topic.stocks
  ^@
  ```

CONFIGURING ADDRESSES AND QUEUES

ADDRESSES, QUEUES, AND ROUTING TYPES

In AMQ Broker, the addressing model comprises three main concepts; *addresses*, *queues*, and *routing types*.

An address represents a messaging endpoint. Within the configuration, a typical address is given a unique name, one or more queues, and a routing type.

A queue is associated with an address. There can be multiple queues per address. Once an incoming message is matched to an address, the message is sent on to one or more of its queues, depending on the routing type configured. Queues can be configured to be automatically created and deleted. You can also configure an address (and hence its associated queues) as *durable*. Messages in a durable queue can survive a crash or restart of the broker, as long as the messages in the queue are also persistent. By contrast, messages in a non-durable queue do not survive a crash or restart of the broker, even if the messages themselves are persistent.

A routing type determines how messages are sent to the queues associated with an address. In AMQ Broker, you can configure an address with two different routing types, as shown in the table.

If you want your messages routed to…	Use this routing type…
A single queue within the matching address, in a point-to-point manner	**anycast**
Every queue within the matching address, in a publish-subscribe manner	**multicast**

NOTE

An address must have at least one defined routing type.

It is *possible* to define more than one routing type per address, but this is not recommended.

If an address *does* have both routing types defined, and the client does not show a preference for either one, the broker defaults to the **multicast** routing type.

Address and queue naming requirements

Be aware of the following requirements when you configure addresses and queues:

- To ensure that a client can connect to a queue, regardless of which wire protocol the client uses, your address and queue names should not include any of the following characters:
  & :: , ? >
- The number sign (#) and asterisk (*****) characters are reserved for wildcard expressions and should not be used in address and queue names.
- Address and queue names should not include spaces.
- To separate words in an address or queue name, use the configured delimiter character. The default delimiter character is a period (**.**)

APPLYING ADDRESS SETTINGS TO SETS OF ADDRESSES

In AMQ Broker, you can apply the configuration specified in an **address-setting** element to a *set* of addresses by using a wildcard expression to represent the matching address name.

The following sections describe how to use wildcard expressions.

AMQ Broker wildcard syntax

AMQ Broker uses a specific syntax for representing wildcards in address settings. Wildcards can also be used in security settings, and when creating consumers.

- A wildcard expression contains words delimited by a period (**.**).
- The number sign (#) and asterisk (*****) characters also have special meaning and can take the place of a word, as follows:
  - The number sign character means "match any sequence of zero or more words". Use this at the end of your expression.
  - The asterisk character means "match a single word". Use this anywhere within your expression.

Matching is not done character by character, but at each delimiter boundary. For example, an **address-setting** element that is configured to match queues with **my** in their name would not match with a queue named **myqueue**.

When more than one **address-setting** element matches an address, the broker overlays configurations, using the configuration of the least specific match as the baseline. Literal expressions are more specific than wildcards, and an asterisk (*****) is more specific than a number sign (#). For example, both **my.destination** and **my.*** match the address **my.destination**. In this case, the broker first applies the configuration found under **my.***, since a wildcard expression is less specific than a literal. Next, the broker overlays the configuration of the **my.destination** address setting element, which overwrites any configuration shared with **my.***. For example, given the following configuration, a queue associated with **my.destination** has **max-delivery-attempts** set to **3** and **last-value-queue** set to **false**.

```
<address-setting match="my.*">
   <max-delivery-attempts>3</max-delivery-attempts>
   <last-value-queue>true</last-value-queue>
</address-setting>
```

```
<address-setting match="my.destination">
   <last-value-queue>false</last-value-queue>
</address-setting>
```

The examples in the following table illustrate how wildcards are used to match a set of addresses.

Example	Description
#	The default **address-setting** used in **broker.xml**. Matches every address. You can continue to apply this catch-all, or you can add a new **address-setting** for each address or group of addresses as the need arises.
news.europe.#	Matches **news.europe**, **news.europe.sport**, **news.europe.politics.fr**, but not **news.usa** or **europe**.
news.*	Matches **news.europe** and **news.usa**, but not **news.europe.sport**.
news.*.sport	Matches **news.europe.sport** and **news.usa.sport**, but not **news.europe.fr.sport**.

Configuring a literal match

In a literal match, wildcard characters are treated as literal characters to match addresses that contain wildcards. For example, the hash (#) character in a literal match can match an address of **orders.#** without matching addresses such as **orders.retail** or **orders.wholesale**.

Procedure

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

2. Before you configure a literal match, use the **literal-match-markers** parameter to define the characters that delimit a literal match. In the following example, parentheses are used to delimit a literal match.

   ```
   <core>
     ...
     <literal-match-markers>()</literal-match-markers>
     ...
   </core>
   ```

3. After you define the markers that delimit a literal match, specify the match, including the markers, in the **address setting match** parameter. The following example configures a literal match for an address called **orders.#** to enable metrics for that specific address.

   ```
   <address-settings>
     <address-setting match="(orders.#)">
       <enable-metrics>true</enable-metrics>
    </address-setting>
   </address-settings>
   ```

Configuring the broker wildcard syntax

The following procedure show how to customize the syntax used for wildcard addresses.

Procedure

4. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.
5. Add a **<wildcard-addresses>** section to the configuration, as in the example below.

```
<configuration>
  <core>
    ...
    <wildcard-addresses> //
      <enabled>true</enabled> //
      <delimiter>,</delimiter> //
      <any-words>@</any-words> //
      <single-word>$</single-word>
    </wildcard-addresses>
    ...
  </core>
</configuration>
```

enabled

When set to **true**, instruct the broker to use your custom settings.

delimiter

Provide a custom character to use as the **delimiter** instead of the default, which is **.**.

any-words

The character provided as the value for **any-words** is used to mean 'match any sequence of zero or more words' and will replace the default #. Use this character at the end of your expression.

single-word

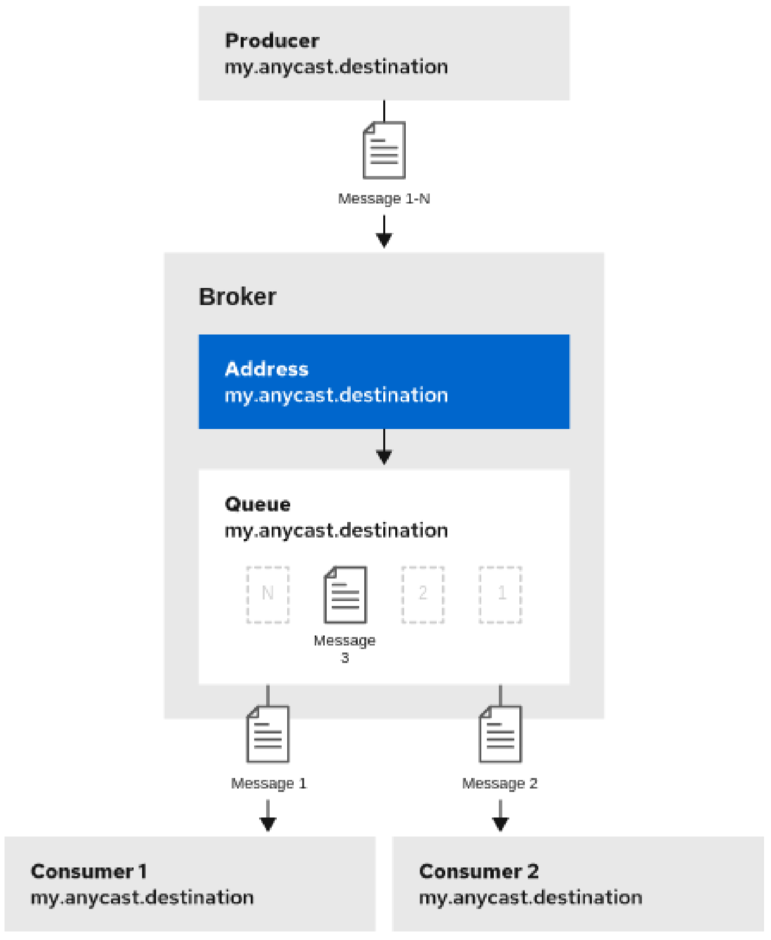
The character provided as the value for **single-word** is used to mean 'match a single word' and will replaced the default *****. Use this character anywhere within your expression.

CONFIGURING ADDRESSES FOR POINT-TO-POINT MESSAGING

Point-to-point messaging is a common scenario in which a message sent by a producer has only one consumer. AMQP and JMS message producers and consumers can make use of point-to-point messaging queues, for example. To ensure that the queues associated with an address receive messages in a point-to-point manner, you define an **anycast** routing type for the given **address** element in your broker configuration.

When a message is received on an address using **anycast**, the broker locates the queue associated with the address and routes the message to it. A consumer might then request to consume messages from that queue. If multiple consumers connect to the same queue, messages are distributed between the consumers equally, provided that the consumers are equally able to handle them.

The following figure shows an example of point-to-point messaging.



Configuring basic point-to-point messaging

The following procedure shows how to configure an address with a single queue for point-to-point messaging.

Procedure

6. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

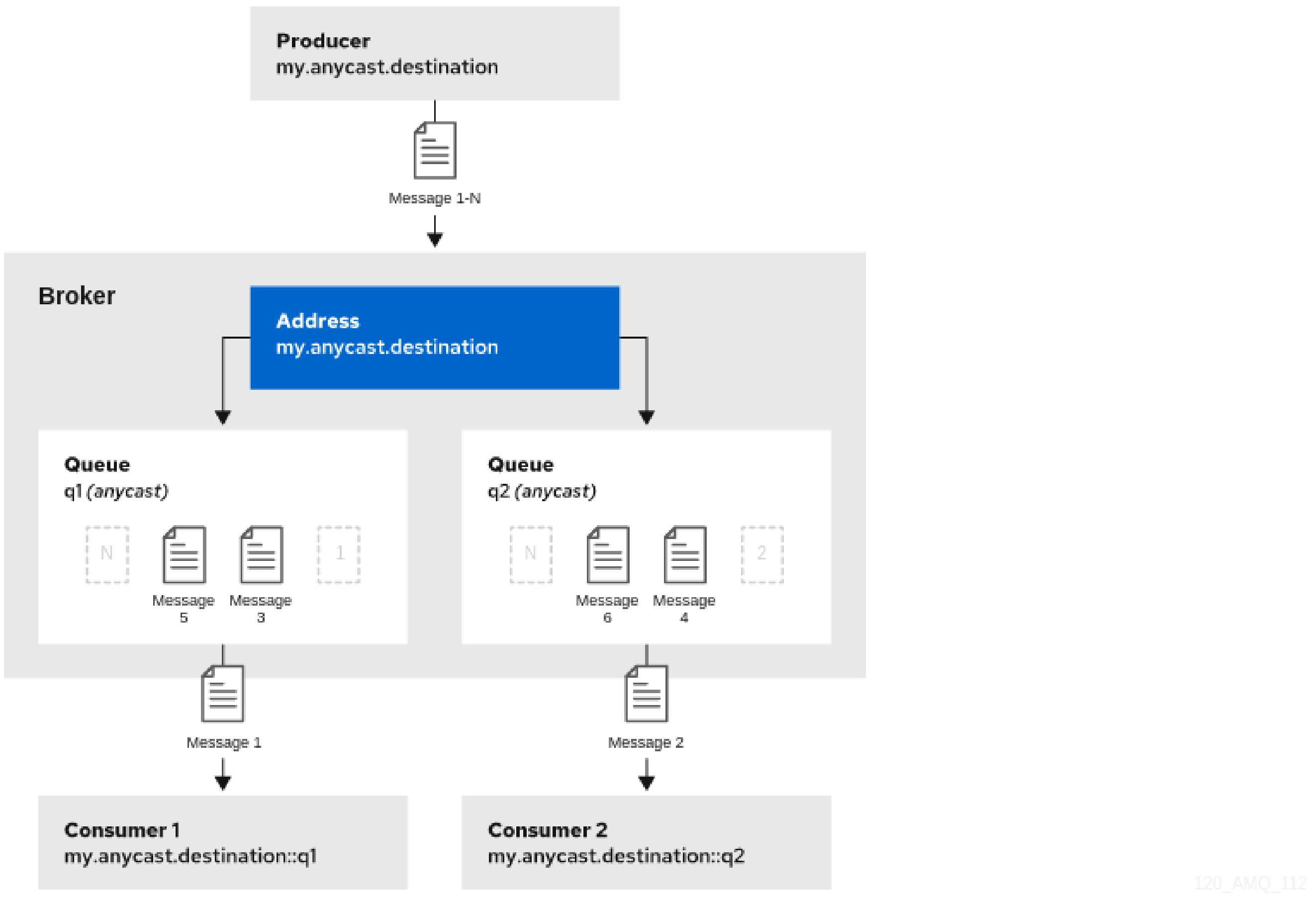
7. Wrap an **anycast** configuration element around the chosen **queue** element of an **address**. Ensure that the values of the **name** attribute for both the **address** and **queue** elements are the same. For example:

```
<configuration ...>
 <core ...>
   ...
   <address name="my.anycast.destination">
     <anycast>
       <queue name="my.anycast.destination"/>
    </anycast>
  </address>
 </core>
</configuration>
```

Configuring point-to-point messaging for multiple queues

You can define more than one queue on an address that uses an **anycast** routing type. The broker distributes messages sent to an **anycast** address evenly across all associated queues. By specifying a *Fully Qualified Queue Name* (FQQN), you can connect a client to a specific queue. If more than one consumer connects to the same queue, the broker distributes messages evenly between the consumers.

The following figure shows an example of point-to-point messaging using two queues.



The following procedure shows how to configure point-to-point messaging for an address that has multiple queues.

Procedure

8. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

9. Wrap an **anycast** configuration element around the **queue** elements in the **address** element. For example:

```
<configuration ...>
  <core ...>
    ...
    <address name="my.anycast.destination">
      <anycast>
        <queue name="q1"/>
        <queue name="q2"/>
      </anycast>
    </address>
  </core>
</configuration>
```

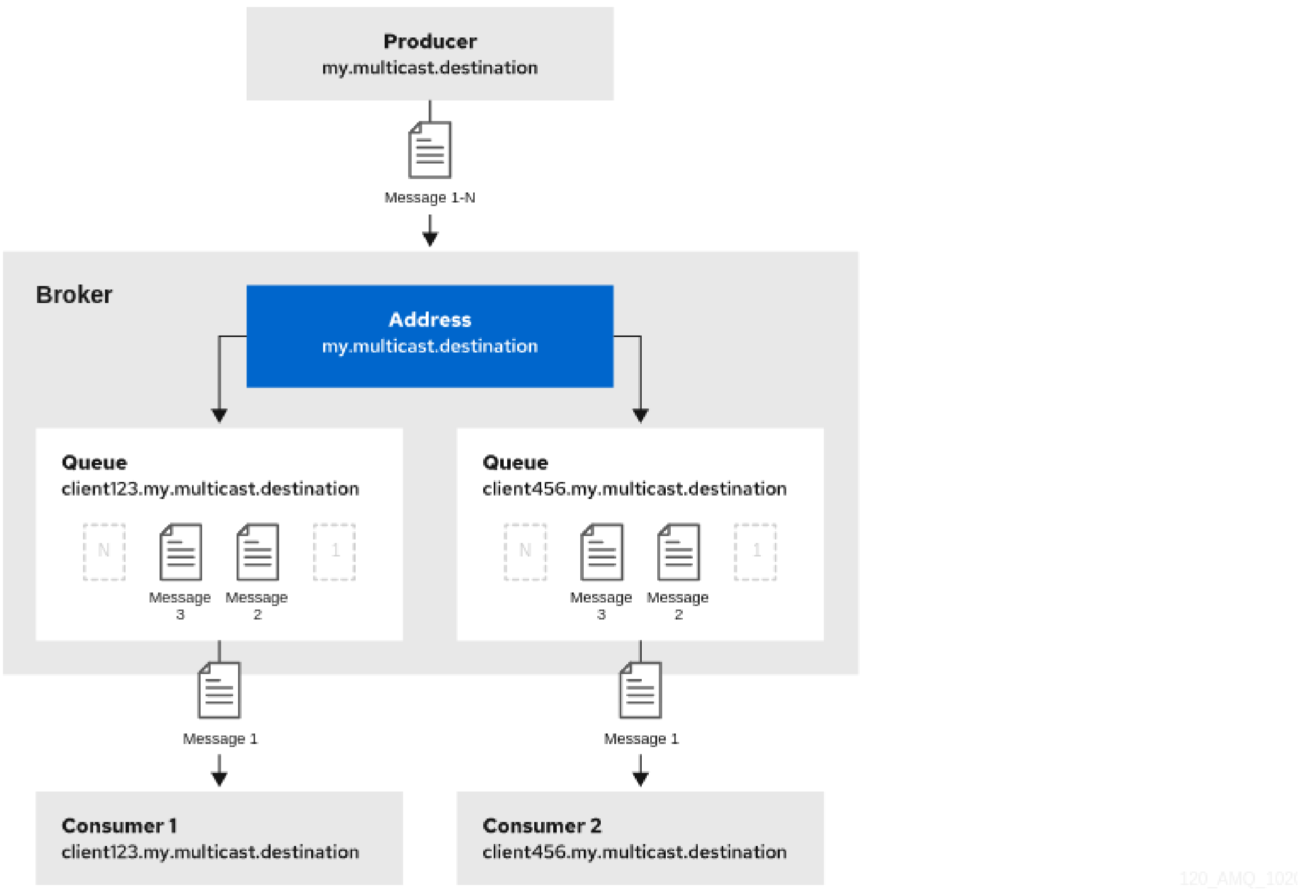
If you have a configuration such as that shown above mirrored across multiple brokers in a cluster, the cluster can load-balance point-to-point messaging in a way that is opaque to producers and consumers. The exact behavior depends on how the message load balancing policy is configured for the cluster.

CONFIGURING ADDRESSES FOR PUBLISH-SUBSCRIBE MESSAGING

In a publish-subscribe scenario, messages are sent to every consumer subscribed to an address. JMS topics and MQTT subscriptions are two examples of publish-subscribe messaging. To ensure that the queues associated with an address receive messages in a publish-subscribe manner, you define a **multicast** routing type for the given **address** element in your broker configuration.

When a message is received on an address with a **multicast** routing type, the broker routes a copy of the message to each queue associated with the address. To reduce the overhead of copying, each queue is sent only a reference to the message, and not a full copy.

The following figure shows an example of publish-subscribe messaging.



The following procedure shows how to configure an address for publish-subscribe messaging.

Procedure

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.
2. Add an empty **multicast** configuration element to the address.

```
<configuration ...>
  <core ...>
    ...
    <address name="my.multicast.destination">
      <multicast/>
    </address>
  </core>
</configuration>
```

3. (Optional) Add one or more **queue** elements to the address and wrap the **multicast** element around them. This step is typically not needed since the broker automatically creates a queue for each subscription requested by a client.

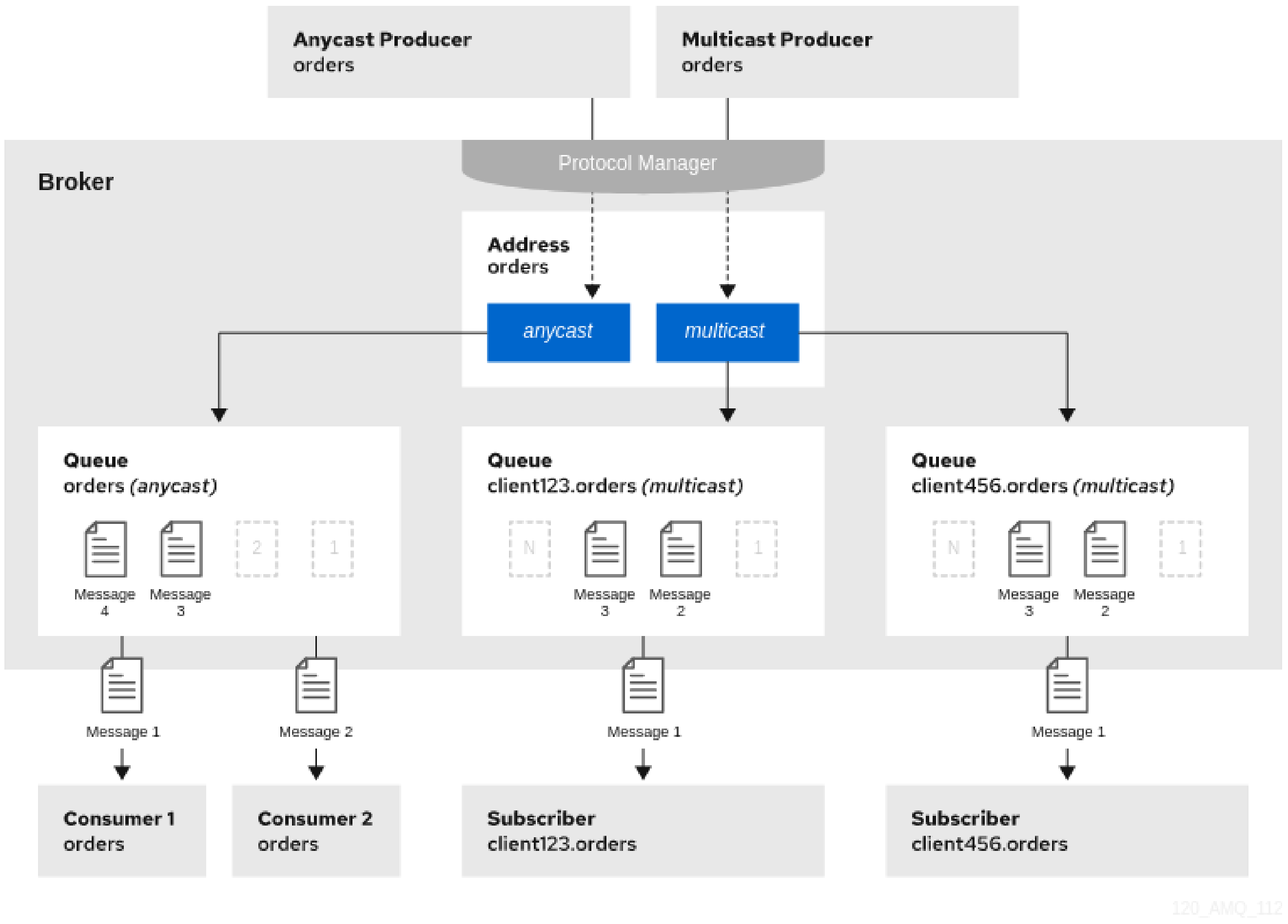
```
<configuration ...>
  <core ...>
    ...
    <address name="my.multicast.destination">
      <multicast>
        <queue  name="client123.my.multicast.destination"/>
        <queue  name="client456.my.multicast.destination"/>
      </multicast>
    </address>
  </core>
</configuration>
```

CONFIGURING AN ADDRESS FOR BOTH POINT-TO-POINT AND PUBLISH-SUBSCRIBE MESSAGING

You can also configure an address with both point-to-point and publish-subscribe semantics.

Configuring an address that uses both point-to-point and publish-subscribe semantics is not typically recommended. However, it *can* be useful when you want, for example, a JMS queue named **orders** and a JMS topic also named **orders**. The different routing types make the addresses appear to be distinct for client connections. In this situation, messages sent by a JMS queue producer use the **anycast** routing type. Messages sent by a JMS topic producer use the **multicast** routing type. When a JMS topic consumer connects to the broker, it is attached to its own subscription queue. A JMS queue consumer, however, is attached to the **anycast** queue.

The following figure shows an example of point-to-point and publish-subscribe messaging used together.



The following procedure shows how to configure an address for both point-to-point and publish-subscribe messaging.

NOTE

The behavior in this scenario is dependent on the protocol being used. For JMS, there is a clear distinction between topic and queue producers and consumers, which makes the logic straightforward. Other protocols like AMQP do not make this distinction. A message being sent via AMQP is routed by both **anycast** and **multicast** and consumers default to **anycast**.

Procedure

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.
2. Wrap an **anycast** configuration element around the **queue** elements in the **address** element. For example:

```
<configuration ...>
 <core ...>
  ...
  <address name="orders">
   <anycast>
    <queue name="orders"/>
   </anycast>
```

```
    </address>
  </core>
</configuration>
```

3. Add an empty **multicast** configuration element to the address.

```
<configuration ...>
  <core ...>
    ...
    <address name="orders">
      <anycast>
        <queue name="orders"/>
      </anycast>
      <multicast/>
    </address>
  </core>
</configuration>
```

NOTE

Typically, the broker creates subscription queues on demand, so there is no need to list specific queue elements inside the **multicast** element.

ADDING A ROUTING TYPE TO AN ACCEPTOR CONFIGURATION

Normally, if a message is received by an address that uses both **anycast** and **multicast**, one of the **anycast** queues receives the message and all of the **multicast** queues. However, clients can specify a special prefix when connecting to an address to specify whether to connect using **anycast** or **multicast**. The prefixes are custom values that are designated using the **anycastPrefix** and **multicastPrefix** parameters within the URL of an acceptor in the broker configuration.

The following procedure shows how to configure prefixes for a given acceptor.

Procedure

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

2. For a given acceptor, to configure an **anycast** prefix, add **anycastPrefix** to the configured URL. Set a custom value. For example:

```
<configuration ...>
  <core ...>
    ...
      <acceptors>
         <!-- Acceptor for every supported protocol -->
         <acceptor name="artemis">tcp://0.0.0.0:61616?
protocols=AMQP;anycastPrefix=anycast://</acceptor>
      </acceptors>
    ...
  </core>
</configuration>
```

Based on the preceding configuration, the acceptor is configured to use **anycast://** for the **anycast** prefix. Client code can specify **anycast://<my.destination>/** if the client needs to send a message to only one of the **anycast** queues.

3. For a given acceptor, to configure a **multicast** prefix, add **multicastPrefix** to the configured URL. Set a custom value. For example:

```
<configuration ...>
  <core ...>
    ...
      <acceptors>
         <!-- Acceptor for every supported protocol -->
         <acceptor name="artemis">tcp://0.0.0.0:61616?
protocols=AMQP;multicastPrefix=multicast://</acceptor>
      </acceptors>
    ...
  </core>
</configuration>
```

Based on the preceding configuration, the acceptor is configured to use **multicast://** for the **multicast** prefix. Client code can specify **multicast://<my.destination>/** if the client needs the message sent to only the **multicast** queues.

CONFIGURING SUBSCRIPTION QUEUES

In *most* cases, it is not necessary to manually create subscription queues because protocol managers create subscription queues automatically when clients first request to subscribe to an address. For durable subscriptions, the generated queue name is usually a concatenation of the client ID and the address.

The following sections show how to manually create subscription queues, when required.

Configuring a durable subscription queue

When a queue is configured as a durable subscription, the broker saves messages for any inactive subscribers and delivers them to the subscribers when they reconnect. Therefore, a client is guaranteed to receive each message delivered to the queue after subscribing to it.

Procedure

10. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

11. Add the **durable** configuration element to a chosen queue. Set a value of **true**.

```
<configuration ...>
  <core ...>
    ...
    <address name="my.durable.address">
      <multicast>
        <queue name="q1">
          <durable>true</durable>
        </queue>
     </multicast>
    </address>
  </core>
</configuration>
```

NOTE

Because queues are durable by default, including the **durable** element and setting the value to **true** is not strictly necessary to create a durable queue. However, explicitly including the element enables you to later change the behavior of the queue to non-durable, if necessary.

Configuring a non-shared durable subscription queue

The broker can be configured to prevent more than one consumer from connecting to a queue at any one time. Therefore, subscriptions to queues configured this way are regarded as "non-shared".

Procedure

12. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

13. Add the **durable** configuration element to each chosen queue. Set a value of **true**.

```
<configuration ...>
  <core ...>
    ...
    <address name="my.non.shared.durable.address">
      <multicast>
        <queue name="orders1">
        <durable>true</durable>
        </queue>
        <queue name="orders2">
          <durable>true</durable>
        </queue>
      </multicast>
    </address>
  </core>
</configuration>
```

NOTE

Because queues are durable by default, including the **durable** element and setting the value to **true** is not strictly necessary to create a durable queue. However, explicitly including the element enables you to later change the behavior of the queue to non-durable, if necessary.

14. Add the **max-consumers** attribute to each chosen queue. Set a value of **1**.

```
<configuration ...>
  <core ...>
    ...
    <address name="my.non.shared.durable.address">
      <multicast>
        <queue name="orders1" max-consumers="1">
          <durable>true</durable>
      </queue>
      <queue name="orders2" max-consumers="1">
        <durable>true</durable>
      </queue>
    </multicast>
  </address>
 </core>
</configuration>
```

Configuring a non-durable subscription queue

Non-durable subscriptions are usually managed by the relevant protocol manager, which creates and deletes temporary queues.

However, if you want to manually create a queue that behaves like a non-durable subscription queue, you can use the **purge-on-no-consumers** attribute on the queue. When **purge-on-no-consumers** is set to **true**, the queue does not start receiving messages until a consumer is connected. In addition, when the last consumer is disconnected from the queue, the queue is *purged* (that is, its messages are removed). The queue does not receive any further messages until a new consumer is connected to the queue.

Procedure

15. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

16. Add the **purge-on-no-consumers** attribute to each chosen queue. Set a value of **true**.

```
<configuration ...>
  <core ...>
    ...
    <address name="my.non.durable.address">
        <multicast>
            <queue name="orders1" purge-on-no-consumers="true"/>
        </multicast>
    </address>
  </core>
</configuration>
```

CREATING AND DELETING ADDRESSES AND QUEUES AUTOMATICALLY

You can configure the broker to automatically create addresses and queues, and to delete them after they are no longer in use. This saves you from having to pre-configure each address before a client can connect to it.

Configuration options for automatic queue creation and deletion

The following table lists the configuration elements available when configuring an **address-setting** element to automatically create and delete queues and addresses.

If you want the **address-setting** to…	Add this configuration…
Create addresses when a client sends a message to or attempts to consume a message from a queue mapped to an address that does not exist.	**auto-create-addresses**
Create a queue when a client sends a message to or attempts to consume a message from a queue.	**auto-create-queues**
Delete an automatically created address when it no longer has any queues.	**auto-delete-addresses**
Delete an automatically created queue when the queue has 0 consumers and 0 messages.	**auto-delete-queues**
Use a specific routing type if the client does not specify one.	**default-address-routing-type**

Configuring automatic creation and deletion of addresses and queues

The following procedure shows how to configure automatic creation and deletion of addresses and queues.

Procedure

17. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

18. Configure an **address-setting** for automatic creation and deletion. The following example uses all of the configuration elements mentioned in the previous table.

```
<configuration ...>
 <core ...>
  ...
  <address-settings>
    <address-setting match="activemq.#">
      <auto-create-addresses>true</auto-create-addresses>
      <auto-delete-addresses>true</auto-delete-addresses>
      <auto-create-queues>true</auto-create-queues>
      <auto-delete-queues>true</auto-delete-queues>
      <default-address-routing-type>ANYCAST</default-address-routing-type>
  </address-setting>
</address-settings>
  ...
 </core>
</configuration>
```

address-setting

The configuration of the **address-setting** element is applied to any address or queue that matches the wildcard address **activemq.#**.

auto-create-addresses

When a client requests to connect to an address that does not yet exist, the broker creates the address.

auto-delete-addresses

An automatically created address is deleted when it no longer has any queues associated with it.

auto-create-queues

When a client requests to connect to a queue that does not yet exist, the broker creates the queue.

auto-delete-queues

An automatically created queue is deleted when it no longer has any consumers or messages.

default-address-routing-type

If the client does not specify a routing type when connecting, the broker uses **ANYCAST** when delivering messages to an address. The default value is **MULTICAST**.

Protocol managers and addresses

A component called a *protocol manager* maps protocol-specific concepts to concepts used in the AMQ Broker address model; queues and routing types. In certain situations, a protocol manager might automatically create queues on the broker.

For example, when a client sends an MQTT subscription packet with the addresses **/house/room1/lights** and **/house/room2/lights**, the MQTT protocol manager understands that the two addresses require **multicast** semantics. Therefore, the protocol manager first looks to ensure that **multicast** is enabled for both addresses. If not, it attempts to dynamically create them. If successful, the protocol manager then creates special subscription queues for each subscription requested by the client.

Each protocol behaves slightly differently. The table below describes what typically happens when subscribe frames to various types of **queue** are requested.

If the queue is of this type…	The typical action for a protocol manager is to…
Durable subscription queue	Look for the appropriate address and ensures that **multicast** semantics is enabled. It then creates a special subscription queue with the client ID and the address as its name and **multicast** as its routing type. The special name allows the protocol manager to quickly identify the required client subscription queues should the client disconnect and reconnect at a later date. When the client unsubscribes the queue is deleted.

If the queue is of this type…	The typical action for a protocol manager is to…
Temporary subscription queue	Look for the appropriate address and ensures that **multicast** semantics is enabled. It then creates a queue with a random (read UUID) name under this address with **multicast** routing type. When the client disconnects the queue is deleted.
Point-to-point queue	Look for the appropriate address and ensures that **anycast** routing type is enabled. If it is, it aims to locate a queue with the same name as the address. If it does not exist, it looks for the first queue available. It this does not exist then it automatically creates the queue (providing auto create is enabled). The queue consumer is bound to this queue. If the queue is auto created, it is automatically deleted once there are no consumers and no messages in it.

SPECIFYING A FULLY QUALIFIED QUEUE NAME

Internally, the broker maps a client’s request for an address to specific queues. The broker decides on behalf of the client to which queues to send messages, or from which queue to receive messages. However, more advanced use cases might require that the client specifies a queue name directly. In these situations the client can use a *fully qualified queue name* (FQQN). An FQQN includes both the address name and the queue name, separated by a **::**.

The following procedure shows how to specify an FQQN when connecting to an address with multiple queues.

Prerequisites

- You have an address configured with two or more queues, as shown in the example below.

```
<configuration ...>
  <core ...>
    ...
    <addresses>
       <address name="my.address">
          <anycast>
             <queue name="q1" />
             <queue name="q2" />
          </anycast>
      </address>
   </addresses>
  </core>
</configuration>
```

Procedure

- In the client code, use both the address name and the queue name when requesting a connection from the broker. Use two colons, **::**, to separate the names. For example:

```
String FQQN = "my.address::q1";
Queue q1 session.createQueue(FQQN);
MessageConsumer consumer = session.createConsumer(q1);
```

CONFIGURING SHARDED QUEUES

A common pattern for processing of messages across a queue where only partial ordering is required is to use *queue sharding*. This means that you define an **anycast** address that acts as a single logical queue, but which is backed by many underlying physical queues.

Procedure

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

2. Add an **address** element and set the **name** attribute. For example:

```
<configuration ...>
  <core ...>
    ...
    <addresses>
       <address name="my.sharded.address"></address>
    </addresses>
  </core>
</configuration>
```

3. Add the **anycast** routing type and include the desired number of sharded queues. In the example below, the queues **q1**, **q2**, and **q3** are added as **anycast** destinations.

```
<configuration ...>
  <core ...>
    ...
    <addresses>
       <address  name="my.sharded.address">
          <anycast>
             <queue name="q1" />
             <queue name="q2" />
             <queue name="q3" />
          </anycast>
      </address>
   </addresses>
</core>
</configuration>
```

Based on the preceding configuration, messages sent to **my.sharded.address** are distributed equally across **q1**, **q2** and **q3**. Clients are able to connect directly to a specific physical queue when using a Fully Qualified Queue Name (FQQN). and receive messages sent to that specific queue only.

To tie particular messages to a particular queue, clients can specify a message group for each message. The broker routes grouped messages to the same queue, and one consumer processes them all.

CONFIGURING LAST VALUE QUEUES

A *last value queue* is a type of queue that discards messages in the queue when a newer message with the same last value key value is placed in the queue. Through this behavior, last value queues retain only the last values for messages of the same key.

A simple use case for a last value queue is for monitoring stock prices, where only the latest value for a particular stock is of interest.

NOTE

If a message without a configured last value key is sent to a last value queue, the broker handles this message as a "normal" message. Such messages are not purged from the queue when a new message with a configured last value key arrives.

You can configure last value queues individually, or for all of the queues associated with a set of addresses.

The following procedures show how to configure last value queues in these ways.

Configuring last value queues individually

The following procedure shows to configure last value queues individually.

19. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

20. For a given queue, add the **last-value-key** key and specify a custom value. For example:

```
<address name="my.address">
   <multicast>
      <queue name="prices1" last-value-key="stock_ticker"/>
   </multicast>
</address>
```

21. Alternatively, you can configure a last value queue that uses the default last value key name of **_AMQ_LVQ_NAME**. To do this, add the **last-value** key to a given queue. Set the value to **true**. For example:

```
<address name="my.address">
   <multicast>
      <queue name="prices1" last-value="true"/>
   </multicast>
</address>
```

Configuring last value queues for addresses

The following procedure shows to configure last value queues for an address or *set* of addresses.

22. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

23. In the **address-setting** element, for a matching address, add **default-last-value-key**. Specify a custom value. For example:

```
<address-setting match="lastValue">
  <default-last-value-key>stock_ticker</default-last-value-key>
</address-setting>
```

Based on the preceding configuration, all queues associated with the **lastValue** address use a last value key of **stock_ticker**. By default, the value of **default-last-value-key** is not set.

24. To configure last value queues for a *set* of addresses, you can specify an address wildcard. For example:

```
<address-setting match="lastValue.*">
  <default-last-value-key>stock_ticker</default-last-value-key>
</address-setting>
```

25. Alternatively, you can configure all queues associated with an address or *set* of addresses to use the default last value key name of **_AMQ_LVQ_NAME**. To do this, add **default-last-value-queue** instead of **default-last-value-key**. Set the value to **true**. For example:

```
<address-setting match="lastValue">
  <default-last-value-queue>true</default-last-value-queue>
</address-setting>
```

Example of last value queue behavior

This example shows the behavior of a last value queue.

In your **broker.xml** configuration file, suppose that you have added configuration that looks like the following:

```
<address name="my.address">
   <multicast>
      <queue name="prices1" last-value-key="stock_ticker"/>
   </multicast>
</address>
```

The preceding configuration creates a queue called **prices1**, with a last value key of **stock_ticker**.

Now, suppose that a client sends two messages. Each message has the same value of **ATN** for the property **stock_ticker**. Each message has a different value for a property called **stock_price**. Each message is sent to the same queue, **prices1**.

```
TextMessage message = session.createTextMessage("First message with last value property set");
message.setStringProperty("stock_ticker", "ATN");
message.setStringProperty("stock_price", "36.83");
producer.send(message);
```

```
TextMessage message = session.createTextMessage("Second message with last value property
set");
message.setStringProperty("stock_ticker",   "ATN");
message.setStringProperty("stock_price",   "37.02");
producer.send(message);
```

When two messages with the same value for the **stock_ticker** last value key (in this case, **ATN**) arrive to the **prices1 queue**, only the latest message remains in the queue, with the first message being purged. At the command line, you can enter the following lines to validate this behavior:

```
TextMessage messageReceived = (TextMessage)messageConsumer.receive(5000);
System.out.format("Received message: %s\n", messageReceived.getText());
```

In this example, the output you see is the second message, since both messages use the same value for the last value key and the second message was received in the queue after the first.

Enforcing non-destructive consumption for last value queues

When a consumer connects to a queue, the normal behavior is that messages sent to that consumer are acquired exclusively by the consumer. When the consumer acknowledges receipt of the messages, the broker removes the messages from the queue.

As an alternative to the normal consumption behaviour, you can configure a queue to enforce *non-destructive* consumption. In this case, when a queue sends a message to a consumer, the message can still be received by other consumers. In addition, the message remains in the queue even when a consumer has consumed it. When you enforce this non-destructive consumption behavior, the consumers are known as queue *browsers*.

Enforcing non-destructive consumption is a useful configuration for last value queues, because it ensures that the queue always holds the latest value for a particular last value key.

The following procedure shows how to enforce non-destructive consumption for a last value queue.

Prerequisites

- You have already configured last-value queues individually, or for all queues associated with an address or *set* of addresses.

Procedure

26. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

27. If you previously configured a queue individually as a last value queue, add the **non-destructive** key. Set the value to **true**. For example:

```
<address name="my.address">
  <multicast>
    <queue name="orders1" last-value-key="stock_ticker" non-destructive="true" />
  </multicast>
</address>
```

28. If you previously configured an address or *set* of addresses for last value queues, add the **default-non-destructive** key. Set the value to **true**. For example:

```
<address-setting match="lastValue">
  <default-last-value-key>stock_ticker </default-last-value-key>
  <default-non-destructive>true</default-non-destructive>
</address-setting>
```

NOTE

By default, the value of **default-non-destructive** is **false**.

MOVING EXPIRED MESSAGES TO AN EXPIRY ADDRESS

For a queue other than a last value queue, if you have only non-destructive consumers, the broker never deletes messages from the queue, causing the queue size to increase over time. To prevent this unconstrained growth in queue size, you can configure when messages expire and specify an address to which the broker moves expired messages.

Configuring message expiry

The following procedure shows how to configure message expiry.

Procedure

29. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

30. In the **core** element, set the **message-expiry-scan-period** to specify how frequently the broker scans for expired messages.

```
<configuration ...>
  <core ...>
    ...
    <message-expiry-scan-period>1000</message-expiry-scan-period>
    ...
```

Based on the preceding configuration, the broker scans queues for expired messages every 1000 milliseconds.

31. In the **address-setting** element for a matching address or *set* of addresses, specify an expiry address. Also, set a message expiration time. For example:

```
<configuration ...>
  <core ...>
    ...
    <address-settings>
      ...
      <address-setting match="stocks">
        ...
        <expiry-address>ExpiryAddress</expiry-address>
        <expiry-delay>10</expiry-delay>
        ...
      </address-setting>
    ...
  <address-settings>
<configuration ...>
```

expiry-address

Expiry address for the matching address or addresses. In the preceding example, the broker sends expired messages for the **stocks** address to an expiry address called **ExpiryAddress**.

expiry-delay

Expiration time, in milliseconds, that the broker applies to messages that are using the default expiration time. By default, messages have an expiration time of **0**, meaning that they don't expire. For messages with an expiration time greater than the default, **expiry-delay** has no effect.

For example, suppose you set **expiry-delay** on an address to **10**, as shown in the preceding example. If a message with the default expiration time of **0** arrives to a queue at this address, then the broker changes the expiration time of the message from **0** to **10**. However, if another message that is using an expiration time of **20** arrives, then its expiration time is unchanged. If you set expiry-delay to **-1**, this feature is disabled. By default, **expiry-delay** is set to **-1**.

32. Alternatively, instead of specifying a value for **expiry-delay**, you can specify minimum and maximum expiry delay values. For example:

```
<configuration ...>
  <core ...>
    ...
    <address-settings>
      ...
      <address-setting match="stocks">
        ...
        <expiry-address>ExpiryAddress</expiry-address>
        <min-expiry-delay>10</min-expiry-delay>
        <max-expiry-delay>100</max-expiry-delay>
        ...
      </address-setting>
      ...
    <address-settings>
<configuration ...>
```

min-expiry-delay

Minimum expiration time, in milliseconds, that the broker applies to messages.

max-expiry-delay

Maximum expiration time, in milliseconds, that the broker applies to messages.

The broker applies the values of **min-expiry-delay** and **max-expiry-delay** as follows:

- For a message with the default expiration time of **0**, the broker sets the expiration time to the specified value of **max-expiry-delay**. If you have not specified a value for **max-expiry-delay**, the broker sets the expiration time to the specified value of **min-expiry-delay**. If you have not specified a value for **min-expiry-delay**, the broker does not change the expiration time of the message.

- For a message with an expiration time above the value of **max-expiry-delay**, the broker sets the expiration time to the specified value of **max-expiry-delay**.

- For a message with an expiration time below the value of **min-expiry-delay**, the broker sets the expiration time to the specified value of **min-expiry-delay**.
- For a message with an expiration between the values of **min-expiry-delay** and **max-expiry-delay**, the broker does not change the expiration time of the message.
- If you specify a value for **expiry-delay** (that is, other than the default value of **-1**), this overrides any values that you specify for **min-expiry-delay** and **max-expiry-delay**.
- The default value for both **min-expiry-delay** and **max-expiry-delay** is **-1** (that is, disabled).

33. In the **addresses** element of your configuration file, configure the address previously specified for **expiry-address**. Define a queue at this address. For example:

```
<addresses>
   ...
   <address name="ExpiryAddress">
      <anycast>
         <queue name="ExpiryQueue"/>
      </anycast>
   </address>
   ...
</addresses>
```

The preceding example configuration associates an expiry queue, **ExpiryQueue**, with the expiry address, **ExpiryAddress**.

Creating expiry resources automatically

A common use case is to segregate expired messages according to their original addresses. For example, you might choose to route expired messages from an address called **stocks** to an expiry queue called **EXP.stocks**. Likewise, you might route expired messages from an address called **orders** to an expiry queue called **EXP.orders**.

This type of routing pattern makes it easy to track, inspect, and administer expired messages. However, a pattern such as this is difficult to implement in an environment that uses mainly automatically-created addresses and queues. In this type of environment, an administrator does not want the extra effort required to manually create addresses and queues to hold expired messages.

As a solution, you can configure the broker to automatically create resources (that is, addressees and queues) to handle expired messages for a given address or *set* of addresses. The following procedure shows an example.

Prerequisites

- You have already configured an expiry address for a given address or *set* of addresses.

Procedure

34. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

35. Locate the **<address-setting>** element that you previously added to the configuration file to define an expiry address for a matching address or *set* of addresses. For example:

```
<configuration ...>

  <core ...>
    ...
    <address-settings>
      ...
      <address-setting match="stocks">
        ...
        <expiry-address>ExpiryAddress</expiry-address>
        ...
      </address-setting>
      ...
    <address-settings>
<configuration ...>
```

36. In the **<address-setting>** element, add configuration items that instruct the broker to automatically create expiry resources (that is, addresses and queues) and how to name these resources. For example:

```
<configuration ...>
  <core ...>
    ...
    <address-settings>
      ...
      <address-setting match="stocks">
        ...
        <expiry-address>ExpiryAddress</expiry-address>
        <auto-create-expiry-resources>true</auto-create-expiry-resources>
        <expiry-queue-prefix>EXP.</expiry-queue-prefix>
        <expiry-queue-suffix></expiry-queue-suffix>
        ...
      </address-setting>
      ...
    <address-settings>
<configuration ...>
```

auto-create-expiry-resources

Specifies whether the broker automatically creates an expiry address and queue to receive expired messages. The default value is **false**.
If the parameter value is set to **true**, the broker automatically creates an **<address>** element that defines an expiry address and an associated expiry queue. The name value of the automatically-created **<address>** element matches the name value specified for **<expiry-address>**.

The automatically-created expiry queue has the **multicast** routing type. By default, the broker names the expiry queue to match the address to which expired messages were originally sent, for example, **stocks**.

The broker also defines a filter for the expiry queue that uses the **_AMQ_ORIG_ADDRESS** property. This filter ensures that the expiry queue receives only messages sent to the corresponding original address.

expiry-queue-prefix

Prefix that the broker applies to the name of the automatically-created expiry queue. The default value is **EXP.**
When you define a prefix value or keep the default value, the name of the expiry queue is a concatenation of the prefix and the original address, for example, **EXP.stocks**.

expiry-queue-suffix

Suffix that the broker applies to the name of an automatically-created expiry queue. The default value is not defined (that is, the broker applies no suffix).

You can directly access the expiry queue using either the queue name by itself (for example, when using the AMQ Broker Core Protocol JMS client) or using the fully qualified queue name (for example, when using another JMS client).

NOTE

Because the expiry address and queue are automatically created, any address settings related to deletion of automatically-created addresses and queues also apply to these expiry resources.

MOVING UNDELIVERED MESSAGES TO A DEAD LETTER ADDRESS

If delivery of a message to a client is unsuccessful, you might not want the broker to make ongoing attempts to deliver the message. To prevent infinite delivery attempts, you can define a *dead letter address* and one or more asscociated *dead letter queues*. After a specified number of delivery attempts, the broker removes an undelivered message from its original queue and sends the message to the configured dead letter address. A system administrator can later consume undelivered messages from a dead letter queue to inspect the messages.

If you do not configure a dead letter address for a given queue, the broker permanently removes undelivered messages from the queue after the specified number of delivery attempts.

Undelivered messages that are consumed from a dead letter queue have the following properties:

_AMQ_ORIG_ADDRESS

String property that specifies the original address of the message

_AMQ_ORIG_QUEUE

String property that specifies the original queue of the message

Configuring a dead letter address

The following procedure shows how to configure a dead letter address and an associated dead letter queue.

Procedure

37. Open the ***<broker_instance_dir>*****/etc/broker.xml** configuration file.

38. In an **<address-setting>** element that matches your queue name(s), set values for the dead letter address name and the maximum number of delivery attempts. For example:

```
<configuration ...>
  <core ...>
    ...
    <address-settings>
      ...
      <address-setting  match="exampleQueue">
        <dead-letter-address>DLA</dead-letter-address>
        <max-delivery-attempts>3</max-delivery-attempts>
      </address-setting>
    ...
    <address-settings>
<configuration ...>
```

match

Address to which the broker applies the configuration in this **address-setting** section. You can specify a wildcard expression for the **match** attribute of the **<address-setting>** element. Using a wildcard expression is useful if you want to associate the dead letter settings configured in the **<address-setting>** element with a matching *set* of addresses.

dead-letter-address

Name of the dead letter address. In this example, the broker moves undelivered messages from the queue *exampleQueue* to the dead letter address, *DLA*.

max-delivery-attempts

Maximum number of delivery attempts made by the broker before it moves an undelivered message to the configured dead letter address. In this example, the broker moves undelivered messages to the dead letter address after three unsuccessful delivery attempts. The default value is **10**. If you want the broker to make an infinite number of redelivery attempts, specify a value of **-1**.

39. In the **addresses** section, add an **address** element for the dead letter address, *DLA*. To associate a dead letter queue with the dead letter address, specify a name value for **queue**. For example:

```
<configuration ...>
  <core ...>
    ...
    <addresses>
      <address name="DLA">
        <anycast>
          <queue name="DLQ" />
        </anycast>
      </address>
    ...
    </addresses>
  </core>
</configuration>
```

In the preceding configuration, you associate a dead letter queue named *DLQ* with the dead letter address, *DLA*.

Creating dead letter queues automatically

A common use case is to segregate undelivered messages according to their original addresses. For example, you might choose to route undelivered messages from an address called **stocks** to a dead letter queue called **DLA.stocks** that has an associated dead letter queue called **DLQ.stocks**. Likewise, you might route undelivered messages from an address called **orders** to a dead letter address called **DLA.orders**.

This type of routing pattern makes it easy to track, inspect, and administrate undelivered messages. However, a pattern such as this is difficult to implement in an environment that uses mainly automatically-created addresses and queues. It is likely that a system administrator for this type of environment does not want the additional effort required to manually create addresses and queues to hold undelivered messages.

As a solution, you can configure the broker to automatically create addressees and queues to handle undelivered messages, as shown in the procedure that follows.

Prerequisites

- You have already configured a dead letter address for a queue or set of queues.

Procedure

40. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

41. Locate the **<address-setting>** element that you previously added to define a dead letter address for a matching queue or set of queues. For example:

```
<configuration ...>
  <core ...>
    ...
    <address-settings>
      ...
      <address-setting  match="exampleQueue">
        <dead-letter-address>DLA</dead-letter-address>
        <max-delivery-attempts>3</max-delivery-attempts>
      </address-setting>
    ...
    <address-settings>
<configuration ...>
```

42. In the **<address-setting>** element, add configuration items that instruct the broker to automatically create dead letter resources (that is, addresses and queues) and how to name these resources. For example:

```
<configuration ...>
  <core ...>
    ...
    <address-settings>
      ...
      <address-setting  match="exampleQueue">
```

```
        <dead-letter-address>DLA</dead-letter-address>
        <max-delivery-attempts>3</max-delivery-attempts>
        <auto-create-dead-letter-resources>true</auto-create-dead-letter-resources>
        <dead-letter-queue-prefix>DLQ.</dead-letter-queue-prefix>
        <dead-letter-queue-suffix></dead-letter-queue-suffix>
      </address-setting>
    ...
    <address-settings>
<configuration ...>
```

auto-create-dead-letter-resources

Specifies whether the broker automatically creates a dead letter address and queue to receive undelivered messages. The default value is **false**.
If **auto-create-dead-letter-resources** is set to **true**, the broker automatically creates an **<address>** element that defines a dead letter address and an associated dead letter queue. The name of the automatically-created **<address>** element matches the name value that you specify for **<dead-letter-address>**.

The dead letter queue that the broker defines in the automatically-created **<address>** element has the **multicast** routing type . By default, the broker names the dead letter queue to match the original address of the undelivered message, for example, **stocks**.

The broker also defines a filter for the dead letter queue that uses the **_AMQ_ORIG_ADDRESS** property. This filter ensures that the dead letter queue receives only messages sent to the corresponding original address.

dead-letter-queue-prefix

Prefix that the broker applies to the name of an automatically-created dead letter queue. The default value is **DLQ.**
When you define a prefix value or keep the default value, the name of the dead letter queue is a concatenation of the prefix and the original address, for example, **DLQ.stocks**.

dead-letter-queue-suffix

Suffix that the broker applies to an automatically-created dead letter queue. The default value is not defined (that is, the broker applies no suffix).

ANNOTATIONS AND PROPERTIES ON EXPIRED OR UNDELIVERED AMQP MESSAGES

Before the broker moves an expired or undelivered AMQP message to an expiry or dead letter queue that you have configured, the broker applies annotations and properties to the message. A client can create a filter based on these properties or annotations, to select particular messages to consume from the expiry or dead letter queue.

NOTE

The properties that the broker applies are *internal* properties These properties are not exposed to clients for regular use, but can be specified by a client in a filter.

The following table shows the annotations and internal properties that the broker applies to expired or undelivered AMQP messages.

Annotation name	Internal property name	Description
x-opt-ORIG-MESSAGE-ID	_AMQ_ORIG_MESSAGE_ID	Original message ID, before the message was moved to an expiry or dead letter queue.
x-opt-ACTUAL-EXPIRY	_AMQ_ACTUAL_EXPIRY	Message expiry time, specified as the number of milliseconds since the last epoch started.
x-opt-ORIG-QUEUE	_AMQ_ORIG_QUEUE	Original queue name of the expired or undelivered message.
x-opt-ORIG-ADDRESS	_AMQ_ORIG_ADDRESS	Original address name of the expired or undelivered message.

DISABLING QUEUES

If you manually define a queue in your broker configuration, the queue is enabled by default.

However, there might be a case where you want to define a queue so that clients can subscribe to it, but are not ready to use the queue for message routing. Alternatively, there might be a situation where you want to stop message flow to a queue, but still keep clients bound to the queue. In these cases, you can disable the queue.

The following example shows how to disable a queue that you have defined in your broker configuration.

Prerequisites

- You should be familiar with how to define an address and associated queue in your broker configuration.

Procedure

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

2. For a queue that you previously defined, add the **enabled** attribute. To disable the queue, set the value of this attribute to **false**. For example:

```
<addresses>
  <address name="orders">
    <multicast>
      <queue name="orders" enabled="false"/>
    </multicast>
  </address>
</addresses>
```

The default value of the **enabled** property is **true**. When you set the value to **false**, message routing to the queue is disabled.

NOTE

If you disable all queues on an address, any messages sent to that address are silently dropped.

LIMITING THE NUMBER OF CONSUMERS CONNECTED TO A QUEUE

Limit the number of consumers connected to a particular queue by using the **max-consumers** attribute. Create an exclusive consumer by setting **max-consumers** flag to **1**. The default value is **-1**, which sets an unlimited number of consumers.

The following procedure shows how to set a limit on the number of consumers that can connect to a queue.

Procedure

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

2. For a given queue, add the **max-consumers** key and set a value.

```
<configuration ...>
  <core ...>
    ...
    <addresses>
       <address name="my.address">
         <anycast>
           <queue name="q3" max-consumers="20"/>
         </anycast>
      </address>
    </addresses>
  </core>
</configuration>
```

   Based on the preceding configuration, only 20 consumers can connect to queue **q3** at the same time.

3. To create an exclusive consumer, set **max-consumers** to **1**.

```
<configuration ...>
  <core ...>
    ...
    <address name="my.address">
      <anycast>
        <queue name="q3" max-consumers="1"/>
      </anycast>
    </address>
  </core>
</configuration>
```

4. To allow an unlimited number of consumers, set **max-consumers** to **-1**.

```
<configuration ...>
  <core ...>
    ...
    <address name="my.address">
      <anycast>
         <queue name="q3" max-consumers="-1"/>
     </anycast>
   </address>
  </core>
</configuration>
```

CONFIGURING EXCLUSIVE QUEUES

Exclusive queues are special queues that route all messages to only one consumer at a time. This configuration is useful when you want all messages to be processed serially by the same consumer. If there are multiple consumers for a queue, only one consumer will receive messages. If that consumer disconnects from the queue, another consumer is chosen.

Configuring exclusive queues individually

The following procedure shows to how to individually configure a given queue as exclusive.

Procedure

43. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

44. For a given queue, add the **exclusive** key. Set the value to **true**.

```
<configuration ...>
  <core ...>
    ...
    <address name="my.address">
      <multicast>
        <queue name="orders1" exclusive="true"/>
      </multicast>
    </address>
  </core>
</configuration>
```

Configuring exclusive queues for addresses

The following procedure shows how to configure an address or *set* of addresses so that all associated queues are exclusive.

45. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

46. In the **address-setting** element, for a matching address, add the **default-exclusive-queue** key. Set the value to **true**.

```
<address-setting match="myAddress">
    <default-exclusive-queue>true</default-exclusive-queue>
</address-setting>
```

Based on the preceding configuration, all queues associated with the **myAddress** address are exclusive. By default, the value of **default-exclusive-queue** is **false**.

47. To configure exclusive queues for a set of addresses, you can specify an address wildcard. For example:

```
<address-setting match="myAddress.*">
   <default-exclusive-queue>true</default-exclusive-queue>
</address-setting>
```

APPLYING SPECIFIC ADDRESS SETTINGS TO TEMPORARY QUEUES

When using JMS, for example, the broker creates *temporary queues* by assigning a universally unique identifier (UUID) as both the address name and the queue name.

The default **<address-setting match="#">** applies the configured address settings to *all* queues, including temporary ones. If you want to apply specific address settings to temporary queues only, you can optionally specify a **temporary-queue-namespace** as described below. You can then specify address settings that match the namespace and the broker applies those settings to all temporary queues.

When a temporary queue is created and a temporary queue namespace exists, the broker prepends the **temporary-queue-namespace** value and the configured delimiter (default **.**) to the address name. It uses that to reference the matching address settings.

Procedure

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

2. Add a **temporary-queue-namespace** value. For example:

```
<temporary-queue-namespace>temp-example</temporary-queue-namespace>
```

3. Add an **address-setting** element with a **match** value that corresponds to the temporary queues namespace. For example:

```
 <address-settings>
    <address-setting match="temp-example.#">
    <enable-metrics>false</enable-metrics>
  </address-setting>
</address-settings>
```

This example disables metrics in all temporary queues created by the broker.

NOTE

Specifying a temporary queue namespace does not affect temporary queues. For example, the namespace does not change the names of temporary queues. The namespace is used to reference the temporary queues.

CONFIGURING RING QUEUES

Generally, queues in AMQ Broker use first-in, first-out (FIFO) semantics. This means that the broker adds messages to the tail of the queue and removes them from the head. A ring queue is a special type of queue that holds a specified, fixed number of messages. The broker maintains the fixed queue size by removing the message at the head of the queue when a new message arrives but the queue already holds the specified number of messages.

For example, consider a ring queue configured with a size of **3** and a producer that sequentially sends messages **A**, **B**, **C**, and **D**. Once message **C** arrives to the queue, the number of messages in the queue has reached the configured ring size. At this point, message **A** is at the head of the queue, while message **C** is at the tail. When message **D** arrives to the queue, the broker adds the message to the tail of the queue. To maintain the fixed queue size, the broker removes the message at the head of the queue (that is, message **A**). Message **B** is now at the head of the queue.

Configuring ring queues

The following procedure shows how to configure a ring queue.

Procedure

48. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

49. To define a default ring size for all queues on matching addresses that don't have an explicit ring size set, specify a value for **default-ring-size** in the **address-setting** element. For example:

```
  <address-settings>
    <address-setting match="ring.#">
      <default-ring-size>3</default-ring-size>
   </address-setting>
</address-settings>
```

The **default-ring-size** parameter is especially useful for defining the default size of auto-created queues. The default value of **default-ring-size** is **-1** (that is, no size limit).

50. To define a ring size on a specific queue, add the **ring-size** key to the **queue** element. Specify a value. For example:

```
  <addresses>
    <address name="myRing">
      <anycast>
        <queue name="myRing" ring-size="5" />
      </anycast>
   </address>
</addresses>
```

NOTE

You can update the value of **ring-size** while the broker is running. The broker dynamically applies the update. If the new **ring-size** value is lower than the previous value, the broker does not immediately delete messages from the head of the queue to enforce the new size. New messages sent to the queue still force the deletion of older messages, but the queue does not reach its new, reduced size until it does so naturally, through the normal consumption of messages by clients.

Troubleshooting ring queues

In-delivery messages and rollbacks

When a message is in delivery to a consumer, the message is in an "in-between" state, where the message is technically no longer on the queue, but is also not yet acknowledged. A message remains in an in-delivery state until acknowledged by the consumer. Messages that remain in an in-delivery state cannot be removed from the ring queue.

Because the broker cannot remove in-delivery messages, a client can send more messages to a ring queue than the ring size configuration seems to allow. For example, consider this scenario:

51. A producer sends three messages to a ring queue configured with **ring-size="3"**.

52. All messages are immediately dispatched to a consumer.
At this point, **messageCount**= **3** and **deliveringCount**= **3**.

53. The producer sends another message to the queue. The message is then dispatched to the consumer.
Now, **messageCount** = **4** and **deliveringCount** = **4**. The message count of **4** is greater than the configured ring size of **3**. However, the broker is obliged to allow this situation because it cannot remove the in-delivery messages from the queue.

54. Now, suppose that the consumer is closed without acknowledging any of the messages.
In this case, the four in-delivery, unacknowledged messages are canceled back to the broker and added to the head of the queue in the reverse order from which they were consumed. This action puts the queue over its configured ring size. Because a ring queue prefers messages at the tail of the queue over messages at the head, the queue discards the first message sent by the producer, because this was the last message added back to the head of the queue. Transaction or core session rollbacks are treated in the same way.

If you are using the core client directly, or using an AMQ Core Protocol JMS client, you can minimize the number of messages in delivery by reducing the value of the **consumerWindowSize** parameter (1024 * 1024 bytes by default).

Scheduled messages

When a scheduled message is sent to a queue, the message is not immediately added to the tail of the queue like a normal message. Instead, the broker holds the scheduled message in an intermediate buffer and schedules the message for delivery onto the head of the queue, according to the details of the message. However, scheduled messages are still reflected in the message count of the queue. As with in-delivery messages, this behavior can make it appear that the broker is not enforcing the ring queue size. For example, consider this scenario:

1. At 12:00, a producer sends a message, **A**, to a ring queue configured with **ring-size="3"**. The message is scheduled for 12:05.
   At this point, **messageCount**= **1** and **scheduledCount**= **1**.

2. At 12:01, producer sends message **B** to the same ring queue.
   Now, **messageCount**= **2** and **scheduledCount**= **1**.

3. At 12:02, producer sends message **C** to the same ring queue.
   Now, **messageCount**= **3** and **scheduledCount**= **1**.

4. At 12:03, producer sends message **D** to the same ring queue.
   Now, **messageCount**= **4** and **scheduledCount**= **1**.

   The message count for the queue is now **4**, one *greater* than the configured ring size of **3**. However, the scheduled message is not technically on the queue yet (that is, it is on the broker and scheduled to be put on the queue). At the scheduled delivery time of 12:05, the broker puts the message on the head of the queue. However, since the ring queue has already reached its configured size, the scheduled message **A** is immediately removed.

Paged messages

Similar to scheduled messages and messages in delivery, paged messages do not count towards the ring queue size enforced by the broker, because messages are actually paged at the address level, not the queue level. A paged message is not technically on a queue, although it is reflected in a queue's **messageCount** value.

It is recommended that you do not use paging for addresses with ring queues. Instead, ensure that the entire address can fit into memory. Or, configure the **address-full-policy** parameter to a value of **DROP**, **BLOCK** or **FAIL**.

CONFIGURING RETROACTIVE ADDRESSES

Configuring an address as *retroactive* enables you to preserve messages sent to that address, including when there are no queues yet bound to the address. When queues are later created and bound to the address, the broker retroactively distributes messages to those queues. If an address is *not* configured as retroactive and does not yet have a queue bound to it, the broker discards messages sent to that address.

When you configure a retroactive address, the broker creates an internal instance of a type of queue known as a *ring queue*. A ring queue is a special type of queue that holds a specified, fixed number of messages. Once the queue has reached the specified size, the next message that arrives to the queue forces the oldest message out of the queue. When you configure a retroactive address, you indirectly specify the size of the internal ring queue. By default, the internal queue uses the **multicast** routing type.

The internal ring queue used by a retroactive address is exposed via the management API. You can inspect metrics and perform other common management operations, such as emptying the queue. The ring queue also contributes to the overall memory usage of the address, which affects behavior such as message paging.

The following procedure shows how to configure an address as retroactive.

Procedure

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

2. Specify a value for the **retroactive-message-count** parameter in the **address-setting** element. The value you specify defines the number of messages you want the broker to preserve. For example:

```
<configuration>
  <core>
    ...
    <address-settings>
       <address-setting match="orders">
          <retroactive-message-count>100</retroactive-message-count>
     </address-setting>
   </address-settings>
...
  </core>
</configuration>
```

> NOTE
>
> You can update the value of **retroactive-message-count** while the broker is running, in either the **broker.xml** configuration file or the management API. However, if you *reduce* the value of this parameter, an additional step is required, because retroactive addresses are implemented via ring queues. A ring queue whose **ring-size** parameter is reduced does not automatically delete messages from the queue to achieve the new **ring-size** value. This behavior is a safeguard against unintended message loss. In this case, you need to use the management API to manually reduce the number of messages in the ring queue.

DISABLING ADVISORY MESSAGES FOR INTERNALLY-MANAGED ADDRESSES AND QUEUES

By default, AMQ Broker creates advisory messages about addresses and queues when an OpenWire client is connected to the broker. Advisory messages are sent to internally-managed addresses created by the broker. These addresses appear on the AMQ Management Console within the same display as user-deployed addresses and queues. Although they provide useful information, advisory messages can cause unwanted consequences when the broker manages a large number of destinations. For example, the messages might increase memory usage or strain connection resources. Also, the AMQ Management Console might become cluttered when attempting to display all of the addresses created to send advisory messages. To avoid these situations, you can use the following parameters to configure the behavior of advisory messages on the broker.

supportAdvisory

Set this option to **true** to enable creation of advisory messages or **false** to disable them. The default value is **true**.

suppressInternalManagementObjects

Set this option to **true** to expose the advisory messages to management services such as JMX registry and AMQ Management Console, or **false** to not expose them. The default value is **true**.

The following procedure shows how to disable advisory messages on the broker.

Procedure

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.
2. For an OpenWire connector, add the **supportAdvisory** and **suppressInternalManagementObjects** parameters to the configured URL. Set the values as described earlier in this section. For example:

```
<acceptor name="artemis">tcp://127.0.0.1:61616?
protocols=CORE,AMQP,OPENWIRE;supportAdvisory=false;suppressInternalManagementObje
cts=false</acceptor>
```

FEDERATING ADDRESSES AND QUEUES

Federation enables transmission of messages between brokers, without requiring the brokers to be in a common cluster. Brokers can be standalone, or in separate clusters. In addition, the source and target brokers can be in different administrative domains, meaning that the brokers might have different configurations, users, and security setups. The brokers might even be using different versions of AMQ Broker.

For example, federation is suitable for reliably sending messages from one cluster to another. This transmission might be across a Wide Area Network (WAN), *Regions* of a cloud infrastructure, or over the Internet. If connection from a source broker to a target broker is lost (for example, due to network failure), the source broker tries to reestablish the connection until the target broker comes back online. When the target broker comes back online, message transmission resumes.

Administrators can use address and queue policies to manage federation. Policy configurations can be matched to specific addresses or queues, or the policies can include wildcard expressions that match configurations to *sets* of addresses or queues. Therefore, federation can be dynamically applied as queues or addresses are added to- or removed from matching sets. Policies can include multiple expressions that include and/or exclude particular addresses and queues. In addition, multiple policies can be applied to brokers or broker clusters.

In AMQ Broker, the two primary federation options are *address federation* and *queue federation*.

NOTE

A broker can include configuration for federated and local-only components. That is, if you configure federation on a broker, you don’t need to federate everything on that broker.

About address federation

Address federation is like a full multicast distribution pattern between connected brokers. For example, every message sent to an address on **BrokerA** is delivered to every queue on that broker. In addition, each of the messages is delivered to **BrokerB** and all attached queues there.

Address federation dynamically links a broker to addresses in remote brokers. For example, if a local broker wants to fetch messages from an address on a remote broker, a queue is automatically created on the remote address. Messages on the remote broker are then consumed to this queue. Finally, messages are copied to the corresponding address on the local broker, as though they were originally published directly to the local address.

The remote broker does not need to be reconfigured to allow federation to create an address on it. However, the local broker *does* need to be granted permissions to the remote address.

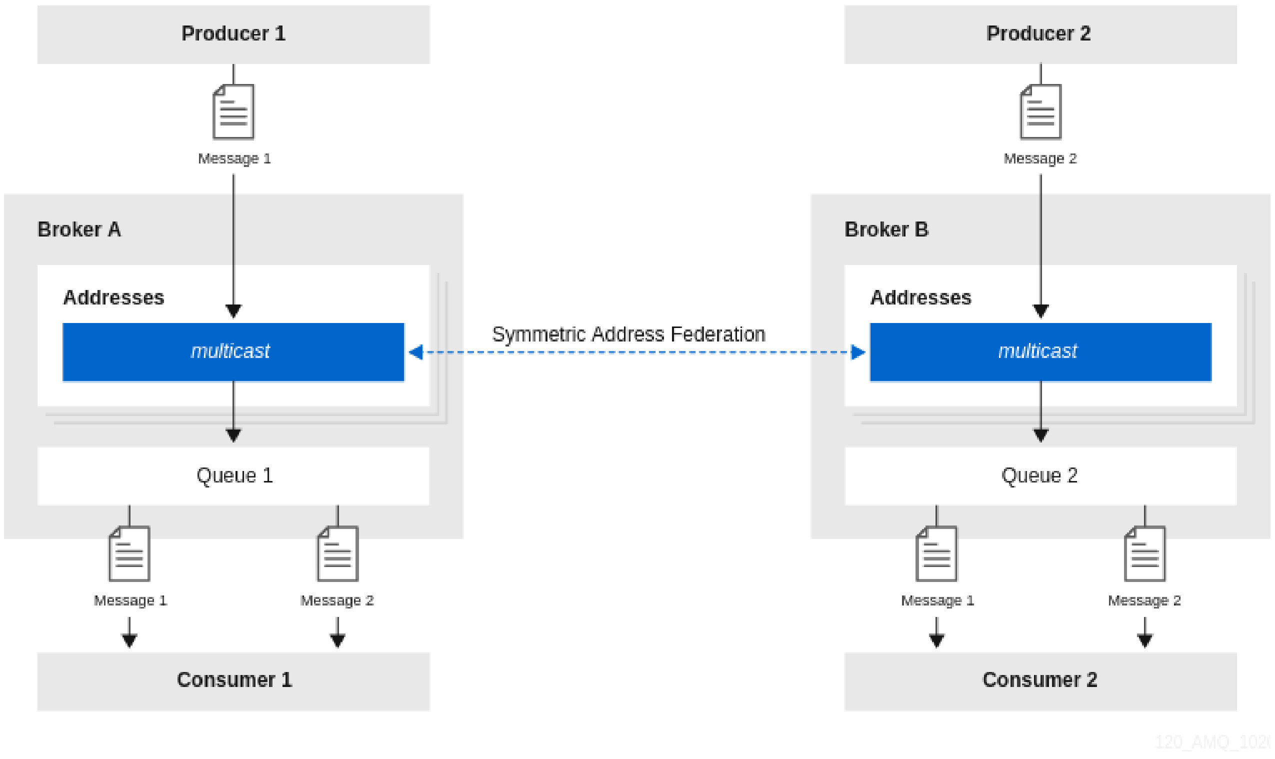
Common topologies for address federation

Some common topologies for the use of address federation are described below.

Symmetric topology

In a symmetric topology, a producer and consumer are connected to each broker. Queues and their consumers can receive messages published by either producer. An example of a symmetric topology is shown below.

Figure 4.1. Address federation in a symmetric topology

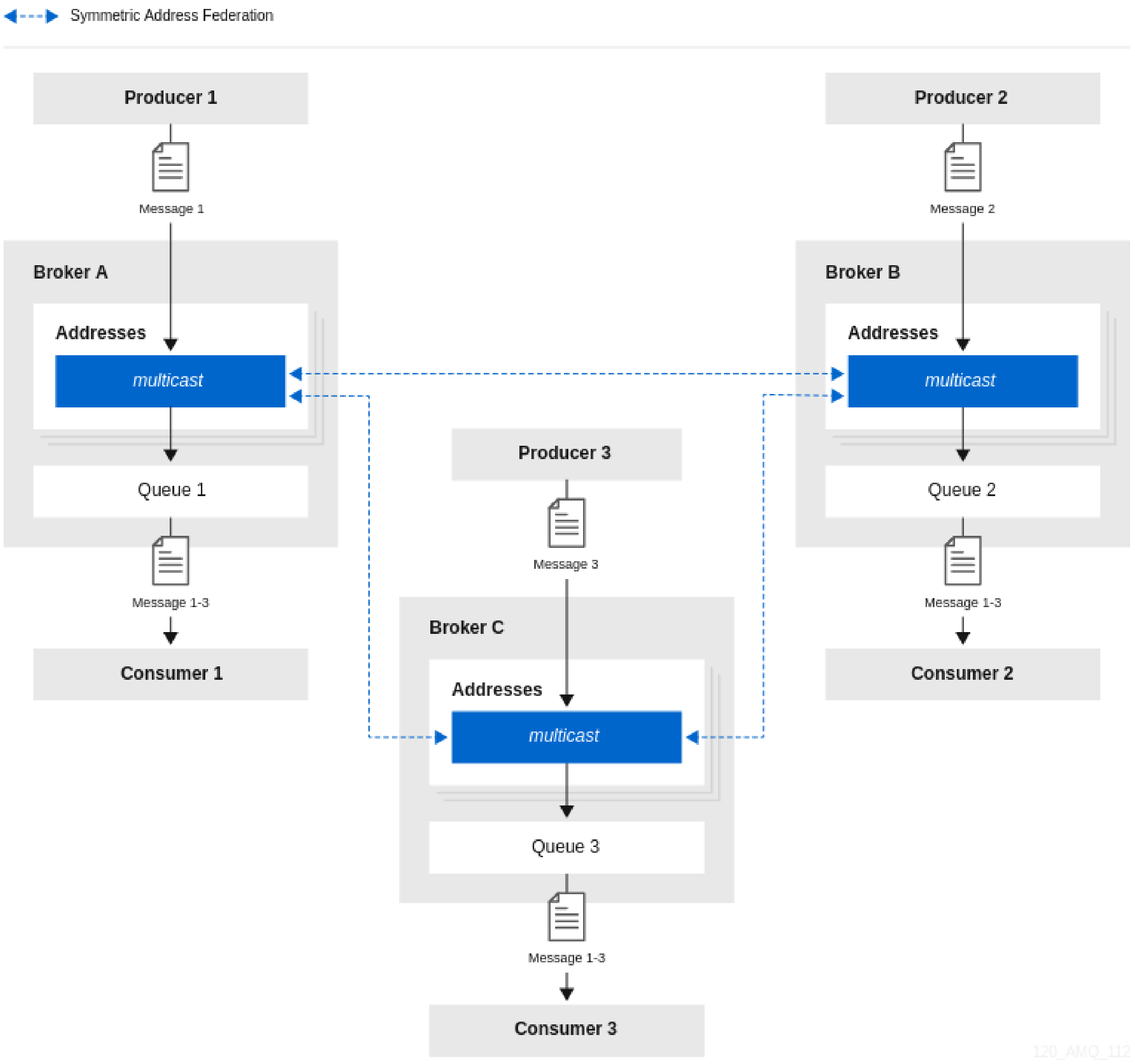


When configuring address federation for a symmetric topology, it is important to set the value of the **max-hops** property of the address policy to **1**. This ensures that messages are copied only once, avoiding cyclic replication. If this property is set to a larger value, consumers will receive multiple copies of the same message.

Full mesh topology

A full mesh topology is similar to a symmetric setup. Three or more brokers symmetrically federate to each other, creating a full mesh. In this setup, a producer and consumer are connected to each broker. Queues and their consumers can receive messages published by any producer. An example of this topology is shown below.

Figure 4.2. Address federation in a full mesh topology

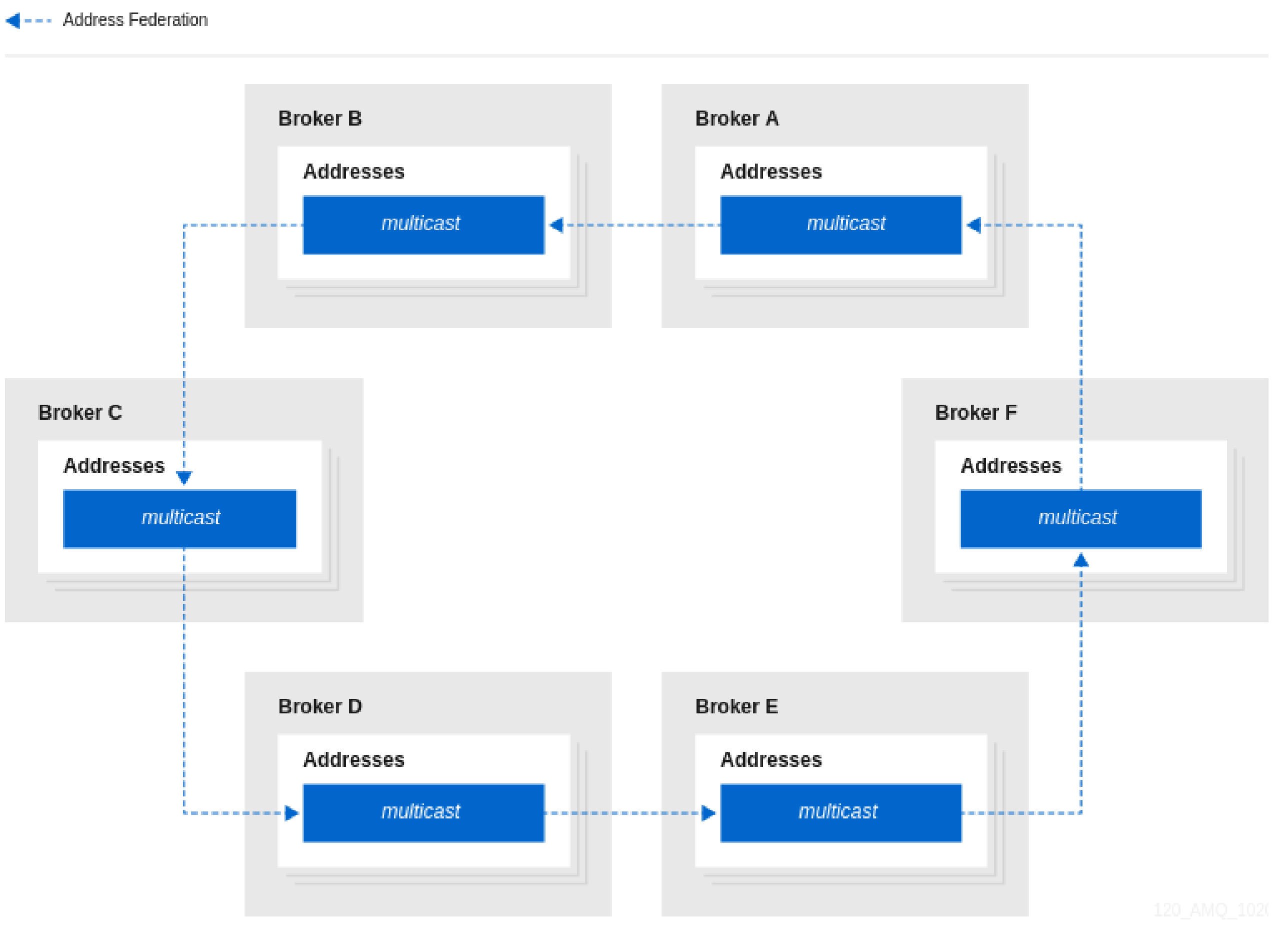


As with a symmetric setup, when configuring address federation for a full mesh topology, it is important to set the value of the **max-hops** property of the address policy to **1**. This ensures that messages are copied only once, avoiding cyclic replication.

Ring topology

In a ring of brokers, each federated address is upstream to just one other in the ring. An example of this topology is shown below.

Figure 4.3. Address federation in a ring topology



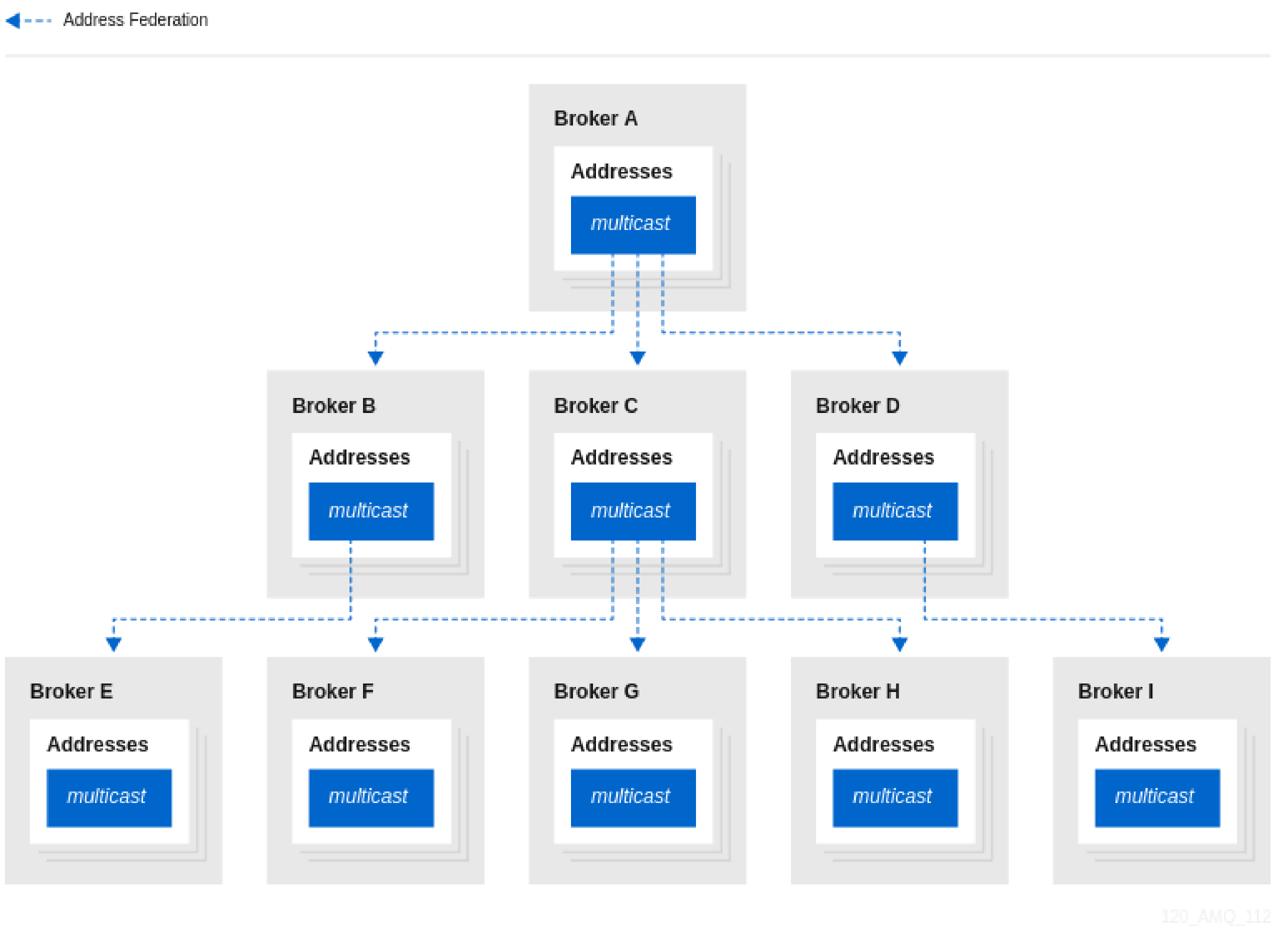
When you configure federation for a ring topology, to avoid cyclic replication, it is important to set the **max-hops** property of the address policy to a value of **n-1**, where *n* is the number of nodes in the ring. For example, in the ring topology shown above, the value of **max-hops** is set to **5**. This ensures that every address in the ring sees the message exactly once.

An advantage of a ring topology is that it is cheap to set up, in terms of the number of physical connections that you need to make. However, a drawback of this type of topology is that if a single broker fails, the whole ring fails.

Fan-out topology

In a fan-out topology, a single master address is linked-to by a tree of federated addresses. Any message published to the master address can be received by any consumer connected to any broker in the tree. The tree can be configured to any depth. The tree can also be extended without the need to re-configure existing brokers in the tree. An example of this topology is shown below.

Figure 4.4. Address federation in a fan-out topology



When you configure federation for a fan-out topology, ensure that you set the **max-hops** property of the address policy to a value of **n-1**, where *n* is the number of levels in the tree. For example, in the fan-out topology shown above, the value of **max-hops** is set to **2**. This ensures that every address in the tree sees the message exactly once.

Support for divert bindings in address federation configuration

When configuring address federation, you can add support for divert bindings in the address policy configuration. Adding this support enables the federation to respond to divert bindings to create a federated consumer for a given address on a remote broker.

For example, suppose that an address called **test.federation.source** is included in the address policy, and another address called **test.federation.target** is not included. Normally, when a queue is created on **test.federation.target**, this would not cause a federated consumer to be created, because the address is not part of the address policy. However, if you create a divert binding such that **test.federation.source** is the source address and **test.federation.target** is the forwarding address, then a durable consumer is created at the forwarding address. The source address still must use the **multicast** routing type , but the target address can use **multicast** or **anycast**.

An example use case is a divert that redirects a JMS topic (**multicast** address) to a JMS queue (**anycast** address). This enables load balancing of messages on the topic for legacy consumers not supporting JMS 2.0 and shared subscriptions.

Configuring federation

You can configure address and queue federation using either the Core protocol or, beginning in 7.12, AMQP. Using AMQP for federation offers the following advantages:

- If clients use AMQP for messaging, using AMQP for federation eliminates the need to convert messages between AMQP and the Core protocol and vice versa, which is required if federation uses the Core protocol.
- AMQP federation supports two-way federation over a single outgoing connection. This eliminates the need for a remote broker to connect back to a local broker, which is a requirement when you use the Core protocol for federation and which might be prevented by network policies.

Configuring federation using AMQP

You can uses the following policies to configure address and queue federation using AMQP:

- A local address policy configures the local broker to watch for demand on addresses and, when that demand exists, create a federation consumer on the matching address on the remote broker to federate messages to the local broker.
- A remote address policy configures the remote broker to watch for demand on addresses and, when that demand exists, create a federation consumer on the matching address on the local broker to federate messages to the remote broker.
- A local queue policy configures the local broker to watch for demand on queues and, when that demand exists, create a federation consumer on the matching queue on the remote broker to federate messages to the local broker.
- A remote queue policy configures the remote broker to watch for demand on queues and, when that demand exists, create a federation consumer on the matching queue on the local broker to federate messages to the remote broker.

Remote address and queue policies are sent to the remote broker and become local policies on the remote broker to provide a reverse federation connection. In applying the policies for the reverse federation connection, the broker that received the policies is the local broker and the broker that sent the policies is the remote broker. By configuring remote address and queue policies on the local broker, you can keep all federation configuration on a single broker, which might be a useful approach for a hub-and-spoke topology, for example.

Configuring address federation using AMQP

Use the **<broker-connections>** element to configure address federation using AMQP.

Prerequisite

The user specified in the **<amqp-connection>** element has read and write permissions to matching addresses and queues on the remote broker.

Procedure

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.
2. Add a **<broker connections>** element that includes an **<amqp-connection>** element. In the **<amqp-connection>** element, specify the connection details for a remote broker and assign a name to the federation configuration. For example:

```
<broker-connections>
  <amqp-connection uri="tcp://<_HOST_>:<_PORT_>" user="federation_user"
password="federation_pwd" name="queue-federation-example">
 </amqp-connection>
</broker-connections>
```

3. Add a **<federation>** element and include one or both of the following:

   - A **<local-address-policy>** element to federate messages from the remote broker to the local broker.
   - A **<remote-address-policy>** element to federate messages from the local broker to the remote broker.

   The following example show a federation element with both local and remote address policies.

```
<broker-connections>
  <amqp-connection uri="tcp://<_HOST_>:<_PORT_>" user="federation_user"
password="federation_pwd" name="queue-federation-example">
    <federation>
      <local-address-policy name="example-local-address-policy" auto-delete="true" auto-
delete-delay="1" auto-delete-message-count="2" max-hops="1" enable-divert-
bindings="true">
         <include  address-match="queue.news.#"  />
      <include  address-match="queue.bbc.news"  />
        <exclude  address-match="queue.news.sport.#"  />
       </local-address-policy>
       <remote-address-policy   name="example-remote-address-policy">
         <include  address-match="queue.usatoday"  />
       </remote-address-policy>
    </federation>
 </amqp-connection>
</broker-connections>
```

   The same parameters are configurable in both a local and remote address policy. The valid parameters are:

   name

      Name of the address policy. All address policy names must be unique within the **<federation>** elements in a **<broker-connections>** element.

   max-hops

      Maximum number of hops that a message can make during federation. The default value of **0** is suitable for most simple federation deployments. However, in certain topologies a greater value might be required to prevent messages from looping.

   auto-delete

      For address federation, a durable queue is created on the broker from which messages are being federated. Set this parameter to true to mark the queue for automatic deletion once the initiating broker disconnects and the delay and message count parameters are met. This is a useful option if you want to automate the cleanup of dynamically-created queues. The default value is **false**, which means that the queue is not automatically deleted.

   auto-delete-delay

      The amount of time in milliseconds before the created queue is eligible for automatic deletion after the initiating broker has disconnected. The default value is **0**.

auto-delete-message-count

The value that the message count for the queue must be less than or equal to before the queue can be automatically deleted. The default value is **0**.

enable-divert-bindings

Setting to **true** enables divert bindings to be listened-to for demand. If a divert binding with an address matches the included addresses for the address policy, any queue bindings that match the forwarding address of the divert creates demand. The default value is **false**.

include

The address-match patterns to match addresses to include in the policy. You can specify multiple patterns. If you do not specify a pattern, all addresses are included in the policy. You can specify an exact address, for example, **queue.bbc.news**. Or, you can use the number sign (#) wildcard character to specify a matching set of addresses. In the preceding example, the local address policy also includes all addresses that start with the **queue.news** string.

exclude

The address-match patterns to match addresses to exclude from the policy. You can specify multiple patterns. If you do not specify a pattern, no addresses are excluded from the policy. You can specify an exact address, for example, **queue.bbc.news**. Or, you can use the number sign (#) wildcard character to specify a matching set of addresses. In the preceding example, the local address policy excludes all addresses that start with the **queue.news.sport** string.

Configuring queue federation using AMQP

Procedure

Use the **<broker connections>** element to configure queue federation for AMQP.

Prerequisite

The user specified in the **<amqp-connection>** element has read and write permissions to matching addresses and queues on the remote broker.

4. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

5. Add a **<broker connections>** element that includes an **<amqp-connection>** element. In the **<amqp-connection>** element, specify the connection details for a remote broker and assign a name to the federation configuration. For example:

```
<broker-connections>
  <amqp-connection uri="tcp://<_HOST_>:<_PORT_>" user="federation_user"
password="federation_pwd" name="queue-federation-example">
 </amqp-connection>
</broker-connections>
```

6. Add a **<federation>** element and include one or both of the following:

- A **<local-queue-policy>** element to federate messages from the remote broker to the local broker.

- A **<remote-queue-policy>** element to federate messages from the local broker to the remote broker.

The following examples show a federation element that contains a local queue policy.

```
<broker-connections>
  <amqp-connection uri="tcp://HOST:PORT" name="federation-example">
    <federation>
      <local-queue-policy name="example-local-queue-policy">
        <include address-match="#" queue-match="#.remote" />
        <exclude address-match="#" queue-match="#.local" />
      </local-queue-policy>
    </federation>
  </amqp-connection>
</broker-connections>
```

name

Name of the queue policy. All queue policy names must be unique within the **<federation>** elements of a **<broker-connections>** element.

include

The **address-match** patterns to match addresses and the **queue-match** patterns to match specific queues on those addresses for inclusion in the policy. As with the **address-match** parameter, you can specify an exact name for the **queue-match** parameter, or you can use a wildcard expression to specify a set of queues. In the preceding example, queues that match the **.remote** string across all addresses, represented by an **address-match** value of #, are included.

exclude

The **address-match** patterns to match addresses and the **queue-match** patterns to match specific queues on those addresses for exclusion from the policy. As with the **address-match** parameter, you can specify an exact name for the **queue-match** parameter, or you can use a wildcard expression to specify a set of queues. In the preceding example, queues that match the **.local** string across all addresses, represented by an **address-match** value of #, are excluded.

priority-adjustment

Adjusts the value of federated consumers to ensure that they have a lower priority value than other local consumers on the same queue. The default value is **-1**, which ensures that the local consumer are prioritized over federated consumers.

include-federated

When the value of this parameter is set to **false**, the configuration does not re-federate an already-federated consumer (that is, a consumer on a federated queue). This avoids a situation where, in a symmetric or closed-loop topology, there are no non-federated consumers and messages flow endlessly around the system.
You might set the value of this parameter to **true** if you do not have a closed-loop topology. For example, suppose that you have a chain of three brokers, BrokerA, BrokerB, and BrokerC, with a producer at BrokerA and a consumer at BrokerC. In this case, you would want BrokerB to re-federate the consumer to BrokerA.

Configuring federation using the Core protocol

You can configure message and queue federation to use the Core protocol.

Configuring federation for a broker cluster

The examples in the sections that follow show how to configure address and queue federation between standalone local and remote brokers. For federation between standalone brokers, the name of the federation configuration, as well as the names of any address and queue policies, must be unique between the local and remote brokers.

However, if you are configuring federation for brokers in a cluster, there is an additional requirement. For clustered brokers, the names of the federation configuration, as well as the names of any address and queues policies within that configuration, must be the same for every broker in that cluster.

Ensuring that brokers in the same cluster use the same federation configuration and address and queue policy names avoids message duplication. For example, if brokers within the same cluster have different federation configuration names, this might lead to a situation where multiple, differently-named forwarding queues are created for the same address, resulting in message duplication for downstream consumers. By contrast, if brokers in the same cluster use the same federation configuration name, this essentially creates replicated, clustered forwarding queues that are load-balanced to the downstream consumers. This avoids message duplication.

Configuring upstream address federation

The following example shows how to configure upstream address federation between standalone brokers. In this example, you configure federation from a local (that is, *downstream*) broker, to some remote (that is, *upstream*) brokers.

Prerequisites

- The following example shows how to configure address federation between standalone brokers. However, you should also be familiar with the requirements for configuring federation for a broker *cluster*

Procedure

7. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

8. Add a new **<federations>** element that includes a **<federation>** element. For example:

```
 <federations>
   <federation name="eu-north-1" user="federation_username"
 password="32a10275cf4ab4e9">
  </federation>
</federations>
```

name

Name of the federation configuration. In this example, the name corresponds to the name of the local broker.

user

Shared user name for connection to the upstream brokers.

password

Shared password for connection to the upstream brokers.

NOTE

If user and password credentials differ for remote brokers, you can separately specify credentials for those brokers when you add them to the configuration.

9. Within the **federation** element, add an **<address-policy>** element. For example:

```
<federations>
   <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">

       <address-policy name="news-address-federation" auto-delete="true" auto-delete-
delay="300000" auto-delete-message-count="-1" enable-divert-bindings="false" max-
hops="1" transformer-ref="news-transformer">
       </address-policy>

   </federation>
</federations>
```

name

Name of the address policy. All address policies that are configured on the broker must have unique names.

auto-delete

During address federation, the local broker dynamically creates a durable queue at the remote address. The value of the **auto-delete** property specifies whether the remote queue should be deleted once the local broker disconnects and the values of the **auto-delete-delay** and **auto-delete-message-count** properties have also been reached. This is a useful option if you want to automate the cleanup of dynamically-created queues. It is also a useful option if you want to prevent a buildup of messages on a remote broker if the local broker is disconnected for a long time. However, you might set this option to **false** if you want messages to always remain queued for the local broker while it is disconnected, avoiding message loss on the local broker.

auto-delete-delay

After the local broker has disconnected, the value of this property specifies the amount of time, in milliseconds, before dynamically-created remote queues are eligible to be automatically deleted.

auto-delete-message-count

After the local broker has been disconnected, the value of this property specifies the maximum number of messages that can still be in a dynamically-created remote queue before that queue is eligible to be automatically deleted.

enable-divert-bindings

Setting this property to **true** enables divert bindings to be listened-to for demand. If there is a divert binding with an address that matches the included addresses for the address policy, then any queue bindings that match the forwarding address of the divert will create demand. The default value is **false**.

max-hops

Maximum number of hops that a message can make during federation. Particular topologies require specific values for this property.

transformer-ref

Name of a transformer configuration. You might add a transformer configuration if you want to transform messages during federated message transmission. Transformer configuration is described later in this procedure.

10. Within the **<address-policy>** element, add address-matching patterns to include and exclude addresses from the address policy. For example:

```
<federations>
   <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">

      <address-policy name="news-address-federation" auto-delete="true" auto-delete-
delay="300000" auto-delete-message-count="-1" enable-divert-bindings="false" max-
hops="1" transformer-ref="news-transformer">

         <include  address-match="queue.bbc.new"  />
         <include  address-match="queue.usatoday"  />
         <include  address-match="queue.news.#"  />

         <exclude  address-match="queue.news.sport.#"  />
      </address-policy>

   </federation>
</federations>
```

include

The value of the **address-match** property of this element specifies addresses to include in the address policy. You can specify an exact address, for example, **queue.bbc.new** or **queue.usatoday**. Or, you can use a wildcard expression to specify a matching *set* of addresses. In the preceding example, the address policy also includes all address names that start with the string **queue.news**.

exclude

The value of the **address-match** property of this element specifies addresses to exclude from the address policy. You can specify an exact address name or use a wildcard expression to specify a matching *set* of addresses. In the preceding example, the address policy excludes all address names that start with the string **queue.news.sport**.

11. (Optional) Within the **federation** element, add a **transformer** element to reference a custom transformer implementation. For example:

```
<federations>
   <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">

      <address-policy name="news-address-federation" auto-delete="true" auto-delete-
delay="300000" auto-delete-message-count="-1" enable-divert-bindings="false" max-
hops="1" transformer-ref="news-transformer">

         <include  address-match="queue.bbc.new"  />
         <include  address-match="queue.usatoday"  />
         <include  address-match="queue.news.#"  />

         <exclude  address-match="queue.news.sport.#"  />
      </address-policy>
```

```
        <transformer name="news-transformer">
            <class-name>org.myorg.NewsTransformer</class-name>
            <property key="key1" value="value1"/>
            <property key="key2" value="value2"/>
        </transformer>

    </federation>
</federations>
```

name

Name of the transformer configuration. This name must be unique on the local broker. This is the name that you specify as a value for the **transformer-ref** property of the address policy.

class-name

Name of a user-defined class that implements the **org.apache.activemq.artemis.core.server.transformer.Transformer** interface. The transformer’s **transform()** method is invoked with the message before the message is transmitted. This enables you to transform the message header or body before it is federated.

property

Used to hold key-value pairs for specific transformer configuration.

12. Within the **federation** element, add one or more **upstream** elements. Each **upstream** element defines a connection to a remote broker and the policies to apply to that connection. For example:

```
<federations>
    <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">

        <upstream name="eu-east-1">
            <static-connectors>
                <connector-ref>eu-east-connector1</connector-ref>
            </static-connectors>
            <policy ref="news-address-federation"/>
        </upstream>

        <upstream name="eu-west-1" >
            <static-connectors>
                <connector-ref>eu-west-connector1</connector-ref>
            </static-connectors>
            <policy ref="news-address-federation"/>
        </upstream>

        <address-policy name="news-address-federation" auto-delete="true" auto-delete-
delay="300000" auto-delete-message-count="-1" enable-divert-bindings="false" max-
hops="1" transformer-ref="news-transformer">

            <include address-match="queue.bbc.new" />
            <include address-match="queue.usatoday" />
            <include address-match="queue.news.#" />

            <exclude address-match="queue.news.sport.#" />
        </address-policy>
        <transformer name="news-transformer">
           <class-name>org.myorg.NewsTransformer</class-name>
           <property key="key1" value="value1"/>
           <property key="key2" value="value2"/>
        </transformer>

     </federation>
  </federations>
```

static-connectors

Contains a list of **connector-ref** elements that reference **connector** elements that are defined elsewhere in the **broker.xml** configuration file of the local broker. A connector defines what transport (TCP, SSL, HTTP, and so on) and server connection parameters (host, port, and so on) to use for outgoing connections. The next step of this procedure shows how to add the connectors that are referenced in the **static-connectors** element.

policy-ref

Name of the address policy configured on the downstream broker that is applied to the upstream broker.

The additional options that you can specify for an **upstream** element are described below:

name

Name of the upstream broker configuration. In this example, the names correspond to upstream brokers called **eu-east-1** and **eu-west-1**.

user

User name to use when creating the connection to the upstream broker. If not specified, the shared user name that is specified in the configuration of the **federation** element is used.

password

Password to use when creating the connection to the upstream broker. If not specified, the shared password that is specified in the configuration of the **federation** element is used.

call-failover-timeout

Similar to **call-timeout**, but used when a call is made during a failover attempt. The default value is **-1**, which means that the timeout is disabled.

call-timeout

Time, in milliseconds, that a federation connection waits for a reply from a remote broker when it transmits a packet that is a blocking call. If this time elapses, the connection throws an exception. The default value is **30000**.

check-period

Period, in milliseconds, between consecutive “keep-alive” messages that the local broker sends to a remote broker to check the health of the federation connection. If the federation connection is healthy, the remote broker responds to each keep-alive message. If the connection is unhealthy, when the downstream broker fails to receive a response from the upstream broker, a mechanism called a *circuit breaker* is used to block federated consumers. See the description of the **circuit-breaker-timeout** parameter for more information. The default value of the **check-period** parameter is **30000**.

circuit-breaker-timeout

A single connection between a downstream and upstream broker might be shared by many federated queue and address consumers. In the event that the connection between the brokers is lost, each federated consumer might try to reconnect at the same time. To avoid this, a mechanism called a *circuit breaker* blocks the consumers. When the specified timeout value elapses, the circuit breaker re-tries the connection. If successful, consumers are unblocked. Otherwise, the circuit breaker is applied again.

connection-ttl

Time, in milliseconds, that a federation connection stays alive if it stops receiving messages from the remote broker. The default value is **60000**.

discovery-group-ref

As an alternative to defining static connectors for connections to upstream brokers, this element can be used to specify a discovery group that is already configured elsewhere in the **broker.xml** configuration file. Specifically, you specify an existing discovery group as a value for the **discovery-group-name** property of this element.

ha

Specifies whether high availability is enabled for the connection to the upstream broker. If the value of this parameter is set to **true**, the local broker can connect to any available broker in an upstream cluster and automatically fails over to a backup broker if the live upstream broker shuts down. The default value is **false**.

initial-connect-attempts

Number of initial attempts that the downstream broker will make to connect to the upstream broker. If this value is reached without a connection being established, the upstream broker is considered permanently offline. The downstream broker no longer routes messages to the upstream broker. The default value is **-1**, which means that there is no limit.

max-retry-interval

Maximum time, in milliseconds, between subsequent reconnection attempts when connection to the remote broker fails. The default value is **2000**.

reconnect-attempts

Number of times that the downstream broker will try to reconnect to the upstream broker if the connection fails. If this value is reached without a connection being re-established, the upstream broker is considered permanently offline. The downstream broker no longer routes messages to the upstream broker. The default value is **-1**, which means that there is no limit.

retry-interval

Period, in milliseconds, between subsequent reconnection attempts, if connection to the remote broker has failed. The default value is **500**.

retry-interval-multiplier

Multiplying factor that is applied to the value of the **retry-interval** parameter. The default value is **1**.

share-connection

If there is both a downstream and upstream connection configured for the same broker, then the same connection will be shared, as long as both of the downstream and upstream configurations set the value of this parameter to **true**. The default value is **false**.

13. On the local broker, add connectors to the remote brokers. These are the connectors referenced in the **static-connectors** elements of your federated address configuration. For example:

```
<connectors>
  <connector name="eu-west-1-connector">tcp://localhost:61616</connector>
  <connector name="eu-east-1-connector">tcp://localhost:61617</connector>
</connectors>
```

Configuring downstream address federation

The following example shows how to configure downstream address federation for standalone brokers.

Downstream address federation enables you to add configuration on the local broker that one or more remote brokers use to connect back to the local broker. The advantage of this approach is that you can keep all federation configuration on a single broker. This might be a useful approach for a hub-and-spoke topology, for example.

NOTE

Downstream address federation reverses the direction of the federation connection versus upstream address configuration. Therefore, when you add remote brokers to your configuration, these become considered as the *downstream* brokers. The downstream brokers use the connection information in the configuration to connect back to the local broker, which is now considered to be upstream. This is illustrated later in this example, when you add configuration for the remote brokers.

Prerequisites

- You should be familiar with the configuration for upstream address federation.
- The following example shows how to configure address federation between standalone brokers. However, you should also be familiar with the requirements for configuring federation for a broker *cluster*.

Procedure

14. On the local broker, open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

15. Add a **<federations>** element that includes a **<federation>** element. For example:

```
<federations>
   <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">
   </federation>
</federations>
```

16. Add an address policy configuration. For example:

```
<federations>
   ...
   <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">

      <address-policy name="news-address-federation" max-hops="1" auto-delete="true"
auto-delete-delay="300000" auto-delete-message-count="-1" transformer-ref="news-
transformer">

         <include  address-match="queue.bbc.new"  />
         <include  address-match="queue.usatoday"  />
         <include  address-match="queue.news.#"  />

```

```
          <exclude address-match="queue.news.sport.#" />
        </address-policy>

    </federation>
  ...
</federations>
```

17. If you want to transform messages before transmission, add a transformer configuration. For example:

```
<federations>
    ...
    <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">

        <address-policy name="news-address-federation" max-hops="1" auto-delete="true"
auto-delete-delay="300000" auto-delete-message-count="-1" transformer-ref="news-
transformer">

            <include address-match="queue.bbc.new" />
            <include address-match="queue.usatoday" />
            <include address-match="queue.news.#" />

            <exclude address-match="queue.news.sport.#" />
        </address-policy>

        <transformer name="news-transformer">
            <class-name>org.myorg.NewsTransformer</class-name>
            <property key="key1" value="value1"/>
            <property key="key2" value="value2"/>
        </transformer>

    </federation>
  ...
</federations>
```

18. Add a **downstream** element for each remote broker. For example:

```
<federations>
    ...
    <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">

        <downstream name="eu-east-1">
                <static-connectors>
                    <connector-ref>eu-east-connector1</connector-ref>
                </static-connectors>
                <upstream-connector-ref>netty-connector</upstream-connector-ref>
                <policy ref="news-address-federation"/>
        </downstream>

        <downstream name="eu-west-1" >
              <static-connectors>
                <connector-ref>eu-west-connector1</connector-ref>
              </static-connectors>
```

```
            <upstream-connector-ref>netty-connector</upstream-connector-ref>
            <policy ref="news-address-federation"/>
        </downstream>

        <address-policy name="news-address-federation" max-hops="1" auto-delete="true"
auto-delete-delay="300000" auto-delete-message-count="-1" transformer-ref="news-
transformer">
            <include address-match="queue.bbc.new" />
            <include address-match="queue.usatoday" />
            <include address-match="queue.news.#" />

            <exclude address-match="queue.news.sport.#" />
        </address-policy>

        <transformer name="news-transformer">
            <class-name>org.myorg.NewsTransformer</class-name>
            <property key="key1" value="value1"/>
            <property key="key2" value="value2"/>
        </transformer>

    </federation>
  ...
</federations>
```

As shown in the preceding configuration, the remote brokers are now considered to be downstream of the local broker. The downstream brokers use the connection information in the configuration to connect back to the local (that is, *upstream*) broker.

19. On the local broker, add connectors and acceptors used by the local and remote brokers to establish the federation connection. For example:

```
<connectors>
  <connector name="netty-connector">tcp://localhost:61616</connector>
  <connector   name="eu-west-1-connector">tcp://localhost:61616</connector>
  <connector   name="eu-east-1-connector">tcp://localhost:61617</connector>
</connectors>

<acceptors>
  <acceptor name="netty-acceptor">tcp://localhost:61616</acceptor>
</acceptors>
```

connector name="netty-connector"

Connector configuration that the local broker sends to the remote broker. The remote broker use this configuration to connect back to the local broker.

connector name="eu-west-1-connector", **connector name="eu-east-1-connector"**

Connectors to remote brokers. The local broker uses these connectors to connect to the remote brokers and share the configuration that the remote brokers need to connect back to the local broker.

acceptor name="netty-acceptor"

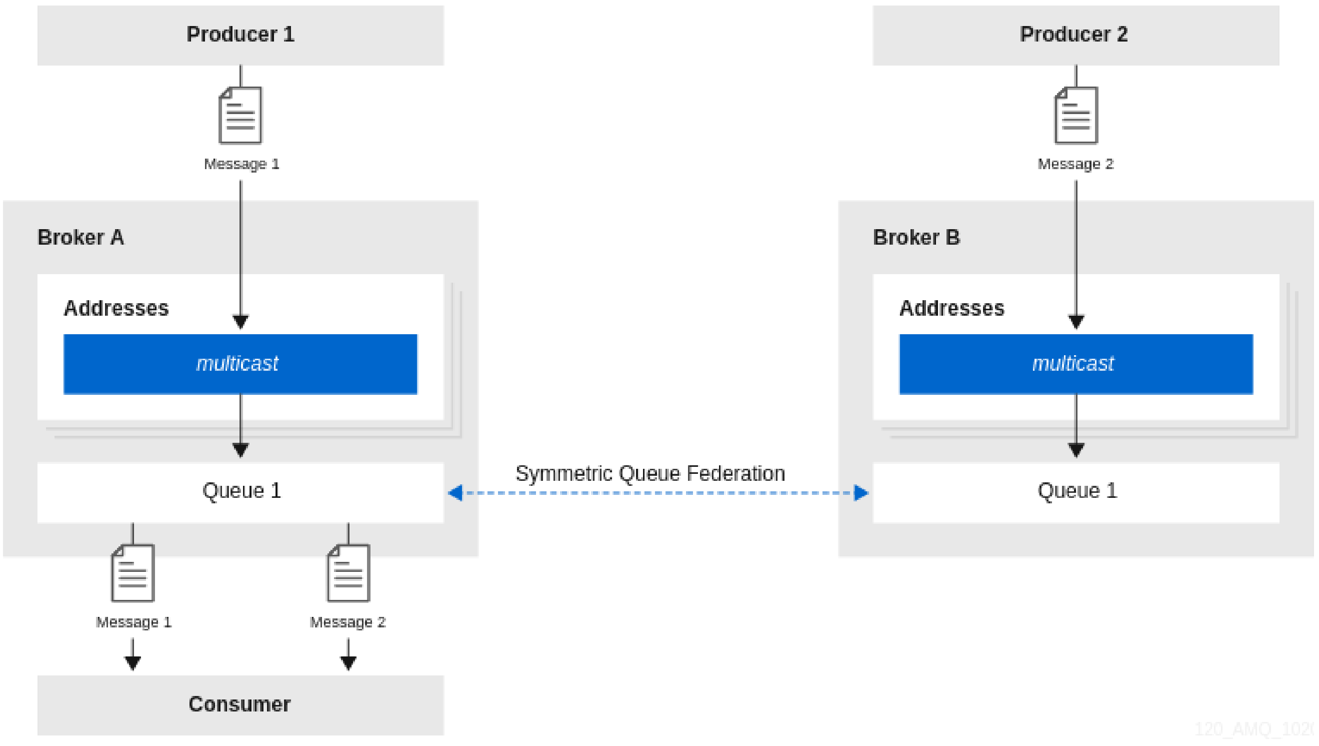
Acceptor on the local broker that corresponds to the connector used by the remote broker to connect back to the local broker.

About queue federation

Queue federation provides a way to balance the load of a single queue on a local broker across other, remote brokers.

To achieve load balancing, a local broker retrieves messages from remote queues in order to satisfy demand for messages from local consumers. An example is shown below.

Figure 4.5. Symmetric queue federation



The remote queues do not need to be reconfigured and they do not have to be on the same broker or in the same cluster. All of the configuration needed to establish the remote links and the federated queue is on the local broker.

Advantages of queue federation

Described below are some reasons you might choose to configure queue federation.

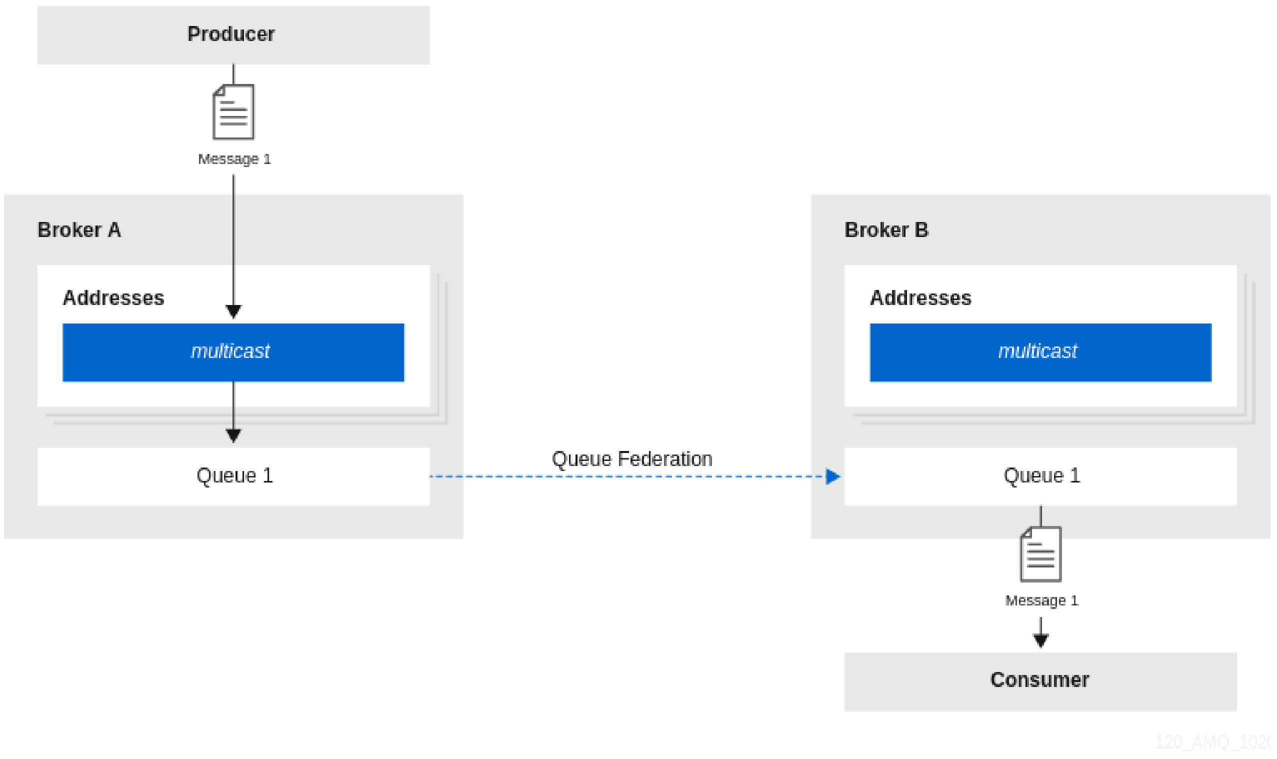
Increasing capacity

Queue federation can create a "logical" queue that is distributed over many brokers. This logical distributed queue has a much higher capacity than a single queue on a single broker. In this setup, as many messages as possible are consumed from the broker they were originally published to. The system moves messages around in the federation only when load balancing is needed.

Deploying multi-region setups

In a multi-region setup, you might have a message producer in one region or venue and a consumer in another. However, you should ideally keep producer and consumer connections local to a given region. In this case, you can deploy brokers in each region where producers and consumers are, and use queue federation to move messages over a Wide Area Network (WAN), between regions. An example is shown below.

Figure 4.6. Multi-region queue federation



Communicating between a secure enterprise LAN and a DMZ

In networking security, a *demilitarized zone* (DMZ) is a physical or logical subnetwork that contains and exposes an enterprise's external-facing services to an untrusted, usually larger, network such as the Internet. The remainder of the enterprise's Local Area Network (LAN) remains isolated from this external network, behind a firewall.

In a situation where a number of message producers are in the DMZ and a number of consumers in the secure enterprise LAN, it might not be appropriate to allow the producers to connect to a broker in the secure enterprise LAN. In this case, you could deploy a broker in the DMZ that the producers can publish messages to. Then, the broker in the enterprise LAN can connect to the broker in the DMZ and use federated queues to receive messages from the broker in the DMZ.

Configuring upstream queue federation

The following example shows how to configure upstream queue federation for standalone brokers. In this example, you configure federation from a local (that is, *downstream*) broker, to some remote (that is, *upstream*) brokers.

Prerequisites

- The following example shows how to configure queue federation between standalone brokers. However, you should also be familiar with the requirements for configuring federation for a broker *cluster*.

Procedure

20. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

21. Within a new **<federations>** element, add a **<federation>** element. For example:

```
<federations>
  <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">
 </federation>
</federations>
```

name

Name of the federation configuration. In this example, the name corresponds to the name of the downstream broker.

user

Shared user name for connection to the upstream brokers.

password

Shared password for connection to the upstream brokers.

NOTE

- If user and password credentials differ for upstream brokers, you can separately specify credentials for those brokers when you add them to the configuration. This is described later in this procedure.

22. Within the **federation** element, add a **<queue-policy>** element. Specify values for properties of the **<queue-policy>** element. For example:

```
<federations>
    <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">

        <queue-policy name="news-queue-federation" include-federated="true" priority-
adjustment="-5" transformer-ref="news-transformer">
        </queue-policy>

    </federation>
</federations>
```

name

Name of the queue policy. All queue policies that are configured on the broker must have unique names.

include-federated

When the value of this property is set to **false**, the configuration does not re-federate an already-federated consumer (that is, a consumer on a federated queue). This avoids a situation where in a symmetric or closed-loop topology, there are no non-federated consumers, and messages flow endlessly around the system.
You might set the value of this property to **true** if you do not have a closed-loop topology. For example, suppose that you have a chain of three brokers, **BrokerA**, **BrokerB**, and **BrokerC**, with a producer at **BrokerA** and a consumer at **BrokerC**. In this case, you would want **BrokerB** to re-federate the consumer to **BrokerA**.

priority-adjustment

When a consumer connects to a queue, its priority is used when the upstream (that is *federated*) consumer is created. The priority of the federated consumer is adjusted by the value of the **priority-adjustment** property. The default value of this property is **-1**, which

ensures that the local consumer get prioritized over the federated consumer during load balancing. However, you can change the value of the priority adjustment as needed.
If the priority adjustment is insufficient to prevent too many messages from moving to federated consumers, which can cause messages to move back and forth between brokers, you can limit the size of the batches of messages that are moved to the federated consumers. To limit the batch size, set the **consumerWindowSize** value to **0** on the connection URI of federated consumers.

```
tcp://<host>:<port>?consumerWindowSize=0
```

With the **consumerWindowSize** value set to **0**, AMQ Broker uses the value of the **defaultConsumerWindowSize** parameter in the address settings for a matching address to determine the batch size of messages that can be moved between brokers. The default value for the **defaultConsumerWindowSize** attribute is **1048576** bytes.

transformer-ref

Name of a transformer configuration. You might add a transformer configuration if you want to transform messages during federated message transmission. Transformer configuration is described later in this procedure.

23. Within the **<queue-policy>** element, add address-matching patterns to include and exclude addresses from the queue policy. For example:

```
<federations>
   <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">

      <queue-policy name="news-queue-federation" include-federated="true" priority-
adjustment="-5" transformer-ref="news-transformer">

         <include  queue-match="#"  address-match="queue.bbc.new"  />
         <include  queue-match="#"  address-match="queue.usatoday"  />
         <include  queue-match="#"  address-match="queue.news.#"  />

         <exclude  queue-match="#.local"  address-match="#"  />

      </queue-policy>

   </federation>
</federations>
```

include

The value of the **address-match** property of this element specifies addresses to include in the queue policy. You can specify an exact address, for example, **queue.bbc.new** or **queue.usatoday**. Or, you can use a wildcard expression to specify a matching *set* of addresses. In the preceding example, the queue policy also includes all address names that start with the string **queue.news**.
In combination with the **address-match** property, you can use the **queue-match** property to include specific queues on those addresses in the queue policy. Like the **address-match** property, you can specify an exact queue name, or you can use a wildcard expression to specify a *set* of queues. In the preceding example, the number sign (#) wildcard character means that all queues on each address or set of addresses are included in the queue policy.

exclude

The value of the **address-match** property of this element specifies addresses to exclude from the queue policy. You can specify an exact address or use a wildcard expression to specify a matching *set* of addresses. In the preceding example, the number sign (#) wildcard character means that any queues that match the **queue-match** property across all addresses are excluded. In this case, any queue that ends with the string **.local** is excluded. This indicates that certain queues are kept as local queues, and not federated.

24. Within the **federation** element, add a **transformer** element to reference a custom transformer implementation. For example:

```
<federations>
  <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">

      <queue-policy name="news-queue-federation" include-federated="true" priority-
adjustment="-5" transformer-ref="news-transformer">

          <include queue-match="#" address-match="queue.bbc.new" />
          <include queue-match="#" address-match="queue.usatoday" />
          <include queue-match="#" address-match="queue.news.#" />

          <exclude queue-match="#.local" address-match="#" />

      </queue-policy>

      <transformer name="news-transformer">
          <class-name>org.myorg.NewsTransformer</class-name>
          <property key="key1" value="value1"/>
          <property key="key2" value="value2"/>
      </transformer>

  </federation>
</federations>
```

name

Name of the transformer configuration. This name must be unique on the broker in question. You specify this name as a value for the **transformer-ref** property of the address policy.

class-name

Name of a user-defined class that implements the **org.apache.activemq.artemis.core.server.transformer.Transformer** interface. The transformer's **transform()** method is invoked with the message before the message is transmitted. This enables you to transform the message header or body before it is federated.

property

Used to hold key-value pairs for specific transformer configuration.

25. Within the **federation** element, add one or more **upstream** elements. Each **upstream** element defines an upstream broker connection and the policies to apply to that connection. For example:

```
<federations>
  <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">
```

```
        <upstream name="eu-east-1">
           <static-connectors>
              <connector-ref>eu-east-connector1</connector-ref>
           </static-connectors>
           <policy ref="news-queue-federation"/>
        </upstream>

        <upstream name="eu-west-1" >
           <static-connectors>
              <connector-ref>eu-west-connector1</connector-ref>
           </static-connectors>
           <policy ref="news-queue-federation"/>
        </upstream>

        <queue-policy name="news-queue-federation" include-federated="true" priority-
adjustment="-5" transformer-ref="news-transformer">

           <include queue-match="#" address-match="queue.bbc.new" />
           <include queue-match="#" address-match="queue.usatoday" />
           <include queue-match="#" address-match="queue.news.#" />

           <exclude queue-match="#.local" address-match="#" />

        </queue-policy>

        <transformer name="news-transformer">
           <class-name>org.myorg.NewsTransformer</class-name>
           <property key="key1" value="value1"/>
           <property key="key2" value="value2"/>
        </transformer>

   </federation>
</federations>
```

static-connectors

Contains a list of **connector-ref** elements that reference **connector** elements that are defined elsewhere in the **broker.xml** configuration file of the local broker. A connector defines what transport (TCP, SSL, HTTP, and so on) and server connection parameters (host, port, and so on) to use for outgoing connections. The following step of this procedure shows how to add the connectors referenced by the **static-connectors** elements of your federated queue configuration.

policy-ref

Name of the queue policy configured on the downstream broker that is applied to the upstream broker.

The additional options that you can specify for an **upstream** element are described below:

name

Name of the upstream broker configuration. In this example, the names correspond to upstream brokers called **eu-east-1** and **eu-west-1**.

user

User name to use when creating the connection to the upstream broker. If not specified, the shared user name that is specified in the configuration of the **federation** element is used.

password

Password to use when creating the connection to the upstream broker. If not specified, the shared password that is specified in the configuration of the **federation** element is used.

call-failover-timeout

Similar to **call-timeout**, but used when a call is made during a failover attempt. The default value is **-1**, which means that the timeout is disabled.

call-timeout

Time, in milliseconds, that a federation connection waits for a reply from a remote broker when it transmits a packet that is a blocking call. If this time elapses, the connection throws an exception. The default value is **30000**.

check-period

Period, in milliseconds, between consecutive “keep-alive” messages that the local broker sends to a remote broker to check the health of the federation connection. If the federation connection is healthy, the remote broker responds to each keep-alive message. If the connection is unhealthy, when the downstream broker fails to receive a response from the upstream broker, a mechanism called a *circuit breaker* is used to block federated consumers. See the description of the **circuit-breaker-timeout** parameter for more information. The default value of the **check-period** parameter is **30000**.

circuit-breaker-timeout

A single connection between a downstream and upstream broker might be shared by many federated queue and address consumers. In the event that the connection between the brokers is lost, each federated consumer might try to reconnect at the same time. To avoid this, a mechanism called a *circuit breaker* blocks the consumers. When the specified timeout value elapses, the circuit breaker re-tries the connection. If successful, consumers are unblocked. Otherwise, the circuit breaker is applied again.

connection-ttl

Time, in milliseconds, that a federation connection stays alive if it stops receiving messages from the remote broker. The default value is **60000**.

discovery-group-ref

As an alternative to defining static connectors for connections to upstream brokers, this element can be used to specify a discovery group that is already configured elsewhere in the **broker.xml** configuration file. Specifically, you specify an existing discovery group as a value for the **discovery-group-name** property of this element.

ha

Specifies whether high availability is enabled for the connection to the upstream broker. If the value of this parameter is set to **true**, the local broker can connect to any available broker in an upstream cluster and automatically fails over to a backup broker if the live upstream broker shuts down. The default value is **false**.

initial-connect-attempts

Number of initial attempts that the downstream broker will make to connect to the upstream broker. If this value is reached without a connection being established, the upstream broker is considered permanently offline. The downstream broker no longer routes messages to the upstream broker. The default value is **-1**, which means that there is no limit.

max-retry-interval

Maximum time, in milliseconds, between subsequent reconnection attempts when connection to the remote broker fails. The default value is **2000**.

reconnect-attempts

Number of times that the downstream broker will try to reconnect to the upstream broker if the connection fails. If this value is reached without a connection being re-established, the upstream broker is considered permanently offline. The downstream broker no longer routes messages to the upstream broker. The default value is **-1**, which means that there is no limit.

retry-interval

Period, in milliseconds, between subsequent reconnection attempts, if connection to the remote broker has failed. The default value is **500**.

retry-interval-multiplier

Multiplying factor that is applied to the value of the **retry-interval** parameter. The default value is **1**.

share-connection

If there is both a downstream and upstream connection configured for the same broker, then the same connection will be shared, as long as both of the downstream and upstream configurations set the value of this parameter to **true**. The default value is **false**.

26. On the local broker, add connectors to the remote brokers. These are the connectors referenced in the **static-connectors** elements of your federated address configuration. For example:

```
<connectors>
  <connector   name="eu-west-1-connector">tcp://localhost:61616</connector>
  <connector   name="eu-east-1-connector">tcp://localhost:61617</connector>
</connectors>
```

Configuring downstream queue federation

The following example shows how to configure downstream queue federation.

Downstream queue federation enables you to add configuration on the local broker that one or more remote brokers use to connect back to the local broker. The advantage of this approach is that you can keep all federation configuration on a single broker. This might be a useful approach for a hub-and-spoke topology, for example.

NOTE

Downstream queue federation reverses the direction of the federation connection versus upstream queue configuration. Therefore, when you add remote brokers to your configuration, these become considered as the *downstream* brokers. The downstream brokers use the connection information in the configuration to connect back to the local broker, which is now considered to be upstream.

Prerequisites

- You should be familiar with the configuration for upstream queue federation.
- The following example shows how to configure queue federation between standalone brokers. However, you should also be familiar with the requirements for configuring federation for a broker *cluster*.

Procedure

27. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.

28. Add a **<federations>** element that includes a **<federation>** element. For example:

```
<federations>
  <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">
 </federation>
</federations>
```

29. Add a queue policy configuration. For example:

```
<federations>
    ...
    <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">

        <queue-policy name="news-queue-federation" priority-adjustment="-5" include-
federated="true" transformer-ref="new-transformer">

            <include queue-match="#" address-match="queue.bbc.new" />
            <include queue-match="#" address-match="queue.usatoday" />
            <include queue-match="#" address-match="queue.news.#" />

            <exclude queue-match="#.local" address-match="#" />

        </queue-policy>

    </federation>
  ...
</federations>
```

30. If you want to transform messages before transmission, add a transformer configuration. For example:

```
<federations>
    ...
    <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">

        <queue-policy name="news-queue-federation" priority-adjustment="-5" include-
federated="true" transformer-ref="news-transformer">

            <include queue-match="#" address-match="queue.bbc.new" />
            <include queue-match="#" address-match="queue.usatoday" />
            <include queue-match="#" address-match="queue.news.#" />

            <exclude queue-match="#.local" address-match="#" />

        </queue-policy>

        <transformer name="news-transformer">
            <class-name>org.myorg.NewsTransformer</class-name>
            <property key="key1" value="value1"/>
            <property key="key2" value="value2"/>
        </transformer>

    </federation>
  ...
</federations>
```

31. Add a **downstream** element for each remote broker. For example:

```
<federations>
    ...
    <federation name="eu-north-1" user="federation_username"
password="32a10275cf4ab4e9">

        <downstream name="eu-east-1">
            <static-connectors>
                <connector-ref>eu-east-connector1</connector-ref>
            </static-connectors>
            <upstream-connector-ref>netty-connector</upstream-connector-ref>
            <policy ref="news-address-federation"/>
        </downstream>

        <downstream name="eu-west-1" >
            <static-connectors>
                <connector-ref>eu-west-connector1</connector-ref>
            </static-connectors>
            <upstream-connector-ref>netty-connector</upstream-connector-ref>
            <policy ref="news-address-federation"/>
        </downstream>

        <queue-policy name="news-queue-federation" priority-adjustment="-5" include-
federated="true" transformer-ref="new-transformer">

            <include queue-match="#" address-match="queue.bbc.new" />
            <include queue-match="#" address-match="queue.usatoday" />
            <include queue-match="#" address-match="queue.news.#" />

            <exclude queue-match="#.local" address-match="#" />

        </queue-policy>

        <transformer name="news-transformer">
            <class-name>org.myorg.NewsTransformer</class-name>
            <property key="key1" value="value1"/>
            <property key="key2" value="value2"/>
        </transformer>

    </federation>
  ...
</federations>
```

As shown in the preceding configuration, the remote brokers are now considered to be downstream of the local broker. The downstream brokers use the connection information in the configuration to connect back to the local (that is, *upstream*) broker.

32. On the local broker, add connectors and acceptors used by the local and remote brokers to establish the federation connection. For example:

```
<connectors>
  <connector name="netty-connector">tcp://localhost:61616</connector>
  <connector  name="eu-west-1-connector">tcp://localhost:61616</connector>
  <connector  name="eu-east-1-connector">tcp://localhost:61617</connector>
</connectors>

<acceptors>
  <acceptor name="netty-acceptor">tcp://localhost:61616</acceptor>
</acceptors>
```

connector name="netty-connector"

Connector configuration that the local broker sends to the remote broker. The remote broker use this configuration to connect back to the local broker.

connector name="eu-west-1-connector" , **connector name="eu-east-1-connector"**

Connectors to remote brokers. The local broker uses these connectors to connect to the remote brokers and share the configuration that the remote brokers need to connect back to the local broker.

acceptor name="netty-acceptor"

Acceptor on the local broker that corresponds to the connector used by the remote broker to connect back to the local broker.