Red Hat AMQ Broker 7.12

Configuring AMQ Broker

For Use with AMQ Broker 7.12

Red Hat AMQ Broker 7.12 Configuring AMQ Broker

For Use with AMQ Broker 7.12

Legal Notice

Abstract

This guide describes how to configure AMQ Broker.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

CHAPTER 1. 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.

1.1. 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.

1.2. 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 ...>

    ...
```

```xml
<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, **#**).

```xml
<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.

```xml
<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>
```

1.3. 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. For more information, see Appendix B, *Address Setting Configuration Elements*.

- 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**)

Additional resources

- To learn how to use the management API, see Using the Management API in Managing AMQ Broker

1.4. 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

1. 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>
```

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

3. 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
      ```

Additional resources

- For more information about XML Inclusions (XIncludes), see https://www.w3.org/TR/xinclude/.

1.4.1. 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**)

Additional resources

- To learn how to use the management API, see Using the Management API in Managing AMQ Broker

1.4.2. 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.

1.5. 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.

1.6. DOCUMENT CONVENTIONS

This document uses the following conventions for the **sudo** command, file paths, and replaceable values.

The **sudo** command

In this document, **sudo** is used for any command that requires root privileges. You should always exercise caution when using **sudo**, as any changes can affect the entire system.

For more information about using **sudo**, see Managing sudo access.

About the use of file paths in this document

In this document, all file paths are valid for Linux, UNIX, and similar operating systems (for example, **/home/...**). If you are using Microsoft Windows, you should use the equivalent Microsoft Windows paths (for example, **C:\Users\...**).

Replaceable values

This document sometimes uses replaceable values that you must replace with values specific to your environment. Replaceable values are lowercase, enclosed by angle brackets (**< >**), and are styled using italics and **monospace** font. Multiple words are separated by underscores (**_**) .

For example, in the following command, replace ***<install_dir>*** with your own directory name.

```
$ <install_dir>/bin/artemis create mybroker
```

CHAPTER 2. CONFIGURING ACCEPTORS AND CONNECTORS IN 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>
```

The sections that follow describe two important configuration elements that are required for network connections and in-VM connections; *acceptors* and *connectors*. These sections show how to configure acceptors and connectors for TCP, HTTP, and SSL/TLS network connections, as well as in-VM connections.

2.1. 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>
```

2.2. 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.

Additional resources

- For details on the available configuration options for acceptors and connectors, see Appendix A, *Acceptor and Connector Configuration Parameters* .

2.3. 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>
```

2.4. CONFIGURING CONNECTORS

The following example shows how to configure a connector.

Procedure

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

2. 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.

3. 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.

Additional resources

- For details on the available configuration options for acceptors and connectors, see Appendix A, *Acceptor and Connector Configuration Parameters* .

- To learn how to configure a broker connector in the AMQ Core Protocol JMS client, see Configuring a broker connector in the AMQ Core Protocol JMS documentation.

2.5. CONFIGURING 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. See Section 5.1, "Securing connections" for more details.

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. For more information, see:

  - Section 2.2, "Configuring acceptors"

  - Section 2.4, "Configuring connectors"

Procedure

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

2. 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**.

3. (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**.

Additional resources

- For details on the available configuration options for TCP connections, see *Appendix A, Acceptor and Connector Configuration Parameters* .

2.6. 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. For more information, see:

  - Section 2.2, "Configuring acceptors"

  - Section 2.4, "Configuring connectors"

Procedure

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

2. 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.

3. (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**.

Additional resources

- An HTTP connection uses the same configuration parameters as TCP, but it also has some of its own. For details on all of the available configuration options for HTTP connections, see *Appendix A, Acceptor and Connector Configuration Parameters* .

- For a full working example that shows how to use HTTP, see the JMS HTTP example .

2.7. CONFIGURING SECURE NETWORK CONNECTIONS

You can secure network connections using TLS/SSL. For more information, see Section 5.1, "Securing connections".

2.8. 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. For more information, see:

  - Section 2.2, "Configuring acceptors"

  - Section 2.4, "Configuring connectors"

Procedure

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

2. 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.

3. (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.

CHAPTER 3. CONFIGURING MESSAGING PROTOCOLS IN NETWORK 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.

3.1. CONFIGURING A NETWORK CONNECTION TO USE A MESSAGING PROTOCOL

You must associate a protocol with a network connection before you can use it. (See Chapter 2, *Configuring acceptors and connectors in network connections* for more information about how to create and configure network connections.) 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</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>
```

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.

Table 3.1. Additional parameters in 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**. To learn more about how credits are used to block AMQP messages, see Section 7.3.2, "Blocking AMQP producers".
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. To learn more about how credits are used to block AMQP messages, see Section 7.3.2, "Blocking AMQP producers".
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**. For more information about configuring a prefix to enable clients to specify a routing type when connecting to an address, see Section 4.6, "Adding a routing type to an acceptor configuration".

Acceptor(s)	Parameter	Description
	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**. For more information about configuring a prefix to enable clients to specify a routing type when connecting to an address, see Section 4.6, "Adding a routing type to an acceptor configuration".

Additional resources

- For information about other parameters that you can configure for Netty network connections, see Appendix A, *Acceptor and Connector Configuration Parameters* .

3.2. 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

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

2. 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.

See the AMQP 1.0 specification for more information about the protocol and its features.

3.2.1. 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:

Table 3.2. Subscription queues

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.

3.2.2. Configuring AMQP security

The broker supports AMQP SASL Authentication. See Security for more information about how to configure SASL-based authentication on the broker.

3.3. 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

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

2. 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.

For more information about the MQTT protocol, see the specification.

- MQTT 3.11 specification

- MQTT 5.0 specification

3.3.1. 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**.

3.4. 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

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

2. 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.

For more details, see the Openwire examples.

3.5. 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

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

2. 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.

For an example of how to configure a broker with STOMP, see the STOMP example.

3.5.1. 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).

3.5.2. 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

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

2. 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**.

3.5.3. 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

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

2. 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.

3.5.4. 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.

Table 3.3. Mapping 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).

See the STOMP specification for more information about message headers.

3.5.5. 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**. See the STOMP specification for more information about the destination frame.

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
  ^@
  ```

CHAPTER 4. CONFIGURING ADDRESSES AND QUEUES

4.1. 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.

Table 4.1. Address routing types

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.

Additional resources

- For more information about configuring:

  - Point-to-point messaging using the **anycast** routing type, see Section 4.3, "Configuring addresses for point-to-point messaging"

  - Publish-subscribe messaging using the **multicast** routing type, see Section 4.4, "Configuring addresses for publish-subscribe messaging"

4.1.1. 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. For more information, see Section 4.2.1, "AMQ Broker wildcard syntax".

- 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 (**.**). For more information, see Section 4.2.1, "AMQ Broker wildcard syntax".

4.2. 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.

4.2.1. 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.

Table 4.2. Matching addresses using wildcards

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**.

4.2.2. 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>
   ```

4.2.3. Configuring the broker wildcard syntax

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

Procedure

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

2. 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.

4.3. 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.

Figure 4.1. Point-to-point messaging
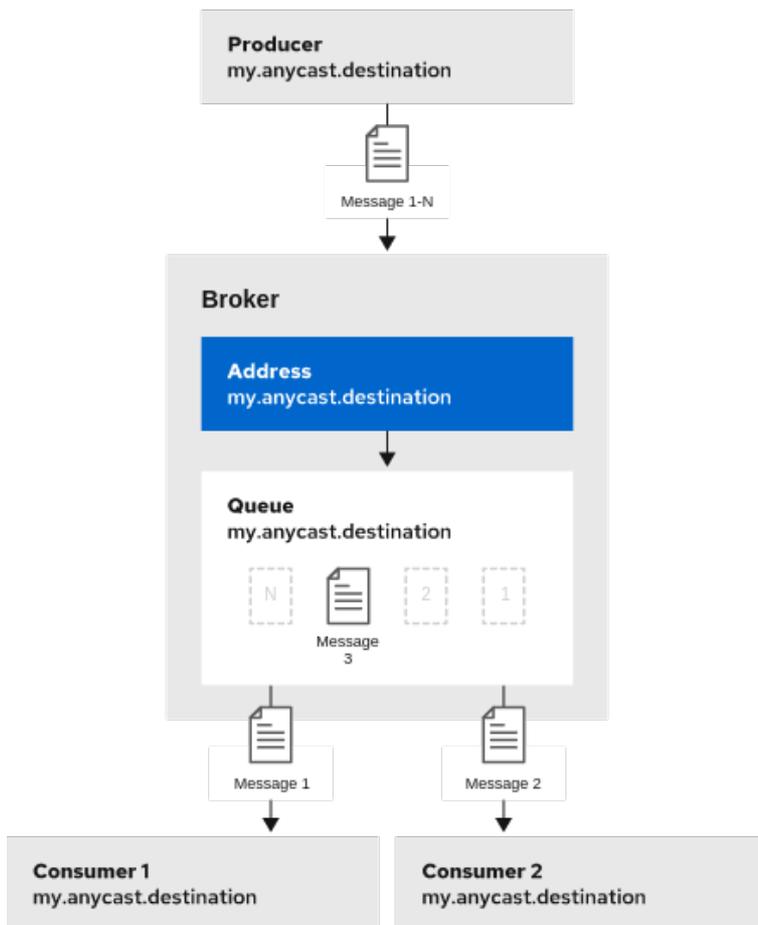
4.3.1. Configuring basic point-to-point messaging

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

Procedure

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

2. 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>
```

4.3.2. 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.

Figure 4.2. Point-to-point messaging using two queues
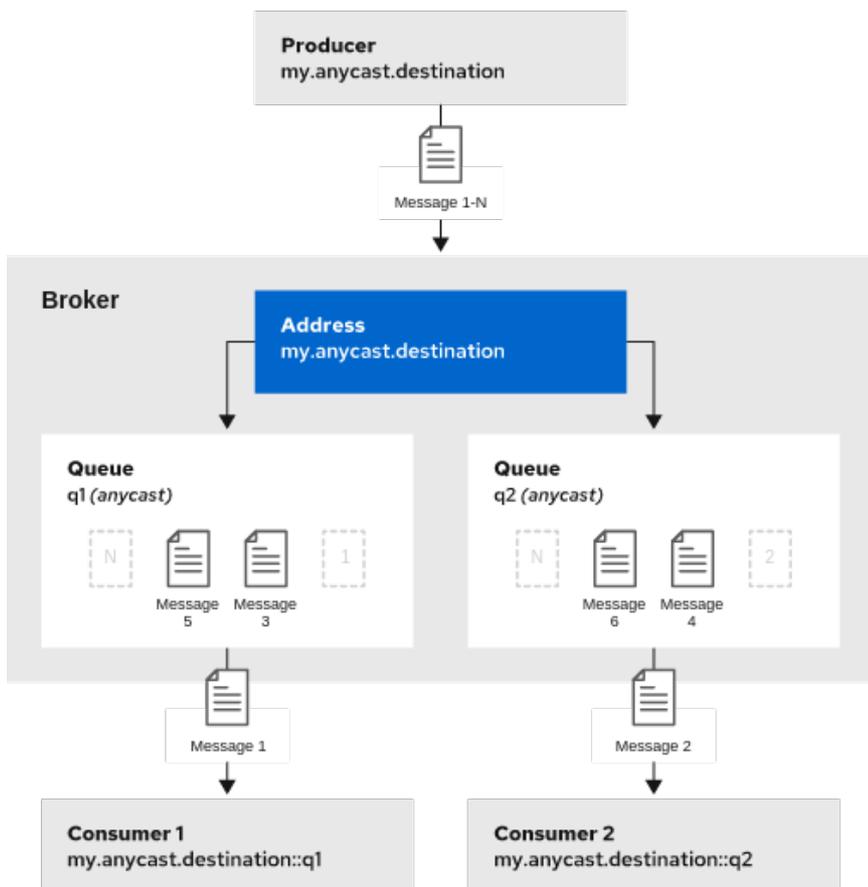
The following procedure shows how to configure point-to-point messaging for an address that has multiple queues.

Procedure

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="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.

Additional resources

- For more information about:

  - Specifying Fully Qualified Queue Names, see Section 4.9, "Specifying a fully qualified queue name".

  - How to configure message load balancing for a broker cluster, see Section 14.1.1, "How broker clusters balance message load".

4.4. 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.

Figure 4.3. Publish-subscribe messaging
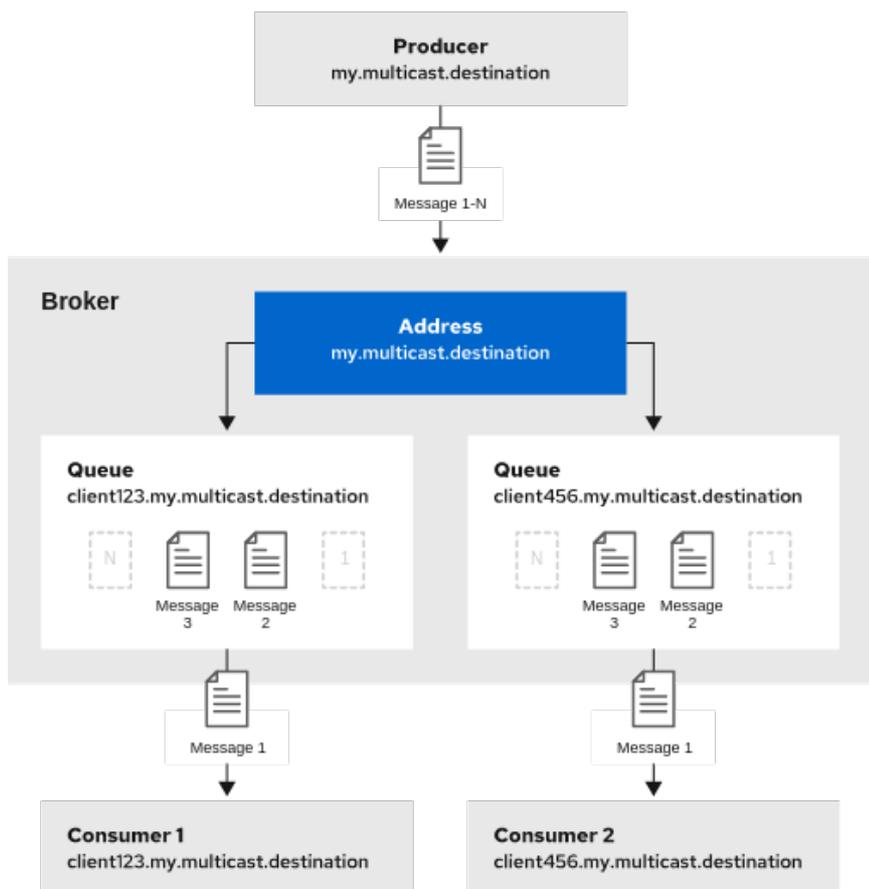
The following procedure shows how to configure an address for publish-subscribe messaging.

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>
```

4.5. 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.

Figure 4.4. Point-to-point and publish-subscribe messaging
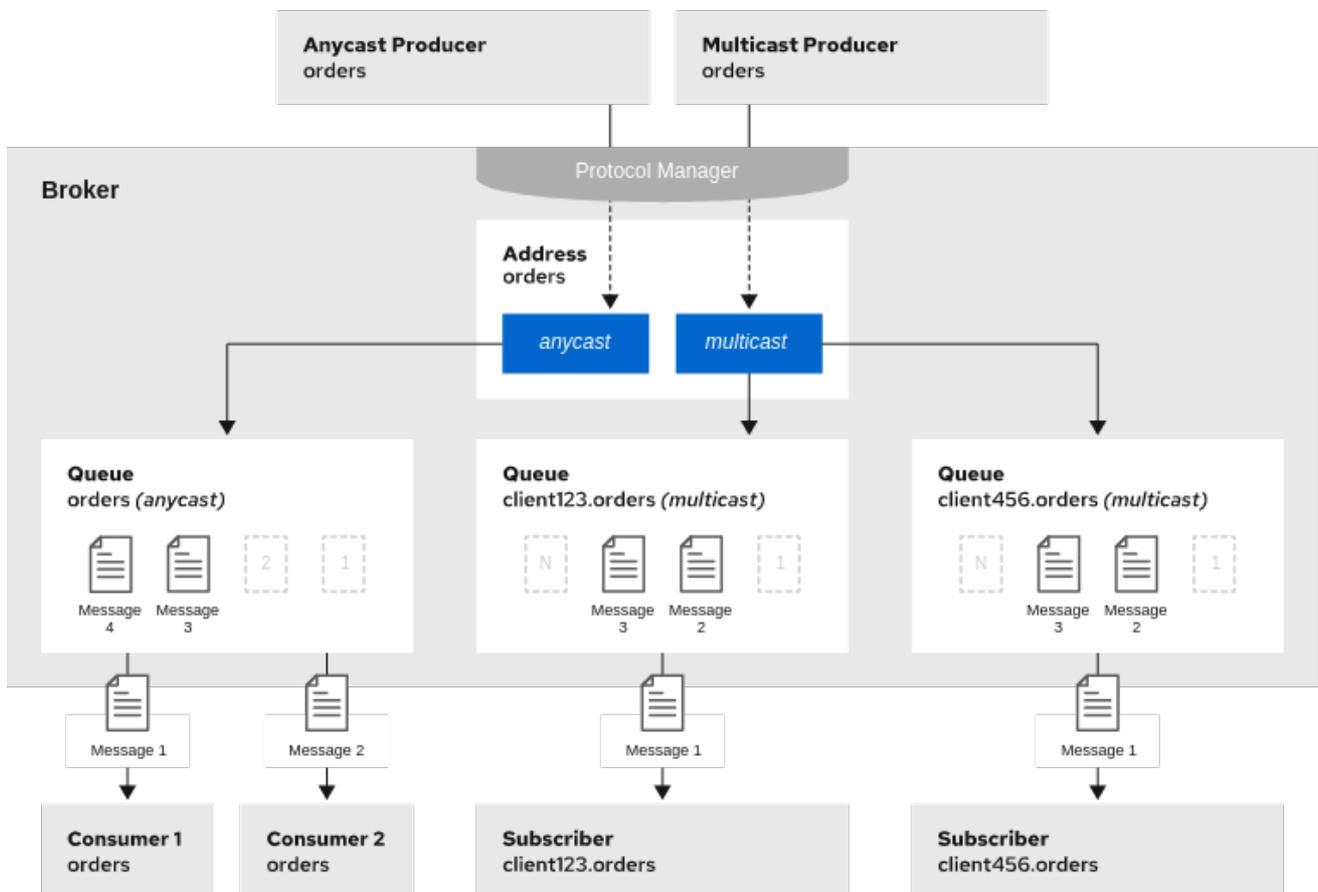
The following procedure shows how to configure an address for both point-to-point and publish-subscribe messaging.

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**. For more information, see Chapter 3, *Configuring messaging protocols in network connections*.

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.

4.6. 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.

4.7. 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. See Section 4.8.3, "Protocol managers and addresses" for more information. 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.

4.7.1. 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

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

2. 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.

4.7.2. 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

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

2. 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.

3. 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>
```

4.7.3. 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

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

2. 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>
```

4.8. 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.

4.8.1. 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.

Table 4.3. Configuration elements 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**

4.8.2. 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

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

2. 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**.

Additional resources

- For more information about:

  - The wildcard syntax that you can use when configuring addresses, see Section 4.2, "Applying address settings to sets of addresses".

  - Routing types, see Section 4.1, "Addresses, queues, and routing types".

4.8.3. 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.

Table 4.4. Protocol manager actions for different queue types

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.
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.

4.9. 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);
```

4.10. 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.

Additional resources

- For more information about:

  - Fully Qualified Queue Names, see Section 4.9, "Specifying a fully qualified queue name"

  - Message grouping, see Using message groups in the AMQ Core Protocol JMS documentation.

4.11. 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.

> **NOTE**
>
> Last-value queues do not work as expected if messages sent to the queues are paged. Set the value of the **address-full-policy** parameter for addresses that have last-value queues to **DROP**, **BLOCK** or **FAIL** to ensure that messages sent to these queues are not paged. For more information, see Section 7.2, "Configuring message dropping".

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.

4.11.1. Configuring last value queues individually

The following procedure shows to configure last value queues individually.

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

2. 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>
```

3. 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>
```

4.11.2. Configuring last value queues for addresses

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

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

2. 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.

3. 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>
```

4. 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>
```

Additional resources

- For more information about the wildcard syntax that you can use when configuring addresses, see Section 4.2, "Applying address settings to sets of addresses" .

4.11.3. 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.

4.11.4. 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. For more information, see:

    - Section 4.11.1, "Configuring last value queues individually"

    - Section 4.11.2, "Configuring last value queues for addresses"

Procedure

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

2. 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>
    ```

3. 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>
    ```
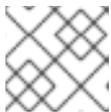
    NOTE

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

4.12. 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.

4.12.1. Configuring message expiry

The following procedure shows how to configure message expiry.

Procedure

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

2. 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.

3. 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**.

4. 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).

5. 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**.

4.12.2. 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. For more information, see Section 4.12.1, "Configuring message expiry".

Procedure

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

2. 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 ...>
   ```

3. 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.

Additional resources

- For more information about address settings used to configure automatic deletion of automatically-created addresses and queues, see Section 4.8.2, "Configuring automatic creation and deletion of addresses and queues".

4.13. 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

4.13.1. Configuring a dead letter address

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

Procedure

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

2. 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**.

3. 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*.

Additional resources

- For more information about using wildcards in address settings, see Section 4.2, "Applying address settings to sets of addresses".

4.13.2. 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. For more information, see Section 4.13.1, "Configuring a dead letter address" .

Procedure

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

2. 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 ...>
```

3. 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).

4.14. 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.
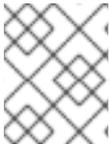
> **NOTE**
>
> The properties that the broker applies are *internal* properties These properties are are not exposed to clients for regular use, but **can** be specified by a client in a filter.

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

Table 4.5. Annotations and properties applied 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.

Additional resources

- For an example of configuring an AMQP client to filter AMQP messages based on annotations, see Section 13.3, "Filtering AMQP Messages Based on Properties on Annotations" .

4.15. 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. For more information, see Chapter 4, *Configuring addresses and queues* .

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.

4.16. 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>
```

4.17. 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.

4.17.1. Configuring exclusive queues individually

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

Procedure

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

2. 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>
```

4.17.2. 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.

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

2. 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**.

3. 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>
   ```

Additional resources

- For more information about the wildcard syntax that you can use when configuring addresses, see Section 4.2, "Applying address settings to sets of addresses" .

4.18. 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.

Additional resources

- For more information about using wildcards in address settings, see Section 4.2, "Applying address settings to sets of addresses".

4.19. 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.

4.19.1. Configuring ring queues

The following procedure shows how to configure a ring queue.

Procedure

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

2. 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).

3. 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.

4.19.2. Troubleshooting ring queues

This section describes situations in which the behavior of a ring queue appears to differ from its configuration.

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:

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

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

3. 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.

4. 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**.

Additional resources

- The broker creates internal instances of ring queues when you configure retroactive addresses. To learn more, see Section 4.20, "Configuring retroactive addresses".

4.20. 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.

Additional resources

- For more information about ring queues, see Section 4.19, "Configuring ring queues".

4.21. 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>
   ```

4.22. 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*. These options are described in the sections that follow.

> **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.

4.22.1. 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.

4.22.2. 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 4.5. Address federation in a symmetric topology
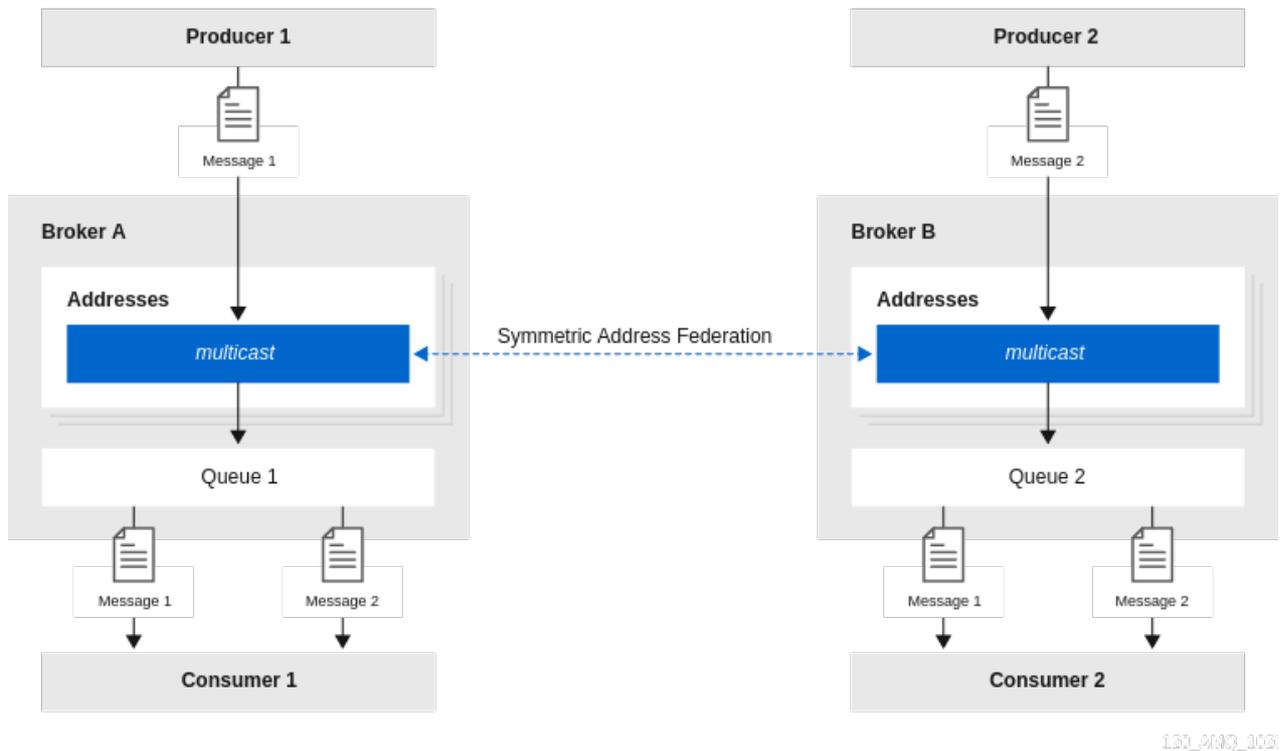
When configuring address federation for a symmetric topology, it is important to set the value of the **max-hops** property of the address policy to **1**. This ensures that messages are copied **only once**, avoiding cyclic replication. If this property is set to a larger value, consumers will receive 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 4.6. Address federation in a full mesh topology
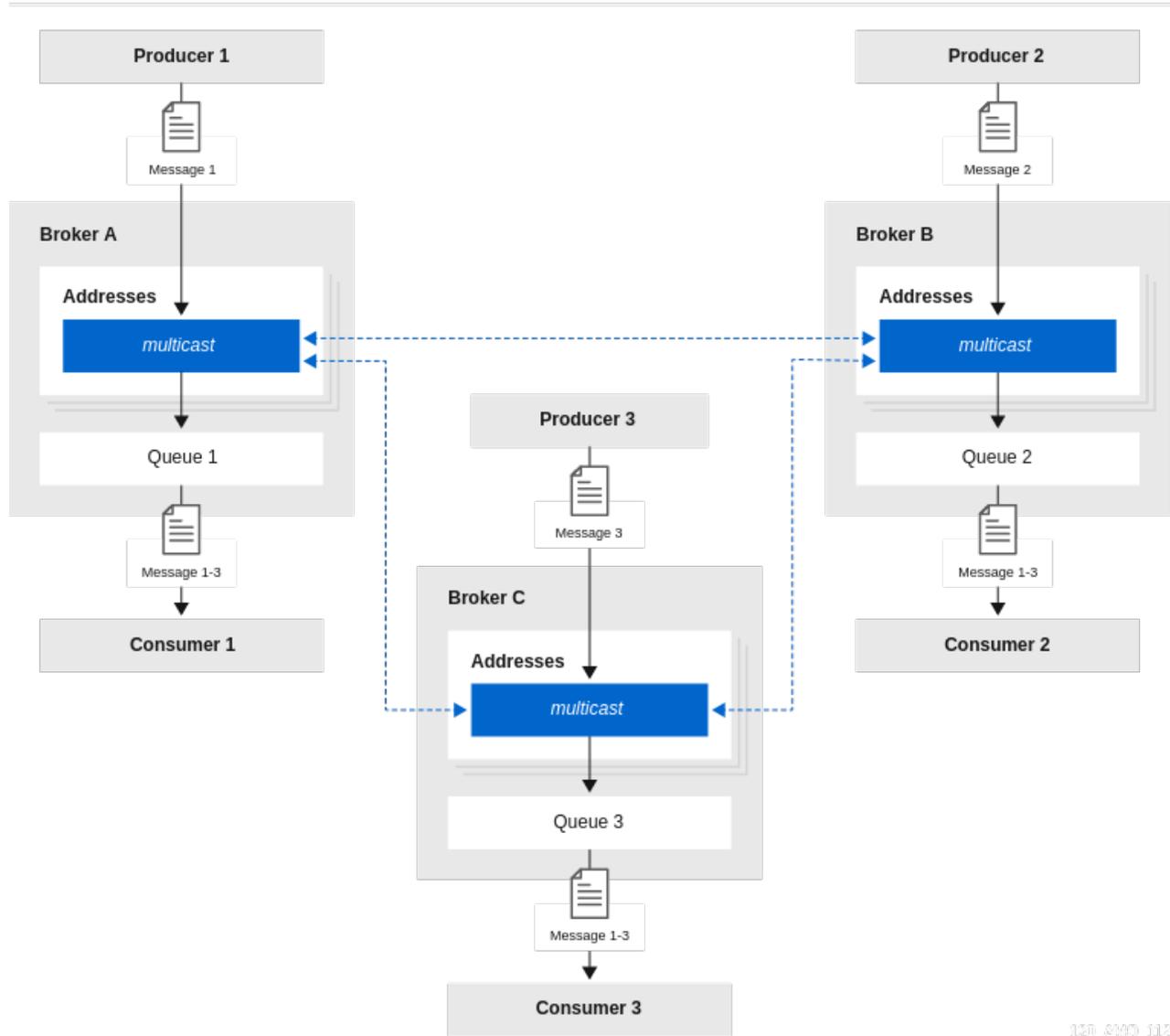
As with a symmetric setup, when configuring address federation for a full mesh topology, it is important to set the value of the **max-hops** property of the address policy to **1**. This ensures that messages are copied **only once**, avoiding cyclic replication.

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 4.7. Address federation in a ring topology
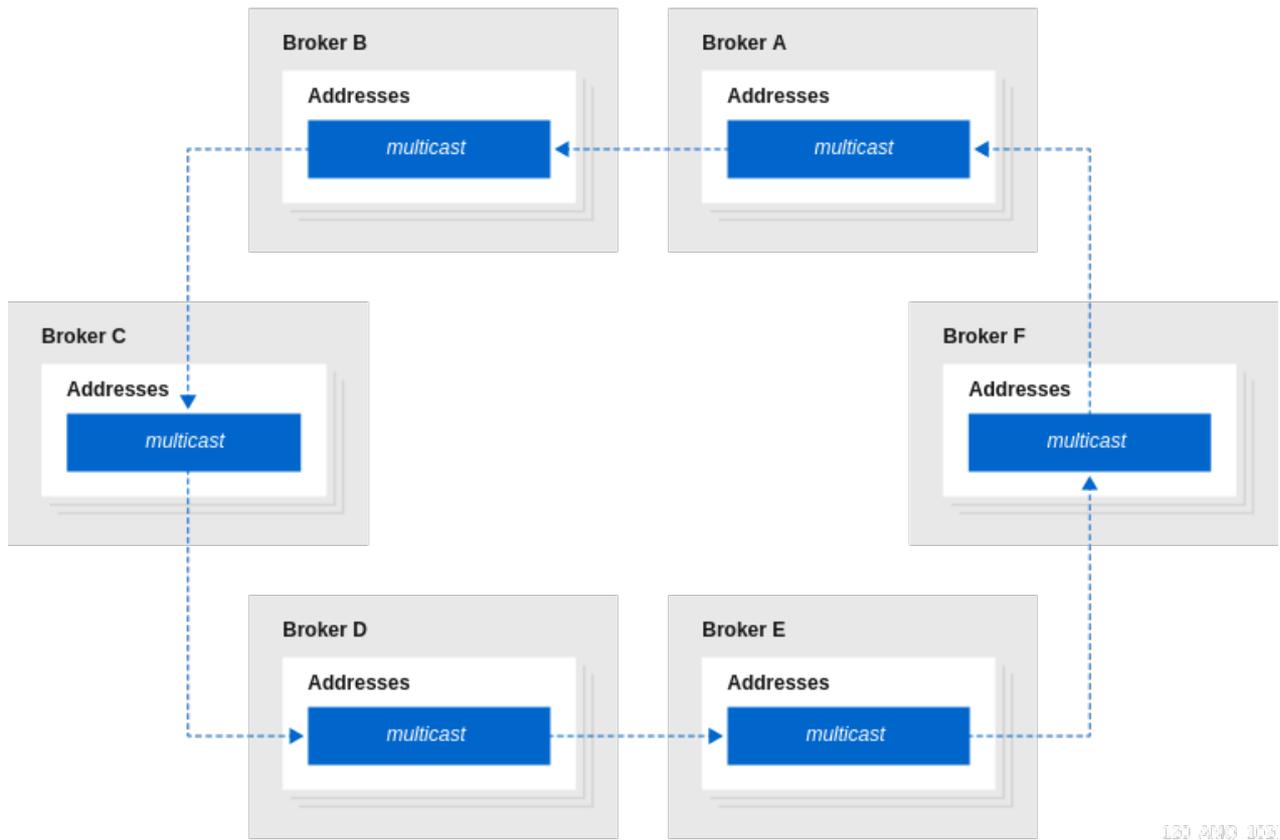
When you configure federation for a ring topology, to avoid cyclic replication, it is important to set the **max-hops** property of the address policy to a value of **n-1**, where *n* is the number of nodes in the ring. For example, in the ring topology shown above, the value of **max-hops** is set to **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 4.8. Address federation in a fan-out topology
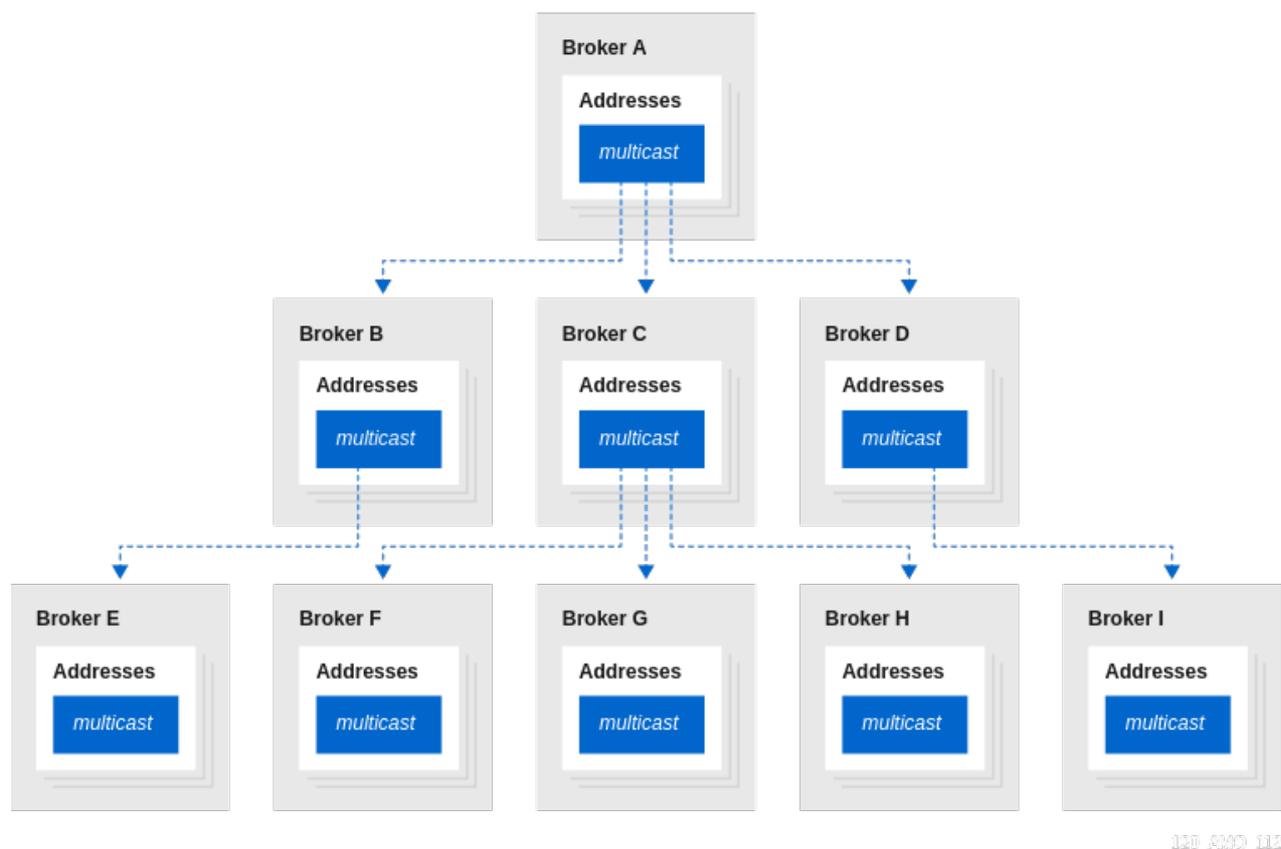
When you configure federation for a fan-out topology, ensure that you set the **max-hops** property of the address policy to a value of **n-1**, where *n* is the number of levels in the tree. For example, in the fan-out topology shown above, the value of **max-hops** is set to **2**. This ensures that every address in the tree sees the message **exactly once**.

4.22.3. 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.

4.22.4. 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.

4.22.4.1. 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.

4.22.4.1.1. 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.

4.22.4.1.2. 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.

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-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.

4.22.4.2. Configuring federation using the Core protocol

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

4.22.4.2.1. 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.

4.22.4.2.2. 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*. For more information, see Section 4.22.4.2.1, "Configuring federation for a broker cluster".

Procedure

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

2. 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. This is described later in this procedure.

3. 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. To learn more about these requirements, see Section 4.22.2, "Common topologies for address federation".

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.

4. 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**.

5. (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.

6. 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. For more information about discovery groups, see Section 14.1.6, "Broker discovery methods" .

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**.

7. 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:

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

4.22.4.2.3. 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. See Section 4.22.4.2.2, "Configuring 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*. For more information, see Section 4.22.4.2.1, "Configuring federation for a broker cluster".

Procedure

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

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

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

3. 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>
```

4. 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>
```

5. 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.

6. 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.

4.22.4.2.4. 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 4.9. Symmetric queue federation
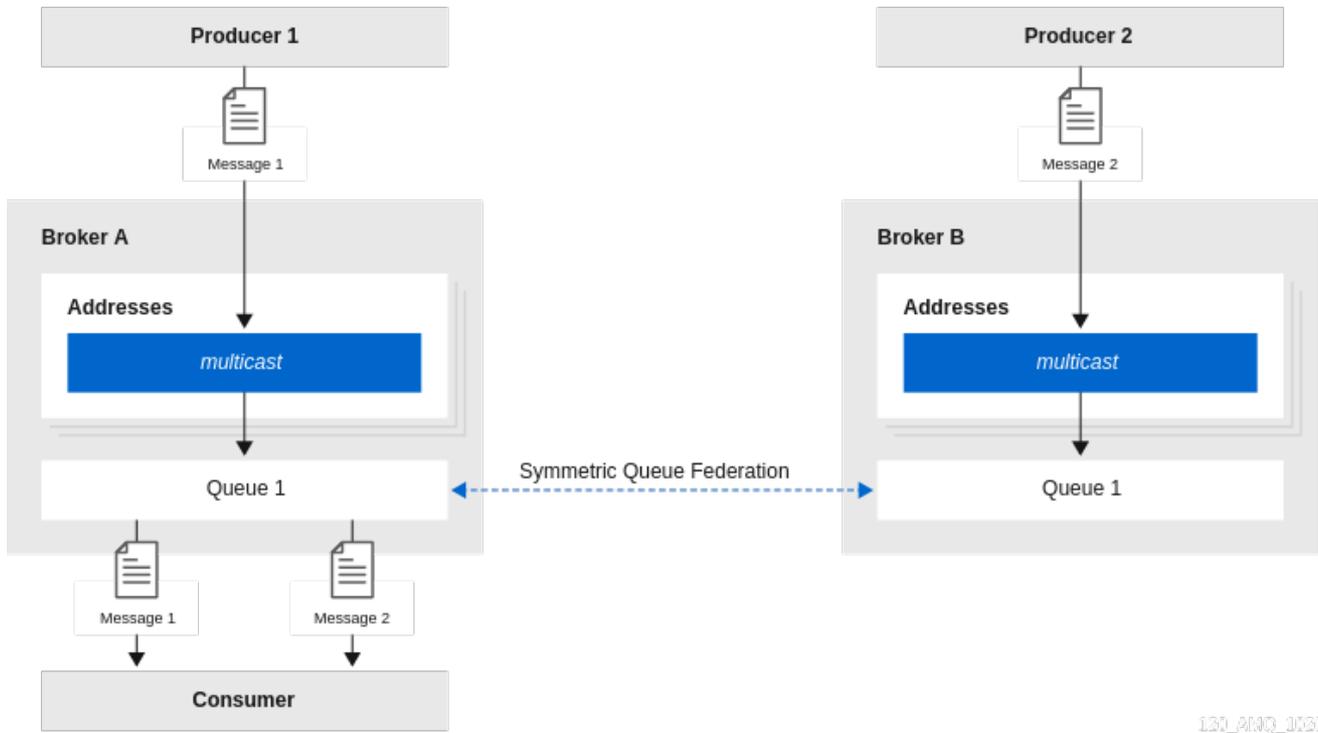
The remote queues do not need to be reconfigured and they do not have to be on the same broker or in the same cluster. All of the configuration needed to establish the remote links and the federated queue is on the local broker.

4.22.4.2.4.1. 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 4.10. Multi-region queue federation
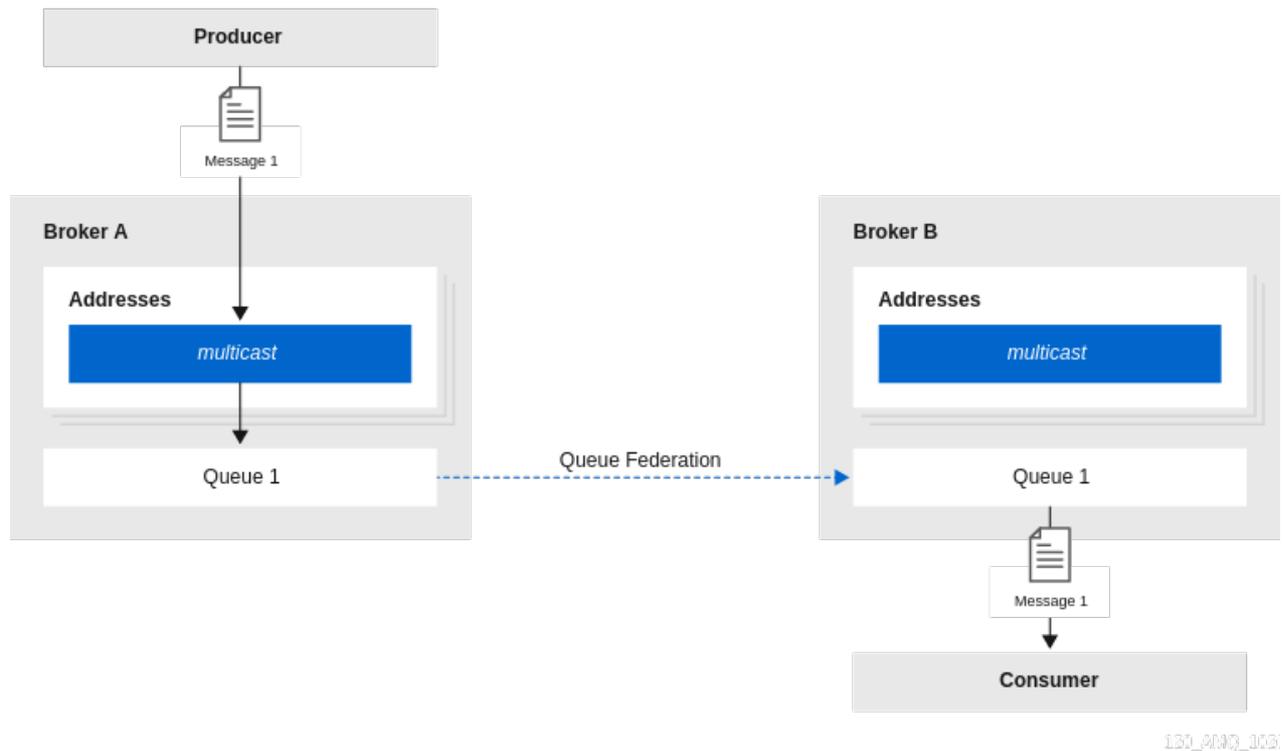
Communicating between a secure enterprise LAN and a DMZ

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.

4.22.4.2.5. 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*. For more information, see Section 4.22.4.2.1, "Configuring federation for a broker cluster".

Procedure

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

2. 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.

3. 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.

4. 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.

5. 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.

6. 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. For more information about discovery groups, see Section 14.1.6, "Broker discovery methods" .

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**.

7. 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>
```

> **NOTE**
>
> If you want large messages to flow across federation connections, set the **consumerWindowSize** parameter to -1 on the connectors used for the federation connections. Setting the **consumerWindowSize** parameter to -1 means that there is no limit set for this parameter, which allows large messages to flow across connections. For example:

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

For more information on large messages, see Chapter 8, *Handling large messages* .

4.22.4.2.6. 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. 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 queue federation. See Section 4.22.4.2.5, "Configuring 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*. For more information, see Section 4.22.4.2.1, "Configuring federation for a broker cluster".

Procedure

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

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

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

3. 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>
   ```

4. 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>
```

5. 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.

6. 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.

> **NOTE**
>
> If you want large messages to flow across the federation connections, set the **consumerWindowSize** parameter to –1 on the connectors used for the federation connections. Setting the **consumerWindowSize** parameter to –1 means that there is no limit set for this parameter, which allows large messages to flow across connections. For example:
>
> ```
> <connectors>
>   <connector name="eu-west-1-connector">tcp://localhost:61616?
> ```

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

For more information on large messages, see Chapter 8, *Handling large messages* .

CHAPTER 5. SECURING BROKERS

5.1. SECURING CONNECTIONS

When brokers are connected to messaging clients, or brokers are connected to other brokers, you can secure these connections using Transport Layer Security (TLS).

There are two TLS configurations that you can use:

- One-way TLS, where only the broker presents a certificate. This is the most common configuration.

- Two-way (or *mutual*) TLS, where both the broker and the client (or other broker) present certificates.

5.1.1. Configuring one-way TLS

The following procedure shows how to configure a given acceptor for one-way TLS.

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

2. For a given acceptor, add the **sslEnabled** key and set the value to **true**. In addition, add the **keyStorePath** and **keyStorePassword** keys. Set values that correspond to your broker key store. For example:

   ```
   <acceptor name="artemis">tcp://0.0.0.0:61616?
   sslEnabled=true;keyStorePath=../etc/broker.keystore;keyStorePassword=1234!</acceptor>
   ```

5.1.2. Configuring two-way TLS

The following procedure shows how to configure two-way TLS.

Prerequisites

- You must have already configured your given acceptor for one-way TLS. For more information, see Section 5.1.1, "Configuring one-way TLS".

Procedure

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

2. For the acceptor that you previously configured for one-way TLS, add the **needClientAuth** key. Set the value to **true**. For example:

   ```
   <acceptor name="artemis">tcp://0.0.0.0:61616?
   sslEnabled=true;keyStorePath=../etc/broker.keystore;keyStorePassword=1234!;needClientAuth
   =true</acceptor>
   ```

3. The configuration in the preceding step assumes that the client's certificate is signed by a trusted provider. If the client's certificate is **not** signed by a trusted provider (it is self-signed, for example) then the broker needs to import the client's certificate into a trust store. In this case, add the **trustStorePath** and **trustStorePassword** keys. Set values that correspond to your broker trust store. For example:

```
<acceptor name="artemis">tcp://0.0.0.0:61616?
sslEnabled=true;keyStorePath=../etc/broker.keystore;keyStorePassword=1234!;needClientAuth
=true;trustStorePath=../etc/client.truststore;trustStorePassword=5678!</acceptor>
```

NOTE

AMQ Broker supports multiple protocols, and each protocol and platform has different ways to specify TLS parameters. However, in the case of a client using Core Protocol (a bridge), the TLS parameters are configured on the connector URL, much like on the broker's acceptor.

If a self-signed certificate is listed as a trusted certificate in a Java Virtual Machine (JVM) truststore, the JVM does not validate the expiry date of the certificate. In a production environment, Red Hat recommends that you use a certificate that is signed by a Certificate Authority.

5.1.3. TLS configuration options

The following table shows all of the available TLS configuration options.

Table 5.1. TLS configuration options

Option	Note
sslEnabled	Specifies whether SSL is enabled for the connection. Must be set to **true** to enable TLS. The default value is **false**.
keyStorePath	**When used on an acceptor:** Path to the TLS keystore on the broker that holds the broker certificates (whether self-signed or signed by an authority). **When used on a connector:** Path to the TLS keystore on the client that holds the client certificates. This is relevant for a connector only if you are using two-way TLS. Although you can configure this value on the broker, it is downloaded and used by the client. If the client needs to use a different path from that set on the broker, it can override the broker setting by using either the standard **javax.net.ssl.keyStore** system property or the AMQ-specific **org.apache.activemq.ssl.keyStore** system property. The AMQ-specific system property is useful if another component on the client is already making use of the standard Java system property.

Option	Note
keyStorePassword	**When used on an acceptor:** Password for the keystore on the broker. **When used on a connector:** Password for the keystore on the client. This is relevant for a connector only if you are using two-way TLS. Although you can configure this value on the broker, it is downloaded and used by the client. If the client needs to use a different password from that set on the broker, then it can override the broker setting by using either the standard **javax.net.ssl.keyStorePassword** system property or the AMQ-specific **org.apache.activemq.ssl.keyStorePassword** system property. The AMQ-specific system property is useful if another component on the client is already making use of the standard Java system property.
trustStorePath	**When used on an acceptor:** Path to the TLS truststore on the broker that holds the keys of all clients that the broker trusts. This is relevant for an acceptor only if you are using two-way TLS. **When used on a connector:** Path to TLS truststore on the client that holds the public keys of all brokers that the client trusts. Although you can configure this value on the broker, it is downloaded and used by the client. If the client needs to use a different path from that set on the server then it can override the server-side setting by using either using the standard **javax.net.ssl.trustStore** system property or the AMQ-specific **org.apache.activemq.ssl.trustStore** system property. The AMQ-specific system property is useful if another component on the client is already making use of the standard Java system property.

Option	Note
trustStorePassword	**When used on an acceptor:** Password for the truststore on the broker. This is relevant for an acceptor only if you are using two-way TLS. **When used on a connector:** Password for the truststore on the client. Although you can configure this value on the broker, it is downloaded and used by the client. If the client needs to use a different password from that set on the broker, then it can override the broker setting by using either the standard **javax.net.ssl.trustStorePassword** system property or the AMQ-specific **org.apache.activemq.ssl.trustStorePassword** system property. The AMQ-specific system property is useful if another component on the client is already making use of the standard Java system property.
enabledCipherSuites	A comma-separated list of cipher suites used for TLS communication for both acceptors or connectors. Specify the most secure cipher suite(s) supported by your client application. If you specify a comma-separated list of cipher suites that are common to both the broker and the client, or you do not specify any cipher suites, the broker and client mutually negotiate a cipher suite to use. If you do not know which cipher suites to specify, you can first establish a broker-client connection with your client running in debug mode to verify the cipher suites that are common to both the broker and the client. Then, configure **enabledCipherSuites** on the broker. The cipher suites available depend on the TLS protocol versions used by the broker and clients. If the default TLS protocol version changes after you upgrade the broker, you might need to select an earlier TLS protocol version to ensure that the broker and the clients can use a common cipher suite. For more information, see **enabledProtocols**.

Option	Note
enabledProtocols	Whether used on an acceptor or connector, this is a comma-separated list of protocols used for TLS communication. If you don't specify a TLS protocol version, the broker uses the JVM's default version.
	If the broker uses the default TLS protocol version for the JVM and that version changes after you upgrade the broker, the TLS protocol versions used by the broker and clients might be incompatible. While it is recommended that you use the later TLS protocol version, you can specify an earlier version in **enabledProtocols** to interoperate with clients that do not support a newer TLS protocol version.
needClientAuth	This property is only for an acceptor. It instructs a client connecting to the acceptor that two-way TLS is required. Valid values are **true** or **false**. The default value is **false**.

5.2. AUTHENTICATING CLIENTS

5.2.1. Client authentication methods

To configure client authentication on the broker, you can use the following methods:

User name- and password-based authentication

    Directly validate user credentials using one of these options:

- Check the credentials against a set of properties files stored locally on the broker. You can also configure a *guest* account that allows limited access to the broker and combine login modules to support more complex use cases.

- Configure a *Lightweight Directory Access Protocol* (LDAP) login module to check client credentials against user data stored in a central X.500 directory server.

Certificate-based authentication

    Configure two-way *Transport Layer Security* (TLS) to require both the broker and client to present certificates for mutual authentication. An administrator must also configure properties files that define approved client users and roles. These properties files are stored on the broker.

Kerberos-based authentication

    Configure the broker to authenticate Kerberos security credentials for the client using the GSSAPI mechanism from the *Simple Authentication and Security Layer* (SASL) framework.

The sections that follow describe how to configure both user-and-password- and certificate-based authentication.

Additional resources

- To learn about complete authentication *and* authorization workflows for LDAP and Kerberos, see:

5.2.2. Configuring user and password authentication based on properties files

AMQ Broker supports a flexible role-based security model for applying security to queues based on their addresses. Queues are bound to addresses either one-to-one (for point-to-point messaging) or many-to-one (for publish-subscribe messaging). When a message is sent to an address, the broker looks up the set of queues that are bound to that address and routes the message to that set of queues.

When you require basic user and password authentication, use **PropertiesLoginModule** to define it. This login module checks user credentials against the following configuration files that are stored locally on the broker:

artemis-users.properties

Used to define users and corresponding passwords

artemis-roles.properties

Used to define roles and assign users to those roles

login.config

Used to configure login modules for user and password authentication and guest access

The **artemis-users.properties** file can contain hashed passwords, for security.

The following sections show how to configure:

- Basic user and password authentication

- User and password authentication that includes guest access

5.2.2.1. Configuring basic user and password authentication

The following procedure shows how to configure basic user and password authentication.

Procedure

1. Open the ***<broker_instance_dir>*/etc/login.config** configuration file. By default, this file in a new AMQ Broker 7.12 instance includes the following lines:

   ```
   activemq {
      org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule sufficient
         debug=false
         reload=true
         org.apache.activemq.jaas.properties.user="artemis-users.properties"
         org.apache.activemq.jaas.properties.role="artemis-roles.properties";
          };
   ```

   activemq

   Alias for the configuration.

   org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule

   The implementation class.

   sufficient

Flag that specifies what level of success is required for the **PropertiesLoginModule**. The values that you can set are:

- **required**: The login module is required to succeed. Authentication continues to proceed down the list of login modules configured under the given alias, regardless of success or failure.

- **requisite**: The login module is required to succeed. A failure immediately returns control to the application. Authentication does not proceed down the list of login modules configured under the given alias.

- **sufficient**: The login module is not required to succeed. If it is successful, control returns to the application and authentication does not proceed further. If it fails, the authentication attempt proceeds down the list of login modules configured under the given alias.

- **optional**: The login module is not required to succeed. Authentication continues down the list of login modules configured under the given alias, regardless of success or failure.

org.apache.activemq.jaas.properties.user

Specifies the properties file that defines a set of users and passwords for the login module implementation.

org.apache.activemq.jaas.properties.role

Specifies the properties file that maps users to defined roles for the login module implementation.

2. Open the ***<broker_instance_dir>*/etc/artemis-users.properties** configuration file.

3. Add users and assign passwords to the users. For example:

```
user1=secret
user2=access
user3=myPassword
```

4. Open the ***<broker_instance_dir>*/etc/artemis-roles.properties** configuration file.

5. Assign role names to the users you previously added to the **artemis-users.properties** file. For example:

```
admin=user1,user2
developer=user3
```

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

7. If necessary, add your security domain alias (in this instance, *activemq*) to the file, as shown below:

```
<jaas-security domain="activemq"/>
```

5.2.2.2. Configuring guest access

For a user who does not have login credentials, or whose credentials fail authentication, you can grant limited access to the broker using a guest account.

You can create a broker instance with guest access enabled using the command-line switch; **--allow-anonymous** (the converse of which is **--require-login**).

The following procedure shows how to configure guest access.

Prerequisites

- This procedure assumes that you have already configured basic user and password authentication. To learn more, see Section 5.2.2.1, "Configuring basic user and password authentication".

Procedure

1. Open the ***<broker_instance_dir>*/etc/login.config** configuration file that you previously configured for basic user and password authentication.

2. After the properties login module configuration that you previously added, add a guest login module configuration. For example:

```
activemq {
  org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule sufficient
      debug=true
      org.apache.activemq.jaas.properties.user="artemis-users.properties"
      org.apache.activemq.jaas.properties.role="artemis-roles.properties";

  org.apache.activemq.artemis.spi.core.security.jaas.GuestLoginModule sufficient
      debug=true
      org.apache.activemq.jaas.guest.user="guest"
      org.apache.activemq.jaas.guest.role="restricted";
};
```

org.apache.activemq.artemis.spi.core.security.jaas.GuestLoginModule

> The implementation class.

org.apache.activemq.jaas.guest.user

> The user name assigned to anonymous users.

org.apache.activemq.jaas.guest.role

> The role assigned to anonymous users.

Based on the preceding configuration, user and password authentication module is activated if the user supplies credentials. Guest authentication is activated if the user supplies no credentials, or if the credentials supplied are incorrect.

5.2.2.2.1. Guest access example

The following example shows configuration of guest access for the use case where only those users with no credentials are logged in as guests. In this example, observe that the order of the login modules is reversed compared with the previous configuration procedure. Also, the flag attached to the properties login module is changed to **requisite**.

```
activemq {
  org.apache.activemq.artemis.spi.core.security.jaas.GuestLoginModule sufficient
      debug=true
    credentialsInvalidate=true
```

```
        org.apache.activemq.jaas.guest.user="guest"
        org.apache.activemq.jaas.guest.role="guests";

    org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule requisite
        debug=true
        org.apache.activemq.jaas.properties.user="artemis-users.properties"
        org.apache.activemq.jaas.properties.role="artemis-roles.properties";
};
```

Based on the preceding configuration, the guest authentication module is activated if no login credentials are supplied.

For this use case, the **credentialsInvalidate** option must be set to **true** in the configuration of the guest login module.

The properties login module is activated if credentials are supplied. The credentials must be valid.

Additional resources

- For more information on the *Java Authentication and Authorization Service* (JAAS), see the documentation from your Java vendor. For example, for an Oracle tutorial on configuring **login.config**, see JAAS Login Configuration File in the Oracle Java documentation.

- To learn how to configure an LDAP login module to validate client credentials, see Section 5.4.1, "Configuring LDAP to authenticate clients".

- For more information about encrypting passwords in configuration files, see Section 5.9.2, "Encrypting a password in a configuration file".

5.2.3. Configuring certificate-based authentication

The *Java Authentication and Authorization Service* (JAAS) certificate login module handles authentication and authorization for clients that are using Transport Layer Security (TLS). The module requires two-way *Transport Layer Security* (TLS) to be in use and clients to be configured with their own certificates. Authentication is performed during the TLS handshake, not directly by the JAAS certificate login module.

The role of the certificate login module is to:

- Constrain the set of acceptable users. Only the user *Distinguished Names* (DNs) explicitly listed in the relevant properties file are eligible to be authenticated.

- Associate a list of groups with the received user identity. This facilitates authorization.

- Require the presence of an incoming client certificate (by default, the TLS layer is configured to treat the presence of a client certificate as optional).

The certificate login module stores a collection of certificate DNs in a pair of flat text files. The files associate a user name and a list of group IDs with each DN.

The certificate login module is implemented by the **org.apache.activemq.artemis.spi.core.security.jaas.TextFileCertificateLoginModule** class.

5.2.3.1. Configuring the broker to use certificate-based authentication

The following procedure shows how to configure the broker to use certificate-based authentication.

Prerequisites

- You must have configured the broker to use two-way Transport Layer Security (TLS). For more information, see Section 5.1.2, "Configuring two-way TLS".

Procedure

1. Obtain the Subject *Distinguished Names* (DNs) from user certificates previously imported to the broker key store.

   a. Export the certificate from the key store file into a temporary file. For example:

      ```
      keytool -export -file <file_name> -alias broker-localhost -keystore broker.ks -storepass
      <password>
      ```

   b. Print the contents of the exported certificate:

      ```
      keytool -printcert -file <file_name>
      ```

      The output is similar to that shown below:

      ```
      Owner: CN=localhost, OU=broker, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
      Issuer: CN=localhost, OU=broker, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
      Serial number: 4537c82e
      Valid from: Thu Oct 19 19:47:10 BST 2006 until: Wed Jan 17 18:47:10 GMT 2007
      Certificate fingerprints:
           MD5:  3F:6C:0C:89:A8:80:29:CC:F5:2D:DA:5C:D7:3F:AB:37
           SHA1: F0:79:0D:04:38:5A:46:CE:86:E1:8A:20:1F:7B:AB:3A:46:E4:34:5C
      ```

      The **Owner** entry is the Subject DN. The format used to enter the Subject DN depends on your platform. The string above could also be represented as;

      ```
      Owner: `CN=localhost,\ OU=broker,\ O=Unknown,\ L=Unknown,\ ST=Unknown,\
      C=Unknown`
      ```

2. Configure certificate-based authentication.

   a. Open the **<broker_instance_dir>/etc/login.config** configuration file. Add the certificate login module and reference the user and roles properties files. For example:

      ```
      activemq {
         org.apache.activemq.artemis.spi.core.security.jaas.TextFileCertificateLoginModule
            debug=true
            org.apache.activemq.jaas.textfiledn.user="artemis-users.properties"
            org.apache.activemq.jaas.textfiledn.role="artemis-roles.properties";
      };
      ```

      org.apache.activemq.artemis.spi.core.security.jaas.TextFileCertificateLoginModule

      The implementation class.

      org.apache.activemq.jaas.textfiledn.user

      Specifies the properties file that defines a set of users and passwords for the login module implementation.

      org.apache.activemq.jaas.textfiledn.role

Specifies the properties file that maps users to defined roles for the login module implementation.

b. Open the ***<broker_instance_dir>*/etc/artemis-users.properties** configuration file. Users and their corresponding DNs are defined in this file. For example:

```
system=CN=system,O=Progress,C=US
user=CN=humble user,O=Progress,C=US
guest=CN=anon,O=Progress,C=DE
```

Based on the preceding configuration, for example, the user named **system** is mapped to the **CN=system,O=Progress,C=US** Subject DN.

c. Open the ***<broker_instance_dir>*/etc/artemis-roles.properties** configuration file. The available roles and the users who hold those roles are defined in this file. For example:

```
admins=system
users=system,user
guests=guest
```

In the preceding configuration, for the **users** role, you list multiple users as a comma-separated list.

d. Ensure that your security domain alias (in this instance, *activemq*) is referenced in **bootstrap.xml**, as shown below:

```
<jaas-security domain="activemq"/>
```

5.2.3.2. Configuring certificate-based authentication for AMQP clients

Use the *Simple Authentication and Security Layer* (SASL) EXTERNAL mechanism configuration parameter to configure your AQMP client for certificate-based authentication when connecting to a broker.

The broker authenticates the *Transport Layer Security* (TLS)/*Secure Sockets Layer* (SSL) certificate of your AMQP client in the same way that it authenticates any certificate:

1. The broker reads the TLS/SSL certificate of the client to obtain an identity from the certificate's subject.

2. The certificate subject is mapped to a broker identity by the certificate login module. The broker then authorizes users based on their roles.

The following procedure shows how to configure certificate-based authentication for AMQP clients. To enable your AMQP client to use certificate-based authentication, you must add configuration parameters to the URI that the client uses to connect to the broker.

Prerequisites

- You must have configured:

  - Two-way TLS. For more information, see Section 5.1.2, "Configuring two-way TLS".

  - The broker to use certificate-based authentication. For more information, see Section 5.2.3.1, "Configuring the broker to use certificate-based authentication" .

Procedure

1. Open the resource containing the URI for editing:

   amqps://localhost:5500

2. Add the parameter **sslEnabled=true** to enable TSL/SSL for the connection:

   amqps://localhost:5500?sslEnabled=true

3. Add parameters related to the client trust store and key store to enable the exchange of TSL/SSL certificates with the broker:

   amqps://localhost:5500?
   sslEnabled=true&trustStorePath=*<trust_store_path>*&trustStorePassword=*<trust_store_pass
   word>*&keyStorePath=*<key_store_path>*&keyStorePassword=*<key_store_password>*

4. Add the parameter **saslMechanisms=EXTERNAL** to request that the broker authenticate the client by using the identity found in its TSL/SSL certificate:

   amqps://localhost:5500?
   sslEnabled=true&trustStorePath=*<trust_store_path>*&trustStorePassword=*<trust_store_pass
   word>*&keyStorePath=*<key_store_path>*&keyStorePassword=*<key_store_password>*&saslM
   echanisms=EXTERNAL

Additional resources

- For more information about certificate-based authentication in AMQ Broker, see Section 5.2.3.1, "Configuring the broker to use certificate-based authentication".

- For more information about configuring your AMQP client, go to the Red Hat Customer Portal for product documentation specific to your client.

5.3. AUTHORIZING CLIENTS

5.3.1. Client authorization methods

To authorize clients to perform operations on the broker such as creating and deleting addresses and queues, and sending and consuming messages, you can use the following methods:

User- and role-based authorization

Configure broker security settings for authenticated users and roles.

Configure LDAP to authorize clients

Configure the *Lightweight Directory Access Protocol* (LDAP) login module to handle both authentication and authorization. The LDAP login module checks incoming credentials against user data stored in a central X.500 directory server and sets permissions based on user roles.

Configure Kerberos to authorize clients

Configure the *Java Authentication and Authorization Service* (JAAS) **Krb5LoginModule** login module to pass credentials to **PropertiesLoginModule** or **LDAPLoginModule** login modules, which map the Kerberos-authenticated users to AMQ Broker roles.

5.3.2. Configuring user- and role-based authorization

5.3.2.1. Setting permissions

Permissions are defined against queues (based on their addresses) via the **<security-setting>** element in the **broker.xml** configuration file. You can define multiple instances of **<security-setting>** in the **<security-settings>** element of the configuration file. You can specify an exact address match or you can define a wildcard match using the number sign (**#**) and asterisk (*****) wildcard characters.

Different permissions can be given to the set of queues that match an address. Those permissions are shown in the following table.

Table 5.2. Queue permissions

To allow users to...	Use this parameter...
Create addresses	**createAddress**
Delete addresses	**deleteAddress**
Create a durable queue under matching addresses	**createDurableQueue**
Delete a durable queue under matching addresses	**deleteDurableQueue**
Create a non-durable queue under matching addresses	**createNonDurableQueue**
Delete a non-durable queue under matching addresses	**deleteNonDurableQueue**
Send a message to matching addresses	**send**
Consume a message from a queue bound to matching addresses	**consume**
Invoke management operations by sending management messages to the management address	**manage**
Browse a queue bound to the matching address	**browse**
Have read-only access to a subset of management operations	**view**
Access the mutating management operations, which is any operation not granted access with the **view** permission.	**edit**

For each permission, you specify a list of roles that are granted the permission. If a given user has any of the roles, they are granted the permission for that set of addresses.

The sections that follow show some configuration examples for permissions.

5.3.2.1.1. Configuring message production for a single address

The following procedure shows how to configure message production permissions for a single address.

Procedure

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

2. Add a single **<security-setting>** element within the **<security-settings>** element. For the **match** key, specify an address. For example:

   ```
   <security-settings>
      <security-setting match="my.destination">
         <permission type="send" roles="producer"/>
      </security-setting>
   </security-settings>
   ```

   Based on the preceding configuration, members of the **producer** role have **send** permissions for address **my.destination**.

5.3.2.1.2. Configuring message consumption for a single address

The following procedure shows how to configure message consumption permissions for a single address.

Procedure

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

2. Add a single **<security-setting>** element within the **<security-settings>** element. For the **match** key, specify an address. For example:

   ```
   <security-settings>
      <security-setting match="my.destination">
         <permission type="consume" roles="consumer"/>
      </security-setting>
   </security-settings>
   ```

   Based on the preceding configuration, members of the **consumer** role have **consume** permissions for address **my.destination**.

5.3.2.1.3. Configuring complete access on all addresses

The following procedure shows how to configure complete access to all addresses and associated queues.

Procedure

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

2. Add a single **<security-setting>** element within the **<security-settings>** element. For the **match** key, to configure access to all addresses, specify the number sign (**#**) wildcard character. For example:

```
<security-settings>
  <security-setting match="#">
    <permission type="createDurableQueue" roles="guest"/>
    <permission type="deleteDurableQueue" roles="guest"/>
    <permission type="createNonDurableQueue" roles="guest"/>
    <permission type="deleteNonDurableQueue" roles="guest"/>
    <permission type="createAddress" roles="guest"/>
    <permission type="deleteAddress" roles="guest"/>
    <permission type="send" roles="guest"/>
    <permission type="browse" roles="guest"/>
    <permission type="consume" roles="guest"/>
    <permission type="manage" roles="guest"/>
  </security-setting>
</security-settings>
```

Based on the preceding configuration, all permissions are granted to members of the *guest* role on all queues. This can be useful in a development scenario where anonymous authentication was configured to assign the **guest** role to every user.

Additional resources

- To learn about configuring more complex use cases, see Section 5.3.2.1.4, "Configuring multiple security settings".

5.3.2.1.4. Configuring multiple security settings

The following example procedure shows how to individually configure multiple security settings for a matching set of addresses. This contrasts with the preceding example in this section, which shows how to grant *complete* access to *all* addresses.

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

2. Add a single **<security-setting>** element within the **<security-settings>** element. For the **match** key, include the number sign (**#**) wildcard character to apply the settings to a matching *set* of addresses. For example:

```
<security-setting match="globalqueues.europe.#">
  <permission type="createDurableQueue" roles="admin"/>
  <permission type="deleteDurableQueue" roles="admin"/>
  <permission type="createNonDurableQueue" roles="admin, guest, europe-users"/>
  <permission type="deleteNonDurableQueue" roles="admin, guest, europe-users"/>
  <permission type="send" roles="admin, europe-users"/>
  <permission type="consume" roles="admin, europe-users"/>
</security-setting>
```

match=globalqueues.europe.#

The number sign (**#**) wildcard character is interpreted by the broker as "any sequence of words". Words are delimited by a period (**.**). In this example, the security settings apply to any address that starts with the string *globalqueues.europe.*

permission type="createDurableQueue"

Only users that have the **admin** role can create or delete durable queues bound to an address that starts with the string *globalqueues.europe.*

permission type="createNonDurableQueue"

Any users with the roles **admin**, **guest**, or **europe-users** can create or delete temporary queues bound to an address that starts with the string *globalqueues.europe.*

permission type="send"

Any users with the roles **admin** or **europe-users** can send messages to queues bound to an address that starts with the string *globalqueues.europe.*

permission type="consume"

Any users with the roles **admin** or **europe-users** can consume messages from queues bound to an address that starts with the string *globalqueues.europe.*

3. (Optional) To apply different security settings to a more narrow set of addresses, add another **<security-setting>** element. For the **match** key, specify a more specific text string. For example:

```
<security-setting match="globalqueues.europe.orders.#">
  <permission type="send" roles="europe-users"/>
  <permission type="consume" roles="europe-users"/>
</security-setting>
```

In the second **security-setting** element, the **globalqueues.europe.orders.#** match is more specific than the **globalqueues.europe.#** match specified in the first **security-setting** element. For any addresses that match **globalqueues.europe.orders.#**, the permissions **createDurableQueue**, **deleteDurableQueue**, **createNonDurableQueue**, **deleteNonDurableQueue** are **not** inherited from the first **security-setting** element in the file. For example, for the address **globalqueues.europe.orders.plastics**, the only permissions that exist are **send** and **consume** for the role **europe-users**.

Therefore, because permissions specified in one **security-setting** block are not inherited by another, you can effectively deny permissions in more specific **security-setting** blocks simply by not specifying those permissions.

5.3.2.1.5. Configuring a queue with a user

When a queue is automatically created, the queue is assigned the user name of the connecting client. This user name is included as metadata on the queue. The name is exposed by JMX and in the AMQ Broker management console.

The following procedure shows how to add a user name to a queue that you have manually defined in the broker configuration.

Procedure

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

2. For a given queue, add the **user** key. Assign a value. For example:

```
<address name="ExampleQueue">
  <anycast>
    <queue name="ExampleQueue">
      <user>admin</user>
    </queue>
  </anycast>
</address>
```

Based on the preceding configuration, the **admin** user is assigned to queue **ExampleQueue**.

NOTE

- Configuring a user on a queue does not change any of the security semantics for that queue - it is only used for metadata on that queue.

- The mapping between users and what roles they have is handled by a component called the *security manager*. The security manager reads user credentials from a properties file stored on the broker. By default, AMQ Broker uses the **org.apache.activemq.artemis.spi.core.security.ActiveMQJAASSecurityManager** security manager. This default security manager provides integration with JAAS and Red Hat JBoss Enterprise Application Platform (JBoss EAP) security. To learn how to use a *custom* security manager, see Section 5.6.2, "Specifying a custom security manager".

5.3.2.2. Configuring role-based access control

Role-based access control (RBAC) is used to restrict access to the attributes and methods of MBeans. MBeans are the way the management API is exposed by AMQ Broker to support management operations.

You can restrict access to MBeans by using either of the following methods:

- Configure the **authorisation** element in the **management.xml** file, which is the default method.

- Configure security settings in the **broker.xml** file.

Unlike when you update the **management.xml** file, you do not need to restart the broker after you make changes to security settings in the **broker.xml** file.

5.3.2.2.1. Configuring role-based access control in the **management.xml** file.

The default method of configuring role based access control for management operations is by configuring the **authorisation** element in the **management.xml** file.

The following example procedure shows how to map roles to particular MBeans and their attributes and methods.

Prerequisites

- You defined users and roles. For more information, see Section 5.2.2.1, "Configuring basic user and password authentication".

Procedure

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

2. Search for the **role-access** element and edit the configuration. For example:

```
<role-access>
  <match domain="org.apache.activemq.artemis">
    <access method="list*" roles="view,update,amq"/>
    <access method="get*" roles="view,update,amq"/>
    <access method="is*" roles="view,update,amq"/>
    <access method="set*" roles="update,amq"/>
```

```
            <access method="*" roles="amq"/>
        </match>
    </role-access>
```

- In this case, a match is applied to any MBean attribute that has the domain name **org.apache.activemq.apache**.

- Access of the **view**, **update**, or **amq** role to a matching MBean attribute is controlled by which of the **list***, **get***, **set***, **is***, and ***** access methods that you add the role to. The **method="*"** (wildcard) syntax is used as a catch-all way to specify every other method that is not listed in the configuration. Each of the access methods in the configuration is converted to an MBean method call.

- An invoked MBean method is matched against the methods listed in the configuration. For example, if you invoke a method called **listMessages** on an MBean with the **org.apache.activemq.artemis** domain, then the broker matches access back to the roles defined in the configuration for the **list** method.

- You can also configure access by using the full MBean method name. For example:

```
<access method="listMessages" roles="view,update,amq"/>
```

3. Start or restart the broker.

   - On Linux: ***\<broker_instance_dir\>*/bin/artemis run**

   - On Windows: ***\<broker_instance_dir\>*\bin\artemis-service.exe start**
     You can also match specific MBeans within a domain by adding a **key** attribute that matches an MBean property.

5.3.2.2.1.1. Role-based access examples

This section shows the following examples of applying role-based access control:

- Mapping roles to all queues in a domain .

- Mapping roles to a specific queue in a domain .

- Mapping roles to all queue names that include a specified prefix .

- Mapping different roles to different sets of queues .

The following example shows how to use the **key** attribute to map roles to all queues in a specified domain.

```
<match domain="org.apache.activemq.artemis" key="subcomponent=queues">
  <access method="list*" roles="view,update,amq"/>
  <access method="get*" roles="view,update,amq"/>
  <access method="is*" roles="view,update,amq"/>
  <access method="set*" roles="update,amq"/>
  <access method="*" roles="amq"/>
</match>
```

The following example shows how to use the **key** attribute to map roles to a specific, named queue. In this example, the named queue is **exampleQueue**.

```
<match domain="org.apache.activemq.artemis" key="queue=exampleQueue">
  <access method="list*" roles="view,update,amq"/>
  <access method="get*" roles="view,update,amq"/>
  <access method="is*" roles="view,update,amq"/>
  <access method="set*" roles="update,amq"/>
  <access method="*" roles="amq"/>
</match>
```

The following example shows how to map roles to every queue whose name includes a specified prefix. In this example, an asterisk (*****) wildcard operator is used to match all queue names that start with the prefix *example*.

```
<match domain="org.apache.activemq.artemis" key="queue=example*">
  <access method="list*" roles="view,update,amq"/>
  <access method="get*" roles="view,update,amq"/>
  <access method="is*" roles="view,update,amq"/>
  <access method="set*" roles="update,amq"/>
  <access method="*" roles="amq"/>
</match>
```

You might want to map roles differently for different sets of the same attribute (for example, different sets of queues). In this case, you can include multiple **match** elements in your configuration file. However, it is then possible to have multiple matches in the same domain.

For example, consider two **<match>** elements configured as follows:

```
<match domain="org.apache.activemq.artemis" key="queue=example*">
```

and

```
<match domain="org.apache.activemq.artemis" key="queue=example.sub*">
```

Based on this configuration, a queue named **example.sub.queue** in the **org.apache.activemq.artemis** domain matches both wildcard key expressions. Therefore, the broker needs a prioritization scheme to decide which set of roles to map to the queue; the roles specified in the first **match** element, or those specified in the second **match** element.

When there are multiple matches in the same domain, the broker uses the following prioritization scheme when mapping roles:

- Exact matches are prioritized over wildcard matches

- Longer wildcard matches are prioritized over shorter wildcard matches

In this example, because the longer wildcard expression matches the queue name of **example.sub.queue** most closely, the broker applies the role-mapping configured in the second **<match>** element.

> **NOTE**
>
> The **default-access** element is a catch-all element for every method call that is not handled using the **role-access** or **whitelist** configurations. The **default-access** and **role-access** elements have the same **match** element semantics.

5.3.2.2.1.2. Configuring the **allowlist** element

An *allowlist* is a set of pre-approved domains or MBeans that do not require user authentication. You can provide a list of domains, or list of MBeans, or both, that must bypass the authentication. For example, you might use an **allowlist** to specify any MBeans that are needed by the AMQ Broker management console to run.

The following example procedure shows how to configure the **allowlist** element.

Procedure

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

2. Search for the **allowlist** element and edit the configuration:

   ```
   <allowlist>
     <entry domain="hawtio"/>
   </allowlist>
   ```

   In this example, any MBean with the domain **hawtio** is allowed access without authentication. You can also use wildcard entries of the form **<entry domain="hawtio" key="type=*"/>** for the MBean properties to match.

3. Start or restart the broker.

   - On Linux: ***<broker_instance_dir>*/bin/artemis run**

   - On Windows: ***<broker_instance_dir>*\bin\artemis-service.exe start**

5.3.2.2.2. Configuring role-based access control in the **broker.xml** file

You can configure role based access control for management operations in the **broker.xml** file instead of the **management.xml** file. Updates to permissions in the **broker.xml** file do not require a broker restart to take effect.

In the **broker.xml** file, you can grant either **view** or **edit** permissions for management operations. The specific management operations that are available to a role that has **view** or **edit** permissions is controlled by a predefined regular expression. Any operation that matches the regular expression can be accessed by a role that has the **view** permission and all other operations require the **edit** permission.

Prerequisites

- You defined users and roles. For more information, see Section 5.2.2.1, "Configuring basic user and password authentication".

Procedure

1. Delete the **authorisation** element configuration from the **management.xml** file to prevent the broker from using the default RBAC configuration in this file.

   a. Edit the **<broker_instance_dir>/etc/management.xml** file.

   b. Delete the **authorisation** element configuration from the file.

   c. Save the **<broker_instance_dir>/etc/management.xml** file.

2. Add an environment variable to the broker JVM to configure the broker to use the RBAC configuration in the **broker.xml** file.

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

   b. Add the following argument to the **JAVA_ARGS list** of Java system arguments:
      -Djavax.management.builder.initial=org.apache.activemq.artemis.core.server.management.ArtemisRbacMBeanServerBuilder

   c. Save the **artemis.profile** file.

3. Open the ***<broker_instance_dir>*/etc/broker.xml** file to configure RBAC for management operations.

4. Search for the **security-settings** element and add a **security-setting** element for management operations.
   The format of the match addresses for management operations is:

   <_management-rbac-prefix_>.<_resource type_>.<_resource name_>.<_operation_>

   The default value of the **management-rbac-prefix** parameter is **mops**.

   In following example RBAC configuration, the number sign (#) in the match address grants the **admin** role **view** and **edit** permissions to all MBeans.

   ```
   <security-settings>
    ..
    <security-setting match="mops.#">
     <permission type="view" roles="admin"/>
     <permission type="edit" roles="admin"/>
    </security-setting>
    ..
   </security-setting>
   ```

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

Other examples of role-based access control for management operations

The following example grants a **manager** role **view** and **edit** permission to an address of **activemq.management**. The asterisk (*) in the operation position grants access to all operations.

```
<security-setting match="mops.address.activemq.management.*">
  <permission type="view" roles="manager"/>
</security-setting>
```

The following example has an empty roles list, which denies all users permission to perform the specified operation, **forceFailover**, using the broker MBean.

```
<security-setting match="mops.broker.forceFailover">
  <permission type="edit" roles=""/>
</security-setting>
```

5.3.2.3. Setting resource limits

Sometimes it is helpful to set particular limits on what certain users can do beyond the normal security settings related to authorization and authentication.

5.3.2.3.1. Configuring connection and queue limits

The following example procedure shows how to limit the number of connections and queues that a user can create.

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

2. Add a **resource-limit-settings** element. Specify values for **max-connections** and **max-queues**. For example:

```
<resource-limit-settings>
  <resource-limit-setting match="myUser">
    <max-connections>5</max-connections>
    <max-queues>3</max-queues>
  </resource-limit-setting>
</resource-limit-settings>
```

max-connections

Defines how many sessions the matched user can create on the broker. The default is **-1**, which means that there is no limit. If you want to limit the number of sessions, take into account that each connection to the broker from an AMQ Core Protocol JMS client creates two sessions.

max-queues

Defines how many queues the matched user can create. The default is **-1**, which means that there is no limit.

> **NOTE**
>
> Unlike the **match** string that you can specify in the **address-setting** element of a broker configuration, the **match** string that you specify in **resource-limit-settings cannot** use wildcard syntax. Instead, the match string defines a specific user to which the resource limit settings are applied.

5.4. USING LDAP FOR AUTHENTICATION AND AUTHORIZATION

The LDAP login module enables authentication and authorization by checking the incoming credentials against user data stored in a central X.500 directory server. It is implemented by **org.apache.activemq.artemis.spi.core.security.jaas.LDAPLoginModule**.

5.4.1. Configuring LDAP to authenticate clients

The following example procedure shows how to use LDAP to authenticate clients.

Procedure

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

2. Within the **security-settings** element, add a **security-setting** element to configure permissions. For example:

```
<security-settings>
   <security-setting match="#">
      <permission type="createDurableQueue" roles="user"/>
      <permission type="deleteDurableQueue" roles="user"/>
      <permission type="createNonDurableQueue" roles="user"/>
      <permission type="deleteNonDurableQueue" roles="user"/>
      <permission type="send" roles="user"/>
      <permission type="consume" roles="user"/>
   </security-setting>
</security-settings>
```

The preceding configuration assigns specific permissions for **all** queues to members of the **user** role.

3. Open the ***<broker_instance_dir>*/etc/login.config** file.

4. Configure the LDAP login module, based on the directory service you are using.

   a. If you are using the Microsoft Active Directory directory service, add a configuration that resembles this example:

```
activemq {
  org.apache.activemq.artemis.spi.core.security.jaas.LDAPLoginModule required
     debug=true
     initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
     connectionURL="LDAP://localhost:389"

connectionUsername="CN=Administrator,CN=Users,OU=System,DC=example,DC=com"

     connectionPassword=redhat.123
     connectionProtocol=s
     connectionTimeout="5000"
     authentication=simple
     userBase="dc=example,dc=com"
     userSearchMatching="(CN={0})"
     userSearchSubtree=true
     readTimeout="5000"
     roleBase="dc=example,dc=com"
     roleName=cn
     roleSearchMatching="(member={0})"
     roleSearchSubtree=true
     ;
};
```

> **NOTE**
>
> If you are using Microsoft Active Directory, and a value that you need to specify for an attribute of **connectionUsername** contains a space (for example, **OU=System Accounts**), then you must enclose the value in a pair of double quotes (**""**) and use a backslash (\) to escape each double quote in the pair. For example, **connectionUsername="CN=Administrator,CN=Users,OU=\"System Accounts\",DC=example,DC=com"**.

b. If you are using the ApacheDS directory service, add a configuration that resembles this example:

```
activemq {
  org.apache.activemq.artemis.spi.core.security.jaas.LDAPLoginModule required
    debug=true
    initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
    connectionURL="ldap://localhost:10389"
    connectionUsername="uid=admin,ou=system"
    connectionPassword=secret
    connectionProtocol=s
    connectionTimeout=5000
    authentication=simple
    userBase="dc=example,dc=com"
    userSearchMatching="(uid={0})"
    userSearchSubtree=true
    userRoleName=
    readTimeout=5000
    roleBase="dc=example,dc=com"
    roleName=cn
    roleSearchMatching="(member={0})"
    roleSearchSubtree=true
    ;
};
```

debug

Turn debugging on (**true**) or off (**false**). The default value is **false**.

initialContextFactory

Must always be set to **com.sun.jndi.ldap.LdapCtxFactory**

connectionURL

Location of the directory server using an LDAP URL, __<ldap://Host:Port>. One can optionally qualify this URL, by adding a forward slash, /, followed by the DN of a particular node in the directory tree. The default port of Apache DS is **10389** while for Microsoft AD the default is **389**.

connectionUsername

Distinguished Name (DN) of the user that opens the connection to the directory server. For example, **uid=admin,ou=system**. Directory servers generally require clients to present username/password credentials in order to open a connection.

connectionPassword

Password that matches the DN from **connectionUsername**. In the directory server, in the *Directory Information Tree* (DIT), the password is normally stored as a **userPassword** attribute in the corresponding directory entry.

connectionProtocol

Any value is supported but is effectively unused. This option must be set explicitly because it has no default value.

connectionTimeout

Specify the maximum time, in milliseconds, that the broker can take to connect to the directory server. If the broker cannot connect to the directory sever within this time, it aborts the connection attempt. If you specify a value of zero or less for this property, the

timeout value of the underlying TCP protocol is used instead. If you do not specify a value, the broker waits indefinitely to establish a connection, or the underlying network times out.

When connection pooling has been requested for a connection, then this property specifies the maximum time that the broker waits for a connection when the maximum pool size has already been reached and all connections in the pool are in use. If you specify a value of zero or less, the broker waits indefinitely for a connection to become available. Otherwise, the broker aborts the connection attempt when the maximum wait time has been reached.

authentication

Specifies the authentication method used when binding to the LDAP server. This parameter can be set to either **simple** (which requires a username and password) or **none** (which allows anonymous access).

userBase

Select a particular subtree of the DIT to search for user entries. The subtree is specified by a DN, which specifies the base node of the subtree. For example, by setting this option to **ou=User,ou=ActiveMQ,ou=system**, the search for user entries is restricted to the subtree beneath the **ou=User,ou=ActiveMQ,ou=system** node.

userSearchMatching

Specify an LDAP search filter, which is applied to the subtree selected by **userBase**. See the Section 5.4.1.1, "Search matching parameters" section below for more information.

userSearchSubtree

Specify the search depth for user entries, relative to the node specified by **userBase**. This option is a Boolean. Specifying a value of **false** means that the search tries to match one of the child entries of the **userBase** node (maps to **javax.naming.directory.SearchControls.ONELEVEL_SCOPE**). Specifying a value of **true** means that the search tries to match any entry belonging to the *subtree* of the **userBase** node (maps to **javax.naming.directory.SearchControls.SUBTREE_SCOPE**).

userRoleName

Name of the attribute of the user entry that contains a list of role names for the user. Role names are interpreted as group names by the broker's authorization plug-in. If this option is omitted, no role names are extracted from the user entry.

readTimeout

Specify the maximum time, in milliseconds, that the broker can wait to receive a response from the directory server to an LDAP request. If the broker does not receive a response from the directory server in this time, the broker aborts the request. If you specify a value of zero or less, or you do not specify a value, the broker waits indefinitely for a response from the directory server to an LDAP request.

roleBase

If role data is stored directly in the directory server, one can use a combination of role options (**roleBase**, **roleSearchMatching**, **roleSearchSubtree**, and **roleName**) as an alternative to (or in addition to) specifying the **userRoleName** option. This option selects a particular subtree of the DIT to search for role/group entries. The subtree is specified by a DN, which specifies the base node of the subtree. For example, by setting this option to **ou=Group,ou=ActiveMQ,ou=system**, the search for role/group entries is restricted to the subtree beneath the **ou=Group,ou=ActiveMQ,ou=system** node.

roleName

Attribute type of the role entry that contains the name of the role/group (such as C, O, OU, etc.). If this option is omitted the role search feature is effectively disabled.

roleSearchMatching

Specify an LDAP search filter, which is applied to the subtree selected by **roleBase**. See the Section 5.4.1.1, "Search matching parameters" section below for more information.

roleSearchSubtree

Specify the search depth for role entries, relative to the node specified by **roleBase**. If set to **false** (which is the default) the search tries to match one of the child entries of the **roleBase** node (maps to **javax.naming.directory.SearchControls.ONELEVEL_SCOPE**). If **true** it tries to match any entry belonging to the subtree of the roleBase node (maps to **javax.naming.directory.SearchControls.SUBTREE_SCOPE**).

> **NOTE**
>
> Apache DS uses the **OID** portion of DN path. Microsoft Active Directory uses the **CN** portion. For example, you might use a DN path such as **oid=testuser,dc=example,dc=com** in Apache DS, while you might use a DN path such as **cn=testuser,dc=example,dc=com** in Microsoft Active Directory.

5. Start or restart the broker (service or process).

5.4.1.1. Search matching parameters

userSearchMatching

Before passing to the LDAP search operation, the string value provided in this configuration parameter is subjected to string substitution, as implemented by the **java.text.MessageFormat** class.
This means that the special string, **{0}**, is substituted by the username, as extracted from the incoming client credentials. After substitution, the string is interpreted as an LDAP search filter (the syntax is defined by the IETF standard RFC 2254).

For example, if this option is set to **(uid={0})** and the received username is **jdoe**, the search filter becomes **(uid=jdoe)** after string substitution.

If the resulting search filter is applied to the subtree selected by the user base, **ou=User,ou=ActiveMQ,ou=system**, it would match the entry, **uid=jdoe,ou=User,ou=ActiveMQ,ou=system**.

roleSearchMatching

This works in a similar manner to the **userSearchMatching** option, except that it supports two substitution strings.
The substitution string **{0}** substitutes the full DN of the matched user entry (that is, the result of the user search). For example, for the user, **jdoe**, the substituted string could be **uid=jdoe,ou=User,ou=ActiveMQ,ou=system**.

The substitution string **{1}** substitutes the received user name. For example, **jdoe**.

If this option is set to **(member=uid={1})** and the received user name is **jdoe**, the search filter becomes **(member=uid=jdoe)** after string substitution (assuming ApacheDS search filter syntax).

If the resulting search filter is applied to the subtree selected by the role base, **ou=Group,ou=ActiveMQ,ou=system**, it matches all role entries that have a **member** attribute equal to **uid=jdoe** (the value of a **member** attribute is a DN).

This option must always be set, even if role searching is disabled, because it has no default value. If OpenLDAP is used, the syntax of the search filter is **(member:=uid=jdoe)**.

Additional resources

- For a short introduction to the search filter syntax, see Oracle JNDI tutorial.

5.4.2. Configuring LDAP authorization

The **LegacyLDAPSecuritySettingPlugin** security settings plugin reads the security information previously handled in AMQ 6 by **LDAPAuthorizationMap** and **cachedLDAPAuthorizationMap** and converts this information to corresponding AMQ 7 security settings, where possible.

The security implementations for brokers in AMQ 6 and AMQ 7 do not match exactly. Therefore, the plugin performs some translation between the two versions to achieve near-equivalent functionality.

The following example shows how to configure the plugin.

Procedure

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

2. Within the **security-settings** element, add the **security-setting-plugin** element. For example:

```
<security-settings>
   <security-setting-plugin class-
name="org.apache.activemq.artemis.core.server.impl.LegacyLDAPSecuritySettingPlugin">
      <setting name="initialContextFactory" value="com.sun.jndi.ldap.LdapCtxFactory"/>
      <setting name="connectionURL"
value="ldap://localhost:1024"/>`ou=destinations,o=ActiveMQ,ou=system`
      <setting name="connectionUsername" value="uid=admin,ou=system"/>
      <setting name="connectionPassword" value="secret"/>
      <setting name="connectionProtocol" value="s"/>
      <setting name="authentication" value="simple"/>
   </security-setting-plugin>
</security-settings>
```

class-name

The implementation is **org.apache.activemq.artemis.core.server.impl.LegacyLDAPSecuritySettingPlugin**.

initialContextFactory

The initial context factory used to connect to LDAP. It must always be set to **com.sun.jndi.ldap.LdapCtxFactory** (that is, the default value).

connectionURL

Specifies the location of the directory server using an LDAP URL, *<ldap://Host:Port>*. You can optionally qualify this URL by adding a forward slash, /, followed by the distinguished name (DN) of a particular node in the directory tree. For example, **ldap://ldapserver:10389/ou=system**. The default value is **ldap://localhost:1024**.

connectionUsername

The DN of the user that opens the connection to the directory server. For example, **uid=admin,ou=system**. Directory servers generally require clients to present username/password credentials in order to open a connection.

connectionPassword

The password that matches the DN from **connectionUsername**. In the directory server, in the *Directory Information Tree* (DIT), the password is normally stored as a **userPassword** attribute in the corresponding directory entry.

connectionProtocol

Currently unused. In the future, this option might allow you to select the Secure Socket Layer (SSL) for the connection to the directory server. This option must be set explicitly because it has no default value.

authentication

Specifies the authentication method used when binding to the LDAP server. Valid values for this parameter are **simple** (username and password) or **none** (anonymous). The default value is **simple**.

> **NOTE**
>
> Simple Authentication and Security Layer (SASL) authentication is **not** supported.

Other settings not shown in the preceding configuration example are:

destinationBase

Specifies the DN of the node whose children provide the permissions for all destinations. In this case, the DN is a literal value (that is, no string substitution is performed on the property value). For example, a typical value of this property is **ou=destinations,o=ActiveMQ,ou=system** The default value is **ou=destinations,o=ActiveMQ,ou=system**.

filter

Specifies an LDAP search filter, which is used when looking up the permissions for any kind of destination. The search filter attempts to match one of the children or descendants of the queue or topic node. The default value is **(cn=*)**.

roleAttribute

Specifies an attribute of the node matched by **filter** whose value is the DN of a role. The default value is **uniqueMember**.

adminPermissionValue

Specifies a value that matches the **admin** permission. The default value is **admin**.

readPermissionValue

Specifies a value that matches the **read** permission. The default value is **read**.

writePermissionValue

Specifies a value that matches the **write** permission. The default value is **write**.

enableListener

Specifies whether to enable a listener that automatically receives updates made in the LDAP server and update the broker's authorization configuration in real time. The default value is **true**.

mapAdminToManage

Specifies whether to map the legacy (that is, AMQ 6) **admin** permission to the AMQ 7 **manage** permission. See details of the mapping semantics in the table below. The default value is **false**. The name of the queue or topic defined in LDAP serves as the "match" for the security setting, the permission value is mapped from the AMQ 6 type to the AMQ 7 type, and the role is mapped as-is. Because the name of the queue or topic defined in LDAP serves as the match for the security setting, the security setting may not be applied as expected to JMS destinations. This is because AMQ 7 always prefixes JMS destinations with "jms.queue." or "jms.topic.", as necessary.

AMQ 6 has three permission types – **read**, **write**, and **admin**. These permission types are described on the ActiveMQ website; Security.

AMQ 7 has the following permission types:

- **createAddress**

- **deleteAddress**

- **createDurableQueue**

- **deleteDurableQueue**

- **createNonDurableQueue**

- **deleteNonDurableQueue**

- **send**

- **consume**

- **manage**

- **browse**
  This table shows how the security settings plugin maps AMQ 6 permission types to AMQ 7 permission types:

Table 5.3. Mapping AMQ 6 permission types to AMQ 7

AMQ 6 permission type	AMQ 7 permission type
read	consume, browse
write	send
admin	createAddress, deleteAddress, createDurableQueue, deleteDurableQueue, createNonDurableQueue, deleteNonDurableQueue, manage (if **mapAdminToManage** is set to **true**)

As described below, there are some cases in which the plugin performs some translation between the AMQ 6 and AMQ 7 permission types to achieve equivalence:

- The mapping does not include the AMQ 7 **manage** permission type by default because there is no analogous permission type in AMQ 6. However, if **mapAdminToManage** is set to **true**, the plugin maps the AMQ 6 **admin** permission to the AMQ 7 **manage**

permission.

- ○ The **admin** permission type in AMQ 6 determines whether the broker automatically creates a destination if the destination does not exist and the user sends a message to it. AMQ 7 automatically allows automatic creation of a destination if the user has permission to send messages to the destination. Therefore, the plugin maps the legacy **admin** permission to the AMQ 7 permissions shown above, by default. The plugin also maps the AMQ 6 **admin** permission to the AMQ 7 **manage** permission if **mapAdminToManage** is set to **true**.

allowQueueAdminOnRead

Whether or not to map the legacy read permission to the createDurableQueue, createNonDurableQueue, and deleteDurableQueue permissions so that JMS clients can create durable and non-durable subscriptions without needing the admin permission. This was allowed in AMQ 6. The default value is false.

This table shows how the security settings plugin maps AMQ 6 permission types to AMQ 7 permission types when **allowQueueAdminOnRead** is **true**:

Table 5.4. Mapping AMQ 6 permission types to AMQ 7 when **allowQueueAdminOnRead** is **true**

AMQ 6 permission type	AMQ 7 permission type
read	consume, browse, createDurableQueue, createNonDurableQueue, deleteDurableQueue
write	send
admin	createAddress, deleteAddress, deleteNonDurableQueue, manage (if **mapAdminToManage** is set to **true**)

5.4.3. Encrypting the password in the login.config file

Because organizations frequently securely store data with LDAP, the **login.config** file can contain the configuration required for the broker to communicate with the organization's LDAP server. This configuration file usually includes a password to log in to the LDAP server, so this password needs to be encrypted.

Prerequisites

- Ensure that you have modified the **login.config** file to add the required properties, as described in Section 5.4.2, "Configuring LDAP authorization".

Procedure

The following procedure shows how to mask the value of the **connectionPassword** parameter found in the *<broker_instance_dir>*/**etc**/**login.config** file.

1. From a command prompt, use the **mask** utility to encrypt the password:

   $ *<broker_instance_dir>*/bin/artemis mask *<password>*

> result: 3a34fd21b82bf2a822fa49a8d8fa115d

2. Open the **<broker_instance_dir>/etc/login.config** file. Locate the **connectionPassword** parameter:

> connectionPassword = *<password>*

3. Replace the plain-text password with the encrypted value:

> connectionPassword = 3a34fd21b82bf2a822fa49a8d8fa115d

4. Wrap the encrypted value with the identifier **"ENC()"**:

> connectionPassword = "ENC(3a34fd21b82bf2a822fa49a8d8fa115d)"

The **login.config** file now contains a masked password. Because the password is wrapped with the **"ENC()"** identifier, AMQ Broker decrypts it before it is used.

Additional resources

- For more information about the configuration files included with AMQ Broker, see AMQ Broker configuration files and locations.

5.4.4. Mapping external roles

You can map roles from external authentication providers such as LDAP to roles used internally by the broker.

To map external roles, create role-mapping entries in a **security-settings** element in the **broker.xml** configuration file. For example:

```
<security-settings>
...
   <role-mapping from="cn=admins,ou=Group,ou=ActiveMQ,ou=system" to="my-admin-role"/>
   <role-mapping from="cn=users,ou=Group,ou=ActiveMQ,ou=system" to="my-user-role"/>
</security-settings>
```

> **NOTE**
>
> - Role mapping is additive. That means the user will keep the original role(s) as well as the newly assigned role(s).
>
> - Role mapping only affects the roles authorizing queue access and does not provide a method to enable web console access.

5.5. USING KERBEROS FOR AUTHENTICATION AND AUTHORIZATION

When sending and receiving messages with the AMQP protocol, clients can send Kerberos security credentials that AMQ Broker authenticates by using the GSSAPI mechanism from the *Simple Authentication and Security Layer* (SASL) framework. Kerberos credentials can also be used for authorization by mapping an authenticated user to an assigned role configured in an LDAP directory or text-based properties file.

You can use SASL in tandem with *Transport Layer Security* (TLS) to secure your messaging applications. SASL provides user authentication, and TLS provides data integrity.

> **IMPORTANT**
>
> - You must deploy and configure a Kerberos infrastructure before AMQ Broker can authenticate and authorize Kerberos credentials. See your operating system documentation for more information about deploying Kerberos.
>
>   - For RHEL 7, see Using Kerberos.
>
>   - For Windows, see Kerberos Authentication Overview.
>
> - Users of an Oracle or IBM JDK should install the Java Cryptography Extension (JCE). See the documentation from the Oracle version of the JCE or the IBM version of the JCE for more information.

The following procedures show how to configure Kerberos for authentication and authorization.

5.5.1. Configuring network connections to use Kerberos

AMQ Broker integrates with Kerberos security credentials by using the GSSAPI mechanism from the *Simple Authentication and Security Layer* (SASL) framework. To use Kerberos in AMQ Broker, each acceptor authenticating or authorizing clients that use a Kerberos credential must be configured to used the GSSAPI mechanism.

The following procedure shows how to configure an acceptor to use Kerberos.

Prerequisites

- You must deploy and configure a Kerberos infrastructure before AMQ Broker can authenticate and authorize Kerberos credentials.

Procedure

1. Stop the broker.

    a. On Linux:

        <broker_instance_dir>/bin/artemis stop

    b. On Windows:

        <broker_instance_dir>\bin\artemis-service.exe stop

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

3. Add the name-value pair **saslMechanisms=GSSAPI** to the query string of the URL for the **acceptor**.

    ```
    <acceptor name="amqp">
      tcp://0.0.0.0:5672?protocols=AMQP;saslMechanisms=GSSAPI
    </acceptor>
    ```

The preceding configuration means that the acceptor uses the GSSAPI mechanism when authenticating Kerberos credentials.

4. (Optional) The **PLAIN** and **ANONYMOUS** SASL mechanisms are also supported. To specify multiple mechanisms, use a comma-separated list. For example:

```
<acceptor name="amqp">
  tcp://0.0.0.0:5672?protocols=AMQP;saslMechanisms=GSSAPI,PLAIN
</acceptor>
```

The result is an acceptor that uses both the **GSSAPI** and **PLAIN** SASL mechanisms.

5. Start the broker.

   a. On Linux:

   ```
   <broker_instance_dir>/bin/artemis run
   ```

   b. On Windows:

   ```
   <broker_instance_dir>\bin\artemis-service.exe start
   ```

Additional resources

- For more information about acceptors, see Section 2.1, "About acceptors".

5.5.2. Authenticating clients with Kerberos credentials

AMQ Broker supports Kerberos authentication of AMQP connections that use the GSSAPI mechanism from the *Simple Authentication and Security Layer* (SASL) framework.

A broker acquires its Kerberos acceptor credentials by using the *Java Authentication and Authorization Service* (JAAS). The JAAS library included with your Java installation is packaged with a login module, **Krb5LoginModule**, that authenticates Kerberos credentials. See the documentation from your Java vendor for more information about their **Krb5LoginModule**. For example, Oracle provides information about their **Krb5LoginModule** login module as part of their Java 8 documentation.

Prerequisites

- You must enable the GSSAPI mechanism of an acceptor before it can authenticate AMQP connections using Kerberos security credentials. For more information, see Section 5.5.1, "Configuring network connections to use Kerberos".

Procedure

1. Stop the broker.

   a. On Linux:

   ```
   <broker_instance_dir>/bin/artemis stop
   ```

   b. On Windows:

   ```
   <broker_instance_dir>\bin\artemis-service.exe stop
   ```

2. Open the ***<broker_instance_dir>*/etc/login.config** configuration file.

3. Add a configuration scope named **amqp-sasl-gssapi**. The following example shows configuration for the **Krb5LoginModule** found in Oracle and OpenJDK versions of the JDK.

```
amqp-sasl-gssapi {
    com.sun.security.auth.module.Krb5LoginModule required
    isInitiator=false
    storeKey=true
    useKeyTab=true
    principal="amqp/my_broker_host@example.com"
    debug=true;
};
```

amqp-sasl-gssapi

By default, the GSSAPI mechanism implementation on the broker uses a JAAS configuration scope named **amqp-sasl-gssapi** to obtain its Kerberos acceptor credentials.

Krb5LoginModule

This version of the **Krb5LoginModule** is provided by the Oracle and OpenJDK versions of the JDK. Verify the fully qualified class name of the **Krb5LoginModule** and its available options by referring to the documentation from your Java vendor.

useKeyTab

The **Krb5LoginModule** is configured to use a Kerberos keytab when authenticating a principal. Keytabs are generated using tooling from your Kerberos environment. See the documentation from your vendor for details about generating Kerberos keytabs.

principal

The Principal is set to **amqp/my_broker_host@example.com**. This value must correspond to the service principal created in your Kerberos environment. See the documentation from your vendor for details about creating service principals.

4. Start the broker.

   a. On Linux:

   ```
   <broker_instance_dir>/bin/artemis run
   ```

   b. On Windows:

   ```
   <broker_instance_dir>\bin\artemis-service.exe start
   ```

5.5.2.1. Using an alternative configuration scope

You can specify an alternative configuration scope by adding the parameter **saslLoginConfigScope** to the URL of an AMQP acceptor. In the following configuration example, the parameter **saslLoginConfigScope** is given the value **alternative-sasl-gssapi**. The result is an acceptor that uses the alternative scope named **alternative-sasl-gssapi**, declared in the ***<broker_instance_dir>*/etc/login.config** configuration file.

```
<acceptor name="amqp">
tcp://0.0.0.0:5672?
protocols=AMQP;saslMechanisms=GSSAPI,PLAIN;saslLoginConfigScope=alternative-sasl-gssapi`
</acceptor>
```

5.5.3. Authorizing clients with Kerberos credentials

AMQ Broker includes an implementation of the JAAS **Krb5LoginModule** login module for use by other security modules when mapping roles. The module adds a Kerberos–authenticated Peer Principal to the Subject's principal set as an AMQ Broker UserPrincipal. The credentials can then be passed to a **PropertiesLoginModule** or **LDAPLoginModule** module, which maps the Kerberos–authenticated Peer Principal to an AMQ Broker role.

> **NOTE**
>
> The Kerberos Peer Principal does not exist as a broker user, only as a role member.

Prerequisites

- You must enable the GSSAPI mechanism of an acceptor before it can authorize AMQP connections using Kerberos security credentials.

Procedure

1. Stop the broker.

   a. On Linux:

      <broker_instance_dir>/bin/artemis stop

   b. On Windows:

      <broker_instance_dir>\bin\artemis-service.exe stop

2. Open the ***<broker_instance_dir>*/etc/login.config** configuration file.

3. Add configuration for the AMQ Broker **Krb5LoginModule** and the **LDAPLoginModule**. Verify the configuration options by referring to the documentation from your LDAP provider.
   An example configuration is shown below.

   ```
   org.apache.activemq.artemis.spi.core.security.jaas.Krb5LoginModule required
       ;
   org.apache.activemq.artemis.spi.core.security.jaas.LDAPLoginModule optional
       initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
       connectionURL="ldap://localhost:1024"
       authentication=GSSAPI
       saslLoginConfigScope=broker-sasl-gssapi
       connectionProtocol=s
       userBase="ou=users,dc=example,dc=com"
       userSearchMatching="(krb5PrincipalName={0})"
       userSearchSubtree=true
       authenticateUser=false
       roleBase="ou=system"
       roleName=cn
       roleSearchMatching="(member={0})"
       roleSearchSubtree=false
       ;
   ```

> **NOTE**
>
> The version of the **Krb5LoginModule** shown in the preceding example is distributed with AMQ Broker and transforms the Kerberos identity into a broker identity that can be used by other AMQ modules for role mapping.

4. Start the broker.

    a. On Linux:

        <broker_instance_dir>/bin/artemis run

    b. On Windows:

        <broker_instance_dir>\bin\artemis-service.exe start

Additional resources

- See Section 5.5.1, "Configuring network connections to use Kerberos" for more information about enabling the GSSAPI mechanism in AMQ Broker.

- See Section 5.2.2.1, "Configuring basic user and password authentication" for more information about **PropertiesLoginModule**.

- See Section 5.4.1, "Configuring LDAP to authenticate clients" for more information about **LDAPLoginModule**.

5.6. SPECIFYING A SECURITY MANAGER

The broker uses a component called the *security manager* to handle authentication and authorization.

AMQ Broker includes two security managers:

- The **ActiveMQJAASSecurityManager** security manager. This security manager provides integration with JAAS and Red Hat JBoss Enterprise Application Platform (JBoss EAP) security. This is the **default** security manager used by AMQ Broker.

- The **ActiveMQBasicSecurityManager** security manager. This basic security manager doesn't support JAAS. Instead, it supports authentication and authorization through user name and password credentials. This security manager supports adding, removing, and updating users using the management API. All user and role data is stored in the broker bindings journal. This means that any changes made to a live broker are also available to its backup broker.

As an alternative to the included security managers, a system administrator might want more control over the implementation of broker security. In this case, it is also possible to specify a *custom* security manager in the broker configuration. A custom security manager is a user-defined class that implements the **org.apache.activemq.artemis.spi.core.security.ActiveMQSecurityManager5** interface.

The examples in the following sub-sections show how to configure the broker to use:

- The basic security manager instead of the default JAAS security manager

- A custom security manager

5.6.1. Using the basic security manager

In addition to the default **ActiveMQJAASSecurityManager** security manager, AMQ Broker also includes the **ActiveMQBasicSecurityManager** security manager.

When you use the basic security manager, all user and role data is stored in the bindings journal (or the bindings *table*, if you are using JDBC persistence). Therefore, if you have configured a live-backup broker group, any user management that you peform on the live broker is automatically reflected on the backup broker upon failover. This avoids the need to separately administer an LDAP server, which is the alternative way to achieve this behavior.

Before you configure and use the basic security manager, be aware of the following:

- The basic security manager is not pluggable like the default JAAS security manager.

- The basic security manager does not support JAAS. Instead, it supports only authentication and authorization through user name and password credentials.

- AMQ Management Console requires JAAS. Therefore, if you use the basic security manager and want to use the console, you also need to configure the **login.config** configuration file for user and password authentication. For more information about configuring user and password authentication, see Section 5.2.2.1, "Configuring basic user and password authentication".

- In AMQ Broker, user management is provided by the broker management API. This management includes the ability to add, list, update, and remove users and roles. You can perform these functions using JMX, management messages, HTTP (using Jolokia or AMQ Management Console), and the AMQ Broker command-line interface. Because the broker directly store this data, the broker must be running in order to manage users. There is no way to manually modify the bindings data.

- Any management access through HTTP (for example, using Jolokia or AMQ Management Console) is handled by the console JAAS login module. MBean access through JConsole or other remote JMX tools is handled by the basic security manager. Management messages are handled by the basic security manager.

5.6.1.1. Configuring the basic security manager

The following procedure shows how to configure the broker to use the basic security manager.

Procedure

1. Open the *<broker-instance-dir>*/etc/boostrap.xml configuration file.

2. In the **security-manager** element, for the **class-name** attribute, specify the full **ActiveMQBasicSecurityManager** class name.

   ```
   <broker xmlns="http://activemq.org/schema">
       ...
       <security-manager class-
   name="org.apache.activemq.artemis.spi.core.security.ActiveMQBasicSecurityManager">
       </security-manager>
       ...
   </broker>
   ```

3. Because you cannot manually modify the bindings data that holds user and role data, and because the broker must be running to manage users, it is advisable to secure the broker upon

first boot. To achieve this, define a *bootstrap user* whose credentials can then be used to add other users.

In the **security-manager** element, add the **bootstrapUser**, **bootstrapPassword**, and **bootstrapRole** properties and specify values. For example:

```
<broker xmlns="http://activemq.org/schema">
  ...
  <security-manager class-
name="org.apache.activemq.artemis.spi.core.security.ActiveMQBasicSecurityManager">
      <property key="bootstrapUser" value="myUser"/>
      <property key="bootstrapPassword" value="myPass"/>
      <property key="bootstrapRole" value="myRole"/>
  </security-manager>
  ...
</broker>
```

bootstrapUser

Name of the bootstrap user.

bootstrapPassword

Passsword of the boostrap user. You can also specify an encrypted password.

bootstrapRole

Role of the boostrap user.

NOTE

If you define the preceding properties for the bootstrap user in your configuration, those credentials are set each time that you start the broker, regardless of any changes you make while the broker is running.

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

5. In the **broker.xml** configuration file, locate the **address-setting** element that is defined by default for the **activemq.management#** address match. These default address settings are shown below.

```
<address-setting match="activemq.management#">
    <dead-letter-address>DLQ</dead-letter-address>
    <expiry-address>ExpiryQueue</expiry-address>
    <redelivery-delay>0</redelivery-delay>
    <!--...-->
    <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>
```

6. Within the address settings for the **activemq.management#** address match, for the bootstrap role name that you specified earlier in this procedure, add the following required permissions:

- **createNonDurableQueue**

- **createAddress**

- **consume**

- **manage**

- **send**

For example:

```
<address-setting match="activemq.management#">
   ...
   <permission type="createNonDurableQueue" roles="myRole"/>
   <permission type="createAddress" roles="myRole"/>
   <permission type="consume" roles="myRole"/>
   <permission type="manage" roles="myRole"/>
   <permission type="send" roles="myRole"/>
</address-setting>
```

Additional resources

- For more information about the **ActiveMQBasicSecurityManager** class, see Class ActiveMQBasicSecurityManager in the ActiveMQ Artemis Core API documentation.

- To learn how to encrypt passwords in configuration files, see Section 5.9, "Encrypting passwords in configuration files".

5.6.2. Specifying a custom security manager

The following procedure shows how to specify a custom security manager in your broker configuration.

Procedure

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

2. In the **security-manager** element, for the **class-name** attribute, specify the class that is a user-defined implementation of the **org.apache.activemq.artemis.spi.core.security.ActiveMQSecurityManager5** interface. For example:

```
<broker xmlns="http://activemq.org/schema">
  ...
  <security-manager class-name="com.myclass.MySecurityManager">
    <property key="myKey1" value="myValue1"/>
    <property key="myKey2" value="myValue2"/>
  </security-manager>
  ...
</broker>
```

Additional resources

- For more information about the **ActiveMQSecurityManager5** interface, see Interface ActiveMQSecurityManager5 in the ActiveMQ Artemis Core API documentation.

5.6.3. Running the custom security manager example program

AMQ Broker has an example program that demonstrates how to implement a custom security manager. In the example, the custom security manager logs details for authentication and authorization and then passes the details to an instance of **ActiveMQJAASSecurityManager** (that is, the default security manager).

The following procedure shows how to run the custom security manager example program.

Prerequisites

- Your machine is set up to run AMQ Broker example programs. For more information, see Running the AMQ Broker examples.

You downloaded the custom security manager example.

Procedure

1. Navigate to the directory that contains the custom security manager example. The following example assumed that you downloaded the example to a directory called **amq-broker-examples**.

   ```
   $ cd amq-broker-examples/examples/features/standard/security-manager
   ```

2. Run the example.

   ```
   $ mvn verify
   ```

> **NOTE**
>
> If you prefer to manually create and start a broker instance when running the example program, replace the command in the preceding step with **mvn -PnoServer verify**.

Additional resources

- For more information about the **ActiveMQJAASSecurityManager** class, see Class ActiveMQJAASSecurityManager in the ActiveMQ Artemis Core API documentation.

5.7. DISABLING SECURITY

Security is **enabled** by default. The following procedure shows how to disable broker security.

Procedure

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

2. In the **core** element, set the value of **security-enabled** to **false**.

   ```
   <security-enabled>false</security-enabled>
   ```

3. If necessary, specify a new value, in milliseconds, for **security-invalidation-interval**. The value of this property specifies when the broker periodically invalidates secure logins. The default value is **10000**.

5.8. TRACKING MESSAGES FROM VALIDATED USERS

To enable tracking and logging the origins of messages (for example, for security-auditing purposes), you can use the **_AMQ_VALIDATED_USER** message key.

In the **broker.xml** configuration file, if the **populate-validated-user** option is set to **true**, then the broker adds the name of the validated user to the message using the **_AMQ_VALIDATED_USER** key. For JMS and STOMP clients, this message key maps to the **JMSXUserID** key.

> **NOTE**
>
> The broker cannot add the validated user name to a message produced by an AMQP JMS client. Modifying the properties of an AMQP message after it has been sent by a client is a violation of the AMQP protocol.

For a user authenticated based on his/her SSL certificate, the validated user name populated by the broker is the name to which the certificate's Distinguished Name (DN) maps.

In the **broker.xml** configuration file, if **security-enabled** is **false** and **populate-validated-user** is **true**, then the broker populates whatever user name, if any, that the client provides. The **populate-validated-user** option is **false** by default.

You can configure the broker to reject a message that doesn't have a user name (that is, the **JMSXUserID** key) already populated by the client when it sends the message. You might find this option useful for AMQP clients, because the broker cannot populate the validated user name itself for messages sent by these clients.

To configure the broker to reject messages without **JMSXUserID** set by the client, add the following configuration to the **broker.xml** configuration file:

```
<reject-empty-validated-user>true</reject-empty-validated-user>
```

By default, **reject-empty-validated-user** is set to **false**.

5.9. ENCRYPTING PASSWORDS IN CONFIGURATION FILES

By default, AMQ Broker stores all passwords in configuration files as plain text. Be sure to secure all configuration files with the correct permissions to prevent unauthorized access. You can also encrypt, or *mask*, the plain text passwords to prevent unwanted viewers from reading them.

5.9.1. About encrypted passwords

An encrypted, or *masked*, password is the encrypted version of a plain text password. The encrypted version is generated by the **mask** command-line utility provided by AMQ Broker. For more information about the **mask** utility, see the command-line help documentation:

```
$ <broker_instance_dir>/bin/artemis help mask
```

To mask a password, replace its plain-text value with the encrypted one. The masked password must be wrapped by the identifier **ENC()** so that it is decrypted when the actual value is needed.

In the following example, the configuration file ***<broker_instance_dir>*/etc/bootstrap.xml** contains masked passwords for the **keyStorePassword** and **trustStorePassword** parameters.

```
<web bind="https://localhost:8443" path="web"
    keyStorePassword="ENC(-342e71445830a32f95220e791dd51e82)"
    trustStorePassword="ENC(32f94e9a68c45d89d962ee7dc68cb9d1)">
    <app url="activemq-branding" war="activemq-branding.war"/>
</web>
```

You can use masked passwords with the following configuration files.

- broker.xml

- bootstrap.xml

- management.xml

- artemis-users.properties

- login.config (for use with the **LDAPLoginModule**)

Configuration files are found at ***<broker_instance_dir>*/etc**.

> **NOTE**
>
> **artemis-users.properties** supports only masked passwords that have been hashed. When a user is created upon broker creation, **artemis-users.properties** contains hashed passwords by default. The default **PropertiesLoginModule** will not decode the passwords in **artemis-users.properties** file but will instead hash the input and compare the two hashed values for password verification. Changing the hashed password to a masked password does not allow access to the AMQ Broker management console.
>
> **broker.xml**, **bootstrap.xml**, **management.xml**, and **login.config** support passwords that are masked but not hashed.

5.9.2. Encrypting a password in a configuration file

The following example shows how to mask the value of **cluster-password** in the **broker.xml** configuration file.

Procedure

1. From a command prompt, use the **mask** utility to encrypt a password:

   ```
   $ <broker_instance_dir>/bin/artemis mask <password>
   ```

   ```
   result: 3a34fd21b82bf2a822fa49a8d8fa115d
   ```

2. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file containing the plain-text password that you want to mask:

```
<cluster-password>
 <password>
</cluster-password>
```

3. Replace the plain-text password with the encrypted value:

```
<cluster-password>
  3a34fd21b82bf2a822fa49a8d8fa115d
</cluster-password>
```

4. Wrap the encrypted value with the identifier **ENC()**:

```
<cluster-password>
  ENC(3a34fd21b82bf2a822fa49a8d8fa115d)
</cluster-password>
```

The configuration file now contains an encrypted password. Because the password is wrapped with the **ENC()** identifier, AMQ Broker decrypts it before it is used.

Additional resources

- For more information about the configuration files included with AMQ Broker, see Section 1.1, "AMQ Broker configuration files and locations".

5.9.3. Setting a codec key to encrypt and decrypt passwords

A codec is required to encrypt and decrypt passwords. If a custom codec is not configured, the **mask** utility uses a default codec to encrypt passwords and AMQ Broker uses the same default codec to decrypt a password. The codec is configured with a default key, which it provides to the underlying encryption algorithm to encrypt and decrypt passwords. Using the default key exposes a risk that the key might be used by a malicious actor to decrypt your passwords.

When you use the **mask** utility to encrypt passwords, you can specify you own key string to avoid using the default codec key. You must then set the same key string in the **ARTEMIS_DEFAULT_SENSITIVE_STRING_CODEC_KEY** environment variable, so the broker can decrypt the passwords. Setting the key in an environment variable makes it more secure because it is not persisted in a configuration file. In addition, you can set the key immediately before you start the broker and unset it immediately after the broker starts.

Procedure

1. Use the **mask** utility to encrypt each password in a configuration file. For the   **key** parameter, specify a string of characters with which to encrypt the password. Use the same key string to encrypt each password.

```
$ <broker_instance_dir>/bin/artemis mask --key <key> <password>
```

> **WARNING**
>
> Ensure that you keep a record of the key string that you specify when you run the **mask** utility to encrypt passwords. You must configure the same key value in an environment variable to allow the broker to decrypt passwords.

For more information about encrypting passwords in configuration files, see Section 5.9.2, "Encrypting a password in a configuration file".

2. From a command prompt, set the **ARTEMIS_DEFAULT_SENSITIVE_STRING_CODEC_KEY** environment variable to the key string that you specified when you encrypted each password.

   ```
   $ export ARTEMIS_DEFAULT_SENSITIVE_STRING_CODEC_KEY= <key>
   ```

3. Start the broker.

   ```
   $ ./artemis run
   ```

4. Unset the **ARTEMIS_DEFAULT_SENSITIVE_STRING_CODEC_KEY** environment variable.

   ```
   $ unset ARTEMIS_DEFAULT_SENSITIVE_STRING_CODEC_KEY
   ```

> **NOTE**
>
> If you unset the **ARTEMIS_DEFAULT_SENSITIVE_STRING_CODEC_KEY** environment variable after you start the broker, you must set it again to the same key string before you start the broker each subsequent time.

5.10. CONFIGURING AUTHENTICATION AND AUTHORIZATION CACHING

By default, AMQ Broker stores information about successful authentication and authorization responses in separate caches. You can change the default number of entries allowed in each cache and the duration for which entries are cached.

1. Open the **<broker-instance-dir>/etc/broker.xml** configuration file.

2. To change the default maximum number of entries, **1000**, allowed in each cache, set the **authentication-cache-size** and the **authorization-cache-size** parameters. For example:

   ```
   <configuration>
    ...
    <core>
     ...
     <authentication-cache-size>2000</authentication-cache-size>
     <authorization-cache-size>1500</authorization-cache-size>
     ...
   ```

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

> **NOTE**
>
> If a cache reaches the limit set, the least recently used entry is removed from the cache.

3. To change the default duration, **10000** milliseconds, for which entries are cached, set the **security-invalidation-interval** parameter. For example:

```
<configuration>
  ...
  <core>
    ...
    <security-invalidation-interval>20000</security-invalidation-interval>
    ...
  </core>
  ...
</configuration>
```

> **NOTE**
>
> If you set the **security-invalidation-interval** parameter to **0**, authentication and authorization caching is disabled.

CHAPTER 6. PERSISTING MESSAGE DATA

AMQ Broker has two options for persisting (that is, *storing*) message data:

Persisting messages in journals

> This is the default option. Journal-based persistence is a high-performance option that writes messages to journals on the file system.

Persisting messages in a database

> This option uses a *Java Database Connectivity* (JDBC) connection to persist messages to a database of your choice.

Alternatively, you can also configure the broker **not** to persist any message data. For more information, see Section 6.3, "Disabling persistence".

The broker uses a different solution for persisting large messages outside the message journal. See Chapter 8, *Handling large messages* for more information.

The broker can also be configured to page messages to disk in low-memory situations. See Section 7.1, "Configuring message paging" for more information.

> **NOTE**
>
> For current information regarding which databases and network file systems are supported by AMQ Broker see Red Hat AMQ 7 Supported Configurations on the Red Hat Customer Portal.

6.1. PERSISTING MESSAGE DATA IN JOURNALS

A broker journal is a set of **append-only** files on disk. Each file is pre-created to a fixed size and initially filled with padding. As messaging operations are performed on the broker, records are appended to end of the journal. Appending records allows the broker to minimize disk head movement and random access operations, which are typically the slowest operation on a disk. When one journal file is full, the broker creates a new one.

The journal file size is configurable, minimizing the number of disk cylinders used by each file. Modern disk topologies are complex, however, and the broker cannot control which cylinder(s) the file is mapped to. Therefore, journal file sizing is difficult to control precisely.

Other persistence-related features that the broker uses are:

- A *garbage collection* algorithm that determines whether a particular journal file is still in use. If the journal file is no longer in use, the broker can reclaim the file for reuse.

- A compaction algorithm that removes dead space from the journal and compresses the data. This results in the journal using fewer files on disk.

- Support for local transactions.

- Support for Extended Architecture (XA) transactions when using JMS clients.

Most of the journal is written in Java. However, interaction with the actual file system is abstracted, so that you can use different, pluggable implementations. AMQ Broker includes the following implementations:

NIO

NIO (New I/O) uses standard Java NIO to interface with the file system. This provides extremely good performance and runs on any platform with a Java 6 or later runtime. For more information about Java NIO, see Java NIO.

AIO

AIO (Aynshcronous I/O) uses a thin native wrapper to talk to the Linux Asynchronous I/O Library (**libaio**). With AIO, the broker is called back after the data has made it to disk, avoiding explicit syncs altogether. By default, the broker tries to use an AIO journal, and falls back to using NIO if AIO is not available.

AIO typically provides even better performance than Java NIO. To learn how to install **libaio**, see Section 6.1.1, "Installing the Linux Asynchronous I/O Library" .

The procedures in the sub-sections that follow show how to configure the broker for journal-based persistence.

6.1.1. Installing the Linux Asynchronous I/O Library

Red Hat recommends using the AIO journal (instead of NIO) for better persistence performance.

> **NOTE**
>
> It is **not** possible to use the AIO journal with other operating systems or earlier versions of the Linux kernel.

To use the AIO journal, you must install the Linux Asynchronous I/O Library (**libaio**). To install **libaio**, use the **yum** command, as shown below:

```
yum install libaio
```

6.1.2. Configuring journal-based persistence

The following procedure describes how to review the default configuration that the broker uses for journal-based persistence. You can use this description to adjust your configuration as needed.

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.
   By default, the broker is configured to use journal-based persistence, as shown below.

   ```xml
   <configuration>
    <core>
     ...
     <persistence-enabled>true</persistence-enabled>
     <journal-type>ASYNCIO</journal-type>
     <bindings-directory>./data/bindings</bindings-directory>
     <journal-directory>./data/journal</journal-directory>
     <journal-datasync>true</journal-datasync>
     <journal-min-files>2</journal-min-files>
     <journal-pool-files>-1</journal-pool-files>
     <journal-device-block-size>4096</journal-device-block-size>
     <journal-file-size>10M</journal-file-size>
     <journal-buffer-timeout>12000</journal-buffer-timeout>
     <journal-max-io>4096</journal-max-io>
   ```

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

persistence-enabled

If the value of this parameter is set to **true**, the broker uses the file-based journal for message persistence.

journal-type

Type of journal to use. If set to **ASYNCIO**, the broker first attempts to use AIO. If AIO is not found, the broker uses NIO.

bindings-directory

File system location of the bindings journal. The default value is relative to the *<broker_instance_dir>* directory.

journal-directory

File system location of the message journal. The default value is relative to the *<broker_instance_dir>* directory.

journal-datasync

If the value of this parameter is set to **true**, the broker uses the **fdatasync** function to confirm disk writes.

journal-min-files

Number of journal files to initially create when the broker starts.

journal-pool-files

Number of files to keep after reclaiming unused files. The default value of **-1** means that no files are deleted during cleanup.

journal-device-block-size

Maximum size, in bytes, of the data blocks used by the journal on your storage device. The default value is 4096 bytes.

journal-file-size

Maximum size, in bytes, of each journal file in the specified journal directory. When this limit is reached, the broker starts a new file. This parameter also supports byte notation (for example, K, M, G), or the binary equivalents (Ki, Mi, Gi). If this parameter is not explicitly specified in your configuration, the default value is 10485760 bytes (10MiB).

journal-buffer-timeout

Specifies how often, in nanoseconds, the broker flushes the journal buffer. AIO typically uses a higher flush rate than NIO, so the broker maintains different default values for both NIO and AIO. If this parameter not explicitly specified in your configuration, the default value for NIO is 3333333 nanoseconds (that is, 300 times per second). The default value for AIO is 50000 nanoseconds (that is, 2000 times per second).

journal-max-io

Maximum number of write requests that can be in the IO queue at any one time. If the queue becomes full, the broker blocks further writes until space is available.
If you are using NIO, this value should always be **1**. If you are using AIO andthis parameter not explicitly specified in your configuration, the default value is **500**.

2. Based on the preceding descriptions, adjust your persistence configuration as needed for your storage device.

Additional resources

- To learn about **all** of the parameters that are available for configuring journal-based persistence, see Appendix E, *Messaging Journal Configuration Elements*.

6.1.3. About the bindings journal

The bindings journal is used to store bindings-related data, such as the set of queues deployed on the broker and their attributes. It also stores data such as ID sequence counters.

The bindings journal always uses NIO because it is typically low throughput when compared to the message journal. Files on this journal are prefixed with **activemq-bindings**. Each file also has an extension of **.bindings** and a default size of 1048576 bytes.

To configure the bindings journal, include the following parameters in the **core** element of the *<broker_instance_dir>*/etc/broker.xml configuration file.

bindings-directory

Directory for the bindings journal. The default value is *<broker_instance_dir>*/data/bindings.

create-bindings-dir

If the value of this parameter is set to **true**, the broker automatically creates the bindings directory in the location specified in **bindings-directory**, if it does not already exist. The default value is **true**.

6.1.4. About the JMS journal

The JMS journal stores all JMS-related data, including JMS queues, topics, and connection factories, as well as any JNDI bindings for these resources. Any JMS resources created via the management API are persisted to this journal, but any resources configured via configuration files are not. The broker creates the JMS journal **only** if JMS is being used.

Files in the JMS journal are prefixed with **activemq-jms**. Each file also has an extension of **.jms** and a default size of 1048576 bytes.

The JMS journal shares its configuration with the bindings journal.

Additional resources

- For more information about the bindings journal, see Section 6.1.3, "About the bindings journal".

6.1.5. Configuring journal retention

You can configure AMQ Broker to keep a copy of each journal file created. After you configure journal retention, you can replay messages in the journal file copies to send the messages to the broker.

6.1.5.1. Configuring journal retention

You can configure AMQ Broker to keep a copy of journal files for a specific time period or until a storage limit is reached, or both.

Procedure

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

2. Within the **core** element, add the **journal-retention-directory** attribute. Specify a **period** or

storage-limit, or both, to control the retention of journal files. In addition, specify the file system location for the journal file copies. The following example configures AMQ Broker to keep journal file copies in the **data/retention** directory for **7** days or until the files use **10GB** of storage.

```
<configuration>
  <core>
    ...
    <journal-retention-directory period="7" unit="DAYS" storage-limit="10G">data/retention</journal-retention-directory>
    ...
  </core>
</configuration>
```

period

Time period to keep the journal file copies. When the time period expires, AMQ Broker removes any files that are older than the specified time period.

unit

The unit of measure to apply to the retention period. The default value is **DAYS**. The other valid values are **HOURS**, **MINUTES** and **SECONDS**.

directory

File system location of the journal file copies. The specified directory is relative to the *<broker_instance_dir>* directory.

storage-limit

The maximum storage that can be used by all journal file copies. If the storage limit is reached, the broker removes the oldest journal file to provide space for a new journal file copy. Setting a storage limit is an effective way of ensuring that the journal file retention does not cause the broker to run out of disk space and shut down.

6.1.5.2. Replaying messages in journal file copies for addresses that exist on the broker

If the address of messages that you want to replay from journal file copies exists on AMQ Broker, use the following procedure to replay the messages. You can replay messages to the original address on the broker or to a different address.

Procedure

1. Log in to AMQ Management Console. For more information see Accessing AMQ Management Console.

2. In the main menu, click **Artemis**.

3. In the folder tree, click **addresses** to display a list of addresses.

4. Click the **Addresses** tab.

5. In the **Action** column for the address from which you want to replay messages, click **operations**.

6. Select a replay operation.

   - If you want the replay operation to search for messages to replay in all journal file copies, click the **replay(String,String)** operation.

- If you want the replay operation to search for messages to replay only in journal file copies created within a specific time period, select the **replay(String,String, String,String)** operation. In the **startScanDate** and **endScanDate** fields, specify the time period.

7. Specify the replay options.

   a. In the **target** field, specify the address on the broker to send replayed messages. If you leave this field blank, messages are replayed to the original address on the broker.

   b. (Optional) In the **filter** field, specify a string to replay messages only that match the filter string. For example, if messages have a storeID property, you can use a filter of **storeID="1000"** to replay all messages that have a store ID value of 1000. If you don't specify a filter, all messages in the scanned journal file copies are replayed to AMQ Broker.

8. Click **Execute**.

Additional resources

- For more information about using AMQ Management Console, see Using AMQ Management Console.

6.1.5.3. Replaying messages in journal file copies for addresses removed from the broker

If the address of messages that you want to replay from the journal file copies was removed from AMQ Broker, use the following procedure to replay messages to a different address on the broker.

Procedure

1. Log in to AMQ Management Console. For more information see Accessing AMQ Management Console.

2. In the main menu, click **Artemis**.

3. In the folder tree, click the top-level server.

4. Click the **Operations** tab.

5. Select a replay operation.

   - If you want the replay operation to search for messages to replay in all journal file copies, click the **replay(String,String,String)** operation.

   - If you want the replay operation to search for messages to replay only in journal file copies created within a specific time period, select the **replay(String,String, String,String,String)** operation. In the **startScanDate** and **endScanDate** fields, specify the time period.

6. Specify the replay options.

   a. In the **address** field, specify the address of messages to replay.

   b. In the **target** field, specify the address on the broker to send replayed messages.

   c. (Optional) In the **filter** field, specify a string to replay messages only that match the filter string. For example, if messages have a storeID property, you can use a filter of **storeID="1000"** to replay all messages that have a store ID value of 1000. If you don't specify a filter, all messages in the scanned journal file copies are replayed to AMQ Broker.

7. Click **Execute**.

Additional resources

- For more information about using AMQ Management Console, see Using AMQ Management Console.

6.1.6. Compacting journal files

AMQ Broker includes a compaction algorithm that removes dead space from the journal and compresses the data so that it takes up less disk space.

The following sub-sections show how to:

- Configure the broker to automatically compact journal files when certain criteria are met

- Run the compaction process manually from the command-line interface

6.1.6.1. Configuring journal file compaction

The broker uses the following criteria to determine when to start compaction:

- The number of files created for the journal.

- The percentage of live data in the journal files.

After the configured values for both of these criteria are reached, the compaction process parses the journal and removes all dead records. Consequently, the journal comprises fewer files.

The following procedure shows how to configure the broker for journal file compaction.

Procedure

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

2. Within the **core** element, add the **journal-compact-min-files** and **journal-compact-percentage** parameters and specify values. For example:

```
<configuration>
 <core>
  ...
   <journal-compact-min-files>15</journal-compact-min-files>
   <journal-compact-percentage>25</journal-compact-percentage>
  ...
 </core>
</configuration>
```

journal-compact-min-files

The minimum number of journal files that the broker has to create before compaction begins. The default value is **10**. Setting the value to **0** disables compaction. You should take care when disabling compaction, because the size of the journal can grow indefinitely.

journal-compact-percentage

The percentage of live data in the journal files. When less than this percentage is considered live data (and the configured value of **journal-compact-min-files** has also been reached), compaction begins. The default value is **30**.

6.1.6.2. Running compaction from the command-line interface

The following procedure shows how to use the command-line interface (CLI) to compact journal files.

Procedure

1. As the owner of the *<broker_instance_dir>* directory, stop the broker. The example below shows the user **amq-broker**.

   ```
   su - amq-broker
   cd <broker_instance_dir>/bin
   $ ./artemis stop
   ```

2. (Optional) Run the following CLI command to get a full list of parameters for the data tool. By default, the tool uses settings found in *<broker_instance_dir>*/**etc**/**broker.xml**.

   ```
   $ ./artemis help data compact.
   ```

3. Run the following CLI command to compact the data.

   ```
   $ ./artemis data compact.
   ```

4. After the tool has successfully compacted the data, restart the broker.

   ```
   $ ./artemis run
   ```

Additional resources

- AMQ Broker includes a number of CLI commands for managing your journal files. See command-line Tools in the Appendix for more information.

6.1.7. Disabling the disk write cache

Most disks contain hardware write caches. A write cache can increase the apparent performance of the disk because writes are lazily written to the disk later. By default, many systems ship with disk write cache enabled. This means that even after syncing from the operating system, there is no guarantee that the data has actually made it to disk. Therefore, if a failure occurs, critical data can be lost.

Some more expensive disks have non-volatile or battery-backed write caches that do not necessarily lose data in event of failure, but you should test them. If your disk does not have such features, you should ensure that write cache is disabled. Be aware that disabling disk write cache can negatively affect performance.

The following procedure shows how to disable the disk write cache on Linux on Windows.

Procedure

1. On Linux, to manage the disk write cache settings, use the tools **hdparm** (for IDE disks) or **sdparm** or **sginfo** (for SDSI/SATA disks).

2. On Windows, to manage the disk writer cache settings, right-click the disk. Select **Properties**.

6.2. PERSISTING MESSAGE DATA IN A DATABASE

When you persist message data in a database, the broker uses a *Java Database Connectivity* (JDBC) connection to store message and bindings data in database tables. The data in the tables is encoded using AMQ Broker journal encoding. For information about supported databases, see Red Hat AMQ 7 Supported Configurations on the Red Hat Customer Portal.

> **IMPORTANT**
>
> An administrator might choose to store message data in a database based on the requirements of an organization's wider IT infrastructure. However, use of a database can negatively effect the performance of a messaging system. Specifically, writing messaging data to database tables via JDBC creates a significant performance overhead for a broker.

6.2.1. Configuring JDBC persistence

The following procedure shows how to configure the broker to store messages and bindings data in database tables.

Procedure

1. Add the appropriate JDBC client libraries to the broker runtime. To do this, add the relevant **.JAR** files to the ***<broker_instance_dir>*/lib** directory.
   If you want to add the **.JAR** files to a different directory, you must add that directory to the Java classpath. For more information see Section 1.5, "Extending the JAVA Classpath"

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

3. Within the **core** element, add a **store** element that contains a **database-store** element.

```
<configuration>
 <core>
  <store>
    <database-store>
    </database-store>
  </store>
 </core>
</configuration>
```

4. Within the **database-store** element, add configuration parameters for JDBC persistence and specify values. For example:

```
<configuration>
 <core>
  <store>
    <database-store>
      <jdbc-connection-url>jdbc:oracle:data/oracle/database-store;create=true</jdbc-connection-url>
      <jdbc-user>ENC(5493dd76567ee5ec269d11823973462f)</jdbc-user>
      <jdbc-password>ENC(56a0db3b71043054269d11823973462f)</jdbc-password>
      <bindings-table-name>BIND_TABLE</bindings-table-name>
```

```
        <message-table-name>MSG_TABLE</message-table-name>
        <large-message-table-name>LGE_TABLE</large-message-table-name>
        <page-store-table-name>PAGE_TABLE</page-store-table-name>
        <node-manager-store-table-name>NODE_TABLE</node-manager-store-table-name>
        <jdbc-driver-class-name>oracle.jdbc.driver.OracleDriver</jdbc-driver-class-name>
        <jdbc-network-timeout>10000</jdbc-network-timeout>
        <jdbc-lock-renew-period>2000</jdbc-lock-renew-period>
        <jdbc-lock-expiration>20000</jdbc-lock-expiration>
        <jdbc-journal-sync-period>5</jdbc-journal-sync-period>
        <jdbc-max-page-size-bytes>100K</jdbc-max-page-size-bytes>
      </database-store>
    </store>
  </core>
</configuration>
```

jdbc-connection-url

Full JDBC connection URL for your database server. The connection URL should include all configuration parameters and the database name.

jdbc-user

Encrypted user name for your database server. For more information about encrypting user names and passwords for use in configuration files, see Section 5.9, "Encrypting passwords in configuration files".

jdbc-password

Encrypted password for your database server. For more information about encrypting user names and passwords for use in configuration files, see Section 5.9, "Encrypting passwords in configuration files".

bindings-table-name

Name of the table in which bindings data is stored. Specifying a table name enables you to share a single database between multiple servers, without interference.

message-table-name

Name of the table in which message data is stored. Specifying this table name enables you to share a single database between multiple servers, without interference.

large-message-table-name

Name of the table in which large messages and related data are persisted. In addition, if a client streams a large message in chunks, the chunks are stored in this table. Specifying this table name enables you to share a single database between multiple servers, without interference.

page-store-table-name

Name of the table in which paged store directory information is stored. Specifying this table name enables you to share a single database between multiple servers, without interference.

node-manager-store-table-name

Name of the table in which the shared store high-availability (HA) locks for live and backup brokers and other HA-related data is stored on the broker server. Specifying this table name enables you to share a single database between multiple servers, without interference. Each live-backup pair that uses shared store HA must use the same table name. You cannot share the same table between multiple (and unrelated) live-backup pairs.

jdbc-driver-class-name

Fully-qualified class name of the JDBC database driver. For information about supported databases, see Red Hat AMQ 7 Supported Configurations on the Red Hat Customer Portal.

jdbc-network-timeout

JDBC network connection timeout, in milliseconds. The default value is 20000 milliseconds. When using a JDBC for shared store HA, it is recommended to set the timeout to a value less than or equal to **jdbc-lock-expiration**.

jdbc-lock-renew-period

Length, in milliseconds, of the renewal period for the current JDBC lock. When this time elapses, the broker can renew the lock. It is recommended to set a value that is several times smaller than the value of **jdbc-lock-expiration**. This gives the broker sufficient time to extend the lease and also gives the broker time to try to renew the lock in the event of a connection problem. The default value is 2000 milliseconds.

jdbc-lock-expiration

Time, in milliseconds, that the current JDBC lock is considered owned (that is, acquired or renewed), even if the value of **jdbc-lock-renew-period** has elapsed.
The broker periodically tries to renew a lock that it owns according to the value of **jdbc-lock-renew-period**. If the broker **fails** to renew the lock (for example, due to a connection problem) the broker keeps trying to renew the lock until the value of **jdbc-lock-expiration** has passed since the lock was last successfully acquired or renewed.

An exception to the renewal behavior described above is when another broker acquires the lock. This can happen if there is a time misalignment between the Database Management System (DBMS) and the brokers, or if there is a long pause for garbage collection. In this case, the broker that originally owned the lock considers the lock lost and does not try to renew it.

After the expiration time elapses, if the JDBC lock has not been renewed by the broker that currently owns it, another broker can establish a JDBC lock.

The default value of **jdbc-lock-expiration** is 20000 milliseconds.

jdbc-journal-sync-period

Duration, in milliseconds, for which the broker journal synchronizes with JDBC. The default value is 5 milliseconds.

jdbc-max-page-size-bytes

Maximum size, in bytes, of each page file when AMQ Broker persists messages to a JDBC database. The default value is **102400**, which is 100KB. The value that you specify also supports byte notation such as "K" "MB", and "GB".

6.2.2. Configuring JDBC connection pooling

If you have configured the broker for JDBC persistence, the broker uses a JDBC connection to store messages and bindings data in database tables.

In the event of a JDBC connection failure, and provided that there is no active connection activity (such as a database read or write) when the failure occurs, the broker stays running and tries to re-establish the database connection. To achieve this, AMQ Broker uses JDBC *connection pooling*.

In general, a *connection pool* provides a set of open connections to a specified database that can be shared between multiple applications. For a broker, if the connection between a broker and the database fails, the broker attempts to reconnect to the database using a different connection from the pool. The pool tests the new connection before the broker receives it.

The following example shows how to configure JDBC connection pooling.

> **IMPORTANT**
>
> If you do not explicitly configure JDBC connection pooling, the broker uses connection pooling with a default configuration. The default configuration uses values from your existing JDBC configuration. For more information, see Default connection pooling configuration.

Prerequisites

- This example builds on the example for configuring JDBC persistence. See Section 6.2.1, "Configuring JDBC persistence"

- To enable connection pooling, AMQ Broker uses the Apache Commons DBCP package. Before configuring JDBC connection pooling for the broker, you should be familiar with what this package provides. For more information, see:

  - Overview of Apache Commons DBCP

  - Apache Commons DBCP Configuration Parameters

Procedure

1. Open the ***<broker-instance-dir>*/etc/broker.xml** configuration file.

2. Within the **database-store** element that you previously added for your JDBC configuration, remove the **jdbc-driver-class-name**, **jdbc-connection-url**, **jdbc-user**, **jdbc-password**, parameters. Later in this procedure, you will replace these with corresponding DBCP configuration parameters.

   > **NOTE**
   >
   > If you do not explicitly remove the preceding parameters, the corresponding DBCP parameters that you add later in this procedure take precedence.

3. Within the **database-store** element, add a **data-source-properties** element. For example:

   ```
   <store>
     <database-store>
       <data-source-properties>
       </data-source-properties>
       <bindings-table-name>BINDINGS</bindings-table-name>
       <message-table-name>MESSAGES</message-table-name>
       <large-message-table-name>LARGE_MESSAGES</large-message-table-name>
       <page-store-table-name>PAGE_STORE</page-store-table-name>
       <node-manager-store-table-name>NODE_MANAGER_STORE</node-manager-store-table-name>
       <jdbc-network-timeout>10000</jdbc-network-timeout>
       <jdbc-lock-renew-period>2000</jdbc-lock-renew-period>
       <jdbc-lock-expiration>20000</jdbc-lock-expiration>
       <jdbc-journal-sync-period>5</jdbc-journal-sync-period>
     </database-store>
   </store>
   ```

4. Within the new **data-source-properties** element, add DBCP data source properties for connection pooling. Specify key–value pairs. For example:

```
<store>
  <database-store>
    <data-source-properties>
      <data-source-property key="driverClassName" value="com.mysql.jdbc.Driver" />
      <data-source-property key="url" value="jdbc:mysql://localhost:3306/artemis" />
      <data-source-property key="username"
value="ENC(5493dd76567ee5ec269d1182397346f)"/>
      <data-source-property key="password"
value="ENC(56a0db3b71043054269d1182397346f)"/>
      <data-source-property key="poolPreparedStatements" value="true" />
      <data-source-property key="maxTotal" value="-1" />
    </data-source-properties>
    <bindings-table-name>BINDINGS</bindings-table-name>
    <message-table-name>MESSAGES</message-table-name>
    <large-message-table-name>LARGE_MESSAGES</large-message-table-name>
    <page-store-table-name>PAGE_STORE</page-store-table-name>
    <node-manager-store-table-name>NODE_MANAGER_STORE</node-manager-store-table-name>
    <jdbc-network-timeout>10000</jdbc-network-timeout>
    <jdbc-lock-renew-period>2000</jdbc-lock-renew-period>
    <jdbc-lock-expiration>20000</jdbc-lock-expiration>
    <jdbc-journal-sync-period>5</jdbc-journal-sync-period>
  </database-store>
</store>
```

driverClassName

Fully-qualified class name of the JDBC database driver.

url

Full JDBC connection URL for your database server.

username

Encrypted user name for your database server. You can also specify this value as unencrypted, plain text. For more information about encrypting user names and passwords for use in configuration files, see Section 5.9, "Encrypting passwords in configuration files".

password

Encrypted password for your database server. You can also specify this value as unencrypted, plain text. For more information about encrypting user names and passwords for use in configuration files, see Section 5.9, "Encrypting passwords in configuration files".

poolPreparedStatements

When the value of this parameter is set to **true**, the pool can have an unlimited number of cached prepared statements. This reduces initialization costs.

maxTotal

Maximum number of connections in the pool. When the value of this parameter is set to **-1**, there is no limit.

If you do not explicitly configure JDBC connection pooling, the broker uses connection pooling with a default configuration. The default configuration is described in the table.

Table 6.1. Default connection pooling configuration

DBCP configuration parameter	Default value
driverClassName	The value of the existing **jdbc-driver-class-name** parameter
url	The value of the existing **jdbc-connection-url** parameter
username	The value of the existing **jdbc-user** parameter
password	The value of the existing **jdbc-password** parameter
poolPreparedStatements	**true**
maxTotal	**-1**

> **NOTE**
>
> Reconnection works **only** if no client is actively sending messages to the broker. If there is an attempt to write to the database tables during reconnection, the broker fails and shuts down.

Additional resources

- For information about databases supported by AMQ Broker, see Red Hat AMQ 7 Supported Configurations on the Red Hat Customer Portal.

- To learn about all of the configuration options available in the Apache Commons DBCP package, see Apache Commons DBCP Configuration Parameters.

6.3. DISABLING PERSISTENCE

In some situations, it might be a requirement that a messaging system **does not** store any data. In these situations you can disable persistence on the broker.

The following procedure shows how to disable persistence.

Procedure

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

2. Within the **core** element, set the value of the **persistence-enabled** parameter to **false**.

   ```
   <configuration>
    <core>
      ...
      <persistence-enabled>false</persistence-enabled>
      ...
    </core>
   </configuration>
   ```

No message data, bindings data, large message data, duplicate ID cache, or paging data is persisted.

CHAPTER 7. CONFIGURING MEMORY USAGE FOR ADDRESSES

AMQ Broker transparently supports huge queues containing millions of messages, even if the machine that is hosting the broker is running with limited memory.

In these situations, it might be not possible to store all of the queues in memory at any one time. To protect against excess memory consumption, you can configure the maximum memory usage that is allowed for each address on the broker. In addition, you can configure the broker to take one of the following actions when memory usage for an address reaches the configured limit:

- Page messages

- Silently drop messages

- Drop messages and notify the sending clients

- Block clients from sending messages

If you configure the broker to page messages when the maximum memory usage for an address is reached, you can configure limits for specific addresses to:

- Limit the disk space used to page incoming messages

- Limit the memory used for paged messages that the broker transfers from disk back to memory when clients are ready to consume messages.

You can also set a disk usage threshold, which overrides all the configured paging limits. If the disk usage threshold is reached, the broker stops paging and blocks all incoming messages.

> **IMPORTANT**
>
> When you use transactions, the broker might allocate extra memory to ensure transactional consistency. In this case, the memory usage reported by the broker might not reflect the total number of bytes being used in memory. Therefore, if you configure the broker to page, drop, or block messages based on a specified maximum memory usage, you should not also use transactions.

7.1. CONFIGURING MESSAGE PAGING

For any address that has a maximum memory usage limit specified, you can also specify what action the broker takes when that usage limit is reached. One of the options that you can configure is *paging*.

If you configure the paging option, when the maximum size of an address is reached, the broker starts to store messages for that address on disk, in files known as *page files*. Each page file has a maximum size that you can configure. Each address that you configure in this way has a dedicated folder in your file system to store paged messages.

Both queue browsers and consumers can navigate through page files when inspecting messages in a queue. However, a consumer that is using a very specific filter might not be able to consume a message that is stored in a page file until existing messages in the queue have been consumed first. For example, suppose that a consumer filter includes a string expression such as **"color='red'"**. If a message that meets this condition follows one million messages with the property **"color='blue'"**, the consumer cannot consume the message until those with **"color='blue'"** have been consumed first.

The broker transfers (that is, *depages*) messages from disk into memory when clients are ready to consume them. The broker removes a page file from disk when all messages in that file have been acknowledged.

> **IMPORTANT**
>
> AMQ Broker orders pending messages in a queue by JMS message priority. However, messages that are paged are not ordered by priority. If you want to preserve the ordering, do not configure paging and size the broker sufficiently to keep all messages in memory.

7.1.1. Specifying a paging directory

The following procedure shows how to specify the location of the paging directory.

Procedure

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

2. Within the **core** element, add the **paging-directory** element. Specify a location for the paging directory in your file system.

   ```
   <configuration ...>
    <core ...>
      ...
       <paging-directory>/path/to/paging-directory</paging-directory>
      ...
    </core>
   </configuration>
   ```

   For each address that you subsequently configure for paging, the broker adds a dedicated directory within the paging directory that you have specified.

7.1.2. Configuring an address for paging

The following procedure shows how to configure an address for paging.

Prerequisites

- You should be familiar with how to configure addresses and address settings. For more information, see Chapter 4, *Configuring addresses and queues* .

Procedure

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

2. For an **address-setting** element that you have configured for a matching address or *set* of addresses, add configuration elements to specify maximum memory usage and define paging behavior. For example:

   ```
   <address-settings>
     <address-setting match="my.paged.address">
       ...
        <max-size-bytes>104857600</max-size-bytes>
        <max-size-messages>20000</max-size-messages>
   ```

```
        <page-size-bytes>10485760</page-size-bytes>
        <address-full-policy>PAGE</address-full-policy>
        ...
    </address-setting>
</address-settings>
```

max-size-bytes

Maximum size, in bytes, of the memory allowed for the address before the broker executes the action specified for the **address-full-policy** attribute. The default value is **-1**, which means that there is no limit. The value that you specify also supports byte notation such as "K", "MB", and "GB".

max-size-messages

Maximum number of messages allowed for the address before the broker executes the action specified for the **address-full-policy** attribute. The default value is –1, which means that there is no message limit.

page-size-bytes

Size, in bytes, of each page file used on the paging system. The default value is **10485760** (that is, 10 MiB). The value that you specify also supports byte notation such as "K", "MB", and "GB".

address-full-policy

Action that the broker takes when the maximum size for an address has been reached. The default value is **PAGE**. Valid values are:

PAGE

The broker pages any further messages to disk.

DROP

The broker silently drops any further messages.

FAIL

The broker drops any further messages and issues exceptions to client message producers.

BLOCK

Client message producers block when they try to send further messages.

If you set limits for the **max-size-bytes** and **max-size-message** attributes, the broker executes the action specified for the **address-full-policy** attribute when either limit is reached. With the configuration in the previous example, the broker starts paging messages for the **my.paged.address** address when the total messages for the address in memory exceeds 20,000 or uses 104857600 bytes of available memory.

Additional paging configuration elements that are **not** shown in the preceding example are described below.

page-sync-timeout

Time, in nanoseconds, between periodic page synchronizations. If you are using an asynchronous IO journal (that is, **journal-type** is set to **ASYNCIO** in the **broker.xml** configuration file), the default value is **3333333**. If you are using a standard Java NIO journal (that is, **journal-type** is set to **NIO**), the default value is the configured value of the **journal-buffer-timeout** parameter.

In the preceding example , when messages sent to the address **my.paged.address** exceed 104857600 bytes in memory, the broker begins paging.

> **NOTE**
>
> If you specify **max-size-bytes** in an **address-setting** element, the value applies to **each** matching address. Specifying this value **does not** mean that the **total** size of all matching addresses is limited to the value of **max-size-bytes**.

7.1.3. Configuring a global paging size

Sometimes, configuring a memory limit *per address* is not practical, for example, when a broker manages many addresses that have different usage patterns. In these situations, you can specify a global memory limit. The global limit is the **total** amount of memory that the broker can use for all addresses. When this memory limit is reached, the broker executes the action specified for the **address-full-policy** attribute for the address associated with each new incoming message.

The following procedure shows how to configure a global paging size.

Prerequisites

- You should be familiar with how to configure an address for paging. For more information, see Section 7.1.2, "Configuring an address for paging".

Procedure

1. Stop the broker.

   a. On Linux:

      <broker_instance_dir>/bin/artemis stop

   b. On Windows:

      <broker_instance_dir>\bin\artemis-service.exe stop

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

3. Within the **core** element, add the **global-max-size** element and specify a value. For example:

   ```
   <configuration>
    <core>
      ...
      <global-max-size>1GB</global-max-size>
      <global-max-messages>900000</global-max-messages>
      ...
    </core>
   </configuration>
   ```

   global-max-size

   > Total amount of memory, in bytes, that the broker can use for all addresses. When this limit is reached, the broker executes the action specified for the **address-full-policy** attribute for the address associated with each incoming message. The default value of **global-max-size** is half of the maximum memory available to the Java virtual machine (JVM) that is hosting the broker.
   > The value for **global-max-size** is in bytes, but also supports byte notation (for example, "K", "Mb", "GB").

In the preceding example, the broker is configured to use a maximum of one gigabyte of available memory when processing messages.

global-max-messages

The total number of messages allowed for all addresses. When this limit is reached, the broker executes the action specified for the **address-full-policy** attribute for the address associated with each incoming message. The default value is -1, which means that there is no message limit.

If you set limits for the **global-max-size** and **global-max-messages** attributes, the broker executes the action specified for the **address-full-policy** attribute when either limit is reached. With the configuration in the previous example, the broker starts paging messages for all addresses when the number of messages in memory exceeds 900,000 or uses 1 GB of available memory.

> **NOTE**
>
> If limits that are set for an individual address, by using the **max-size-bytes** or **max-size-message** attributes, are reached before the limits set for the **global-max-size** or **global-max-messages** attributes, the broker executes the action specified for the **address-full-policy** attribute for that address.

4. Start the broker.

   a. On Linux:

      <broker_instance_dir>/bin/artemis run

   b. On Windows:

      <broker_instance_dir>\bin\artemis-service.exe start

7.1.4. Limiting disk usage during paging for specific addresses

You can limit the amount of disk space that the broker can use before it stops paging incoming messages for an individual address or set of addresses.

Procedure

1. Stop the broker.

   a. On Linux:

      <broker_instance_dir>/bin/artemis stop

   b. On Windows:

      <broker_instance_dir>\bin\artemis-service.exe stop

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

3. Within the **core** element, add attributes to specify paging limits based on disk usage or number of messages, or both, and specify the action to take if either limit is reached. For example:

```
<address-settings>
  <address-setting match="match="my.paged.address"">
    ...
    <page-limit-bytes>10G</page-limit-bytes>
    <page-limit-messages>1000000</page-limit-messages>
    <page-full-policy>FAIL</page-full-policy>
    ...
  </address-setting>
</address-settings>
```

page-limit-bytes

Maximum size, in bytes, of the disk space allowed for paging incoming messages for the address before the broker executes the action specified for the **page-full-policy** attribute. The value that you specify supports byte notation such as "K", "MB", and "GB". The default value is -1, which means that there is no limit.

page-limit-messages

Maximum number of incoming message that can be paged for the address before the broker executes the action specified for the **page-full-policy** attribute. The default value is -1, which means that there is no message limit.

page-full-policy

Action that the broker takes when a limit set in the **page-limit-bytes** or **page-limit-messages** attributes is reached for an address. Valid values are:
DROP The broker silently drops any further messages.

FAIL The broker drops any further messages and notifies the sending clients

In the preceding example, the broker pages message for the **my.paged.address** address until paging uses 10GB of disk space or until a total of one million messages are paged.

4. Start the broker.

    a. On Linux:

        <broker_instance_dir>/bin/artemis run

    b. On Windows:

        <broker_instance_dir>\bin\artemis-service.exe start

7.1.5. Controlling the flow of paged messages into memory

If AMQ Broker is configured to page messages to disk, the broker reads paged messages and transfers the messages into memory when clients are ready to consume messages. To prevent messages from consuming excess memory, you can limit the memory used by each address for messages that the broker transfers from disk to memory.

IMPORTANT

If client applications leave too many messages pending acknowledgment, the broker does not read paged messages until the pending messages are acknowledged, which can cause message starvation on the broker.

For example, if the limit for the transfer of paged messages into memory, which is 20 MB by default, is reached, the broker waits for an acknowledgment from a client before it reads any more messages. If, at the same time, clients are waiting to receive sufficient messages before they send an acknowledgment to the broker, which is determined by the batch size used by clients, the broker is starved of messages.

To avoid starvation, either increase the broker limits that control the transfer of paged message into memory or reduce the number of delivering messages. You can reduce the number of delivering messages by ensuring that clients either commit message acknowledgments sooner or use a timeout and commit acknowledgments when no more messages are received from the broker.

You can see the number and size of delivering messages in a queue's **Delivering Count** and **Delivering Bytes** metrics in AMQ Management Console.

Procedure

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

2. For an **address-settings** element that you have configured for a matching address or set of addresses, specify limits on the transfer of paged messages into memory. For example:

   ```
   address-settings>
      <address-setting match="my.paged.address">
         ...
         <max-read-page-messages>104857600</max-read-page-messages>
         <max-read-page-bytes>20MB</max-read-page-bytes>
         ...
      </address-setting>
   </address-settings>
   ```

   Max-read-page-messages Maximum number of paged messages that the broker can read from disk into memory per–address. The default value is -1, which means that no limit applies.

   Max-read-page-bytes Maximum size in bytes of paged messages that the broker can read from disk into memory per–address. The default value is 20 MB.

   When it applies these limits, the broker counts both messages in memory that are ready for delivery to consumers and messages that are currently delivering. If consumers are slow to acknowledge messages, delivering messages can cause the memory or message limit to be reached and prevent the broker from reading new messages into memory. As a result, the broker can be starved of messages.

3. If consumers are slow to acknowledge messages and the configured **max-read-page-messages** or **max-read-page-bytes** limits are reached by messages that are currently delivering, specify separate limits on the transfer of paged messages into memory for messages that are currently delivering. For example

   ```
   address-settings>
      <address-setting match="my.paged.address">
   ```

```
        ...
        <prefetch-page-bytes>20MB</prefetch-page-bytes>
        <prefetch-page-messages>104857600</prefetch-page-messages>
        ...
    </address-setting>
</address-settings>
```

prefetch-page-bytes Memory, in bytes, that is available to read paged messages into memory per-queue. The default value is 20 MB.

prefetch-page-messages Number of paged messages that the broker can read from disk into memory per-queue. The default value is -1, which means that no limit applies.

If you specify limits for the **prefetch-page-bytes** or **prefetch-page-messages** parameters to limit the amount of memory or number of messages used by messages currently delivering, set higher limits for the **max-read-page-bytes** or **max-read-page-message** parameter to provide capacity to read new messages into memory.

> **NOTE**
>
> If the value of the max-read-page-bytes parameter is reached before the value of the prefetch-page-bytes parameter, the broker stops reading further paged messages into memory.

7.1.6. Setting a disk usage threshold

You can set a disk usage threshold which, if reached, causes the broker to stop paging and block all incoming messages.

Procedure

1. Stop the broker.

   a. On Linux:

      <broker_instance_dir>/bin/artemis stop

   b. On Windows:

      <broker_instance_dir>\bin\artemis-service.exe stop

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

3. Within the **core** element add the **max-disk-usage** configuration element and specify a value. For example:

   ```
   <configuration>
     <core>
       ...
       <max-disk-usage>80</max-disk-usage>
       ...
     </core>
   </configuration>
   ```

max-disk-usage

Maximum percentage of the available disk space that the broker can use. When this limit is reached, the broker blocks incoming messages. The default value is **90**.
In the preceding example, the broker is limited to using eighty percent of available disk space.

4. Start the broker.

a. On Linux:

> *<broker_instance_dir>*/bin/artemis run

b. On Windows:

> *<broker_instance_dir>*\bin\artemis-service.exe start

7.2. CONFIGURING MESSAGE DROPPING

Section 7.1.2, "Configuring an address for paging" shows how to configure an address for paging. As part of that procedure, you set the value of **address-full-policy** to **PAGE**.

To *drop* messages (rather than paging them) when an address reaches its specified maximum size, set the value of the **address-full-policy** to one of the following:

DROP

When the maximum size of a given address has been reached, the broker silently drops any further messages.

FAIL

When the maximum size of a given address has been reached, the broker drops any further messages and issues exceptions to producers.

7.3. CONFIGURING MESSAGE BLOCKING

The following procedures show how to configure message blocking when a given address reaches the maximum size limit that you have specified.

> **NOTE**
>
> You can configure message blocking **only** for the Core, OpenWire, and AMQP protocols.

7.3.1. Blocking Core and OpenWire producers

The following procedure shows how to configure message blocking for Core and OpenWire message producers when a given address reaches the maximum size limit that you have specified.

Prerequisites

- You should be familiar with how to configure addresses and address settings. For more information, see Chapter 4, *Configuring addresses and queues* .

Procedure

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

2. For an **address-setting** element that you have configured for a matching address or *set* of addresses, add configuration elements to define message blocking behavior. For example:

```
<address-settings>
  <address-setting match="my.blocking.address">
      ...
      <max-size-bytes>300000</max-size-bytes>
      <address-full-policy>BLOCK</address-full-policy>
      ...
  </address-setting>
</address-settings>
```

max-size-bytes

> Maximum size, in bytes, of the memory allowed for the address before the broker executes the policy specified for **address-full-policy**. The value that you specify also supports byte notation such as "K", "MB", and "GB".

> **NOTE**
>
> If you specify **max-size-bytes** in an **address-setting** element, the value applies to **each** matching address. Specifying this value **does not** mean that the **total** size of all matching addresses is limited to the value of **max-size-bytes**.

address-full-policy

> Action that the broker takes when then the maximum size for an address has been reached.

In the preceding example, when messages sent to the address **my.blocking.address** exceed 300000 bytes in memory, the broker begins blocking further messages from Core or OpenWire message producers.

7.3.2. Blocking AMQP producers

Protocols such as Core and OpenWire use a window–size flow control system. In this system, credits represent bytes and are allocated to producers. If a producer wants to send a message, the producer must wait until it has sufficient credits for the size of the message.

By contrast, AMQP flow control credits do not represent bytes. Instead, AMQP credits represent the number of messages a producer is permitted to send, regardless of message size. Therefore, it is possible, in some situations, for AMQP producers to significantly exceed the **max-size-bytes** value of an address.

Therefore, to block AMQP producers, you must use a different configuration element, **max-size-bytes-reject-threshold**. For a matching address or set of addresses, this element specifies the maximum size, in bytes, of all AMQP messages in memory. When the total size of all messages in memory reaches the specified limit, the broker blocks AMQP producers from sending further messages.

The following procedure shows how to configure message blocking for AMQP message producers.

Prerequisites

- You should be familiar with how to configure addresses and address settings. For more information, see Chapter 4, *Configuring addresses and queues* .

Procedure

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

2. For an **address-setting** element that you have configured for a matching address or *set* of addresses, specify the maximum size of all AMQP messages in memory. For example:

   ```
   <address-settings>
     <address-setting match="my.amqp.blocking.address">
       ...
       <max-size-bytes-reject-threshold>300000</max-size-bytes-reject-threshold>
       ...
     </address-setting>
   </address-settings>
   ```

   max-size-bytes-reject-threshold

   Maximum size, in bytes, of the memory allowed for the address before the broker blocks further AMQP messages. The value that you specify also supports byte notation such as "K", "MB", and "GB". By default, **max-size-bytes-reject-threshold** is set to **-1**, which means that there is no maximum size.

   > **NOTE**
   >
   > If you specify **max-size-bytes-reject-threshold** in an **address-setting** element, the value applies to **each** matching address. Specifying this value **does not** mean that the **total** size of all matching addresses is limited to the value of **max-size-bytes-reject-threshold**.

In the preceding example, when messages sent to the address **my.amqp.blocking.address** exceed 300000 bytes in memory, the broker begins blocking further messages from AMQP producers.

7.4. UNDERSTANDING MEMORY USAGE ON MULTICAST ADDRESSES

When a message is routed to an address that has multicast queues bound to it, there is only one copy of the message in memory. Each queue has only a *reference* to the message. Because of this, the associated memory is released only after **all** queues referencing the message have delivered it.

In this type of situation, if you have a slow consumer, the entire address might experience a negative performance impact.

For example, consider this scenario:

- An address has ten queues that use the multicast routing type.

- Due to a slow consumer, one of the queues does not deliver its messages. The other nine queues continue to deliver messages and are empty.

- Messages continue to arrive to the address. The queue with the slow consumer continues to accumulate references to the messages, causing the broker to keep the messages in memory.

- When the maximum size of the address is reached, the broker starts to page messages.

In this scenario because of a single slow consumer, consumers on **all** queues are forced to consume messages from the page system, requiring additional IO.

Additional resources

- To learn how to configure flow control to regulate the flow of data between the broker and producers and consumers, see Flow control in the AMQ Core Protocol JMS documentation.

CHAPTER 8. HANDLING LARGE MESSAGES

Clients might send large messages that can exceed the size of the broker's internal buffer, causing unexpected errors. To prevent this situation, you can configure the broker to store messages as files when the messages are larger than a specified minimum value. Handling large messages in this way means that the broker does not hold the messages in memory. Instead, you specify a directory on disk or in a database table in which the broker stores large message files.

When the broker stores a message as a large message, the queue retains a reference to the file in the large messages directory or database table.

Large message handling is available for the Core Protocol, AMQP, OpenWire and STOMP protocols.

For the Core Protocol and OpenWire protocols, clients specify the minimum large message size in their connection configurations. For the AMQP and STOMP protocols, you specify the minimum large message size in the acceptor defined for each protocol in the broker configuration.

> **NOTE**
>
> It is recommended that you **do not** use different protocols for producing and consuming large messages. To do this, the broker might need to perform several conversions of the message. For example, say that you want to send a message using the AMQP protocol and receive it using OpenWire. In this situation, the broker must first read the entire body of the large message and convert it to use the Core protocol. Then, the broker must perform another conversion, this time to the OpenWire protocol. Message conversions such as these cause significant processing overhead on the broker.

The minimum large message size that you specify for any of the preceding protocols is affected by system resources such as the amount of disk space available, as well as the sizes of the messages. It is recommended that you run performance tests using several values to determine an appropriate size.

The procedures in this section show how to:

- Configure the broker to store large messages

- Configure acceptors for the AMQP and STOMP protocols for large message handling

This section also links to additional resources about configuring AMQ Core Protocol and AMQ OpenWire JMS clients to work with large messages.

8.1. CONFIGURING THE BROKER FOR LARGE MESSAGE HANDLING

The following procedure shows how to specify a directory on disk or a database table in which the broker stores large message files.

Procedure

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

2. Specify where you want the broker to store large message files.

   a. If you are storing large messages on disk, add the **large-messages-directory** parameter within the **core** element and specify a file system location. For example:

      <configuration>

```
  <core>
    ...
    <large-messages-directory>/path/to/my-large-messages-directory</large-messages-
    directory>
    ...
  </core>
</configuration>
```

NOTE

If you do not explicitly specify a value for **large-messages-directory**, the
broker uses a default value of ***<broker_instance_dir>*/data/largemessages**

b. If you are storing large messages in a database table, add the **large-message-table**
parameter to the **database-store** element and specify a value. For example:

```
<store>
  <database-store>
    ...
    <large-message-table>MY_TABLE</large-message-table>
    ...
  </database-store>
</store>
```

NOTE

If you do not explicitly specify a value for **large-message-table**, the broker
uses a default value of **LARGE_MESSAGE_TABLE**.

Additional resources

- For more information about configuring a database store, see Section 6.2, "Persisting message
  data in a database".

8.2. CONFIGURING AMQP ACCEPTORS FOR LARGE MESSAGE HANDLING

The following procedure shows how to configure an AMQP acceptor to handle an AMQP message larger
than a specified size as a large message.

Procedure

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file.
   The default AMQP acceptor in the broker configuration file looks as follows:

```
<acceptors>
  ...
  <acceptor name="amqp">tcp://0.0.0.0:5672?
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=AMQP;useEpoll=true;a
mqpCredits=1000;amqpLowCredits=300</acceptor>
  ...
</acceptors>
```

2. In the default AMQP acceptor (or another AMQP acceptor that you have configured), add the **amqpMinLargeMessageSize** property and specify a value. For example:

```
<acceptors>
    ...
    <acceptor name="amqp">tcp://0.0.0.0:5672?
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=AMQP;useEpoll=true;a
mqpCredits=1000;amqpLowCredits=300;amqpMinLargeMessageSize=204800</acceptor>
    ...
</acceptors>
```

In the preceding example, the broker is configured to accept AMQP messages on port 5672. Based on the value of **amqpMinLargeMessageSize**, if the acceptor receives an AMQP message with a body larger than or equal to 204800 bytes (that is, 200 kilobytes), the broker stores the message as a large message. If you do not explicitly specify a value for this property, the broker uses a default value of 102400 (that is, 100 kilobytes).

> **NOTE**
>
> - If you set **amqpMinLargeMessageSize** to -1, large message handling for AMQP messages is disabled.
>
> - If the broker receives a persistent AMQP message that does not exceed the value of **amqpMinLargeMessageSize**, but which *does* exceed the size of the messaging journal buffer (specified using the **journal-buffer-size** configuration parameter), the broker converts the message to a large Core Protocol message, before storing it in the journal.

8.3. CONFIGURING STOMP ACCEPTORS FOR LARGE MESSAGE HANDLING

The following procedure shows how to configure a STOMP acceptor to handle a STOMP message larger than a specified size as a large message.

Procedure

1. Open the *<broker_instance_dir>*/**etc/broker.xml** configuration file.
   The default AMQP acceptor in the broker configuration file looks as follows:

```
<acceptors>
    ...
    <acceptor name="stomp">tcp://0.0.0.0:61613?
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=STOMP;useEpoll=true
</acceptor>
    ...
</acceptors>
```

2. In the default STOMP acceptor (or another STOMP acceptor that you have configured), add the **stompMinLargeMessageSize** property and specify a value. For example:

```
<acceptors>
    ...
    <acceptor name="stomp">tcp://0.0.0.0:61613?
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576;protocols=STOMP;useEpoll=true;
```

```
stompMinLargeMessageSize=204800</acceptor>

    ...
</acceptors>
```

In the preceding example, the broker is configured to accept STOMP messages on port 61613. Based on the value of **stompMinLargeMessageSize**, if the acceptor receives a STOMP message with a body larger than or equal to 204800 bytes (that is, 200 kilobytes), the broker stores the message as a large message. If you do not explicitly specify a value for this property, the broker uses a default value of 102400 (that is, 100 kilobytes).

> **NOTE**
>
> To deliver a large message to a STOMP consumer, the broker automatically converts the message from a large message to a normal message before sending it to the client. If a large message is compressed, the broker decompresses it before sending it to STOMP clients.

8.4. LARGE MESSAGES AND JAVA CLIENTS

There are two options available to Java developers who are writing clients that use large messages.

One option is to use instances of **InputStream** and **OutputStream**. For example, a **FileInputStream** can be used to send a message taken from a large file on a physical disk. A **FileOutputStream** can then be used by the receiver to stream the message to a location on its local file system.

Another option is to stream a JMS **BytesMessage** or **StreamMessage** directly. For example:

```
BytesMessage rm = (BytesMessage)cons.receive(10000);
byte data[] = new byte[1024];
for (int i = 0; i < rm.getBodyLength(); i += 1024)
{
  int numberOfBytes = rm.readBytes(data);
  // Do whatever you want with the data
}
```

Additional resources

- To learn about working with large messages in the AMQ Core Protocol JMS client, see:

  - Large message options

  - Writing to a streamed large message

  - Reading from a streamed large message

- For an example of working with large messages, see the large message example . To learn more about running example programs, see Running the AMQ Broker examples.

CHAPTER 9. DETECTING DEAD CONNECTIONS

Sometimes clients stop unexpectedly and do not have a chance to clean up their resources. If this occurs, it can leave resources in a faulty state and result in the broker running out of memory or other system resources. The broker detects that a client's connection was not properly shut down at garbage collection time. The connection is then closed and a message similar to the one below is written to the log. The log captures the exact line of code where the client session was instantiated. This enables you to identify the error and correct it.

```
[Finalizer] 20:14:43,244 WARNING [org.apache.activemq.artemis.core.client.impl.DelegatingSession]
I'm closing a JMS Conection you left open. Please make sure you close all connections explicitly
before let
ting them go out of scope!
[Finalizer] 20:14:43,244 WARNING [org.apache.activemq.artemis.core.client.impl.DelegatingSession]
The session you didn't close was created here:
java.lang.Exception
   at org.apache.activemq.artemis.core.client.impl.DelegatingSession.<init>
(DelegatingSession.java:83)
   at org.acme.yourproject.YourClass (YourClass.java:666)  ❶
```

❶ The line in the client code where the connection was instantiated.

9.1. CONNECTION TIME-TO-LIVE

Because the network connection between the client and the server can fail and then come back online, allowing a client to reconnect, AMQ Broker waits to clean up inactive server-side resources. This wait period is called a time-to-live (TTL). The default TTL for a network-based connection is **60000** milliseconds (1 minute). The default TTL on an in-VM connection is **-1**, which means the broker never times out the connection on the broker side.

Configuring Time-To-Live on the Broker

If you do not want clients to specify their own connection TTL, you can set a global value on the broker side. This can be done by specifying the **connection-ttl-override** element in the broker configuration.

The logic to check connections for TTL violations runs periodically on the broker, as determined by the **connection-ttl-check-interval** element.

Procedure

- Edit ***<broker_instance_dir>*/etc/broker.xml** by adding the **connection-ttl-override** configuration element and providing a value for the time-to-live, as in the example below.

```
<configuration>
 <core>
  ...
  <connection-ttl-override>30000</connection-ttl-override>  ❶
  <connection-ttl-check-interval>1000</connection-ttl-check-interval>  ❷
  ...
 </core>
</configuration>
```

❶ The global TTL for all connections is set to 30000 milliseconds. The default value is **-1**, which allows clients to set their own TTL.

2   The interval between checks for dead connections is set to 1000 milliseconds. By default, the checks are done every 2000 milliseconds.

9.2. DISABLING ASYNCHRONOUS CONNECTION EXECUTION

Most packets received on the broker side are executed on the **remoting** thread. These packets represent short-running operations and are always executed on the **remoting** thread for performance reasons. However, some packet types are executed using a thread pool instead of the **remoting** thread, which adds a little network latency.

The packet types that use the thread pool are implemented within the Java classes listed below. The classes are all found in the package **org.apache.actiinvemq.artemis.core.protocol.core.impl.wireformat**.

- RollbackMessage

- SessionCloseMessage

- SessionCommitMessage

- SessionXACommitMessage

- SessionXAPrepareMessage

- SessionXARollbackMessage

Procedure

- To disable asynchronous connection execution, add the **async-connection-execution-enabled** configuration element to **<broker_instance_dir>/etc/broker.xml** and set it to **false**, as in the example below. The default value is **true**.

  ```
  <configuration>
   <core>
    ...
    <async-connection-execution-enabled>false</async-connection-execution-enabled>
    ...
   </core>
  </configuration>
  ```

Additional resources

- To learn how to configure the AMQ Core Protocol JMS client to detect dead connections, see Detecting dead connections in the AMQ Core Protocol JMS documentation.

- To learn how to configure a connection time-to-live in the AMQ Core Protocol JMS client, see Configuring time-to-live in the AMQ Core Protocol JMS documentation.

CHAPTER 10. DETECTING DUPLICATE MESSAGES

You can configure the broker to automatically detect and filter duplicate messages. This means that you do not have to implement your own duplicate detection logic.

Without duplicate detection, in the event of an unexpected connection failure, a client cannot determine whether a message it sent to the broker was received. In this situation, the client might assume that the broker did not receive the message, and resend it. This results in a duplicate message.

For example, suppose that a client sends a message to the broker. If the broker or connection fails *before* the message is received and processed by the broker, the message never arrives at its address. The client does not receive a response from the broker due to the failure. If the broker or connection fails *after* the message is received and processed by the broker, the message is routed correctly, but the client still does not receive a response.

In addition, using a transaction to determine success does not necessarily help in these cases. If the broker or connection fails while the transaction commit is being processed, the client is still unable to determine whether it successfully sent the message.

In these situations, to correct the assumed failure, the client resends the most recent message. The result might be a duplicate message that negatively impacts your system. For example, if you are using the broker in an order-fulfilment system, a duplicate message might mean that a purchase order is processed twice.

The following procedures show how to configure duplicate message detection to protect against these types of situations.

10.1. CONFIGURING THE DUPLICATE ID CACHE

To enable the broker to detect duplicate messages, producers must provide unique values for the message property **_AMQ_DUPL_ID** when sending each message. The broker maintains caches of received values of the **_AMQ_DUPL_ID** property. When a broker receives a new message on an address, it checks the cache for that address to ensure that it has not previously processed a message with the same value for this property.

Each address has its own cache. Each cache is circular and fixed in size. This means that new entries replace the oldest ones as cache space demands.

The following procedure shows how to globally configure the ID cache used by each address on the broker.

Procedure

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

2. Within the **core** element, add the **id-cache-size** and **persist-id-cache** properties and specify values. For example:

```
<configuration>
 <core>
  ...
  <id-cache-size>5000</id-cache-size>
  <persist-id-cache>false</persist-id-cache>
 </core>
</configuration>
```

id-cache-size

Maximum size of the ID cache, specified as the number of individual entries in the cache. The default value is 20,000 entries. In this example, the cache size is set to 5,000 entries.
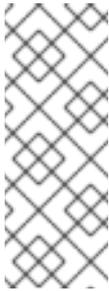
> **NOTE**
>
> When the maximum size of the cache is reached, it is possible for the broker to start processing duplicate messages. For example, suppose that you set the size of the cache to **3000**. If a previous message arrived **more** than 3,000 messages before the arrival of a new message with the same value of **_AMQ_DUPL_ID**, the broker cannot detect the duplicate. This results in both messages being processed by the broker.

persist-id-cache

When the value of this property is set to **true**, the broker persists IDs to disk as they are received. The default value is **true**. In the example above, you disable persistence by setting the value to **false**.

Additional resources

- To learn how to set the duplicate ID message property using the AMQ Core Protocol JMS client, see Using duplicate message detection in the AMQ Core Protocol JMS client documentation.

10.2. CONFIGURING DUPLICATE DETECTION FOR CLUSTER CONNECTIONS

You can configure cluster connections to insert a duplicate ID header for each message that moves across the cluster.

Prerequisites

- You should have already configured a broker cluster. For more information, see Section 14.2, "Creating a broker cluster"

Procedure

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

2. Within the **core** element, for a given cluster connection, add the **use-duplicate-detection** property and specify a value. For example:

```
<configuration>
 <core>
   ...
   <cluster-connections>
     <cluster-connection name="my-cluster">
      <use-duplicate-detection>true</use-duplicate-detection>
      ...
     </cluster-connection>
      ...
   </cluster-connections>
 </core>
</configuration>
```

■

use-duplicate-detection

When the value of this property is set to **true**, the cluster connection inserts a duplicate ID header for each message that it handles.

CHAPTER 11. INTERCEPTING MESSAGES

With AMQ Broker you can intercept packets entering or exiting the broker, allowing you to audit packets or filter messages. Interceptors can change the packets they intercept, which makes them powerful, but also potentially dangerous.

You can develop interceptors to meet your business requirements. Interceptors are protocol specific and must implement the appropriate interface.

Interceptors must implement the **intercept()** method, which returns a boolean value. If the value is **true**, the message packet continues onward. If **false**, the process is aborted, no other interceptors are called, and the message packet is not processed further.

11.1. CREATING INTERCEPTORS

You can create your own incoming and outgoing interceptors. All interceptors are protocol specific and are called for any packet entering or exiting the server respectively. This allows you to create interceptors to meet business requirements such as auditing packets. Interceptors can change the packets they intercept. This makes them powerful as well as potentially dangerous, so be sure to use them with caution.

Interceptors and their dependencies must be placed in the Java classpath of the broker. You can use the ***<broker_instance_dir>*/lib** directory since it is part of the classpath by default.

Procedure

The following examples demonstrate how to create an interceptor that checks the size of each packet passed to it. Note that the examples implement a specific interface for each protocol.

- Implement the appropriate interface and override its **intercept()** method.

  - If you are using the AMQP protocol, implement the **org.apache.activemq.artemis.protocol.amqp.broker.AmqpInterceptor** interface.

    ```java
    package com.example;

    import org.apache.activemq.artemis.protocol.amqp.broker.AMQPMessage;
    import org.apache.activemq.artemis.protocol.amqp.broker.AmqpInterceptor;
    import org.apache.activemq.artemis.spi.core.protocol.RemotingConnection;

    public class MyInterceptor implements AmqpInterceptor
    {
      private final int ACCEPTABLE_SIZE = 1024;

      @Override
      public boolean intercept(final AMQPMessage message, RemotingConnection connection)
      {
        int size = message.getEncodeSize();
        if (size <= ACCEPTABLE_SIZE) {
          System.out.println("This AMQPMessage has an acceptable size.");
          return true;
        }
        return false;
      }
    }
    ```

- If you are using Core Protocol, your interceptor must implement the **org.apache.artemis.activemq.api.core.Interceptor** interface.

```
package com.example;

import org.apache.artemis.activemq.api.core.Interceptor;
import org.apache.activemq.artemis.core.protocol.core.Packet;
import org.apache.activemq.artemis.spi.core.protocol.RemotingConnection;

public class MyInterceptor implements Interceptor
{
  private final int ACCEPTABLE_SIZE = 1024;

  @Override
  boolean intercept(Packet packet, RemotingConnection connection)
  throws ActiveMQException
  {
    int size = packet.getPacketSize();
    if (size <= ACCEPTABLE_SIZE) {
      System.out.println("This Packet has an acceptable size.");
      return true;
    }
    return false;
  }
}
```

- If you are using the MQTT protocol, implement the **org.apache.activemq.artemis.core.protocol.mqtt.MQTTInterceptor** interface.

```
package com.example;

import org.apache.activemq.artemis.core.protocol.mqtt.MQTTInterceptor;
import io.netty.handler.codec.mqtt.MqttMessage;
import org.apache.activemq.artemis.spi.core.protocol.RemotingConnection;

public class MyInterceptor implements Interceptor
{
  private final int ACCEPTABLE_SIZE = 1024;

  @Override
  boolean intercept(MqttMessage mqttMessage, RemotingConnection connection)
  throws ActiveMQException
  {
    byte[] msg = (mqttMessage.toString()).getBytes();
    int size = msg.length;
    if (size <= ACCEPTABLE_SIZE) {
      System.out.println("This MqttMessage has an acceptable size.");
      return true;
    }
    return false;
  }
}
```

- If you are using the STOMP protocol, implement the **org.apache.activemq.artemis.core.protocol.stomp.StompFrameInterceptor** interface.

```java
package com.example;

import org.apache.activemq.artemis.core.protocol.stomp.StompFrameInterceptor;
import org.apache.activemq.artemis.core.protocol.stomp.StompFrame;
import org.apache.activemq.artemis.spi.core.protocol.RemotingConnection;

public class MyInterceptor implements Interceptor
{
  private final int ACCEPTABLE_SIZE = 1024;

  @Override
  boolean intercept(StompFrame stompFrame, RemotingConnection connection)
  throws ActiveMQException
  {
    int size = stompFrame.getEncodedSize();
    if (size <= ACCEPTABLE_SIZE) {
      System.out.println("This StompFrame has an acceptable size.");
      return true;
    }
    return false;
  }
}
```

11.2. CONFIGURING THE BROKER TO USE INTERCEPTORS

Once you have created an interceptor, you must configure the broker to use it.

Prerequisites

You must create an interceptor class and add it (and its dependencies) to the Java classpath of the broker before you can configure it for use by the broker. You can use the **<broker_instance_dir>/lib** directory since it is part of the classpath by default.

Procedure

- Configure the broker to use an interceptor by adding configuration to **<broker_instance_dir>/etc/broker.xml**

  - If your interceptor is intended for incoming messages, add its **class-name** to the list of **remoting-incoming-interceptors**.

    ```xml
    <configuration>
     <core>
       ...
        <remoting-incoming-interceptors>
           <class-name>org.example.MyIncomingInterceptor</class-name>
        </remoting-incoming-interceptors>
       ...
     </core>
    </configuration>
    ```

  - If your interceptor is intended for outgoing messages, add its **class-name** to the list of **remoting-outgoing-interceptors**.

    ```xml
    <configuration>
    ```

```
<core>

  ...
  <remoting-outgoing-interceptors>
    <class-name>org.example.MyOutgoingInterceptor</class-name>
  </remoting-outgoing-interceptors>
 </core>
</configuration>
```

Additional resources

- To learn how to configure interceptors in the AMQ Core Protocol JMS client, see Using message interceptors in the AMQ Core Protocol JMS documentation.

CHAPTER 12. DIVERTING MESSAGES AND SPLITTING MESSAGE FLOWS

In AMQ Broker, you can configure objects called *diverts* that enable you to transparently divert messages from one address to another address, without changing any client application logic. You can also configure a divert to forward a **copy** of a message to a specified forwarding address, effectively splitting the message flow.

12.1. HOW MESSAGE DIVERTS WORK

Diverts enable you to transparently divert messages routed to one address to some other address, without changing any client application logic. Think of the set of diverts on a broker server as a type of routing table for messages.

A divert can be *exclusive*, meaning that a message is diverted to a specified forwarding address without going to its original address.

A divert can also be *non-exclusive*, meaning that a message continues to go to its original address, while the broker sends a copy of the message to a specified forwarding address. Therefore, you can use non-exclusive diverts for splitting message flows. For example, you might split a message flow if you want to separately monitor every order sent to an order queue.

Multiple diverts can be configured for a single address. When an address has both exclusive and non-exclusive diverts configured, the broker processes the exclusive diverts first. If a particular message has already been diverted by an exclusive divert, the broker does not process any non-exclusive diverts for that message. In this case, the message never goes to the original address.

When a broker diverts a message, the broker assigns a new message ID and sets the message address to the new forwarding address. You can retrieve the original message ID and address values via the **_AMQ_ORIG_ADDRESS** (string type) and **_AMQ_ORIG_MESSAGE_ID** (long type) message properties. If you are using the Core API, use the **Message.HDR_ORIGINAL_ADDRESS** and **Message.HDR_ORIG_MESSAGE_ID** properties.

> **NOTE**
>
> You can divert a message only to an address on the same broker server. If you want to divert to an address on a different server, a common solution is to first divert the message to a local store-and-forward queue. Then, set up a bridge that consumes from that queue and forwards messages to an address on a different broker. Combining diverts with bridges enables you to create a distributed network of routing connections between geographically distributed broker servers. In this way, you can create a global messaging mesh.

12.2. CONFIGURING MESSAGE DIVERTS

To configure a divert in your broker instance, add a **divert** element within the **core** element of your **broker.xml** configuration file.

```
<core>
...
  <divert name= >
     <address> </address>
     <forwarding-address> </forwarding-address>
     <filter string= >
```

```
    <routing-type> </routing-type>
    <exclusive> </exclusive>
  </divert>
...
</core>
```

divert

> Named instance of a divert. You can add multiple **divert** elements to your **broker.xml** configuration file, as long as each divert has a unique name.

address

> Address **from which** to divert messages

forwarding-address

> Address **to which** to forward messages

filter

> Optional message filter. If you configure a filter, only messages that match the filter string are diverted. If you do not specify a filter, all messages are considered a match by the divert.

routing-type

> Routing type of the diverted message. You can configure the divert to:

> - Apply the **anycast** or **multicast** routing type to a message
>
> - *Strip* (that is, remove) the existing routing type
>
> - *Pass through* (that is, preserve) the existing routing type

Control of the routing type is useful in situations where the message has its routing type already set, but you want to divert the message to an address that uses a different routing type. For example, the broker cannot route a message with the **anycast** routing type to a queue that uses **multicast** unless you set the **routing-type** parameter of the divert to **MULTICAST**. Valid values for the **routing-type** parameter of a divert are **ANYCAST**, **MULTICAST**, **PASS**, and **STRIP**. The default value is **STRIP**.

exclusive

> Specify whether the divert is exclusive (set the property to **true**) or non- exclusive (set the property to **false**).

The following subsections show configuration examples for exclusive and non-exclusive diverts.

12.2.1. Exclusive divert example

Shown below is an example configuration for an exclusive divert. An exclusive divert diverts all matching messages from the originally-configured address to a new address. Matching messages do not get routed to the original address.

```
<divert name="prices-divert">
  <address>priceUpdates</address>
  <forwarding-address>priceForwarding</forwarding-address>
  <filter string="office='New York'"/>
  <exclusive>true</exclusive>
</divert>
```

In the preceding example, you define a divert called **prices-divert** that diverts any messages sent to the

address **priceUpdates** to another local address, **priceForwarding**. You also specify a message filter string. Only messages with the message property **office** and the value **New York** are diverted. All other messages are routed to their original address. Finally, you specify that the divert is exclusive.

12.2.2. Non-exclusive divert example

Shown below is an example configuration for a non-exclusive divert. In a non-exclusive divert, a message continues to go to its original address, while the broker also sends a copy of the message to a specified forwarding address. Therefore, a non-exclusive divert is a way to split a message flow.

```
<divert name="order-divert">
  <address>orders</address>
  <forwarding-address>spyTopic</forwarding-address>
  <exclusive>false</exclusive>
</divert>
```

In the preceding example, you define a divert called **order-divert** that takes a copy of every message sent to the address **orders** and sends it to a local address called **spyTopic**. You also specify that the divert is non-exclusive.

Additional resources

For a detailed example that uses both exclusive and non-exclusive diverts, and a bridge to forward messages to another broker, see the divert example (external).

CHAPTER 13. FILTERING MESSAGES

AMQ Broker provides a powerful filter language based on a subset of the SQL 92 expression syntax. The filter language uses the same syntax as used for JMS selectors, but the predefined identifiers are different. The table below lists the identifiers that apply to a AMQ Broker message.

Table 13.1. Identfiers for filtering messages

Identifier	Attribute
AMQPriority	The priority of a message. Message priorities are integers with valid values from **0** through **9**. **0** is the lowest priority and **9** is the highest.
AMQExpiration	The expiration time of a message. The value is a long integer.
AMQDurable	Whether a message is durable or not. The value is a string. Valid values are **DURABLE** or **NON_DURABLE**.
AMQTimestamp	The timestamp of when the message was created. The value is a long integer.
AMQSize	The value of the **encodeSize** property of the message. The value of **encodeSize** is the space, in bytes, that the message takes up in the journal. Because the broker uses a double-byte character set to encode messages, the actual size of the message is half the value of **encodeSize**.

Any other identifiers used in core filter expressions are assumed to be properties of the message. For documentation on selector syntax for JMS Messages, see the Java EE API.

13.1. CONFIGURING A QUEUE TO USE A FILTER

You can add a filter to the queues you configure in ***<broker_instance_dir>*/etc/broker.xml**. Only messages that match the filter expression enter the queue.

Procedure

- Add the **filter** element to the desired **queue** and include the filter you want to apply as the value of the element. In the example below, the filter **NEWS='technology'** is added to the queue **technologyQueue**.

  ```
  <configuration>
   <core>
    ...
     <addresses>
       <address name="myQueue">
         <anycast>
           <queue name="myQueue">
             <filter string="NEWS='technology'"/>
           </queue>
         </anycast>
       </address>
  ```

```
        </addresses>
      </core>
    </configuration>
```

13.2. FILTERING JMS MESSAGE PROPERTIES

The JMS specification states that a String property must not be converted to a numeric type when used in a selector. For example, if a message has the **age** property set to the String value **21**, the selector **age > 18** must not match it. This restriction limits STOMP clients because they can send only messages with String properties.

Configuring a Filter to Convert a String to a Number

To convert String properties to a numeric type, add the prefix **convert_string_expressions:** to the value of the **filter**.

Procedure

- Edit *<broker_instance_dir>*/etc/broker.xml by applying the prefix **convert_string_expressions:** to the desired **filter**. The example below edits the **filter** value from **age > 18** to **convert_string_expressions:age > 18**.

```
<configuration>
  <core>
    ...
    <addresses>
      <address name="myQueue">
        <anycast>
          <queue name="myQueue">
            <filter string="convert_string_expressions='age > 18'"/>
          </queue>
        </anycast>
      </address>
    </addresses>
  </core>
</configuration>
```

13.3. FILTERING AMQP MESSAGES BASED ON PROPERTIES ON ANNOTATIONS

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 the 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 are not exposed to clients for regular use, but **can** be specified by a client in a filter.

Shown below are examples of filters based on message properties and annotations. Filtering based on properties is the recommended approach, when possible, because this approach requires less processing by the broker.

Filter based on message properties

```
ConnectionFactory factory = new JmsConnectionFactory("amqp://localhost:5672");
Connection connection = factory.createConnection();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
connection.start();
javax.jms.Queue queue = session.createQueue("my_DLQ");
MessageConsumer consumer = session.createConsumer(queue,
"_AMQ_ORIG_ADDRESS='original_address_name'");
Message message = consumer.receive();
```

Filter based on message annotations

```
ConnectionFactory factory = new JmsConnectionFactory("amqp://localhost:5672");
Connection connection = factory.createConnection();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
connection.start();
javax.jms.Queue queue = session.createQueue("my_DLQ");
MessageConsumer consumer = session.createConsumer(queue, "\"m.x-opt-ORIG-
ADDRESS\"='original_address_name'");
Message message = consumer.receive();
```

> **NOTE**
>
> When consuming AMQP messages based on an annotation, the client must include append a **m.** prefix to the message annotation, as shown in the preceding example.

Additional resources

- For more information about the annotations and properties that the broker applies to expired or undelivered AMQP messages, see Section 4.14, "Annotations and properties on expired or undelivered AMQP messages".

13.4. FILTERING XML MESSAGES

AMQ Broker provides a way of filtering Text messages that contain an XML body using XPath. XPath (XML Path Language) is a query language for selecting nodes from an XML document.

> **NOTE**
>
> Only text based messages are supported. Filtering large messages is not supported.

To filter text based messages, you need to create a Message Selector of the form **XPATH '<xpath-expression>**.

An example of a Message Body

```
<root>
   <a key='first' num='1'/>
   <b key='second' num='2'>b</b>
</root>
```

Filter based on an XPath query

> XPATH 'root/a'

> **WARNING**
>
> Since XPath applies to the body of the message and requires parsing of XML, filtering can be significantly slower than normal filters.

XPath filters are supported with and between producers and consumers using the following protocols:

- OpenWire JMS

- Core (and Core JMS)

- STOMP

- AMQP

Configuring the XML Parser
By default the XML Parser used by the Broker is the Platform default DocumentBuilderFactory instance used by the JDK.

The XML parser used for XPath default configuration includes the following settings:

- **http://xml.org/sax/features/external-general-entities**: **false**

- **http://xml.org/sax/features/external-parameter-entities**: **false**

- **http://apache.org/xml/features/disallow-doctype-decl**: **true**

However, in order to deal with any implementation-specific issues the features can be customized by configuring System properties in the **artemis.profile** configuration file.

org.apache.activemq.documentBuilderFactory.feature:prefix

Example feature configuration

> -Dorg.apache.activemq.documentBuilderFactory.feature:http://xml.org/sax/features/external-general-entities=true

CHAPTER 14. SETTING UP A BROKER CLUSTER

A cluster consists of multiple broker instances that have been grouped together. Broker clusters enhance performance by distributing the message processing load across multiple brokers. In addition, broker clusters can minimize downtime through high availability.

You can connect brokers together in many different cluster topologies. Within the cluster, each active broker manages its own messages and handles its own connections.

You can also balance client connections across the cluster and redistribute messages to avoid broker starvation.

14.1. UNDERSTANDING BROKER CLUSTERS

Before creating a broker cluster, you should understand some important clustering concepts.

14.1.1. How broker clusters balance message load

When brokers are connected to form a cluster, AMQ Broker automatically balances the message load between the brokers. This ensures that the cluster can maintain high message throughput.

Consider a symmetric cluster of four brokers. Each broker is configured with a queue named **OrderQueue**. The **OrderProducer** client connects to **Broker1** and sends messages to **OrderQueue**. **Broker1** forwards the messages to the other brokers in round-robin fashion. The **OrderConsumer** clients connected to each broker consume the messages.

Figure 14.1. Message load balancing in a cluster
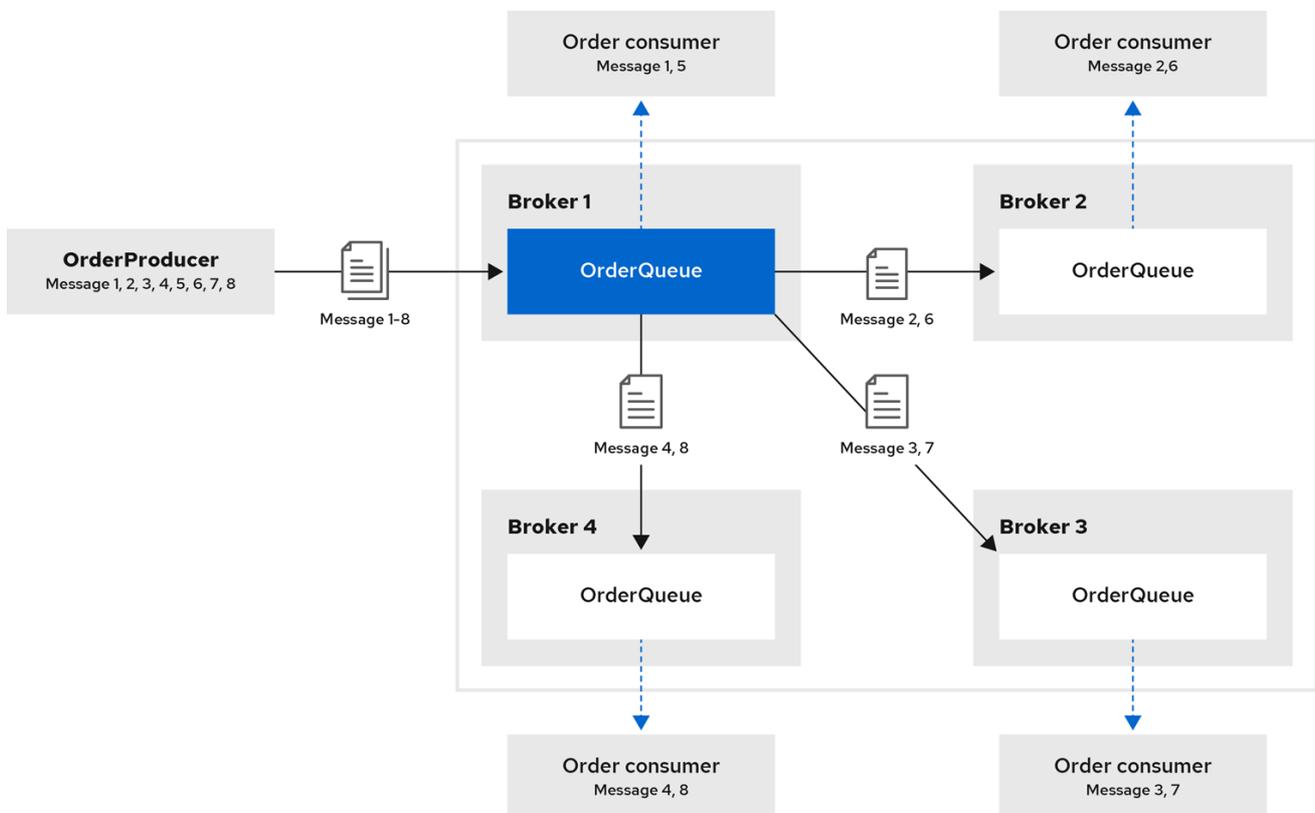
Without message load balancing, the messages sent to **Broker1** would stay on **Broker1** and only **OrderConsumer1** would be able to consume them.

The AMQ Broker automatically load balances messages by default, distributing the first group of messages to the first broker and the second group of messages to the second broker. The order in which the brokers were started determines which broker is first, second and so on.

You can configure:

- the cluster to load balance messages to brokers that have a matching queue.

- the cluster to load balance messages to brokers that have a matching queue with active consumers.

- the cluster to not load balance, but to perform redistribution of messages from queues that do not have any consumers to queues that do have consumers.

- an address to automatically redistribute messages from queues that do not have any consumers to queues that do have consumers.

Additional resources

- The message load balancing policy is configured with the **message-load-balancing** property in each broker's cluster connection. For more information, see Appendix C, *Cluster Connection Configuration Elements*.

- For more information about message redistribution, see Section 14.4.2, "Configuring message redistribution".

14.1.2. How broker clusters improve reliability

Broker clusters make high availability and failover possible, which makes them more reliable than standalone brokers. By configuring high availability, you can ensure that client applications can continue to send and receive messages even if a broker encounters a failure event.

With high availability, the brokers in the cluster are grouped into live-backup groups. A live-backup group consists of a live broker that serves client requests, and one or more backup brokers that wait passively to replace the live broker if it fails. If a failure occurs, the backup brokers replaces the live broker in its live-backup group, and the clients reconnect and continue their work.

14.1.3. Cluster limitations

The following limitation applies when you use AMQ broker in a clustered environment.

Temporary Queues

During a failover, if a client has consumers that use temporary queues, these queues are automatically recreated. The recreated queue name does not match the original queue name, which causes message redistribution to fail and can leave messages stranded in existing temporary queues. Red Hat recommends that you avoid using temporary queues in a cluster. For example, applications that use a request/reply pattern should use fixed queues for the JMSReplyTo address.

14.1.4. Understanding node IDs

The broker *node ID* is a Globally Unique Identifier (GUID) generated programmatically when the journal for a broker instance is first created and initialized. The node ID is stored in the **server.lock** file. The

node ID is used to uniquely identify a broker instance, regardless of whether the broker is a standalone instance, or part of a cluster. Live-backup broker pairs share the same node ID, since they share the same journal.

In a broker cluster, broker instances (nodes) connect to each other and create bridges and internal "store-and-forward" queues. The names of these internal queues are based on the node IDs of the other broker instances. Broker instances also monitor cluster broadcasts for node IDs that match their own. A broker produces a warning message in the log if it identifies a duplicate ID.

When you are using the replication high availability (HA) policy, a master broker that starts and has **check-for-live-server** set to **true** searches for a broker that is using its node ID. If the master broker finds another broker using the same node ID, it either does not start, or initiates failback, based on the HA configuration.

The node ID is *durable*, meaning that it survives restarts of the broker. However, if you delete a broker instance (including its journal), then the node ID is also permanently deleted.

Additional resources

- For more information about configuring the replication HA policy, see Configuring replication high availability.

14.1.5. Common broker cluster topologies

You can connect brokers to form either a *symmetric* or *chain* cluster topology. The topology you implement depends on your environment and messaging requirements.

Symmetric clusters
In a symmetric cluster, every broker is connected to every other broker. This means that every broker is no more than one hop away from every other broker.

Figure 14.2. Symmetric cluster topology
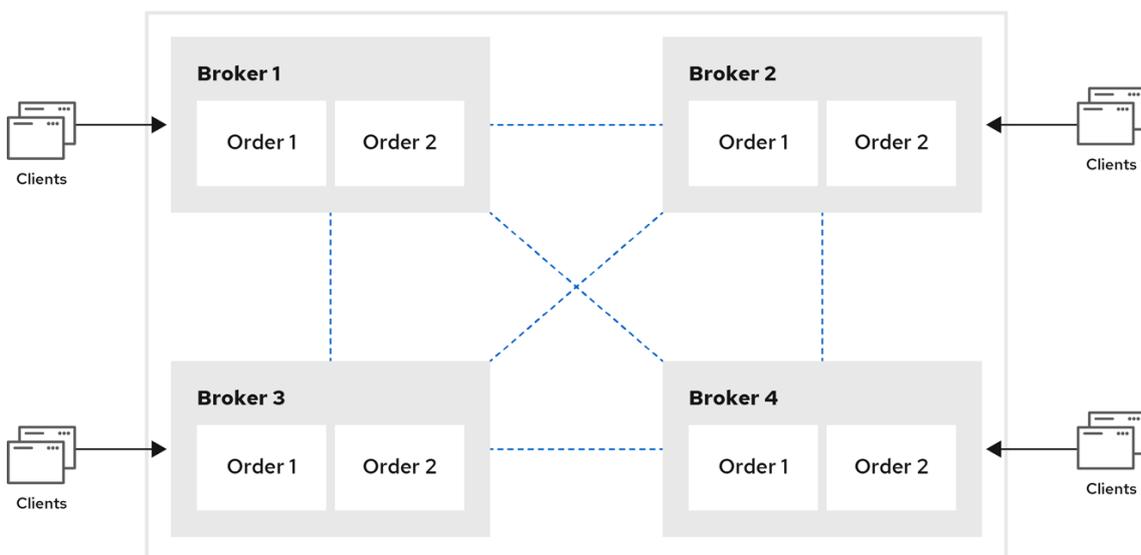
Each broker in a symmetric cluster is aware of all of the queues that exist on every other broker in the cluster and the consumers that are listening on those queues. Therefore, symmetric clusters are able to load balance and redistribute messages more optimally than a chain cluster.

Symmetric clusters are easier to set up than chain clusters, but they can be difficult to use in environments in which network restrictions prevent brokers from being directly connected.

Chain clusters
In a chain cluster, each broker in the cluster is not connected to every broker in the cluster directly. Instead, the brokers form a chain with a broker on each end of the chain and all other brokers just connecting to the previous and next brokers in the chain.

Figure 14.3. Chain cluster topology
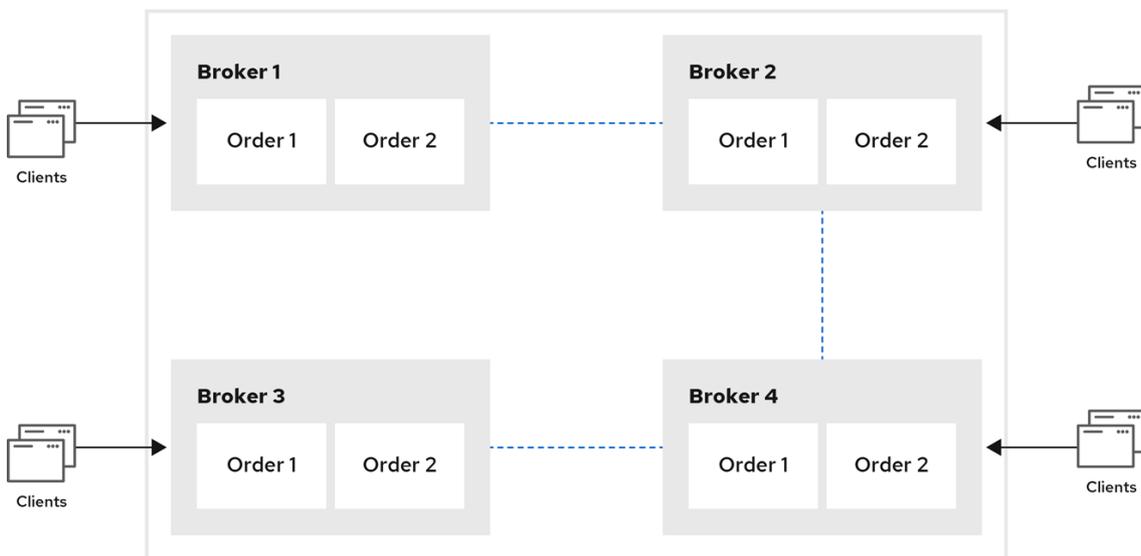
Chain clusters are more difficult to set up than symmetric clusters, but can be useful when brokers are on separate networks and cannot be directly connected. By using a chain cluster, an intermediary broker can indirectly connect two brokers to enable messages to flow between them even though the two brokers are not directly connected.

14.1.6. Broker discovery methods

Discovery is the mechanism by which brokers in a cluster propagate their connection details to each other. AMQ Broker supports both *dynamic discovery* and *static discovery*.

Dynamic discovery
Each broker in the cluster broadcasts its connection settings to the other members through either UDP multicast or JGroups. In this method, each broker uses:

- A *broadcast group* to push information about its cluster connection to other potential members of the cluster.

- A *discovery group* to receive and store cluster connection information about the other brokers in the cluster.

Static discovery
If you are not able to use UDP or JGroups in your network, or if you want to manually specify each member of the cluster, you can use static discovery. In this method, a broker "joins" the cluster by connecting to a second broker and sending its connection details. The second broker then propagates those details to the other brokers in the cluster.

14.1.7. Cluster sizing considerations

Before creating a broker cluster, consider your messaging throughput, topology, and high availability requirements. These factors affect the number of brokers to include in the cluster.

> **NOTE**
>
> After creating the cluster, you can adjust the size by adding and removing brokers. You can add and remove brokers without losing any messages.

Messaging throughput

The cluster should contain enough brokers to provide the messaging throughput that you require. The more brokers in the cluster, the greater the throughput. However, large clusters can be complex to manage.

Topology

You can create either symmetric clusters or chain clusters. The type of topology you choose affects the number of brokers you may need.

For more information, see Section 14.1.5, "Common broker cluster topologies".

High availability

If you require high availability (HA), consider choosing an HA policy before creating the cluster. The HA policy affects the size of the cluster, because each master broker should have at least one slave broker.

For more information, see Section 14.3, "Implementing high availability".

14.2. CREATING A BROKER CLUSTER

You create a broker cluster by configuring a cluster connection on each broker that should participate in the cluster. The cluster connection defines how the broker should connect to the other brokers.

You can create a broker cluster that uses static discovery or dynamic discovery (either UDP multicast or JGroups).

Prerequisites

- You should have determined the size of the broker cluster.
  For more information, see Section 14.1.7, "Cluster sizing considerations".

14.2.1. Creating a broker cluster with static discovery

You can create a broker cluster by specifying a static list of brokers. Use this static discovery method if you are unable to use UDP multicast or JGroups on your network.

Procedure

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

2. Within the **<core>** element, add the following connectors:

   - A connector that defines how other brokers can connect to this one

   - One or more connectors that define how this broker can connect to other brokers in the cluster

```
<configuration>
  <core>
    ...
    <connectors>
      <connector name="netty-connector">tcp://localhost:61617</connector>   1
      <connector name="broker2">tcp://localhost:61618</connector>   2
      <connector name="broker3">tcp://localhost:61619</connector>
    </connectors>
    ...
  </core>
</configuration>
```

1 This connector defines connection information that other brokers can use to connect to this one. This information will be sent to other brokers in the cluster during discovery.

2 The **broker2** and **broker3** connectors define how this broker can connect to two other brokers in the cluster, one of which will always be available. If there are other brokers in the cluster, they will be discovered by one of these connectors when the initial connection is made.

For more information about connectors, see Section 2.3, "About connectors".

3. Add a cluster connection and configure it to use static discovery.
   By default, the cluster connection will load balance messages for all addresses in a symmetric topology.

```
<configuration>
  <core>
    ...
    <cluster-connections>
      <cluster-connection name="my-cluster">
        <connector-ref>netty-connector</connector-ref>
        <static-connectors>
          <connector-ref>broker2-connector</connector-ref>
          <connector-ref>broker3-connector</connector-ref>
        </static-connectors>
      </cluster-connection>
    </cluster-connections>
    ...
  </core>
</configuration>
```

cluster-connection

Use the **name** attribute to specify the name of the cluster connection.

connector-ref

The connector that defines how other brokers can connect to this one.

static-connectors

One or more connectors that this broker can use to make an initial connection to another broker in the cluster. After making this initial connection, the broker will discover the other brokers in the cluster. You only need to configure this property if the cluster uses static discovery.

4. Configure any additional properties for the cluster connection.

These additional cluster connection properties have default values that are suitable for most common use cases. Therefore, you only need to configure these properties if you do not want the default behavior. For more information, see Appendix C, *Cluster Connection Configuration Elements*.

5. Create the cluster user and password.
   AMQ Broker ships with default cluster credentials, but you should change them to prevent unauthorized remote clients from using these default credentials to connect to the broker.

   > **IMPORTANT**
   >
   > The cluster password must be the same on every broker in the cluster.

   ```
   <configuration>
     <core>
        ...
        <cluster-user>cluster_user</cluster-user>
        <cluster-password>cluster_user_password</cluster-password>
        ...
     </core>
   </configuration>
   ```

6. Repeat this procedure on each additional broker.
   You can copy the cluster configuration to each additional broker. However, do not copy any of the other AMQ Broker data files (such as the bindings, journal, and large messages directories). These files must be unique among the nodes in the cluster or the cluster will not form properly.

Additional resources

- For an example of a broker cluster that uses static discovery, see the **clustered-static-discovery** example.

14.2.2. Creating a broker cluster with UDP-based dynamic discovery

You can create a broker cluster in which the brokers discover each other dynamically through UDP multicast.

Procedure

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

2. Within the **<core>** element, add a connector.
   This connector defines connection information that other brokers can use to connect to this one. This information will be sent to other brokers in the cluster during discovery.

   ```
   <configuration>
     <core>
        ...
        <connectors>
           <connector name="netty-connector">tcp://localhost:61617</connector>
        </connectors>
        ...
     </core>
   </configuration>
   ```

■

3. Add a UDP broadcast group.

The broadcast group enables the broker to push information about its cluster connection to the other brokers in the cluster. This broadcast group uses UDP to broadcast the connection settings:

```
<configuration>
  <core>
    ...
    <broadcast-groups>
      <broadcast-group name="my-broadcast-group">
        <local-bind-address>172.16.9.3</local-bind-address>
        <local-bind-port>-1</local-bind-port>
        <group-address>231.7.7.7</group-address>
        <group-port>9876</group-port>
        <broadcast-period>2000</broadcast-period>
        <connector-ref>netty-connector</connector-ref>
      </broadcast-group>
    </broadcast-groups>
    ...
  </core>
</configuration>
```

The following parameters are required unless otherwise noted:

broadcast-group

Use the **name** attribute to specify a unique name for the broadcast group.

local-bind-address

The address to which the UDP socket is bound. If you have multiple network interfaces on your broker, you should specify which one you want to use for broadcasts. If this property is not specified, the socket will be bound to an IP address chosen by the operating system. This is a UDP-specific attribute.

local-bind-port

The port to which the datagram socket is bound. In most cases, use the default value of **-1**, which specifies an anonymous port. This parameter is used in connection with **local-bind-address**. This is a UDP-specific attribute.

group-address

The multicast address to which the data will be broadcast. It is a class D IP address in the range **224.0.0.0** – **239.255.255.255** inclusive. The address **224.0.0.0** is reserved and is not available for use. This is a UDP-specific attribute.

group-port

The UDP port number used for broadcasting. This is a UDP-specific attribute.

broadcast-period (optional)

The interval in milliseconds between consecutive broadcasts. The default value is 2000 milliseconds.

connector-ref

The previously configured cluster connector that should be broadcasted.

4. Add a UDP discovery group.

The discovery group defines how this broker receives connector information from other brokers. The broker maintains a list of connectors (one entry for each broker). As it receives broadcasts

from a broker, it updates its entry. If it does not receive a broadcast from a broker for a length of time, it removes the entry.

This discovery group uses UDP to discover the brokers in the cluster:

```
<configuration>
  <core>
    ...
    <discovery-groups>
      <discovery-group name="my-discovery-group">
        <local-bind-address>172.16.9.7</local-bind-address>
        <group-address>231.7.7.7</group-address>
        <group-port>9876</group-port>
        <refresh-timeout>10000</refresh-timeout>
      </discovery-group>
    <discovery-groups>
    ...
  </core>
</configuration>
```

The following parameters are required unless otherwise noted:

discovery-group

Use the **name** attribute to specify a unique name for the discovery group.

local-bind-address (optional)

If the machine on which the broker is running uses multiple network interfaces, you can specify the network interface to which the discovery group should listen. This is a UDP-specific attribute.

group-address

The multicast address of the group on which to listen. It should match the **group-address** in the broadcast group that you want to listen from. This is a UDP-specific attribute.

group-port

The UDP port number of the multicast group. It should match the **group-port** in the broadcast group that you want to listen from. This is a UDP-specific attribute.

refresh-timeout (optional)

The amount of time in milliseconds that the discovery group waits after receiving the last broadcast from a particular broker before removing that broker's connector pair entry from its list. The default is 10000 milliseconds (10 seconds).
Set this to a much higher value than the **broadcast-period** on the broadcast group. Otherwise, brokers might periodically disappear from the list even though they are still broadcasting (due to slight differences in timing).

5. Create a cluster connection and configure it to use dynamic discovery.
By default, the cluster connection will load balance messages for all addresses in a symmetric topology.

```
<configuration>
  <core>
    ...
    <cluster-connections>
      <cluster-connection name="my-cluster">
```

```
            <connector-ref>netty-connector</connector-ref>
            <discovery-group-ref discovery-group-name="my-discovery-group"/>
        </cluster-connection>
      </cluster-connections>
      ...
    </core>
</configuration>
```

cluster-connection

> Use the **name** attribute to specify the name of the cluster connection.

connector-ref

> The connector that defines how other brokers can connect to this one.

discovery-group-ref

> The discovery group that this broker should use to locate other members of the cluster. You
> only need to configure this property if the cluster uses dynamic discovery.

6. Configure any additional properties for the cluster connection.
   These additional cluster connection properties have default values that are suitable for most
   common use cases. Therefore, you only need to configure these properties if you do not want
   the default behavior. For more information, see Appendix C, *Cluster Connection Configuration
   Elements*.

7. Create the cluster user and password.
   AMQ Broker ships with default cluster credentials, but you should change them to prevent
   unauthorized remote clients from using these default credentials to connect to the broker.

   > **IMPORTANT**
   >
   > The cluster password must be the same on every broker in the cluster.

   ```
   <configuration>
     <core>
        ...
        <cluster-user>cluster_user</cluster-user>
        <cluster-password>cluster_user_password</cluster-password>
        ...
     </core>
   </configuration>
   ```

8. Repeat this procedure on each additional broker.
   You can copy the cluster configuration to each additional broker. However, do not copy any of
   the other AMQ Broker data files (such as the bindings, journal, and large messages directories).
   These files must be unique among the nodes in the cluster or the cluster will not form properly.

Additional resources

- For an example of a broker cluster configuration that uses dynamic discovery with UDP, see the
  clustered-queue example.

14.2.3. Creating a broker cluster with JGroups–based dynamic discovery

If you are already using JGroups in your environment, you can use it to create a broker cluster in which the brokers discover each other dynamically.

Prerequisites

- JGroups must be installed and configured.
  For an example of a JGroups configuration file, see the **clustered-jgroups** example.

Procedure

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

2. Within the **<core>** element, add a connector.
   This connector defines connection information that other brokers can use to connect to this one. This information will be sent to other brokers in the cluster during discovery.

   ```xml
   <configuration>
     <core>
       ...
       <connectors>
         <connector name="netty-connector">tcp://localhost:61617</connector>
       </connectors>
       ...
     </core>
   </configuration>
   ```

3. Within the **<core>** element, add a JGroups broadcast group.
   The broadcast group enables the broker to push information about its cluster connection to the other brokers in the cluster. This broadcast group uses JGroups to broadcast the connection settings:

   ```xml
   <configuration>
     <core>
       ...
       <broadcast-groups>
         <broadcast-group name="my-broadcast-group">
           <jgroups-file>test-jgroups-file_ping.xml</jgroups-file>
           <jgroups-channel>activemq_broadcast_channel</jgroups-channel>
           <broadcast-period>2000</broadcast-period>
           <connector-ref>netty-connector</connector-ref>
         </broadcast-group>
       </broadcast-groups>
       ...
     </core>
   </configuration>
   ```

   The following parameters are required unless otherwise noted:

   broadcast-group

   Use the **name** attribute to specify a unique name for the broadcast group.

   jgroups-file

   The name of JGroups configuration file to initialize JGroups channels. The file must be in the Java resource path so that the broker can load it.

jgroups-channel

The name of the JGroups channel to connect to for broadcasting.

broadcast-period (optional)

The interval, in milliseconds, between consecutive broadcasts. The default value is 2000 milliseconds.

connector-ref

The previously configured cluster connector that should be broadcasted.

4. Add a JGroups discovery group.
   The discovery group defines how connector information is received. The broker maintains a list of connectors (one entry for each broker). As it receives broadcasts from a broker, it updates its entry. If it does not receive a broadcast from a broker for a length of time, it removes the entry.

   This discovery group uses JGroups to discover the brokers in the cluster:

   ```
   <configuration>
     <core>
        ...
        <discovery-groups>
           <discovery-group name="my-discovery-group">
              <jgroups-file>test-jgroups-file_ping.xml</jgroups-file>
              <jgroups-channel>activemq_broadcast_channel</jgroups-channel>
              <refresh-timeout>10000</refresh-timeout>
           </discovery-group>
        <discovery-groups>
        ...
     </core>
   </configuration>
   ```

   The following parameters are required unless otherwise noted:

   discovery-group

   Use the **name** attribute to specify a unique name for the discovery group.

   jgroups-file

   The name of JGroups configuration file to initialize JGroups channels. The file must be in the Java resource path so that the broker can load it.

   jgroups-channel

   The name of the JGroups channel to connect to for receiving broadcasts.

   refresh-timeout (optional)

   The amount of time in milliseconds that the discovery group waits after receiving the last broadcast from a particular broker before removing that broker's connector pair entry from its list. The default is 10000 milliseconds (10 seconds).
   Set this to a much higher value than the **broadcast-period** on the broadcast group. Otherwise, brokers might periodically disappear from the list even though they are still broadcasting (due to slight differences in timing).

5. Create a cluster connection and configure it to use dynamic discovery.
   By default, the cluster connection will load balance messages for all addresses in a symmetric topology.

```
<configuration>
  <core>
    ...
    <cluster-connections>
      <cluster-connection name="my-cluster">
        <connector-ref>netty-connector</connector-ref>
        <discovery-group-ref discovery-group-name="my-discovery-group"/>
      </cluster-connection>
    </cluster-connections>
    ...
  </core>
</configuration>
```

cluster-connection

Use the **name** attribute to specify the name of the cluster connection.

connector-ref

The connector that defines how other brokers can connect to this one.

discovery-group-ref

The discovery group that this broker should use to locate other members of the cluster. You only need to configure this property if the cluster uses dynamic discovery.

6. Configure any additional properties for the cluster connection.
   These additional cluster connection properties have default values that are suitable for most common use cases. Therefore, you only need to configure these properties if you do not want the default behavior. For more information, see Appendix C, *Cluster Connection Configuration Elements*.

7. Create the cluster user and password.
   AMQ Broker ships with default cluster credentials, but you should change them to prevent unauthorized remote clients from using these default credentials to connect to the broker.

   > **IMPORTANT**
   >
   > The cluster password must be the same on every broker in the cluster.

```
<configuration>
  <core>
    ...
    <cluster-user>cluster_user</cluster-user>
    <cluster-password>cluster_user_password</cluster-password>
    ...
  </core>
</configuration>
```

8. Repeat this procedure on each additional broker.
   You can copy the cluster configuration to each additional broker. However, do not copy any of the other AMQ Broker data files (such as the bindings, journal, and large messages directories). These files must be unique among the nodes in the cluster or the cluster will not form properly.

Additional resources

- For an example of a broker cluster that uses dynamic discovery with JGroups, see the clustered-jgroups example .

14.3. IMPLEMENTING HIGH AVAILABILITY

You can improve its reliability by implementing high availability (HA), enabling the broker cluster continue to function even if one or more brokers go offline.

Implementing HA involves several steps:

1. Configure a broker cluster for your HA implementation as described in Section 14.2, "Creating a broker cluster".

2. You should understand what live-backup groups are, and choose an HA policy that best meets your requirements. See Understanding how HA works in AMQ Broker .

3. When you have chosen a suitable HA policy, configure the HA policy on each broker in the cluster. See:

   - Configuring shared store high availability

   - Configuring replication high availability

   - Configuring limited high availability with live-only

   - Configuring high availability with colocated backups

4. Configure your client applications to use failover .

> **NOTE**
>
> In the later event that you need to troubleshoot a broker cluster configured for high availability, it is recommended that you enable Garbage Collection (GC) logging for each Java Virtual Machine (JVM) instance that is running a broker in the cluster. To learn how to enable GC logs on your JVM, consult the official documentation for the Java Development Kit (JDK) version used by your JVM. For more information on the JVM versions that AMQ Broker supports, see Red Hat AMQ 7 Supported Configurations .

14.3.1. Understanding high availability

In AMQ Broker, you implement high availability (HA) by grouping the brokers in the cluster into *live-backup groups*. In a live-backup group, a live broker is linked to a backup broker, which can take over for the live broker if it fails. AMQ Broker also provides several different strategies for failover (called *HA policies*) within a live-backup group.

14.3.1.1. How live-backup groups provide high availability

In AMQ Broker, you implement high availability (HA) by linking together the brokers in your cluster to form *live-backup groups*. Live-backup groups provide *failover*, which means that if one broker fails, another broker can take over its message processing.

A live-backup group consists of one live broker (sometimes called the *master* broker) linked to one or more backup brokers (sometimes called *slave* brokers). The live broker serves client requests, while the backup brokers wait in passive mode. If the live broker fails, a backup broker replaces the live broker, enabling the clients to reconnect and continue their work.

14.3.1.2. High availability policies

A high availability (HA) policy defines how failover happens in a live-backup group. AMQ Broker provides several different HA policies:

Shared store (recommended)

The live and backup brokers store their messaging data in a common directory on a shared file system; typically a Storage Area Network (SAN) or Network File System (NFS) server. You can also store broker data in a specified database if you have configured JDBC-based persistence. With shared store, if a live broker fails, the backup broker loads the message data from the shared store and takes over for the failed live broker.

In most cases, you should use shared store instead of replication. Because shared store does not replicate data over the network, it typically provides better performance than replication. Shared store also avoids network isolation (also called "split brain") issues in which a live broker and its backup become live at the same time.

Figure 14.4. Shared store high availability
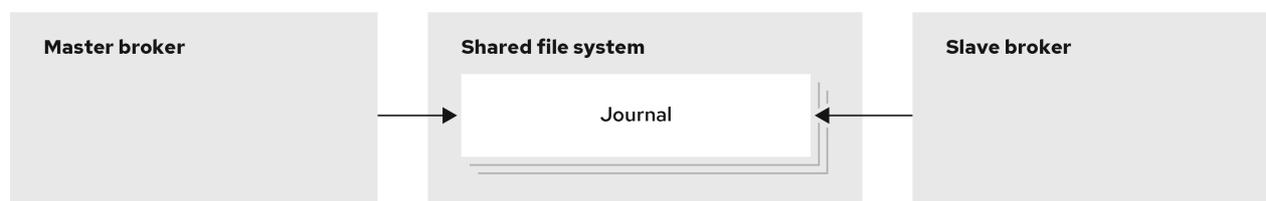
Replication

The live and backup brokers continuously synchronize their messaging data over the network. If the live broker fails, the backup broker loads the synchronized data and takes over for the failed live broker.

Data synchronization between the live and backup brokers ensures that no messaging data is lost if the live broker fails. When the live and backup brokers initially join together, the live broker replicates all of its existing data to the backup broker over the network. Once this initial phase is complete, the live broker replicates persistent data to the backup broker as the live broker receives it. This means that if the live broker drops off the network, the backup broker has all of the persistent data that the live broker has received up to that point.

Because replication synchronizes data over the network, network failures can result in network isolation in which a live broker and its backup become live at the same time.

Figure 14.5. Replication high availability
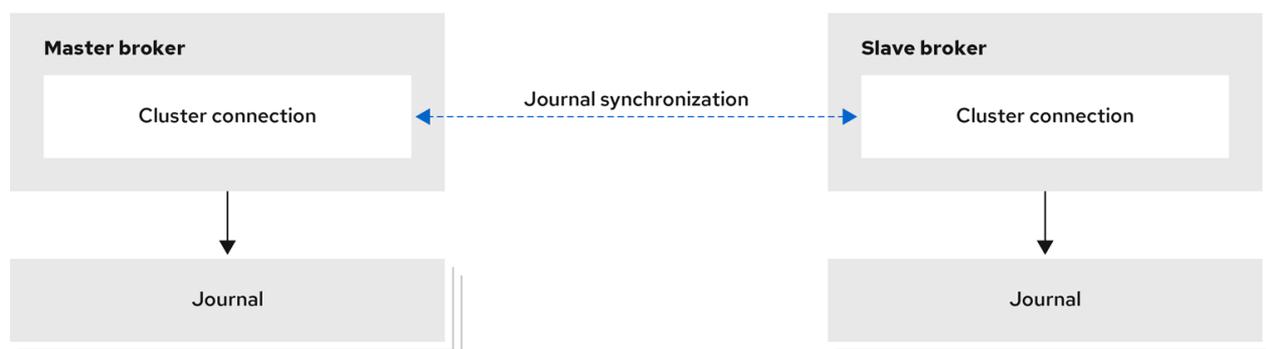
Live-only (limited HA)

When a live broker is stopped gracefully, it copies its messages and transaction state to another live broker and then shuts down. Clients can then reconnect to the other broker to continue sending and receiving messages.

Figure 14.6. Live-only high availability
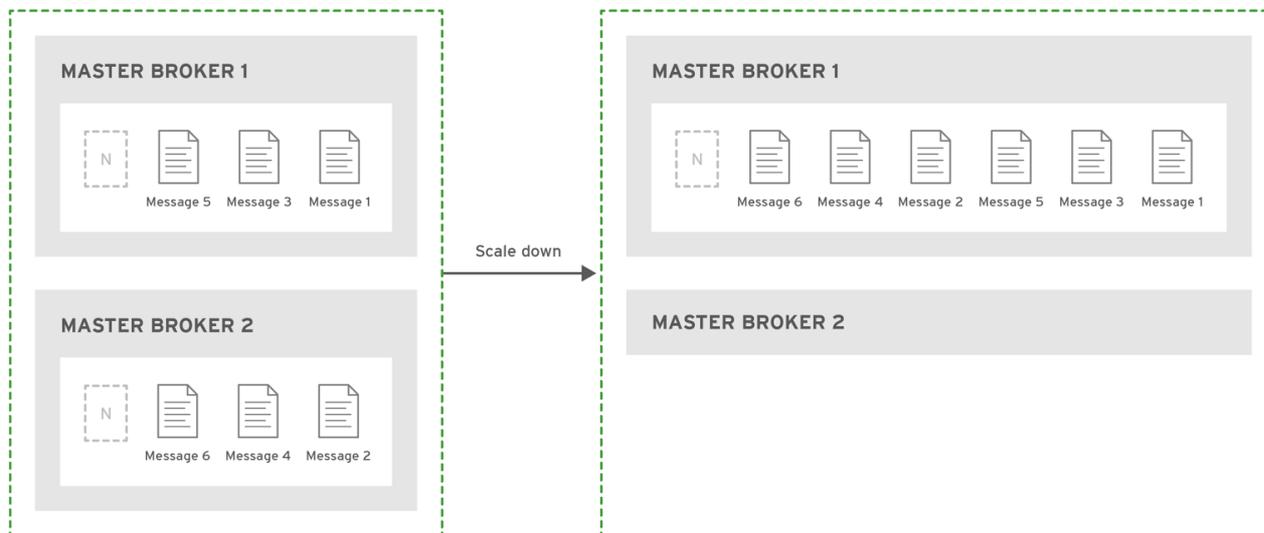
Additional resources

- For more information about the persistent message data that is shared between brokers in a live-backup group, see Section 6.1, "Persisting message data in journals" .

14.3.1.3. Replication policy limitations

When you use replication to provide high availability, a risk exists that both live and backup brokers can become live at the same time, which is referred to as "split brain".

Split brain can happen if a live broker and its backup lose their connection. In this situation, both a live broker and its backup can become active at the same time. Because there is no message replication between the brokers in this situation, they each serve clients and process messages without the other knowing it. In this case, each broker has a completely different journal. Recovering from this situation can be very difficult and in some cases, not possible.

- To **eliminate** any possibility of split brain, use the **shared store** HA policy.

- If you do use the replication HA policy, take the following steps to reduce the risk of split brain occurring.
  If you want the brokers to use the ZooKeeper Coordination Service to coordinate brokers, deploy ZooKeeper on at least three nodes. If the brokers lose connection to one ZooKeeper node, using at least three nodes ensures that a majority of nodes are available to coordinate the brokers when a live-backup broker pair experiences a replication interruption.

  If you want to use the embedded broker coordination, which uses the other available brokers in the cluster to provide a quorum vote, you can reduce (but not eliminate) the chance of encountering split brain by using **at least three live-backup pairs**. Using at least three live-backup pairs ensures that a majority result can be achieved in any quorum vote that takes place when a live-backup broker pair experiences a replication interruption.

Some additional considerations when you use the replication HA policy are described below:

- When a live broker fails and the backup transitions to live, no further replication takes place until a new backup broker is attached to the live, or failback to the original live broker occurs.

- If the backup broker in a live-backup group fails, the live broker continues to serve messages. However, messages are not replicated until another broker is added as a backup, or the original backup broker is restarted. During that time, messages are persisted only to the live broker.

- If the brokers use the embedded broker coordination and both brokers in a live-backup pair are shut down, to avoid message loss, you must restart the most recently active broker first. If the most recently active broker was the backup broker, you need to manually reconfigure this broker as a master broker to enable it to be restarted first.

14.3.2. Configuring shared store high availability

You can use the shared store high availability (HA) policy to implement HA in a broker cluster. With shared store, both live and backup brokers access a common directory on a shared file system; typically a Storage Area Network (SAN) or Network File System (NFS) server. You can also store broker data in a specified database if you have configured JDBC-based persistence. With shared store, if a live broker fails, the backup broker loads the message data from the shared store and takes over for the failed live broker.

In general, a SAN offers better performance (for example, speed) versus an NFS server, and is the recommended option, if available. If you need to use an NFS server, see Red Hat AMQ 7 Supported Configurations for more information about network file systems that AMQ Broker supports.

In most cases, you should use shared store HA instead of replication. Because shared store does not replicate data over the network, it typically provides better performance than replication. Shared store also avoids network isolation (also called "split brain") issues in which a live broker and its backup become live at the same time.

> **NOTE**
>
> When using shared store, the startup time for the backup broker depends on the size of the message journal. When the backup broker takes over for a failed live broker, it loads the journal from the shared store. This process can be time consuming if the journal contains a lot of data.

14.3.2.1. Configuring an NFS shared store

When using shared store high availability, you must configure both the live and backup brokers to use a common directory on a shared file system. Typically, you use a Storage Area Network (SAN) or Network File System (NFS) server.

Listed below are some recommended configuration options when mounting an exported directory from an NFS server on each of your broker machine instances.

sync

Specifies that all changes are immediately flushed to disk.

intr

Allows NFS requests to be interrupted if the server is shut down or cannot be reached.

noac

Disables attribute caching. This behavior is needed to achieve attribute cache coherence among multiple clients.

soft

Specifies that if the NFS server is unavailable, the error should be reported rather than waiting for the server to come back online.

lookupcache=none

Disables lookup caching.

timeo=n

The time, in deciseconds (tenths of a second), that the NFS client (that is, the broker) waits for a response from the NFS server before it retries a request. For NFS over TCP, the default **timeo** value is **600** (60 seconds). For NFS over UDP, the client uses an adaptive algorithm to estimate an appropriate timeout value for frequently used request types, such as read and write requests.

retrans=n

The number of times that the NFS client retries a request before it attempts further recovery action. If the **retrans** option is not specified, the NFS client tries each request three times.

> **IMPORTANT**
>
> It is important to use reasonable values when you configure the **timeo** and **retrans** options. A default **timeo** wait time of 600 deciseconds (60 seconds) combined with a **retrans** value of 5 retries can result in a five-minute wait for AMQ Broker to detect an NFS disconnection.

Additional resources

- To learn how to mount an exported directory from an NFS server, see Mounting an NFS share with mount in the Red Hat Enterprise Linux documentation.

- For information about network file systems supported by AMQ Broker, see Red Hat AMQ 7 Supported Configurations.

14.3.2.2. Configuring shared store high availability

This procedure shows how to configure shared store high availability for a broker cluster.

Prerequisites

- A shared storage system must be accessible to the live and backup brokers.

  - Typically, you use a Storage Area Network (SAN) or Network File System (NFS) server to provide the shared store. For more information about supported network file systems, see Red Hat AMQ 7 Supported Configurations .

  - If you have configured JDBC-based persistence, you can use your specified database to provide the shared store. To learn how to configure JDBC persistence, see Section 6.2, "Persisting message data in a database".

Procedure

1. Group the brokers in your cluster into live-backup groups.
   In most cases, a live-backup group should consist of two brokers: a live broker and a backup broker. If you have six brokers in your cluster, you would need three live-backup groups.

2. Create the first live-backup group consisting of one live broker and one backup broker.

   a. Open the live broker's ***<broker_instance_dir>*/etc/broker.xml** configuration file.

   b. If you are using:

      i. A network file system to provide the shared store, verify that the live broker's paging, bindings, journal, and large messages directories point to a shared location that the backup broker can also access.

```
<configuration>
  <core>
     ...
     <paging-directory>../sharedstore/data/paging</paging-directory>
     <bindings-directory>../sharedstore/data/bindings</bindings-directory>
     <journal-directory>../sharedstore/data/journal</journal-directory>
     <large-messages-directory>../sharedstore/data/large-messages</large-messages-directory>
     ...
  </core>
</configuration>
```

      ii. A database to provide the shared store, ensure that both the master and backup broker can connect to the same database and have the same configuration specified in the **database-store** element of the **broker.xml** configuration file. An example configuration is shown below.

```
<configuration>
  <core>
    <store>
      <database-store>
        <jdbc-connection-url>jdbc:oracle:data/oracle/database-store;create=true</jdbc-connection-url>
        <jdbc-user>ENC(5493dd76567ee5ec269d11823973462f)</jdbc-user>
        <jdbc-password>ENC(56a0db3b71043054269d11823973462f)</jdbc-password>
        <bindings-table-name>BIND_TABLE</bindings-table-name>
        <message-table-name>MSG_TABLE</message-table-name>
        <large-message-table-name>LGE_TABLE</large-message-table-name>
        <page-store-table-name>PAGE_TABLE</page-store-table-name>
        <node-manager-store-table-name>NODE_TABLE<node-manager-store-table-name>
        <jdbc-driver-class-name>oracle.jdbc.driver.OracleDriver</jdbc-driver-class-name>
        <jdbc-network-timeout>10000</jdbc-network-timeout>
        <jdbc-lock-renew-period>2000</jdbc-lock-renew-period>
        <jdbc-lock-expiration>15000</jdbc-lock-expiration>
        <jdbc-journal-sync-period>5</jdbc-journal-sync-period>
      </database-store>
    </store>
  </core>
</configuration>
```

   c. Configure the live broker to use shared store for its HA policy.

```
<configuration>
```

```
      <core>
         ...
         <ha-policy>
            <shared-store>
               <master>
                  <failover-on-shutdown>true</failover-on-shutdown>
               </master>
            </shared-store>
         </ha-policy>
         ...
      </core>
   </configuration>
```

failover-on-shutdown

If this broker is stopped normally, this property controls whether the backup broker should become live and take over.

d. Open the backup broker's ***<broker_instance_dir>*/etc/broker.xml** configuration file.

e. If you are using:

   i. A network file system to provide the shared store, verify that the backup broker's paging, bindings, journal, and large messages directories point to the same shared location as the live broker.

```
<configuration>
   <core>
      ...
      <paging-directory>../sharedstore/data/paging</paging-directory>
      <bindings-directory>../sharedstore/data/bindings</bindings-directory>
      <journal-directory>../sharedstore/data/journal</journal-directory>
      <large-messages-directory>../sharedstore/data/large-messages</large-messages-directory>
      ...
   </core>
</configuration>
```

   ii. A database to provide the shared store, ensure that both the master and backup brokers can connect to the same database and have the same configuration specified in the **database-store** element of the **broker.xml** configuration file.

f. Configure the backup broker to use shared store for its HA policy.

```
<configuration>
   <core>
      ...
      <ha-policy>
         <shared-store>
            <slave>
               <failover-on-shutdown>true</failover-on-shutdown>
               <allow-failback>true</allow-failback>
               <restart-backup>true</restart-backup>
            </slave>
         </shared-store>
      </ha-policy>
```

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

failover-on-shutdown

> If this broker has become live and then is stopped normally, this property controls whether the backup broker (the original live broker) should become live and take over.

allow-failback

> If failover has occurred and the backup broker has taken over for the live broker, this property controls whether the backup broker should fail back to the original live broker when it restarts and reconnects to the cluster.

> NOTE
>
> Failback is intended for a live-backup pair (one live broker paired with a single backup broker). If the live broker is configured with multiple backups, then failback will not occur. Instead, if a failover event occurs, the backup broker will become live, and the next backup will become its backup. When the original live broker comes back online, it will not be able to initiate failback, because the broker that is now live already has a backup.

restart-backup

> This property controls whether the backup broker automatically restarts after it fails back to the live broker. The default value of this property is **true**.

3. Repeat Step 2 for each remaining live-backup group in the cluster.

14.3.3. Configuring replication high availability

You can use the replication high availability (HA) policy to implement HA in a broker cluster. With replication, persistent data is synchronized between the live and backup brokers. If a live broker encounters a failure, message data is synchronized to the backup broker and it takes over for the failed live broker.

You should use replication as an alternative to shared store, if you do not have a shared file system. However, replication can result in a scenario in which a live broker and its backup become live at the same time.

> NOTE
>
> Because the live and backup brokers must synchronize their messaging data over the network, replication adds a performance overhead. This synchronization process blocks journal operations, but it does not block clients. You can configure the maximum amount of time that journal operations can be blocked for data synchronization.

If the replication connection between the live-backup broker pair is interrupted, the brokers require a way to coordinate to determine if the live broker is still active or if it is unavailable and a failover to the backup broker is required. To provide this coordination, you can configure the brokers to use either of the following coordination methods.

- The Apache ZooKeeper coordination service.

- The embedded broker coordination, which uses other brokers in the cluster to provide a quorum vote.

14.3.3.1. Choosing a coordination method

Red Hat recommends that you use the Apache ZooKeeper coordination service to coordinate broker activation. When choosing a coordination method, it is useful to understand the differences in infrastructure requirements and the management of data consistency between both coordination methods.

Infrastructure requirements

- If you use the ZooKeeper coordination service, you can operate with a single live-backup broker pair. However, you must connect the brokers to at least 3 Apache ZooKeeper nodes to ensure that brokers can continue to function if they lose connection to one node. To provide a coordination service to brokers, you can share existing ZooKeeper nodes that are used by other applications. For more information on setting up Apache ZooKeeper, see the Apache ZooKeeper documentation.

- If you want to use the embedded broker coordination, which uses the other available brokers in the cluster to provide a quorum vote, you must have at least three live-backup broker pairs. Using at least three live-backup pairs ensures that a majority result can be achieved in any quorum vote that occurs when a live-backup broker pair experiences a replication interruption.

Data consistency

- If you use the Apache ZooKeeper coordination service, ZooKeeper tracks the version of the data on each broker so only the broker that has the most up-to-date journal data can activate as the live broker, irrespective of whether the broker is configured as a primary or backup broker for replication purposes. Version tracking eliminates the possibility that a broker can activate with an out-of-date journal and start serving clients.

- If you use the embedded broker coordination, no mechanism exists to track the version of the data on each broker to ensure that only the broker that has the most up-to-date journal can become the live broker. Therefore, it is possible for a broker that has an out-of-date journal to become live and start serving clients, which causes a divergence in the journal.

14.3.3.2. How brokers coordinate after a replication interruption

This section explains how both coordination methods work after a replication connection is interrupted.

Using the ZooKeeper coordination service

If you use the ZooKeeper coordination service to manage replication interruptions, both brokers must be connected to multiple Apache ZooKeeper nodes.

- If, at any time, the live broker loses connection to a majority of the ZooKeeper nodes, it shuts down to avoid the risk of "split brain" occurring.

- If, at any time, the backup broker loses connection to a majority of the ZooKeeper nodes, it stops receiving replication data and waits until it can connect to a majority of the ZooKeeper nodes before it acts as a backup broker again. When the connection is restored to a majority of the ZooKeeper nodes, the backup broker uses ZooKeeper to determine if it needs to discard its data and search for a live broker from which to replicate, or if it can become the live broker with its current data.

ZooKeeper uses the following control mechanisms to manage the failover process:

- A shared lease lock that can be owned only by a single live broker at any time.

- An activation sequence counter that tracks the latest version of the broker data. Each broker tracks the version of its journal data in a local counter stored in its server lock file, along with its NodeID. The live broker also shares its version in a coordinated activation sequence counter on ZooKeeper.

If the replication connection between the live broker and the backup broker is lost, the live broker increases both its local activation sequence counter value and the coordinated activation sequence counter value on ZooKeeper by **1** to advertise that it has the most up-to-date data. The backup broker's data is now considered stale and the broker cannot become the live broker until the replication connection is restored and the up-to-date data is synchronized.

After the replication connection is lost, the backup broker checks if the ZooKeeper lock is owned by the live broker and if the coordinated activation sequence counter on ZooKeeper matches its local counter value.

- If the lock is owned by the live broker, the backup broker detects that the activation sequence counter on ZooKeeper was updated by the live broker when the replication connection was lost. This indicates that the live broker is running so the backup broker does not try to failover.

- If the lock is not owned by the live broker, the live broker is not alive. If the value of the activation sequence counter on the backup broker is the same as the coordinated activation sequence counter value on ZooKeeper, which indicates that the backup broker has up-to-date data, the backup broker fails over.

- If the lock is not owned by the live broker but the value of the activation sequence counter on the backup broker is less than the counter value on ZooKeeper, the data on the backup broker is not up-to-date and the backup broker cannot fail over.

Using the embedded broker coordination

If a live-backup broker pair use the embedded broker coordination to coordinate a replication interruption, the following two types of quorum votes can be initiated.

Table 14.1. Quorum voting

Vote type	Description	Initiator	Required configuration	Participants	Action based on vote result

Vote type	Description	Initiator	Required configuration	Participants	Action based on vote result
Backup vote	If a backup broker loses its replication connection to the live broker, the backup broker decides whether or not to start based on the result of this vote.	Backup broker	None. A backup vote happens automatically when a backup broker loses connection to its replication partner. However, you can control the properties of a backup vote by specifying custom values for these parameters: • **quorum-vote-wait** • **vote-retries** • **vote-retry-wait**	Other live brokers in the cluster	The backup broker starts if it receives a majority (that is, a *quorum*) vote from the other live brokers in the cluster, indicating that its replication partner is no longer available.
Live vote	If a live broker loses connection to its replication partner, the live broker decides whether to continue running based on this vote.	Live broker	A live vote happens when a live broker loses connection to its replication partner and **vote-on-replication-failure** is set to **true**. A backup broker that has become active is considered a live broker, and can initiate a live vote.	Other live brokers in the cluster	The live broker shuts down if it **doesn't** receive a majority vote from the other live brokers in the cluster, indicating that its cluster connection is still active.

IMPORTANT

Listed below are some important things to note about how the configuration of your broker cluster affects the behavior of quorum voting.

- For a quorum vote to succeed, the size of your cluster must allow a majority result to be achieved. Therefore, your cluster should have **at least three** live-backup broker pairs.

- The more live-backup broker pairs that you add to your cluster, the more you increase the overall fault tolerance of the cluster. For example, suppose you have three live-backup pairs. If you lose a complete live-backup pair, the two remaining live-backup pairs cannot achieve a majority result in any subsequent quorum vote. This situation means that any further replication interruption in the cluster might cause a live broker to shut down, and prevent its backup broker from starting up. By configuring your cluster with, say, five broker pairs, the cluster can experience at least two failures, while still ensuring a majority result from any quorum vote.

- If you intentionally **reduce** the number of live-backup broker pairs in your cluster, the previously established threshold for a majority vote does not automatically decrease. During this time, any quorum vote triggered by a lost replication connection cannot succeed, making your cluster more vulnerable to split brain. To make your cluster recalculate the majority threshold for a quorum vote, first shut down the live-backup pairs that you are removing from your cluster. Then, restart the remaining live-backup pairs in the cluster. When all of the remaining brokers have been restarted, the cluster recalculates the quorum vote threshold.

14.3.3.3. Configuring replication for a broker cluster using the ZooKeeper coordination service

You must specify the same replication configuration for both brokers in a live-backup pair that uses the Apache ZooKeeper coordination service. The brokers then coordinate to determine which broker is the primary broker and which is the backup broker.

Prerequisites

- At least 3 Apache ZooKeeper nodes to ensure that brokers can continue to operate if they lose the connection to one node.

- The broker machines have a similar hardware specification, that is, you do not have a preference for which machine runs the live broker and which runs the backup broker at any point in time.

- ZooKeeper must have sufficient resources to ensure that pause times are significantly less than the ZooKeeper server tick time. Depending on the expected load of the broker, consider carefully if the broker and ZooKeeper node can share the same node. For more information, see https://zookeeper.apache.org/.

Procedure

1. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file for both brokers in the live-backup pair.

2. Configure the same replication configuration for both brokers in the pair. For example:

   ```
   <configuration>
   ```

```
<core>
    ...
    <ha-policy>
      <replication>
        <primary>
          <coordination-id>production-001</coordination-id>
          <manager>
            <properties>
              <property key="connect-string"
value="192.168.1.10:6666,192.168.2.10:6667,192.168.3.10:6668"/>
            </properties>
          </manager>
        </primary>
      </replication>
    </ha-policy>
    ...
  </core>
</configuration>
```

primary

Configure the replication type as primary to indicate that either broker can be the primary broker depending on the result of the broker coordination.

Coordination-id

Specify a common string value for both brokers in the live-backup pair. Brokers with the same **Coordination-id** string coordinate activation together. During the coordination process, both brokers use the **Coordination-id** string as the node Id and attempt to obtain a lock in ZooKeeper. The first broker that obtains a lock and has up-to-date data starts as a live broker and the other broker becomes the backup.

properties

Specify a **property** element within which you can specify a set of key-value pairs to provide the connection details for the ZooKeeper nodes:

Table 14.2. ZooKeeper connection details

Key	Value
connect-string	Specify a comma-separated list of the IP addresses and port numbers of the ZooKeeper nodes. For example, **value="192.168.1.10:6666,192.168.2.10:6667,192.168.3.10:6668"**.

Key	Value
session-ms	The duration that the broker waits before it shuts down after losing connection to a majority of the ZooKeeper nodes. The default value is **18000** ms. A valid value is between 2 times and 20 times the ZooKeeper server tick time.
	NOTE The ZooKeeper pause time for garbage collection must be less than 0.33 of the value of the **session-ms** property in order to allow the ZooKeeper heartbeat to function reliably. If it is not possible to ensure that pause times are less than this limit, increase the value of the **session-ms** property for each broker and accept a slower failover.
	IMPORTANT Broker replication partners automatically exchange "ping" packets every 2 seconds to confirm that the partner broker is available. When a backup broker does not receive a response from the live broker, the backup waits for a response until the broker's connection time-to-live (ttl) expires. The default connection-ttl is **60000** ms which means that a backup broker attempts to fail over after 60 seconds. It is recommended that you set the connection-ttl value to a similar value to the **session-ms** property value to allow a faster failover. To set a new connection-ttl, configure the **connection-ttl-override** property.
namespace (optional)	If the brokers share the ZooKeeper nodes with other applications, you can create a ZooKeeper namespace to store the files that provide a coordination service to brokers. You must specify the same namespace for both brokers in a live-backup pair.

3. Configure any additional HA properties for the brokers.
   These additional HA properties have default values that are suitable for most common use cases. Therefore, you only need to configure these properties if you do not want the default behavior. For more information, see Appendix F, *Additional Replication High Availability Configuration Elements*.

4. Repeat steps 1 to 3 to configure each additional live-backup broker pair in the cluster.

Additional resources

- For examples of broker clusters that use replication for HA, see the HA examples.

- For more information about node IDs, see Understanding node IDs .

14.3.3.4. Configuring a broker cluster for replication high availability using the embedded broker coordination

Replication using the embedded broker coordination requires at least three live-backup pairs to lessen (but not eliminate) the risk of "split brain".

The following procedure describes how to configure replication high-availability (HA) for a six-broker cluster. In this topology, the six brokers are grouped into three live-backup pairs: each of the three live brokers is paired with a dedicated backup broker.

Prerequisites

- You must have a broker cluster with at least six brokers.
  The six brokers are configured into three live-backup pairs. For more information about adding brokers to a cluster, see Chapter 14, *Setting up a broker cluster* .

Procedure

1. Group the brokers in your cluster into live-backup groups.
   In most cases, a live-backup group should consist of two brokers: a live broker and a backup broker. If you have six brokers in your cluster, you need three live-backup groups.

2. Create the first live-backup group consisting of one live broker and one backup broker.

   a. Open the live broker's **<broker_instance_dir>/etc/broker.xml** configuration file.

   b. Configure the live broker to use replication for its HA policy.

   ```
   <configuration>
     <core>
        ...
        <ha-policy>
           <replication>
              <master>
                 <check-for-live-server>true</check-for-live-server>
                 <group-name>my-group-1</group-name>
                 <vote-on-replication-failure>true</vote-on-replication-failure>
                 ...
              </master>
           </replication>
        </ha-policy>
        ...
     </core>
   </configuration>
   ```

   check-for-live-server

   > If the live broker fails, this property controls whether clients should fail back to it when it restarts.
   >
   > If you set this property to **true**, when the live broker restarts after a previous failover, it searches for another broker in the cluster with the same node ID. If the live broker finds another broker with the same node ID, this indicates that a backup broker successfully started upon failure of the live broker. In this case, the live broker synchronizes its data with the backup broker. The live broker then requests the backup broker to shut down. If the backup broker is configured for failback, as shown below, it shuts down. The live broker then resumes its active role, and clients reconnect to it.

> **WARNING**
>
> If you do not set **check-for-live-server** to **true** on the live broker, you might experience duplicate messaging handling when you restart the live broker after a previous failover. Specifically, if you restart a live broker with this property set to **false**, the live broker does not synchronize data with its backup broker. In this case, the live broker might process the same messages that the backup broker has already handled, causing duplicates.

group-name

A name for this live-backup group (optional). To form a live-backup group, the live and backup brokers must be configured with the same group name. If you don't specify a group-name, a backup broker can replicate with any live broker.

vote-on-replication-failure

This property controls whether a live broker initiates a quorum vote called a *live vote* in the event of an interrupted replication connection.
A live vote is a way for a live broker to determine whether it or its partner is the cause of the interrupted replication connection. Based on the result of the vote, the live broker either stays running or shuts down.

> **IMPORTANT**
>
> For a quorum vote to succeed, the size of your cluster must allow a majority result to be achieved. Therefore, when you use the replication HA policy, your cluster should have **at least three** live-backup broker pairs.
>
> The more broker pairs you configure in your cluster, the more you increase the overall fault tolerance of the cluster. For example, suppose you have three live-backup broker pairs. If you lose connection to a complete live-backup pair, the two remaining live-backup pairs can no longer achieve a majority result in a quorum vote. This situation means that any subsequent replication interruption might cause a live broker to shut down, and prevent its backup broker from starting up. By configuring your cluster with, say, five broker pairs, the cluster can experience at least two failures, while still ensuring a majority result from any quorum vote.

c. Configure any additional HA properties for the live broker.
   These additional HA properties have default values that are suitable for most common use cases. Therefore, you only need to configure these properties if you do not want the default behavior. For more information, see Appendix F, *Additional Replication High Availability Configuration Elements*.

d. Open the backup broker's ***<broker_instance_dir>*/etc/broker.xml** configuration file.

e. Configure the backup broker to use replication for its HA policy.

```
  <core>
    ...
    <ha-policy>
      <replication>
        <slave>
          <allow-failback>true</allow-failback>
          <group-name>my-group-1</group-name>
          <vote-on-replication-failure>true</vote-on-replication-failure>
          ...
        </slave>
      </replication>
    </ha-policy>
    ...
  </core>
</configuration>
```

allow-failback

If failover has occurred and the backup broker has taken over for the live broker, this property controls whether the backup broker should fail back to the original live broker when it restarts and reconnects to the cluster.

> **NOTE**
>
> Failback is intended for a live-backup pair (one live broker paired with a single backup broker). If the live broker is configured with multiple backups, then failback will not occur. Instead, if a failover event occurs, the backup broker will become live, and the next backup will become its backup. When the original live broker comes back online, it will not be able to initiate failback, because the broker that is now live already has a backup.

group-name

A name for this live-backup group (optional). To form a live-backup group, the live and backup brokers must be configured with the same group name. If you don't specify a group-name, a backup broker can replicate with any live broker.

vote-on-replication-failure

This property controls whether a live broker initiates a quorum vote called a *live vote* in the event of an interrupted replication connection. A backup broker that has become active is considered a live broker and can initiate a live vote.
A live vote is a way for a live broker to determine whether it or its partner is the cause of the interrupted replication connection. Based on the result of the vote, the live broker either stays running or shuts down.

f. (Optional) Configure properties of the quorum votes that the backup broker initiates.

```
<configuration>
  <core>
    ...
    <ha-policy>
      <replication>
        <slave>
          ...
```

```
                <vote-retries>12</vote-retries>
                <vote-retry-wait>5000</vote-retry-wait>
            ...
          </slave>
       </replication>
    </ha-policy>
    ...
  </core>
</configuration>
```

vote-retries

> This property controls how many times the backup broker retries the quorum vote in order to receive a majority result that allows the backup broker to start up.

vote-retry-wait

> This property controls how long, in milliseconds, that the backup broker waits between each retry of the quorum vote.

g. Configure any additional HA properties for the backup broker.
   These additional HA properties have default values that are suitable for most common use cases. Therefore, you only need to configure these properties if you do not want the default behavior. For more information, see Appendix F, *Additional Replication High Availability Configuration Elements*.

3. Repeat step 2 for each additional live-backup group in the cluster.
   If there are six brokers in the cluster, repeat this procedure two more times; once for each remaining live-backup group.

Additional resources

- For examples of broker clusters that use replication for HA, see the HA examples.

- For more information about node IDs, see Understanding node IDs .

14.3.4. Configuring limited high availability with live-only

The live-only HA policy enables you to shut down a broker in a cluster without losing any messages. With live-only, when a live broker is stopped gracefully, it copies its messages and transaction state to another live broker and then shuts down. Clients can then reconnect to the other broker to continue sending and receiving messages.

The live-only HA policy only handles cases when the broker is stopped gracefully. It does not handle unexpected broker failures.

While live-only HA prevents message loss, it may not preserve message order. If a broker configured with live-only HA is stopped, its messages will be appended to the ends of the queues of another broker.
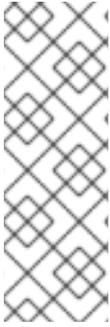
> **NOTE**
>
> When a broker is preparing to scale down, it sends a message to its clients before they are disconnected informing them which new broker is ready to process their messages. However, clients should reconnect to the new broker only after their initial broker has finished scaling down. This ensures that any state, such as queues or transactions, is available on the other broker when the client reconnects. The normal reconnect settings apply when the client is reconnecting, so you should set these high enough to deal with the time needed to scale down.

This procedure describes how to configure each broker in the cluster to scale down. After completing this procedure, whenever a broker is stopped gracefully, it will copy its messages and transaction state to another broker in the cluster.

Procedure

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

2. Configure the broker to use the live-only HA policy.

   ```
   <configuration>
     <core>
       ...
         <ha-policy>
           <live-only>
           </live-only>
         </ha-policy>
       ...
     </core>
   </configuration>
   ```

3. Configure a method for scaling down the broker cluster.
   Specify the broker or group of brokers to which this broker should scale down.

   Table 14.3. Methods for scaling down a broker cluster

To scale down to...	Do this...
A specific broker in the cluster	Specify the connector of the broker to which you want to scale down. ``` <live-only> <scale-down> <connectors> <connector-ref>broker1-connector</connector-ref> </connectors> </scale-down> </live-only> ```

To scale down to...	Do this...

Any broker in the cluster	Specify the broker cluster's discovery group.
	
	`<live-only>`
	`  <scale-down>`
	`    <discovery-group-ref discovery-group-name="my-discovery-group"/>`
	`  </scale-down>`
	`</live-only>`

A broker in a particular broker group	Specify a broker group.
	
	`<live-only>`
	`  <scale-down>`
	`    <group-name>my-group-name</group-name>`
	`  </scale-down>`
	`</live-only>`

4. Repeat this procedure for each remaining broker in the cluster.

Additional resources

- For an example of a broker cluster that uses live-only to scale down the cluster, see the **scale-down** example.

14.3.5. Configuring high availability with colocated backups

Rather than configure live-backup groups, you can colocate backup brokers in the same JVM as another live broker. In this configuration, each live broker is configured to request another live broker to create and start a backup broker in its JVM.

Figure 14.7. Colocated live and backup brokers
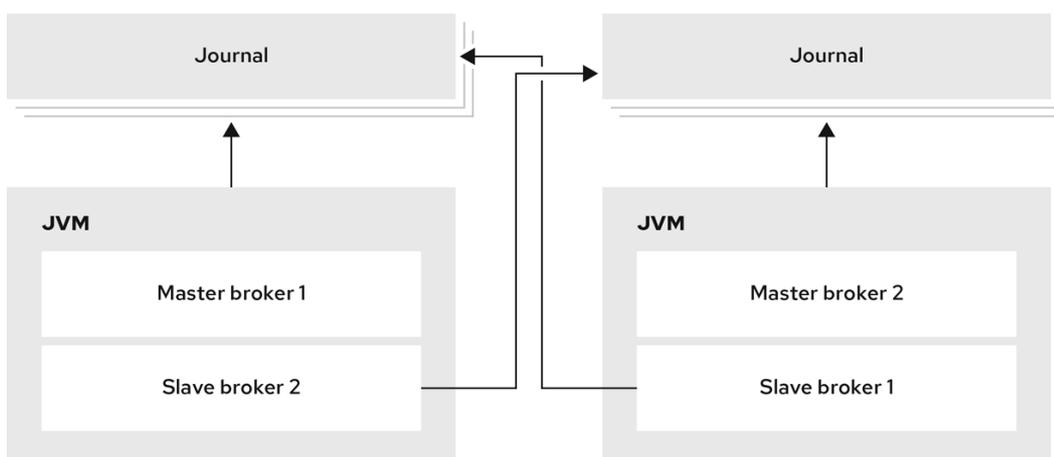
You can use colocation with either shared store or replication as the high availability (HA) policy. The new backup broker inherits its configuration from the live broker that creates it. The name of the backup is set to **colocated_backup_n** where **n** is the number of backups the live broker has created.

In addition, the backup broker inherits the configuration for its connectors and acceptors from the live broker that creates it. By default, port offset of 100 is applied to each. For example, if the live broker has an acceptor for port 61616, the first backup broker created will use port 61716, the second backup will use 61816, and so on.

Directories for the journal, large messages, and paging are set according to the HA policy you choose. If you choose shared store, the requesting broker notifies the target broker which directories to use. If replication is chosen, directories are inherited from the creating broker and have the new backup's name appended to them.

This procedure configures each broker in the cluster to use shared store HA, and to request a backup to be created and colocated with another broker in the cluster.

Procedure

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

2. Configure the broker to use an HA policy and colocation.
   In this example, the broker is configured with shared store HA and colocation.

```
<configuration>
  <core>
    ...
    <ha-policy>
      <shared-store>
        <colocated>
          <request-backup>true</request-backup>
          <max-backups>1</max-backups>
          <backup-request-retries>-1</backup-request-retries>
          <backup-request-retry-interval>5000</backup-request-retry-interval/>
          <backup-port-offset>150</backup-port-offset>
          <excludes>
            <connector-ref>remote-connector</connector-ref>
          </excludes>
          <master>
            <failover-on-shutdown>true</failover-on-shutdown>
          </master>
          <slave>
            <failover-on-shutdown>true</failover-on-shutdown>
            <allow-failback>true</allow-failback>
            <restart-backup>true</restart-backup>
          </slave>
        </colocated>
      </shared-store>
    </ha-policy>
    ...
  </core>
</configuration>
```

request-backup

By setting this property to **true**, this broker will request a backup broker to be created by another live broker in the cluster.

max-backups

The number of backup brokers that this broker can create. If you set this property to **0**, this broker will not accept backup requests from other brokers in the cluster.

backup-request-retries

The number of times this broker should try to request a backup broker to be created. The default is **-1**, which means unlimited tries.

backup-request-retry-interval

The amount of time in milliseconds that the broker should wait before retrying a request to create a backup broker. The default is **5000**, or 5 seconds.

backup-port-offset

The port offset to use for the acceptors and connectors for a new backup broker. If this broker receives a request to create a backup for another broker in the cluster, it will create the backup broker with the ports offset by this amount. The default is **100**.

excludes (optional)

Excludes connectors from the backup port offset. If you have configured any connectors for external brokers that should be excluded from the backup port offset, add a **<connector-ref>** for each of the connectors.

master

The shared store or replication failover configuration for this broker.

slave

The shared store or replication failover configuration for this broker's backup.

3. Repeat this procedure for each remaining broker in the cluster.

Additional resources

- For examples of broker clusters that use colocated backups, see the HA examples.

14.3.6. Configuring clients to fail over

After configuring high availability in a broker cluster, you configure your clients to fail over. Client failover ensures that if a broker fails, the clients connected to it can reconnect to another broker in the cluster with minimal downtime.

> **NOTE**
>
> In the event of transient network problems, AMQ Broker automatically reattaches connections to the same broker. This is similar to failover, except that the client reconnects to the same broker.

You can configure two different types of client failover:

Automatic client failover

The client receives information about the broker cluster when it first connects. If the broker to which it is connected fails, the client automatically reconnects to the broker's backup, and the backup broker re-creates any sessions and consumers that existed on each connection before failover.

Application-level client failover

As an alternative to automatic client failover, you can instead code your client applications with your own custom reconnection logic in a failure handler.

Procedure

- Use AMQ Core Protocol JMS to configure your client application with automatic or application-level failover.
  For more information, see Using the AMQ Core Protocol JMS Client .

14.4. ENABLING MESSAGE REDISTRIBUTION

If your broker cluster is configured with **message-load-balancing** set to **ON_DEMAND** or **OFF_WITH_REDISTRIBUTION**, you can configure *message redistribution* to prevent messages from being "stuck" in a queue that does not have a consumer to consume the messages.

This section contains information about:

- Understanding message distribution

- Configuring message redistribution

14.4.1. Understanding message redistribution

Broker clusters use load balancing to distribute the message load across the cluster. When configuring load balancing in the cluster connection, you can enable redistribution using the following **message-load-balancing** settings:

- **ON_DEMAND** - enable load balancing and allow redistribution

- **OFF_WITH_REDISTRIBUTION** - disable load balancing but allow redistribution

In both cases, the broker forwards messages only to other brokers that have matching consumers. This behavior ensures that messages are not moved to queues that do not have any consumers to consume the messages. However, if the consumers attached to a queue close after the messages are forwarded to the broker, those messages become "stuck" in the queue and are not consumed. This issue is sometimes called *starvation*.

Message redistribution prevents starvation by automatically redistributing the messages from queues that have no consumers to brokers in the cluster that do have matching consumers.

With **OFF_WITH_REDISTRIBUTION**, the broker only forwards messages to other brokers that have matching consumers if there are no active local consumers, enabling you to prioritize a broker while providing an alternative when consumers are not available.

Message redistribution supports the use of filters (also know as *selectors*), that is, messages are redistributed when they do not match the selectors of the available local consumers.

Additional resources

- For more information about cluster load balancing, see Section 14.1.1, "How broker clusters balance message load".

14.4.2. Configuring message redistribution

This procedure shows how to configure message redistribution with load balancing. If you want message redistribution without load balancing, set **<message-load-balancing>** is set to **OFF_WITH_REDISTRIBUTION**.

Procedure

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

2. In the **<cluster-connection>** element, verify that **<message-load-balancing>** is set to **ON_DEMAND**.

```
<configuration>
  <core>
     ...
     <cluster-connections>
        <cluster-connection name="my-cluster">
           ...
           <message-load-balancing>ON_DEMAND</message-load-balancing>
           ...
        </cluster-connection>
     </cluster-connections>
  </core>
</configuration>
```

3. Within the **<address-settings>** element, set the redistribution delay for a queue or set of queues.
   In this example, messages load balanced to **my.queue** will be redistributed 5000 milliseconds after the last consumer closes.

```
<configuration>
  <core>
     ...
     <address-settings>
        <address-setting match="my.queue">
           <redistribution-delay>5000</redistribution-delay>
        </address-setting>
     </address-settings>
     ...
  </core>
</configuration>
```

address-setting

Set the **match** attribute to be the name of the queue for which you want messages to be redistributed. You can use the broker wildcard syntax to specify a range of queues. For more information, see Section 4.2, "Applying address settings to sets of addresses" .

redistribution-delay

The amount of time (in milliseconds) that the broker should wait after this queue's final consumer closes before redistributing messages to other brokers in the cluster. If you set this to **0**, messages will be redistributed immediately. However, you should typically set a delay before redistributing – it is common for a consumer to close but another one to be quickly created on the same queue.

4. Repeat this procedure for each additional broker in the cluster.

Additional resources

- For an example of a broker cluster configuration that redistributes messages, see the **queue-message-redistribution** example.

14.5. CONFIGURING CLUSTERED MESSAGE GROUPING

Message grouping enables clients to send groups of messages of a particular type to be processed serially by the same consumer. By adding a grouping handler to each broker in the cluster, you ensure that clients can send grouped messages to any broker in the cluster and still have those messages consumed in the correct order by the same consumer.

> **NOTE**
>
> Grouping and clustering techniques can be summarized as follows:
>
> - Message grouping imposes an order on message consumption. In a group, each message must be fully consumed and acknowledged prior to proceeding with the next message. This methodology leads to serial message processing, where concurrency is not an option.
>
> - Clustering aims to horizontally scale brokers to boost message throughput. Horizontal scaling is achieved by adding additional consumers that can process messages concurrently.
>
> Because these techniques contradict each other, avoid using clustering and grouping together.

There are two types of grouping handlers: *local handlers* and *remote handlers*. They enable the broker cluster to route all of the messages in a particular group to the appropriate queue so that the intended consumer can consume them in the correct order.

Prerequisites

- There should be at least one consumer on each broker in the cluster.
  When a message is pinned to a consumer on a queue, all messages with the same group ID will be routed to that queue. If the consumer is removed, the queue will continue to receive the messages even if there are no consumers.

Procedure

1. Configure a local handler on one broker in the cluster.
   If you are using high availability, this should be a master broker.

   a. Open the broker's **<broker_instance_dir>/etc/broker.xml** configuration file.

   b. Within the **<core>** element, add a local handler:
      The local handler serves as an arbiter for the remote handlers. It stores route information and communicates it to the other brokers.

      ```
      <configuration>
        <core>
          ...
          <grouping-handler name="my-grouping-handler">
      ```

```
          <type>LOCAL</type>
          <timeout>10000</timeout>
      </grouping-handler>
      ...
    </core>
</configuration>
```

grouping-handler

> Use the **name** attribute to specify a unique name for the grouping handler.

type

> Set this to **LOCAL**.

timeout

> The amount of time to wait (in milliseconds) for a decision to be made about where to route the message. The default is 5000 milliseconds. If the timeout is reached before a routing decision is made, an exception is thrown, which ensures strict message ordering. When the broker receives a message with a group ID, it proposes a route to a queue to which the consumer is attached. If the route is accepted by the grouping handlers on the other brokers in the cluster, then the route is established: all brokers in the cluster will forward messages with this group ID to that queue. If the broker's route proposal is rejected, then it proposes an alternate route, repeating the process until a route is accepted.

2. If you are using high availability, copy the local handler configuration to the master broker's slave broker.
   Copying the local handler configuration to the slave broker prevents a single point of failure for the local handler.

3. On each remaining broker in the cluster, configure a remote handler.

   a. Open the broker's **<broker_instance_dir>/etc/broker.xml** configuration file.

   b. Within the **<core>** element, add a remote handler:

   ```
   <configuration>
     <core>
       ...
       <grouping-handler name="my-grouping-handler">
         <type>REMOTE</type>
         <timeout>5000</timeout>
       </grouping-handler>
       ...
     </core>
   </configuration>
   ```

grouping-handler

> Use the **name** attribute to specify a unique name for the grouping handler.

type

> Set this to **REMOTE**.

timeout

> The amount of time to wait (in milliseconds) for a decision to be made about where to route the message. The default is 5000 milliseconds. Set this value to at least half of the value of the local handler.

Additional resources

- For an example of a broker cluster configured for message grouping, see the JMS clustered grouping example.

14.6. CONNECTING CLIENTS TO A BROKER CLUSTER

You can use the Red Hat build of Apache Qpid JMS clients to connect to the cluster. By using JMS, you can configure your messaging clients to discover the list of brokers dynamically or statically. You can also configure client-side load balancing to distribute the client sessions created from the connection across the cluster.

Procedure

- Use AMQ Core Protocol JMS to configure your client application to connect to the broker cluster.
  For more information, see Using the AMQ Core Protocol JMS Client .

14.7. PARTITIONING CLIENT CONNECTIONS

Partitioning client connections involves routing connections for individual clients to the same broker each time the client initiates a connection.

Two use cases for partitioning client connections are:

- Partitioning clients of durable subscriptions to ensure that a subscriber always connects to the broker where the durable subscriber queue is located.

- Minimizing the need to move data by attracting clients to data where it originates, also known as data gravity.

Durable subscriptions

A durable subscription is represented as a queue on a broker and is created when a durable subscriber first connects to the broker. This queue remains on the broker and receives messages until the client unsubscribes. Therefore, you want the client to connect to the same broker repeatedly to consume the messages that are in the subscriber queue.

To partition clients for durable subscription queues, you can filter the client ID in client connections.

Data gravity

If you scale up the number of brokers in your environment without considering data gravity, some of the performance benefits are lost because of the need to move messages between brokers. To support date gravity, you should partition your client connections so that client consumers connect to the broker on which the messages that they need to consume are produced.

To partition client connections to support data gravity, you can filter any of the following attributes of a client connection:

- a role assigned to the connecting user (ROLE_NAME)

- the username of the user (USER_NAME)

- the hostname of the client (SNI_HOST)

- the IP address of the client (SOURCE_IP)

14.7.1. Partitioning client connections to support durable subscriptions

To partition clients for durable subscriptions, you can filter client IDs in incoming connections by using a consistent hashing algorithm or a regular expression.

Prerequisites

Clients are configured so they can connect to all of the brokers in the cluster, for example, by using a load balancer or by having all of the broker instances configured in the connection URL. If a broker rejects a connection because the client details do not match the partition configuration for that broker, the client must be able to connect to the other brokers in the cluster to find a broker that accepts connections from it.

14.7.1.1. Filtering client IDs using a consistent hashing algorithm

You can configure each broker in a cluster to use a consistent hashing algorithm to hash the client ID in each client connection. After the broker hashes the client ID, it performs a modulo operation on the hashed value to return an integer value, which identifies the target broker for the client connection. The broker compares the integer value returned to a unique value configured on the broker. If there is a match, the broker accepts the connection. If the values don't match, the broker rejects the connection. This process is repeated on each broker in the cluster until a match is found and a broker accepts the connection.

Procedure

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

2. Create a **connection-routers** element and create a **connection-route** to filter client IDs by using a consistent hashing algorithm. For example:

```xml
<configuration>
  <core>
    ...
    <connection-routers>
      <connection-route name="consistent-hash-routing">
        <key>CLIENT_ID</target-key>
        <local-target-filter>NULL|0</local-target-filter>
        <policy name="CONSISTENT_HASH_MODULO">
          <property key="modulo" value="<number_of_brokers_in_cluster>">
          </property>
        </policy>
      </connection-route>
    </connection-routers>
    ...
  </core>
</configuration>
```

connection-route

For the **connection-route name**, specify an identifying string for this connection routing configuration. You must add this name to each broker acceptor that you want to apply the consistent hashing filter to.

key

The key type to apply the filter to. To filter the client ID, specify **CLIENT_ID** in the **key** field.

local-target-filter

The value that the broker compares to the integer value returned by the modulo operation to determine if there is a match and the broker can accept the connection. The value of **NULL|0** in the example provides a match for connections that have no client ID (NULL) and connections where the number returned by the modulo operation is **0**.

policy

Accepts a **modulo** property key, which performs a modulo operation on the hashed client ID to identify the target broker. The value of the **modulo** property key must equal the number of brokers in the cluster.
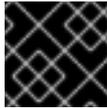
IMPORTANT

The **policy name** must be **CONSISTENT_HASH_MODULO**.

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

4. Create a **connection-routers** element and create a **connection route** to filter client IDs by using a consistent hashing algorithm.
   In the following example, the **local-target-filter** value of **NULL|1** provides a match for connections that have no client ID (NULL) and connections where the value returned by the modulo operation is **1**.

```
<configuration>
  <core>
    ...
    <connection-routers>
      <connection-route name="consistent-hash-routing">
        <key>CLIENT_ID</target-key>
        <local-target-filter>NULL|1</local-target-filter>
        <policy name="CONSISTENT_HASH_MODULO">
          <property key="modulo" value="<number_of_brokers_in_cluster>">
          </property>
        </policy>
      </connection-route>
    </connection-routers>
    ...
  </core>
</configuration>
```

5. Repeat this procedure to create a consistent hash filter for each additional broker in the cluster.

14.7.1.2. Filtering client IDs using a regular expression

You can partition client connections by configuring brokers to apply a regular expression filter to a part of the client ID in client connections. A broker only accepts a connection if the result of the regular expression filter matches the local target filter configured for the broker. If a match is not found, the broker rejects the connection. This process is repeated on each broker in the cluster until a match is found and a broker accepts the connection.

Prerequisites

- A common string in each client ID that can be filtered by a regular expression.

Procedure

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

2. Create a **connection-routers** element and create a **connection-route** to filter part of the client ID. For example:

```
<configuration>
  <core>
     ...
     <connection-routers>
        <connection-route name="regex-routing">
          <key>CLIENT_ID</target-key>
          <key-filter>^.{3}</key-filter>
          <local-target-filter>NULL|CL1</local-target-filter>
        </connection-route>
     </connection-routers>
     ...
  </core>
</configuration>
```

connection-route

For the **connection-route name**, specify an identifying string for this routing configuration. You must add this name to each broker acceptor that you want to apply the regular expression filter to.

key

The key to apply the filter to. To filter the client ID, specify **CLIENT_ID** in the **key** field.

key-filter

The part of the client ID string to which the regular expression is applied to extract a key value. In the example for the first broker above, the broker extracts a key value that is the first 3 characters of the client ID. If, for example, the client ID string is **CL100.consumer**, the broker extracts a key value of **CL1**. After the broker extracts the key value, it compares it to the value of the **local-target-filter**.

If an incoming connection does not have a client ID, or if the broker is unable to extract a key value by using the regular expression specified for the **key-filter**, the key value is set to NULL.

local-target-filter

The value that the broker compares to the key value to determine if there is a match and the broker can accept the connection. A value of **NULL|CL1**, as shown in the example for the first broker above, matches connections that have no client ID (NULL) or have a 3-character prefix of **CL1** in the client ID.

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

4. Create a **connection-routers** element and create a **connection route** to filter connections based on a part of the client ID.
In the following filter example, the broker uses a regular expression to extract a key value that is the first 3 characters of the client ID. The broker compares the values of **NULL** and **CL2** to the key value to determine if there is a match and the broker can accept the connection.

```
<configuration>
  <core>
     ...
     <connection-routers>
        <connection-route name="regex-routing">
           <key>CLIENT_ID</target-key>
           <key-filter>^.{3}</key-filter>
           <local-target-filter>NULL|CL2</local-target-filter>
        </connection-route>
     </connection-routers>
     ...
  </core>
</configuration>
```

5. Repeat this procedure and create the appropriate connection routing filter for each additional broker in the cluster.

14.7.2. Partitioning client connections to support data gravity

To support date gravity, you can partition client connections so that client consumers connect to the broker where the messages that they need to consume are produced. For example, if you have a set of addresses that are used by producer and consumer applications, you can configure the addresses on a specific broker. You can then partition client connections for both producers and consumers that use those addresses so they can only connect to that broker.

You can partition client connections based on attributes such as the role assigned to the connecting user, the username of the user, or the hostname or IP address of the client. This section shows an example of how to partition client connections by filtering user roles assigned to client users. If clients are required to authenticate to connect to brokers, you can assign roles to client users and filter connections so only users that match the role criteria can connect to a broker.

Prerequisites

- Clients are configured so they can connect to all of the brokers in the cluster, for example, by using a load balancer or by having all of the broker instances configured in the connection URL. If a broker rejects a connection because the client does not match the partitioning filter criteria configured for that broker, the client must be able to connect to the other brokers in the cluster to find a broker that accepts connections from it.

Procedure

1. Open the ***<broker_instance_dir>*/etc/artemis-roles.properties** file for the first broker. Add a **broker1users** role and add users to the role.

2. Open the ***<broker_instance_dir>*/etc/broker.xml** configuration file for the first broker.

3. Create a **connection-routers** element and create a **connection-route** to filter connections based on the roles assigned to users. For example:

```
<configuration>
  <core>
     ...
     <connection-routers>
        <connection-route name="role-based-routing">
           <key>ROLE_NAME</target-key>
```

```
            <key-filter>broker1users</key-filter>
            <local-target-filter>broker1users</local-target-filter>
         </connection-route>
      </connection-routers>
      ...
   </core>
</configuration>
```

connection-route

For the **connection-route name**, specify an identifying string for this routing configuration. You must add this name to each broker acceptor that you want to apply the role filter to.

key

The key to apply the filter to. To configure role-based filtering, specify **ROLE_NAME** in the **key** field.

key-filter

The string or regular expression that the broker uses to filter the user's roles and extract a key value. If the broker finds a matching role, it sets the key value to that role. If it does not find a matching role, the broker sets the key value to NULL. In the above example, the broker applies a filter of **broker1users** to the client user's roles. After the broker extracts the key value, it compares it to the value of the **local-target-filter**.

local-target-filter

The value that the broker compares to the key value to determine if there is a match and the broker can accept the connection. In the example, the broker compares a value of **broker1users** to the key value. It there is a match, which means that the user has a **broker1users** role, the broker accepts the connection.

4. Repeat this procedure and specify the appropriate role in the filter to partition clients on other brokers in the cluster.

14.7.3. Adding connection routes to acceptors

After you configure a connection route on a broker, you must add the route to one or more of the broker's acceptors to partition client connections. After you add a connection route to an acceptor, the broker applies the filter configured in the connection route to connections received by the acceptor.

Procedure

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

2. For each acceptor on which you want to enable partitioning, append the **router** key and specify the **connection-route name**. In the following example, a **connection-route name** of **consistent-hash-routing** is added to the **artemis** acceptor.

```
<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,
HORNETQ,MQTT,OPENWIRE;useEpoll=true;amqpCredits=1000;amqpLowCredits=300;route
r="consistent-hash-routing" </acceptor>
```

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

3. Repeat this procedure to specify the appropriate connection route filter for each broker in the cluster.

CHAPTER 15. CONFIGURING A MULTI-SITE, FAULT-TOLERANT MESSAGING SYSTEM USING CEPH

Large-scale enterprise messaging systems commonly have discrete broker clusters located in geographically distributed data centers. In the event of a data center outage, system administrators might need to preserve existing messaging data and ensure that client applications can continue to produce and consume messages. You can use specific broker topologies and Red Hat Ceph Storage, a software-defined storage platform, to ensure continuity of your messaging system during a data center outage. This type of solution is called a *multi-site, fault-tolerant architecture* .

> **NOTE**
>
> If you only require AMQP protocol support, consider Chapter 16, *Configuring a multi-site, fault-tolerant messaging system using broker connections*.

The following sections explain how to protect your messaging system from data center outages using Red Hat Ceph Storage:

- How Red Hat Ceph Storage clusters work

- Installing and configuring a Red Hat Ceph Storage cluster

- Adding backup brokers to take over from live brokers in the event of a data center outage

- Configuring your broker servers with the Ceph client role

- Configuring each broker to use the shared store high-availability (HA) policy, specifying where in the Ceph File System each broker stores its messaging data

- Configuring client applications to connect to new brokers in the event of a data center outage

- Restarting a data center after an outage

> **NOTE**
>
> Multi-site fault tolerance is not a replacement for high-availability (HA) broker redundancy **within** data centers. Broker redundancy based on live-backup groups provides automatic protection against single broker failures within single clusters. By contrast, multi-site fault tolerance protects against large-scale data center outages.

> **NOTE**
>
> To use Red Hat Ceph Storage to ensure continuity of your messaging system, you must configure your brokers to use the shared store high-availability (HA) policy. You cannot configure your brokers to use the replication HA policy. For more information about these policies, see Implementing High Availability .

15.1. HOW RED HAT CEPH STORAGE CLUSTERS WORK

Red Hat Ceph Storage is a clustered object storage system. Red Hat Ceph Storage uses data sharding of objects and policy-based replication to guarantee data integrity and system availability.

Red Hat Ceph Storage uses an algorithm called CRUSH (Controlled Replication Under Scalable Hashing) to determine how to store and retrieve data by automatically computing data storage

locations. You configure Ceph items called *CRUSH maps*, which detail cluster topography and specify how data is replicated across storage clusters.

CRUSH maps contain lists of Object Storage Devices (OSDs), a list of 'buckets' for aggregating the devices into a failure domain hierarchy, and rules that tell CRUSH how it should replicate data in a Ceph cluster's pools.

By reflecting the underlying physical organization of the installation, CRUSH maps can model — and thereby address — potential sources of correlated device failures, such as physical proximity, shared power sources, and shared networks. By encoding this information into the cluster map, CRUSH can separate object replicas across different failure domains (for example, data centers) while still maintaining a pseudo-random distribution of data across the storage cluster. This helps to prevent data loss and enables the cluster to operate in a degraded state.

Red Hat Ceph Storage clusters require a number of nodes (physical or virtual) to operate. Clusters must include the following types of nodes:

Monitor nodes

Each Monitor (MON) node runs the monitor daemon (**ceph-mon**), which maintains a master copy of the cluster map. The cluster map includes the cluster topology. A client connecting to the Ceph cluster retrieves the current copy of the cluster map from the Monitor, which enables the client to read from and write data to the cluster.

> **IMPORTANT**
>
> A Red Hat Ceph Storage cluster can run with one Monitor node; however, to ensure high availability in a production cluster, Red Hat supports only deployments with at least three Monitor nodes. A minimum of three Monitor nodes means that in the event of the failure or unavailability of one Monitor, a quorum exists for the remaining Monitor nodes in the cluster to elect a new leader.

Manager nodes

Each Manager (MGR) node runs the Ceph Manager daemon (**ceph-mgr**), which is responsible for keeping track of runtime metrics and the current state of the Ceph cluster, including storage utilization, current performance metrics, and system load. Usually, Manager nodes are colocated (that is, on the same host machine) with Monitor nodes.

Object Storage Device nodes

Each Object Storage Device (OSD) node runs the Ceph OSD daemon (**ceph-osd**), which interacts with logical disks attached to the node. Ceph stores data on OSD nodes. Ceph can run with very few OSD nodes (the default is three), but production clusters realize better performance at modest scales, for example, with 50 OSDs in a storage cluster. Having multiple OSDs in a storage cluster enables system administrators to define isolated failure domains within a CRUSH map.

Metadata Server nodes

Each Metadata Server (MDS) node runs the MDS daemon (**ceph-mds**), which manages metadata related to files stored on the Ceph File System (CephFS). The MDS daemon also coordinates access to the shared cluster.

Additional resources

For more information about Red Hat Ceph Storage, see What is Red Hat Ceph Storage?

15.2. INSTALLING RED HAT CEPH STORAGE

AMQ Broker multi-site, fault-tolerant architectures use Red Hat Ceph Storage 3. By replicating data across data centers, a Red Hat Ceph Storage cluster effectively creates a shared store available to brokers in separate data centers. You configure your brokers to use the shared store high-availability (HA) policy and store messaging data in the Red Hat Ceph Storage cluster.

Red Hat Ceph Storage clusters intended for production use should have a minimum of:

- Three Monitor (MON) nodes

- Three Manager (MGR) nodes

- Three Object Storage Device (OSD) nodes containing multiple OSD daemons

- Three Metadata Server (MDS) nodes

> **IMPORTANT**
>
> You can run the OSD, MON, MGR, and MDS nodes on either the same or separate physical or virtual machines. However, to ensure fault tolerance within your Red Hat Ceph Storage cluster, it is good practice to distribute each of these types of nodes across distinct data centers. In particular, you must ensure that in the event of a single data center outage, your storage cluster still has a minimum of two available MON nodes. Therefore, if you have three MON nodes in you cluster, each of these nodes must run on separate host machines in separate data centers. Do not run two MON nodes in a single data center, because failure of this data center will leave your storage cluster with only one remaining MON node. In this situation, the storage cluster can no longer operate.

The procedures linked-to from this section show you how to install a Red Hat Ceph Storage 3 cluster that includes MON, MGR, OSD, and MDS nodes.

Prerequisites

- For information about preparing a Red Hat Ceph Storage installation, see:

  - [Prerequisites](#)

  - [Requirements Checklist for Installing Red Hat Ceph Storage](#)

Procedure

- For procedures that show how to install a Red Hat Ceph 3 storage cluster that includes MON, MGR, OSD, and MDS nodes, see:

  - [Installing a Red Hat Ceph Storage Cluster](#)

  - [Installing Metadata Servers](#)

15.3. CONFIGURING A RED HAT CEPH STORAGE CLUSTER

This example procedure shows how to configure your Red Hat Ceph storage cluster for fault tolerance. You create CRUSH buckets to aggregate your Object Storage Device (OSD) nodes into data centers that reflect your real-life, physical installation. In addition, you create a rule that tells CRUSH how to

replicate data in your storage pools. These steps update the default CRUSH map that was created by your Ceph installation.

Prerequisites

- You have already installed a Red Hat Ceph Storage cluster. For more information, see Installing Red Hat Ceph Storage.

- You should understand how Red Hat Ceph Storage uses Placement Groups (PGs) to organize large numbers of data objects in a pool, and how to calculate the number of PGs to use in your pool. For more information, see Placement Groups (PGs).

- You should understand how to set the number of object replicas in a pool. For more information, Set the Number of Object Replicas .

Procedure

1. Create CRUSH buckets to organize your OSD nodes. Buckets are lists of OSDs, based on physical locations such as data centers. In Ceph, these physical locations are known as *failure domains*.

   ```
   ceph osd crush add-bucket dc1 datacenter
   ceph osd crush add-bucket dc2 datacenter
   ```

2. Move the host machines for your OSD nodes to the data center CRUSH buckets that you created. Replace host names **host1**–**host4** with the names of your host machines.

   ```
   ceph osd crush move host1 datacenter=dc1
   ceph osd crush move host2 datacenter=dc1
   ceph osd crush move host3 datacenter=dc2
   ceph osd crush move host4 datacenter=dc2
   ```

3. Ensure that the CRUSH buckets you created are part of the **default** CRUSH tree.

   ```
   ceph osd crush move dc1 root=default
   ceph osd crush move dc2 root=default
   ```

4. Create a rule to map storage object replicas across your data centers. This helps to prevent data loss and enables your cluster to stay running in the event of a single data center outage. The command to create a rule uses the following syntax: **ceph osd crush rule create-replicated <rule-name> <root> <failure-domain> <class>**. An example is shown below.

   ```
   ceph osd crush rule create-replicated multi-dc default datacenter hdd
   ```

   > **NOTE**
   >
   > In the preceding command, if your storage cluster uses solid–state drives (SSD), specify **ssd** instead of **hdd** (hard disk drives).

5. Configure your Ceph data and metadata pools to use the rule that you created. Initially, this might cause data to be backfilled to the storage destinations determined by the CRUSH algorithm.

```
ceph osd pool set cephfs_data crush_rule multi-dc
ceph osd pool set cephfs_metadata crush_rule multi-dc
```

6. Specify the numbers of Placement Groups (PGs) and Placement Groups for Placement (PGPs) for your metadata and data pools. The PGP value should be equal to the PG value.

```
ceph osd pool set cephfs_metadata pg_num 128
ceph osd pool set cephfs_metadata pgp_num 128

ceph osd pool set cephfs_data pg_num 128
ceph osd pool set cephfs_data pgp_num 128
```

7. Specify the numbers of replicas to be used by your data and metadata pools.

```
ceph osd pool set cephfs_data min_size 1
ceph osd pool set cephfs_metadata min_size 1

ceph osd pool set cephfs_data size 2
ceph osd pool set cephfs_metadata size 2
```

The following figure shows the Red Hat Ceph Storage cluster created by the preceding example procedure. The storage cluster has OSDs organized into CRUSH buckets corresponding to data centers.

Figure 15.1. Red Hat Ceph Storage Custer example
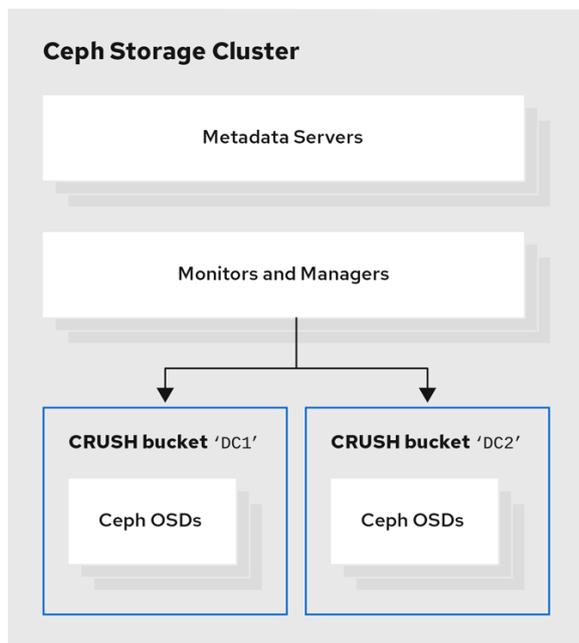
The following figure shows a possible layout of the first data center, including your broker servers. Specifically, the data center hosts:

- The servers for two live-backup broker pairs

- The OSD nodes that you assigned to the first data center in the preceding procedure

- Single Metadata Server, Monitor and Manager nodes. The Monitor and Manager nodes are usually co-located on the same machine.

Figure 15.2. Single data center with Ceph Storage Cluster and two live-backup broker pairs
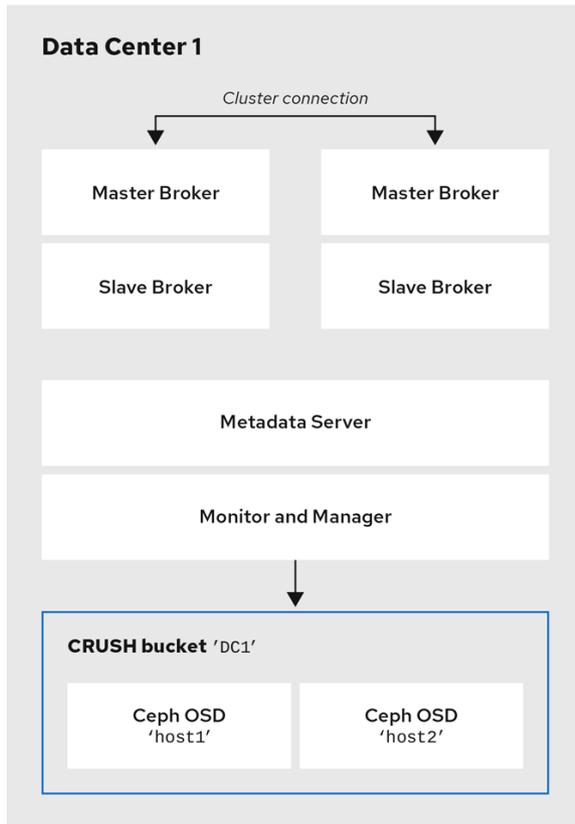
IMPORTANT

You can run the OSD, MON, MGR, and MDS nodes on either the same or separate physical or virtual machines. However, to ensure fault tolerance within your Red Hat Ceph Storage cluster, it is good practice to distribute each of these types of nodes across distinct data centers. In particular, you must ensure that in the event of a single data center outage, you storage cluster still has a minimum of two available MON nodes. Therefore, if you have three MON nodes in you cluster, each of these nodes must run on separate host machines in separate data centers.

The following figure shows a complete example topology. To ensure fault tolerance in your storage cluster, the MON, MGR, and MDS nodes are distributed across three separate data centers.

Figure 15.3. Multiple data centers with Ceph storage clusters and two live-backup broker pairs
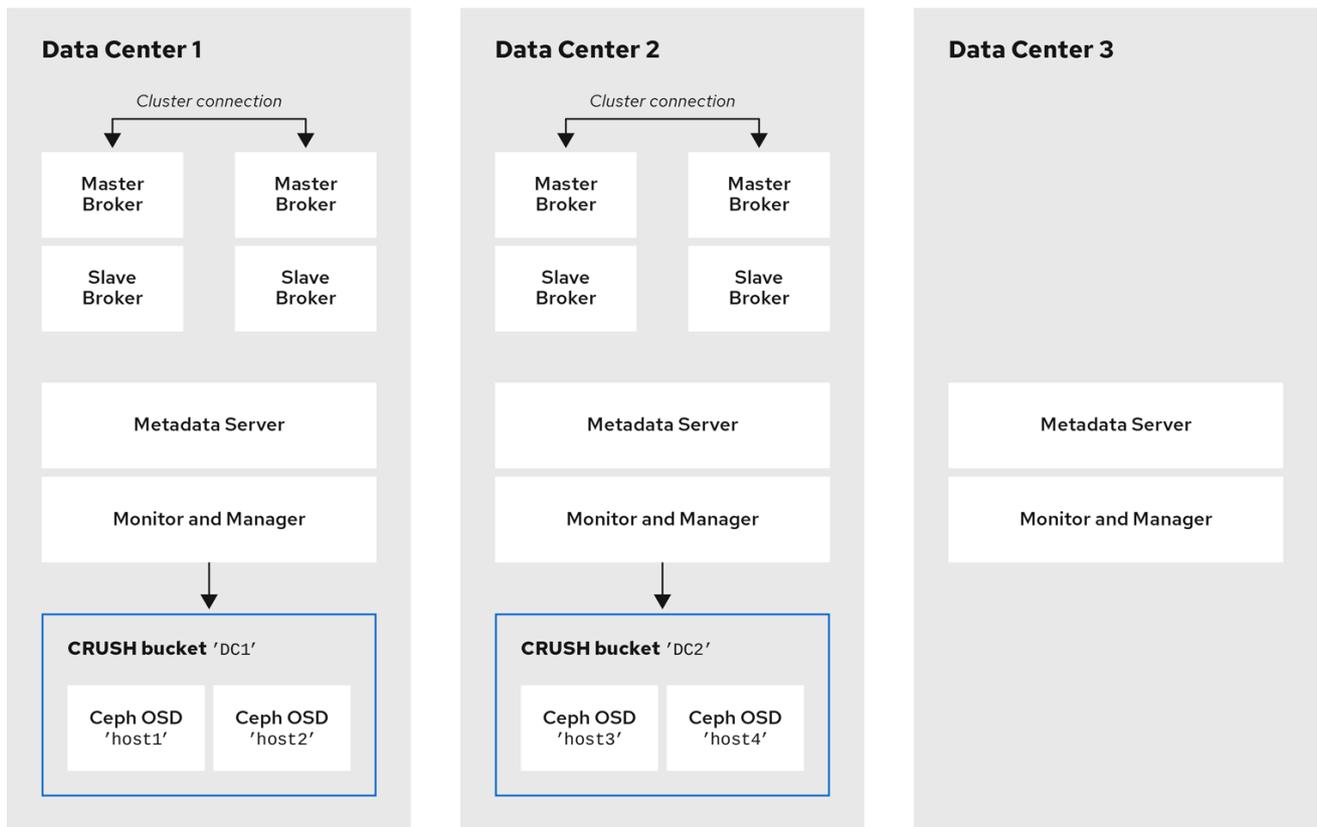
NOTE

Locating the host machines for certain OSD nodes in the same data center as your broker servers does not mean that you store messaging data on those specific OSD nodes. You configure the brokers to store messaging data in a specified directory in the Ceph File System. The Metadata Server nodes in your cluster then determine how to distribute the stored data across all available OSDs in your data centers and handle replication of this data across data centers. the sections that follow show how to configure brokers to store messaging data on the Ceph File System.

The figure below illustrates replication of data between the two data centers that have broker servers.

Figure 15.4. Ceph repliaction between two data centers that have broker servers
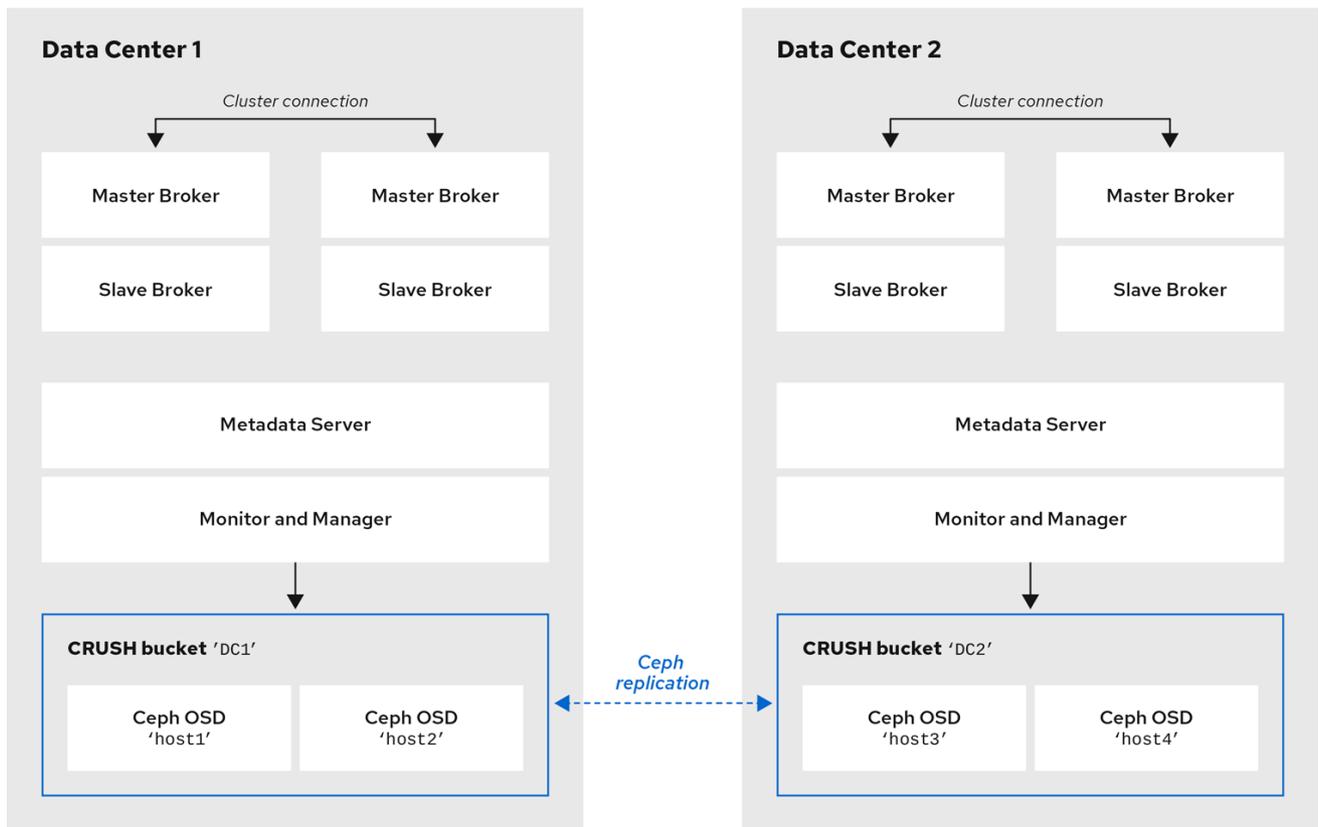
Additional resources

For more information about:

- Administrating CRUSH for your Red Hat Ceph Storage cluster, see CRUSH Administration.

- The full set of attributes that you can set on a storage pool, see Pool Values.

15.4. MOUNTING THE CEPH FILE SYSTEM ON YOUR BROKER SERVERS

Before you can configure brokers in your messaging system to store messaging data in your Red Hat Ceph Storage cluster, you first need to mount a Ceph File System (CephFS).

The procedure linked-to from this section shows you how to mount the CephFS on your broker servers.

Prerequisites

- You have:

  - Installed and configured a Red Hat Ceph Storage cluster. For more information, see Installing Red Hat Ceph Storage and Configuring a Red Hat Ceph Storage cluster .

  - Installed and configured three or more Ceph Metadata Server daemons (**ceph-mds**). For more information, see Installing Metadata Servers and Configuring Metadata Server Daemons.

- Created the Ceph File System from a Monitor node. For more information, see Creating the Ceph File System.

- Created a Ceph File System client user with a key that your broker servers can use for authorized access. For more information, see Creating Ceph File System Client Users .

Procedure

For instructions on mounting the Ceph File System on your broker servers, see Mounting the Ceph File System as a kernel client.

15.5. CONFIGURING BROKERS IN A MULTI-SITE, FAULT-TOLERANT MESSAGING SYSTEM

To configure your brokers as part of a multi-site, fault-tolerant messaging system, you need to:

- Add idle backup brokers to take over from live brokers in the event of a data center failure

- Configure all broker servers with the Ceph client role

- Configure each broker to use the shared store high-availability (HA) policy, specifying where in the Ceph File System the broker stores its messaging data

15.5.1. Adding backup brokers

Within each of your data centers, you need to add idle backup brokers that can take over from live master-slave broker groups that shut down in the event of a data center outage. You should replicate the configuration of live master brokers in your idle backup brokers. You also need to configure your backup brokers to accept client connections in the same way as your existing brokers.

In a later procedure, you see how to configure an idle backup broker to join an existing master-slave broker group. You must locate the idle backup broker in a separate data center to that of the live master-slave broker group. It is also recommended that you manually start the idle backup broker only in the event of a data center failure.

The following figure shows an example topology.

Figure 15.5. Idle backup broker in a multi-site, fault-tolerant messaging system
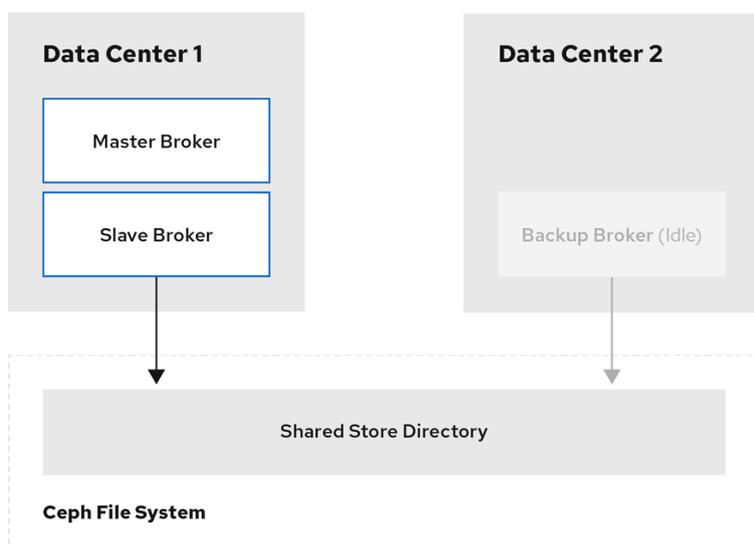
Additional resources

- To learn how to create additional broker instances, see Creating a standalone broker .

- For information about configuring broker network connections, see Chapter 2, *Configuring acceptors and connectors in network connections*.

15.5.2. Configuring brokers as Ceph clients

When you have added the backup brokers that you need for a fault-tolerant system, you must configure all of the broker servers with the Ceph client role. The client role enable brokers to store data in your Red Hat Ceph Storage cluster.

To learn how to configure Ceph clients, see Installing the Ceph Client Role .

15.5.3. Configuring shared store high availability

The Red Hat Ceph Storage cluster effectively creates a shared store that is available to brokers in different data centers. To ensure that messages remain available to broker clients in the event of a failure, you configure each broker in your live-backup group to use:

- The shared store high availability (HA) policy

- The same journal, paging, and large message directories in the Ceph File System

The following procedure shows how to configure the shared store HA policy on the master, slave, and idle backup brokers of your live-backup group.

Procedure

1. Edit the **broker.xml** configuration file of each broker in the live-backup group. Configure each broker to use the same paging, bindings, journal, and large message directories in the Ceph File System.

   ```
   # Master Broker - DC1
   <paging-directory>mnt/cephfs/broker1/paging</paging-directory>
   <bindings-directory>/mnt/cephfs/data/broker1/bindings</bindings-directory>
   <journal-directory>/mnt/cephfs/data/broker1/journal</journal-directory>
   <large-messages-directory>mnt/cephfs/data/broker1/large-messages</large-messages-
   directory>

   # Slave Broker - DC1
   <paging-directory>mnt/cephfs/broker1/paging</paging-directory>
   <bindings-directory>/mnt/cephfs/data/broker1/bindings</bindings-directory>
   <journal-directory>/mnt/cephfs/data/broker1/journal</journal-directory>
   <large-messages-directory>mnt/cephfs/data/broker1/large-messages</large-messages-
   directory>

   # Backup Broker (Idle) - DC2
   <paging-directory>mnt/cephfs/broker1/paging</paging-directory>
   <bindings-directory>/mnt/cephfs/data/broker1/bindings</bindings-directory>
   <journal-directory>/mnt/cephfs/data/broker1/journal</journal-directory>
   <large-messages-directory>mnt/cephfs/data/broker1/large-messages</large-messages-
   directory>
   ```

2. Configure the backup broker as a master within it's HA policy, as shown below. This configuration setting ensures that the backup broker immediately becomes the master when you manually start it. Because the broker is an idle backup, the **failover-on-shutdown** parameter that you can specify for an active master broker does not apply in this case.

```
<configuration>
  <core>
    ...
    <ha-policy>
      <shared-store>
        <master>
        </master>
      </shared-store>
    </ha-policy>
    ...
  </core>
</configuration>
```

Additional resources

- For more information about configuring the shared store high availability policy for live-backup broker groups, see Configuring shared store high availability .

15.6. CONFIGURING CLIENTS IN A MULTI-SITE, FAULT-TOLERANT MESSAGING SYSTEM

An internal client application is one that is running on a machine located in the same data center as the broker server. The following figure shows this topology.

Figure 15.6. Internal clients in a multi-site, fault-tolerant messaging system
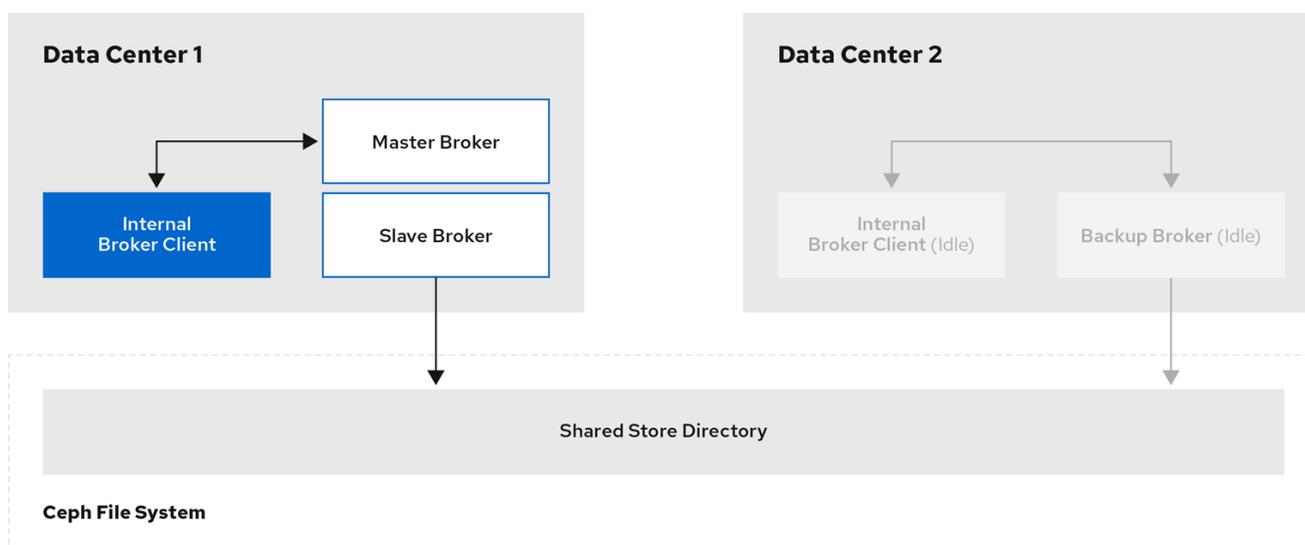
An external client application is one running on a machine located outside the broker data center. The following figure shows this topology.

Figure 15.7. External clients in a multi-site, fault-tolerant messaging system
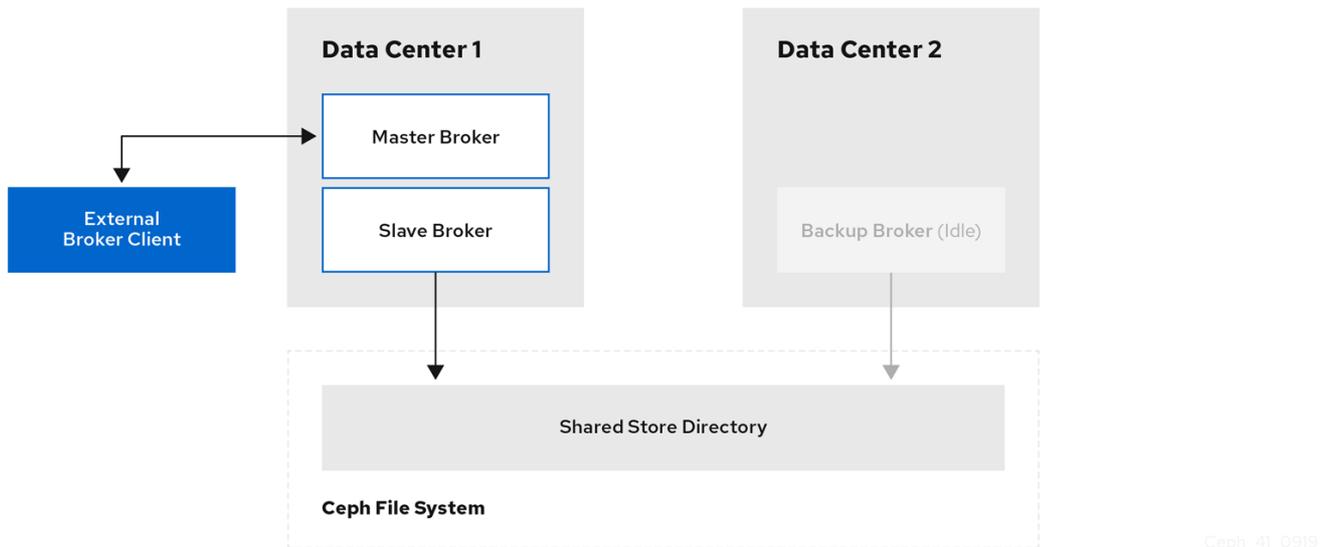
The following sub-sections describe show examples of configuring your internal and external client applications to connect to a backup broker in another data center in the event of a data center outage.

15.6.1. Configuring internal clients

If you experience a data center outage, internal client applications will shut down along with your brokers. To mitigate this situation, you must have another instance of the client application available in a separate data center. In the event of a data center outage, you manually start your backup client to connect to a backup broker that you have also manually started.

To enable the backup client to connect to a backup broker, you need to configure the client connection similarly to that of the client in your primary data center.

Example

A basic connection configuration for an AMQ Core Protocol JMS client to a master-slave broker group is shown below. In this example, **host1** and **host2** are the host servers for the master and slave brokers.

```
<ConnectionFactory connectionFactory = new
ActiveMQConnectionFactory("(tcp://host1:port,tcp://host2:port)?
ha=true&retryInterval=100&retryIntervalMultiplier=1.0&reconnectAttempts=-1");
```

To configure a backup client to connect to a backup broker in the event of a data center outage, use a similar connection configuration, but specify only the host name of your backup broker server. In this example, the backup broker server is **host3.**

```
<ConnectionFactory connectionFactory = new ActiveMQConnectionFactory("(tcp://host3:port)?
ha=true&retryInterval=100&retryIntervalMultiplier=1.0&reconnectAttempts=-1");
```

Additional resources

- For more information about configuring broker network connections, see Chapter 2, *Configuring acceptors and connectors in network connections*.

15.6.2. Configuring external clients

To enable an external broker client to continue producing or consuming messaging data in the event of a data center outage, you must configure the client to fail over to a broker in another data center. In the case of a multi-site, fault-tolerant system, you configure the client to fail over to the backup broker that you manually start in the event of an outage.

Examples

Shown below are examples of configuring the AMQ Core Protocol JMS and Red Hat build of Apache Qpid JMS clients to fail over to a backup broker in the event that the primary master-slave group is unavailable. In these examples, **host1** and **host2** are the host servers for the primary master and slave brokers, while **host3** is the host server for the backup broker that you manually start in the event of a data center outage.

- To configure an AMQ Core Protocol JMS client, include the backup broker on the ordered list of brokers that the client attempts to connect to.

  ```
  <ConnectionFactory connectionFactory = new
  ActiveMQConnectionFactory("(tcp://host1:port,tcp://host2:port,tcp://host3:port)?
  ha=true&retryInterval=100&retryIntervalMultiplier=1.0&reconnectAttempts=-1");
  ```

- To configure an Red Hat build of Apache Qpid JMS client, include the backup broker in the failover URI that you configure on the client.

  ```
  failover:(amqp://host1:port,amqp://host2:port,amqp://host3:port)?
  jms.clientID=myclient&failover.maxReconnectAttempts=20
  ```

Additional resources

- For more information about configuring failover on:

  - The AMQ Core Protocol JMS client, see Failover options.

  - The Red Hat build of Apache Qpid JMS client, see Failover options.

  - Other supported clients, consult the client-specific documentation in Product Documentation for Red Hat AMQ Clients.

15.7. VERIFYING STORAGE CLUSTER HEALTH DURING A DATA CENTER OUTAGE

When you have configured your Red Hat Ceph Storage cluster for fault tolerance, the cluster continues to run in a degraded state without losing data, even when one of your data centers fails.

This procedure shows how to verify the status of your cluster while it runs in a degraded state.

Procedure

1. To verify the status of your Ceph storage cluster, use the **health** or **status** commands:

   ```
   # ceph health
   # ceph status
   ```

2. To watch the ongoing events of the cluster on the command line, open a new terminal. Then, enter:

```
# ceph -w
```

When you run any of the preceding commands, you see output indicating that the storage cluster is still running, but in a degraded state. Specifically, you should see a warning that resembles the following:

```
health: HEALTH_WARN
        2 osds down
        Degraded data redundancy: 42/84 objects degraded (50.0%), 16 pgs unclean, 16 pgs degraded
```

Additional resources

- For more information about monitoring the health of your Red Hat Ceph Storage cluster, see Monitoring.

15.8. MAINTAINING MESSAGING CONTINUITY DURING A DATA CENTER OUTAGE

The following procedure shows you how to keep brokers and associated messaging data available to clients during a data center outage. Specifically, when a data center fails, you need to:

- Manually start any idle backup brokers that you created to take over from brokers in your failed data center.

- Connect internal or external clients to the new active brokers.

Prerequisites

- You must have:

  - Installed and configured a Red Hat Ceph Storage cluster. For more information, see Installing Red Hat Ceph Storage and Configuring a Red Hat Ceph Storage cluster .

  - Mounted the Ceph File System. For more information, see Mounting the Ceph File System on your broker servers.

  - Added idle backup brokers to take over from live brokers in the event of a data center failure. For more information, see Adding backup brokers.

  - Configured your broker servers with the Ceph client role. For more information, see Configuring brokers as Ceph clients .

  - Configured each broker to use the shared store high availability (HA) policy, specifying where in the Ceph File System each broker stores its messaging data . For more information, see Configuring shared store high availability .

  - Configured your clients to connect to backup brokers in the event of a data center outage. For more information, see Configuring clients in a multi-site, fault-tolerant messaging system.

Procedure

1. For each master-slave broker pair in the failed data center, manually start the idle backup broker that you added.

Figure 15.8. Backup broker started after data center outage
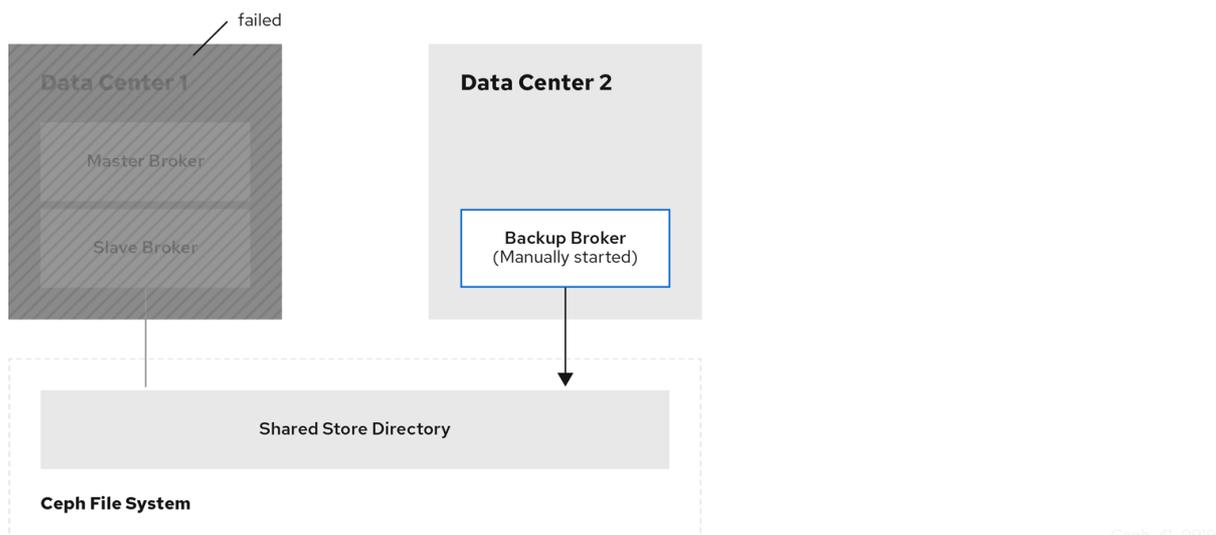
2. Reestablish client connections.

    a. If you were using an internal client in the failed data center, manually start the backup client that you created. As described in Configuring clients in a multi-site, fault-tolerant messaging system, you must configure the client to connect to the backup broker that you manually started.

       The following figure shows the new topology.

       Figure 15.9. Internal client connected to backup broker after data center outage
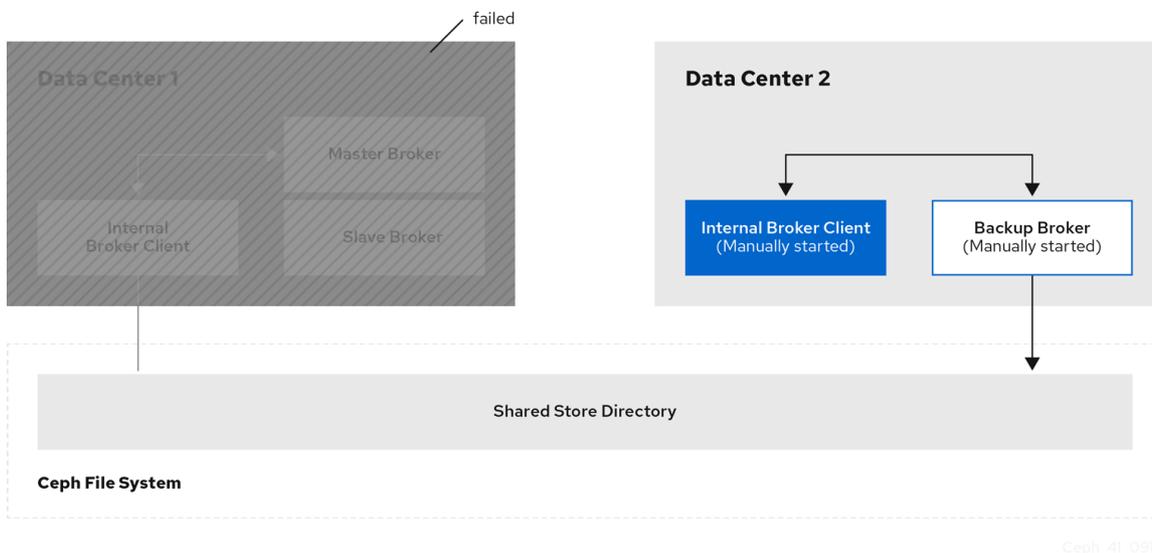
    b. If you have an external client, manually connect the external client to the new active broker or observe that the clients automatically fails over to the new active broker, based on its configuration. For more information, see Configuring external clients.

       The following figure shows the new topology.

Figure 15.10. External client connected to backup broker after data center outage
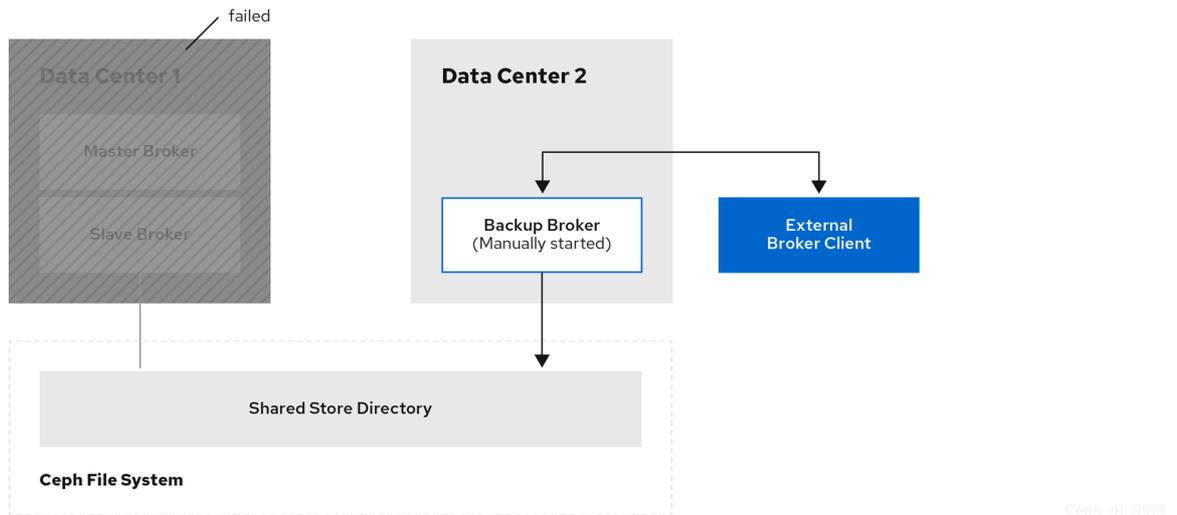
15.9. RESTARTING A PREVIOUSLY FAILED DATA CENTER

When a previously failed data center is back online, follow these steps to restore the original state of your messaging system:

- Restart the servers that host the nodes of your Red Hat Ceph Storage cluster

- Restart the brokers in your messaging system

- Re-establish connections from your client applications to your restored brokers

The following sub-sections show to perform these steps.

15.9.1. Restarting storage cluster servers

When you restart Monitor, Metadata Server, Manager, and Object Storage Device (OSD) nodes in a previously failed data center, your Red Hat Ceph Storage cluster self-heals to restore full data redundancy. During this process, Red Hat Ceph Storage automatically backfills data to the restored OSD nodes, as needed.

To verify that your storage cluster is automatically self-healing and restoring full data redundancy, use the commands previously shown in Verifying storage cluster health during a data center outage . When you re-execute these commands, you see that the percentage degradation indicated by the previous **HEALTH_WARN** message starts to improve until it returns to 100%.

15.9.2. Restarting broker servers

The following procedure shows how to restart your broker servers when your storage cluster is no longer operating in a degraded state.

Procedure

1. Stop any client applications connected to backup brokers that you manually started when the data center outage occurred.

2. Stop the backup brokers that you manually started.

a. On Linux:

> *<broker_instance_dir>*/bin/artemis stop

b. On Windows:

> *<broker_instance_dir>*\bin\artemis-service.exe stop

3. In your previously failed data center, restart the original master and slave brokers.

a. On Linux:

> *<broker_instance_dir>*/bin/artemis run

b. On Windows:

> *<broker_instance_dir>*\bin\artemis-service.exe start

The original master broker automatically resumes its role as master when you restart it.

15.9.3. Reestablishing client connections

When you have restarted your broker servers, reconnect your client applications to those brokers. The following subsections describe how to reconnect both internal and external client applications.

15.9.3.1. Reconnecting internal clients

Internal clients are those running in the same, previously failed data center as the restored brokers. To reconnect internal clients, restart them. Each client application reconnects to the restored master broker that is specified in its connection configuration.

For more information about configuring broker network connections, see Chapter 2, *Configuring acceptors and connectors in network connections*.

15.9.3.2. Reconnecting external clients

External clients are those running outside the data center that previously failed. Based on your client type, and the information in Configuring external broker clients, you either configured the client to automatically fail over to a backup broker, or you manually established this connection. When you restore your previously failed data center, you reestablish a connection from your client to the restored master broker in a similar way, as described below.

- If you configured your external client to automatically fail over to a backup broker, the client automatically fails back to the original master broker when you shut down the backup broker and restart the original master broker.

- If you manually connected the external client to a backup broker when a data center outage occurred, you must manually reconnect the client to the original master broker that you restart.

CHAPTER 16. CONFIGURING A MULTI-SITE, FAULT-TOLERANT MESSAGING SYSTEM USING BROKER CONNECTIONS

Large-scale enterprise messaging systems commonly have discrete broker clusters located in geographically distributed data centers. In the event of a data center outage, system administrators might need to preserve existing messaging data and ensure that client applications can continue to produce and consume messages. You can use broker connections to ensure continuity of your messaging system during a data center outage. This type of solution is called a *multi-site, fault-tolerant architecture*.

> **NOTE**
>
> Only the AMQP protocol is supported for communication between brokers for broker connections. A client can use any supported protocol. Currently, messages are converted to AMQP through the mirroring process.

The following sections explain how to protect your messaging system from data center outages using broker connections:

- Section 16.1, "About broker connections"

- Section 16.2, "Configuring broker mirroring"

> **NOTE**
>
> Multi-site fault tolerance is not a replacement for high-availability (HA) broker redundancy **within** data centers. Broker redundancy based on live-backup groups provides automatic protection against single broker failures within single clusters. In contrast, multi-site fault tolerance protects against large-scale data center outages.

16.1. ABOUT BROKER CONNECTIONS

With broker connections, a broker can establish a connection to another broker and mirror messages to and from that broker.

AMQP server connections

A broker can initiate connections to other endpoints using the AMQP protocol using broker connections. This means, for example, that the broker can connect to other AMQP servers and create elements on those connections.

The following types of operations are supported on an AMQP server connection:

- Mirrors - The broker uses an AMQP connection to another broker and duplicates messages and sends acknowledgements over the wire.

- Senders - Messages received on specific queues are transferred to another broker.

- Receivers - The broker pulls messages from another broker.

- Peers - The broker creates both senders and receivers on AMQ Interconnect endpoints.

This chapter describes how to use broker connections to create a fault-tolerant system. See Chapter 17, *Bridging brokers* for information about sender, receiver, and peer options.

The following events are sent through mirroring:

- Message sending – Messages sent to one broker will be "replicated" to the target broker.

- Message acknowledgement – Acknowledgements removing messages at one broker will be sent to the target broker.

- Queue and address creation.

- Queue and address deletion.

> **NOTE**
>
> If the message is pending for a consumer on the target mirror, the acknowledgement will not succeed and the message might be delivered by both brokers.

Mirroring does not block any operation and does not affect the performance of a broker.

The broker only mirrors messages arriving from the point in time the mirror was configured. Previously existing messages will not be forwarded to other brokers.

16.2. CONFIGURING BROKER MIRRORING

You can use broker connections to mirror messages between a pair of brokers. Only one of the brokers can be active at any time.

Prerequisites

- You have two working brokers.

Procedure

1. Create a **broker-connections** element in the **broker.xml** file for the first broker, for example:

   ```xml
   <broker-connections>
     <amqp-connection uri="tcp://<hostname>:<port>" name="DC1">
       <mirror/>
     </amqp-connection>
   </broker-connections>
   ```

   \<hostname>

   The hostname of the other broker instance.

   \<port>

   The port used by the broker on the other host.

   All messages on the first broker are mirrored to the second broker, but messages that existed before the mirror was created are not mirrored.

2. If you want the first broker to mirror messages synchronously to ensure that the mirrored broker is up-to-date for disaster recovery, set the **sync=true** attribute in the **amqp-connection** element of the broker, as shown in the following example.

Synchronous mirroring requires that messages sent by a broker to a mirrored broker are written to the volumes of both brokers at the same time. Once the write operation is complete on both brokers, the source broker acknowledges that the write request is complete and control is returned to clients.

```
<broker-connections>
  <amqp-connection uri="tcp://<hostname>:<port>" name="DC2">
    <mirror sync="true"/>
  </amqp-connection>
</broker-connections>
```

> **NOTE**
>
> If the write request cannot be completed on the mirrored broker, for example, if the broker is unavailable, client connections are blocked until a mirror is available to complete the most recent write request.

> **NOTE**
>
> The broker connections name in the example, **DC1**, is used to create a queue named **$ACTIVEMQ_ARTEMIS_MIRROR_mirror**. Make sure that the corresponding broker is configured to accept those messages, even though the queue is not visible on that broker.

3. Create a **broker-connections** element in the **broker.xml** file for the second broker, for example:

```
<broker-connections>
  <amqp-connection uri="tcp://<hostname>:<port>" name="DC2">
    <mirror/>
  </amqp-connection>
</broker-connections>
```

4. If you want the second broker to mirror messages synchronously, set the **sync=true** attribute in the **amqp-connection** element of the broker. For example:

```
<broker-connections>
  <amqp-connection uri="tcp://<hostname>:<port>" name="DC2">
    <mirror sync="true"/>
  </amqp-connection>
</broker-connections>
```

5. (Optional) Configure the following parameters for the mirror, as required.

queue-removal

Specifies whether either queue or address removal events are sent. The default value is **true**.

message-acknowledgments

Specifies whether message acknowledgments are sent. The default value is **true**.

queue-creation

Specifies whether either queue or address creation events are sent. The default value is **true**.

For example:

```
<broker-connections>
  <amqp-connection uri="tcp://<hostname>:<port>" name="DC2">
    <mirror sync="true" queue-removal="false" message-acknowledgments ="false" queue-
creation="false"/>
  </amqp-connection>
</broker-connections>
```

6. (Optional) Customize the broker retry attempts to acknowledge messages on the target mirror. An acknowledgment might be received on a target mirror for a message that is not in the queue memory. To give the broker sufficient time to retry acknowledging the message on the target mirror, you can customize the following parameters for your environment:

   mirrorAckManagerQueueAttempts

   The number of attempts the broker makes to find a message in memory. The default value is **5**. If the broker does not find the message in memory after the specified number of attempts, the broker searches for the message in page files.

   mirrorAckManagerPageAttempts

   The number of attempts the broker makes to find a message in page files if the message was not found in memory. The default value is **2**.

   mirrorAckManagerRetryDelay

   The interval, in milliseconds, between attempts the broker makes to find a message to acknowledge in memory and then in page files.

   Specify any of these parameters outside of the **broker-connections** element. For example:

   ```
   <mirrorAckManagerQueueAttempts>8</mirrorAckManagerQueueAttempts>

   <broker-connections>
     <amqp-connection uri="tcp://<hostname>:<port>" name="DC2">
       <mirror/>
     </amqp-connection>
   </broker-connections>
   ```

7. (Optional) If messages are paged on the target mirror, set the **mirrorPageTransaction** to **true** if you want the broker to coordinate writing duplicate detection information with writing messages to page files.
   If the **mirrorPageTransaction** attribute is set to **false**, which is the default, and a communication failure occurs between the brokers, a duplicate message can, in rare circumstances, be written to the target mirror.

   Setting this parameter to **true** increases the broker's memory usage.

8. Configure clients using the instructions documented in Section 15.6, "Configuring clients in a multi-site, fault-tolerant messaging system", noting that with broker connections, there is no shared storage.

   > **IMPORTANT**
   >
   > Red Hat does not support client applications consuming messages from both brokers in a mirror configuration. To prevent clients from consuming messages on both brokers, disable the client acceptors on one of the brokers.

CHAPTER 17. BRIDGING BROKERS

Bridges provide a method to connect two brokers, forwarding messages from one to the other.

The following bridges are available:

Core

> The **core-bridge** example demonstrates a core bridge deployed on one broker, which consumes messages from a local queue and forwards them to an address on a second broker.

Mirror

> See Chapter 16, *Configuring a multi-site, fault-tolerant messaging system using broker connections*

Sender and receiver

> See Section 17.1, "Sender and receiver configurations for broker connections"

Peer

> See Section 17.2, "Peer configurations for broker connections"

> **NOTE**
>
> The **broker.xml** element for Core bridges is **bridge**. The other bridging techniques use the **<broker-connection>** element.

17.1. SENDER AND RECEIVER CONFIGURATIONS FOR BROKER CONNECTIONS

It is possible to connect a broker to another broker by creating a sender or receiver broker connection element in the **<broker-connections>** section of **broker.xml**.

For a **sender**, the broker creates a message consumer on a queue that sends messages to another broker.

For a **receiver**, the broker creates a message producer on an address that receives messages from another broker.

Both elements function as a message bridge. However, there is no additional overhead required to process messages. Senders and receivers behave just like any other consumer or producer in a broker.

Specific queues can be configured by senders or receivers. Wildcard expressions can be used to match senders and receivers to specific addresses or *sets* of addresses. When configuring a sender or receiver, the following properties can be set:

- **address-match**: Match the sender or receiver to a specific address or set of addresses, using a wildcard expression.

- **queue-name**: Configure the sender or receiver for a specific queue.

- Using address expressions:

  ```
  <broker-connections>
    <amqp-connection uri="tcp://HOST:PORT" name="other-server">
     <sender address-match="queues.#"/>
     <!-- notice the local queues for remotequeues.# need to be created on this broker -->
     <receiver address-match="remotequeues.#"/>
  ```

```
    </amqp-connection>
  </broker-connections>

  <addresses>
   <address name="remotequeues.A">
    <anycast>
      <queue name="remoteQueueA"/>
    </anycast>
   </address>
   <address name="queues.B">
    <anycast>
      <queue name="localQueueB"/>
    </anycast>
   </address>
  </addresses>
```

- Using queue names:

```
  <broker-connections>
   <amqp-connection uri="tcp://HOST:PORT" name="other-server">
    <receiver queue-name="remoteQueueA"/>
    <sender queue-name="localQueueB"/>
   </amqp-connection>
  </broker-connections>

  <addresses>
   <address name="remotequeues.A">
    <anycast>
      <queue name="remoteQueueA"/>
    </anycast>
   </address>
   <address name="queues.B">
    <anycast>
      <queue name="localQueueB"/>
    </anycast>
   </address>
  </addresses>
```

> **NOTE**
>
> Receivers can only be matched to a local queue that already exists. Therefore, if receivers are being used, ensure that queues are pre-created locally. Otherwise, the broker cannot match the remote queues and addresses.

> **NOTE**
>
> Do not create a sender and a receiver with the same destination because this creates an infinite loop of sends and receives.

17.2. PEER CONFIGURATIONS FOR BROKER CONNECTIONS

The broker can be configured as a peer which connects to a AMQ Interconnect instance and instructs it that the broker will act as a store-and-forward queue for a given AMQP waypoint address configured on that router. In this scenario, clients connect to a router to send and receive messages using a waypoint

address, and the router routes these messages to or from the queue on the broker.

This peer configuration creates a sender and receiver pair for each destination matched in the broker connections configuration on the broker. These pairs include configurations that enable the router to collaborate with the broker. This feature avoids the requirement for the router to initiate a connection and create auto-links.

With a peer configuration, the same properties are present as when there are senders and receivers. For example, a configuration where queues with names beginning **queue**. act as storage for the matching router waypoint address would be:

```
<broker-connections>
  <amqp-connection uri="tcp://HOST:PORT" name="router">
    <peer address-match="queues.#"/>
  </amqp-connection>
</broker-connections>

<addresses>
  <address name="queues.A">
    <anycast>
      <queue name="queues.A"/>
    </anycast>
  </address>
  <address name="queues.B">
    <anycast>
      <queue name="queues.B"/>
    </anycast>
  </address>
</addresses>
```

There must be a matching address waypoint configuration on the router. This instructs it to treat the particular router addresses the broker attaches to as waypoints. For example, see the following prefix-based router address configuration:

```
address {
    prefix: queue
    waypoint: yes
}
```

NOTE

Do not use the **peer** option to connect directly to another broker. If you use this option to connect to another broker, all messages become immediately ready to consume, creating an infinite echo of sends and receives.

CHAPTER 18. LOGGING

AMQ Broker uses the Apache Log4j 2 logging utility to provide message logging. When you install a broker, it has a default Log4j 2 configuration in the ***<broker_instance_dir>*/etc/log4j2.properties** file. With the default configuration, the loggers write to both the console and to a file.

The loggers available in AMQ Broker are shown in the following table.

Table 18.1. AMQ Broker loggers

Logger	Description
org.apache.activemq.artemis.core.server	Logs the broker core
org.apache.activemq.artemis.journal	Logs Journal calls
org.apache.activemq.artemis.utils	Logs utility calls
org.apache.activemq.artemis.jms	Logs JMS calls
org.apache.activemq.artemis.integration.bootstrap	Logs bootstrap calls
org.apache.activemq.audit.base	Logs access to all JMX object methods
org.apache.activemq.audit.message	Logs message operations such as production, consumption, and browsing of messages
org.apache.activemq.audit.resource	Logs authentication events, creation or deletion of broker resources from JMX or the AMQ Broker management console, and browsing of messages in the management console

18.1. CHANGING THE LOGGING LEVEL

You can configure the logging level for each logger in the ***<logger name>*.level** line after the logger name, as shown in the following example for the **apache.activemq.artemis.core.server** logger:

```
logger.artemis_server.name=org.apache.activemq.artemis.core.server
logger.artemis_server.level=INFO
```

The default logging level for audit loggers is **OFF**, which means that logging is disabled. The default logging level for the other loggers available in AMQ Broker is **INFO**. For information on the logging levels available in Log4j 2, see the Log4j 2 documentation .

18.2. ENABLING AUDIT LOGGING

Three audit loggers are available for you to enable; a base audit logger, a message audit logger, and a resource audit logger.

Base audit logger (org.apache.activemq.audit.base)

Logs *access* to all JMX object methods, such as creation and deletion of addresses and queues. The log **does not** indicate whether these operations succeeded or failed.

Message audit logger (org.apache.activemq.audit.message)

Logs message-related broker operations, such as production, consumption, or browsing of messages.

Resource audit logger (org.apache.activemq.audit.resource)

Logs authentication success or failure from clients, routes, and the AMQ Broker management console. Also logs creation, update, or deletion of queues from either JMX or the management console, and browsing of messages in the management console.

You can enable each audit logger independently of the others. By default, the logging level is set to **OFF**, which means logging is disabled, for each audit logger. To enable one of the audit loggers, change the logging level from **OFF** to **INFO**. For example:

```
logger.audit_base = INFO, audit_log_file
```

> **NOTE**
>
> **INFO** is the only available logging level for the **logger.org.apache.activemq.audit.base**, **logger.org.apache.activemq.audit.message**, and **logger.org.apache.activemq.audit.resource** audit loggers.

> **IMPORTANT**
>
> The message audit logger runs on a performance-intensive path on the broker. Enabling the logger might negatively affect the performance of the broker, particularly if the broker is running under a high messaging load. Red Hat recommends that you do not enable audit logging on messaging systems where high throughput is required.

18.3. CLIENT OR EMBEDDED SERVER LOGGING

If you want to enable logging on a client, you need to include a logging implementation in your application that supports the SLF4J facade. If you are using Maven, add the following dependencies for Log4j 2:

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>artemis-jms-client</artifactId>
  <version>2.28.0</version>
</dependency>
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-slf4j-impl</artifactId>
  <version>2.19.0</version>
</dependency>
```

You can supply the Log4j 2 configuration in a **log4j2.properties** file in the classpath. Or, you can specify a custom configuration file by using the **log4j2.configurationFile** system property. For example:

```
-Dlog4j2.configurationFile=file:///path/to/custom-log4j2-config.properties
```

The following is an example **log4j2.properties** file for a client:

```
# Log4J 2 configuration

# Monitor config file every X seconds for updates
monitorInterval = 5

rootLogger = INFO, console, log_file

logger.activemq.name=org.apache.activemq
logger.activemq.level=INFO

# Console appender
appender.console.type=Console
appender.console.name=console
appender.console.layout.type=PatternLayout
appender.console.layout.pattern=%d %-5level [%logger] %msg%n

# Log file appender
appender.log_file.type = RollingFile
appender.log_file.name = log_file
appender.log_file.fileName = log/application.log
appender.log_file.filePattern = log/application.log.%d{yyyy-MM-dd}
appender.log_file.layout.type = PatternLayout
appender.log_file.layout.pattern = %d %-5level [%logger] %msg%n
appender.log_file.policies.type = Policies
appender.log_file.policies.cron.type = CronTriggeringPolicy
appender.log_file.policies.cron.schedule = 0 0 0 * * ?
appender.log_file.policies.cron.evaluateOnStartup = true
```

18.4. AMQ BROKER PLUGIN SUPPORT

AMQ supports custom plugins. You can use plugins to log information about many different types of events that would otherwise only be available through debug logs. Multiple plugins can be registered, tied, and executed together. The plugins are executed based on the order of the registration, that is, the first plugin registered is always executed first.

You can create custom plugins and implement them using the **ActiveMQServerPlugin** interface. This interface ensures that the plugin is on the classpath, and is registered with the broker. As all the interface methods are implemented by default, you have to add only the required behavior that needs to be implemented.

18.4.1. Adding plugins to the class path

Add the custom created broker plugins to the broker runtime by adding the relevant **.jar** files to the **<broker_instance_dir>/lib** directory.

If you are using an embedded system, place the **.jar** file under the regular class path of your embedded application.

18.4.2. Registering a plugin

You must register a plugin by adding the **broker-plugins** element in the **broker.xml** configuration file. You can specify the plugin configuration value using the **property** child elements. These properties are read and passed into the plugin's init (Map<String, String>) operation after the plugin has been instantiated.

```
<broker-plugins>
  <broker-plugin class-name="some.plugin.UserPlugin">
    <property key="property1" value="val_1" />
    <property key="property2" value="val_2" />
  </broker-plugin>
</broker-plugins>
```

18.4.3. Registering a plugin programmatically

To register a plugin programmatically, use the **registerBrokerPlugin()** method and pass in a new instance of your plugin. The example below shows the registration of the **UserPlugin** plugin:

```
Configuration config = new ConfigurationImpl();

config.registerBrokerPlugin(new UserPlugin());
```

18.4.4. Logging specific events

By default, AMQ broker provides the **LoggingActiveMQServerPlugin** plugin to log specific broker events. The **LoggingActiveMQServerplugin** plugin is commented-out by default and does not log any information.

The following table describes each plugin property. Set a configuration property value to **true** to log events.

Table 18.2. **LoggingActiveMQServerPlugin** plugin properties

Property	Description
LOG_CONNECTION_EVENTS	Logs information when a connection is created or destroyed.
LOG_SESSION_EVENTS	Logs information when a session is created or closed.
LOG_CONSUMER_EVENTS	Logs information when a consumer is created or closed.
LOG_DELIVERING_EVENTS	Logs information when message is delivered to a consumer and when a message is acknowledged by a consumer.
LOG_SENDING_EVENTS	Logs information when a message has been sent to an address and when a message has been routed within the broker.
LOG_INTERNAL_EVENTS	Logs information when a queue created or destroyed, when a message is expired, when a bridge is deployed, and when a critical failure occurs.
LOG_ALL_EVENTS	Logs information for all the above events.

To configure the **LoggingActiveMQServerPlugin** plugin to log connection events, uncomment the **<broker-plugins>** section in the **broker.xml** configuration file. The value of all the events is set to **true** in the commented default example.

```
<configuration ...>
...
<!-- Uncomment the following if you want to use the Standard LoggingActiveMQServerPlugin plugin
to log in events -->
      <broker-plugins>
      <broker-plugin class-
name="org.apache.activemq.artemis.core.server.plugin.impl.LoggingActiveMQServerPlugin">
      <property key="LOG_ALL_EVENTS" value="true"/>
      <property key="LOG_CONNECTION_EVENTS" value="true"/>
      <property key="LOG_SESSION_EVENTS" value="true"/>
      <property key="LOG_CONSUMER_EVENTS" value="true"/>
      <property key="LOG_DELIVERING_EVENTS" value="true"/>
      <property key="LOG_SENDING_EVENTS" value="true"/>
      <property key="LOG_INTERNAL_EVENTS" value="true"/>
      </broker-plugin>
    </broker-plugins>
...
</configuration>
```

When you change the configuration parameters inside the **<broker-plugins>** section, you must restart the broker to reload the configuration updates. These configuration changes are **not** reloaded based on the **configuration-file-refresh-period** setting.

When the log level is set to **INFO**, an entry is logged after the event has occurred. If the log level is set to **DEBUG**, log entries are generated for both before and after the event, for example, **beforeCreateConsumer()** and **afterCreateConsumer()**. If the log Level is set to **DEBUG**, the logger logs more information for a notification, when available.

CHAPTER 19. TUNING GUIDELINES

Review the following guidelines to tune AMQ Broker.

19.1. TUNING PERSISTENCE

Review the following information for tips on improving persistence performance.

- Persist messages to a file-based journal.
  Use a file-based journal for message persistence. AMQ Broker can also persist messages to a Java Database Connectivity (JDBC) database, but this has a performance cost when compared to using a file-based journal.

- Put the message journal on its own physical volume.
  One of the advantages of an append-only journal is that disk head movement is minimized. This advantage is lost if the disk is shared. When multiple processes, such as a transaction coordinator, databases, and other journals, read and write from the same disk, performance is impacted because the disk head must skip around between different files. If you are using paging or large messages, make sure that they are also put on separate volumes.

- Tune the **journal-min-files** parameter value.
  Set the **journal-min-files** parameter to the number of files that fits your average sustainable rate. If new files are created frequently in the journal data directory, meaning that much data is being persisted, you need to increase the minimum number of files that the journal maintains. This allows the journal to reuse, rather than create, new data files.

- Optimize the journal file size.
  Align the value of the **journal-file-size** parameter to the capacity of a cylinder on the disk. The default value of 10 MB should be enough on most systems.

- Use the asynchronous IO (AIO) journal type.
  For Linux operating systems, keep your journal type as AIO. AIO scales better than Java new I/O (NIO).

- Tune the **journal-buffer-timeout** parameter value.
  Increasing the value of the **journal-buffer-timeout** parameter results in increased throughput at the expense of latency.

- Tune the **journal-max-io** parameter value.
  If you are using AIO, you might be able to improve performance by increasing the **journal-max-io** parameter value. Do not change this value if you are using NIO.

- Tune the **journal-pool-files** parameter.
  Set the **journal-pool-files** parameter, which is the upper threshold of the journal file pool, to a number that is close to your maximum expected load. When required, the journal expands beyond the upper threshold, but shrinks to the threshold, when possible. This allows reuse of files without consuming more disk space than required. If you see new files being created too often in the journal data directory, increase the **journal-pool-size** parameter. Increasing this parameter allows the journal to reuse more existing files instead of creating new files, which improves performance.

- Disable the **journal-data-sync** parameter if you do not require durability guarantees on journal writes.

If you do not require guaranteed durability on journal writes if a power failure occurs, disable the **journal-data-sync** parameter and use a journal type of **NIO** or **MAPPED** for better performance.

19.2. TUNING JAVA MESSAGE SERVICE (JMS)

If you use the JMS API, review the following information for tips on how to improve performance.

- Disable the message ID.
  If you do not need message IDs, disable them by using the **setDisableMessageID()** method on the **MessageProducer** class. Setting the value to **true** eliminates the need to create a unique ID and decreases the size of the message.

- Disable the message timestamp.
  If you do not need message timestamps, disable them by using the **setDisableMessageTimeStamp()** method on the **MessageProducer** class. Setting the value to **true** eliminates the overhead of creating the timestamp and decreases the size of the message.

- Avoid using **ObjectMessage**.
  ObjectMessage is used to send a message that has a serialized object, meaning the body of the message, or payload, is sent over the wire as a stream of bytes. The Java serialized form of even small objects is quite large and takes up significant space on the wire. It is also slow when compared to custom marshalling techniques. Use **ObjectMessage** only if you cannot use one of the other message types, for example, if you do not know the type of the payload until runtime.

- Avoid **AUTO_ACKNOWLEDGE**.
  The choice of acknowledgment mode for a consumer impacts performance because of the additional overhead and traffic incurred by sending the acknowledgment message over the network. **AUTO_ACKNOWLEDGE** incurs this overhead because it requires that an acknowledgment is sent from the server for each message received on the client. If possible, use **DUPS_OK_ACKNOWLEDGE**, which acknowledges messages in a lazy manner or **CLIENT_ACKNOWLEDGE**, meaning the client code will call a method to acknowledge the message. Or, batch up many acknowledgments with one acknowledge or commit in a transacted session.

- Avoid durable messages.
  By default, JMS messages are durable. If you do not need durable messages, set them to be non-durable. Durable messages incur additional overhead because they are persisted to storage.

- Use **TRANSACTED_SESSION** mode to send and receive messages in a single transaction.
  By batching messages in a single transaction, AMQ Broker requires only one network round trip on the commit, not on every send or receive.

19.3. TUNING TRANSPORT SETTINGS

Review the following information for tips on tuning transport settings.

- If your operating system supports TCP auto-tuning, as is the case with later versions of Linux, do not increase the TCP send and receive buffer sizes to try to improve performance. Setting the buffer sizes manually on a system that has auto-tuning can prevent auto-turing from working and actually reduce broker performance. If your operating system does not support TCP auto-tuning and the broker is running on a fast machine and network, you might improve the broker performance by increasing the TCP send and receive buffer sizes. For more information, see Appendix A, *Acceptor and Connector Configuration Parameters* .

- If you expect many concurrent connections on your broker, or if clients are rapidly opening and closing connections, ensure that the user running the broker has permission to create enough file handles. The way you do this varies between operating systems. On Linux systems, you can increase the number of allowable open file handles in the **/etc/security/limits.conf** file. For example, add the lines:

```
serveruser   soft  nofile  20000
serveruser   hard  nofile  20000
```

  This example allows the **serveruser** user to open up to 20000 file handles.

- Set a value for the **batchDelay** netty TCP parameter and set the **directDeliver** netty TCP parameter to **false** to maximize throughput for very small messages.

19.4. TUNING THE BROKER VIRTUAL MACHINE

Review the following information for tips on how to tune various virtual machine settings.

- Use the latest Java virtual machine for best performance.

- Allocate as much memory as possible to the server.
  AMQ Broker can run with low memory by using paging. However, you get improved performance if AMQ Broker can keep all queues in memory. The amount of memory you require depends on the size and number of your queues and the size and number of your messages. Use the -Xms and -Xmx JVM arguments to set the available memory.

- Tune the heap size.
  During periods of high load, it is likely that AMQ Broker generates and destroys large numbers of objects, which can result in a build up of stale objects. This increases the risk of the broker running out of memory and causing a full garbage collection, which might introduce pauses and unintentional behaviour. To reduce this risk, ensure that the maximum heap size (-Xmx) for the JVM is set to at least five times the value of the **global-max-size** parameter. For example, if the broker is under high load and running with a **global-max-size** of 1 GB, set the maximum heap size to 5 GB.

19.5. TUNING OTHER SETTINGS

Review the following information for additional tips on improving performance.

- Use asynchronous send acknowledgements.
  If you need to send non-transactional, durable messages and do not need a guarantee that they have reached the server by the time the call to send() returns, do not set them to be sent blocking. Instead, use asynchronous send acknowledgements to get the send acknowledgements returned in a separate stream. However, in the case of a server crash, some messages might be lost.

- Use pre-acknowledge mode.
  With pre-acknowledge mode, messages are acknowledged before they are sent to the client. This reduces the amount of acknowledgment traffic on the wire. However, if that client crashes, messages are not redelivered if the client reconnects.

- Disable security.
  A small performance improvement results from disabling security by setting the **security-enabled** parameter to **false**.

- Disable persistence.
  You can turn off message persistence by setting the **persistence-enabled** parameter to **false**.

- Sync transactions lazily.
  Setting the **journal-sync-transactional** parameter to **false** provides better performance when persisting transactions, at the expense of some possibility of loss of transactions on failure.

- Sync non-transactional lazily.
  Setting the **journal-sync-non-transactional** parameter to **false** provides better performance when persisting non-transactions, at the expense of some possibility of loss of durable messages on failure.

- Send messages non-blocking.
  To avoid waiting for a network round trip for every message sent, set the **block-on-durable-send** and **block-on-non-durable-send** parameters to **false** if you are using Java Messaging Service (JMS) and Java Naming and Directory Interface (JNDI). Or, set them directly on the **ServerLocator** by calling the **setBlockOnDurableSend()** and **setBlockOnNonDurableSend()** methods.

- Optimize the **consumer-window-size**.
  If you have very fast consumers, you can increase the value of the **consumer-window-size** parameter to effectively disable consumer flow control.

- Use the core API instead of the JMS API.
  JMS operations must be translated into core operations before the server can handle them, resulting in lower performance than when you use the core API. When using the core API, try to use methods that take **SimpleString** as much as possible. **SimpleString**, unlike java.lang.String, does not require copying before it is written to the wire. Therefore, if you reuse **SimpleString** instances between calls, you can avoid some unnecessary copying. Note that the core API is not portable to other brokers.

19.6. AVOIDING ANTI PATTERNS

- Reuse connections, sessions, consumers, and producers where possible.
  The most common messaging anti-pattern is the creation of a new connection, session, and producer for every message sent or consumed. These objects take time to create and might involve several network round trips, which is a poor use of resources.

  > **NOTE**
  >
  > Some popular libraries such as the Spring JMS Template use these anti-patterns. If you are using the Spring JMS Template, you might see poor performance. The Spring JMS Template can be used safely only on an application server which caches JMS sessions, for example, using Java Connector Architecture, and then only for sending messages. It cannot be used safely to consume messages synchronously, even on an application server.

- Avoid fat messages.
  Verbose formats such as XML take up significant space on the wire and performance suffers as a result. Avoid XML in message bodies if you can.

- Do not create temporary queues for each request.
  This common anti-pattern involves the temporary queue request-response pattern. With the temporary queue request-response pattern, a message is sent to a target, and a reply-to header is set with the address of a local temporary queue. When the recipient receives the message,

they process it and then send back a response to the address specified in the reply-to header. A common mistake made with this pattern is to create a new temporary queue on each message sent, which drastically reduces performance. Instead, the temporary queue should be reused for many requests.

- Do not use message-driven beans unless it is necessary.
  Using message-driven beans to consume messages is slower than consuming messages by using a simple JMS message consumer.

APPENDIX A. ACCEPTOR AND CONNECTOR CONFIGURATION PARAMETERS

The tables below detail some of the available parameters used to configure Netty network connections. Parameters and their values are appended to the URI of the connection string. See Configuring acceptors and connectors in network connections for more information. Each table lists the parameters by name and notes whether they can be used with acceptors or connectors or with both. You can use some parameters, for example, only with acceptors.

> **NOTE**
>
> All Netty parameters are defined in the class **org.apache.activemq.artemis.core.remoting.impl.netty.TransportConstants**. Source code is available for download on the customer portal.

Table A.1. Netty TCP parameters

Parameter	Use with...	Description
batchDelay	Both	Before writing packets to the acceptor or connector, the broker can be configured to batch up writes for a maximum of **batchDelay** milliseconds. This can increase overall throughput for very small messages. It does so at the expense of an increase in average latency for message transfer. The default value is **0** ms.
connectionsAllowed	Acceptors	Limits the number of connections that the acceptor allows. When this limit is reached, a DEBUG-level message is issued to the log and the connection is refused. The type of client in use determines what happens when the connection is refused. The default value is **-1**, which means that there is no limit to the number of connections that the acceptor allows.
directDeliver	Both	When a message arrives on the server and is delivered to waiting consumers, by default, the delivery is done on the same thread as that on which the message arrived. This gives good latency in environments with relatively small messages and a small number of consumers, but at the cost of overall throughput and scalability - especially on multi-core machines. If you want the lowest latency and a possible reduction in throughput then you can use the default value for **directDeliver**, which is **true**. If you are willing to take some small extra hit on latency but want the highest throughput set **directDeliver** to **false**.

Parameter	Use with...	Description
handshake-timeout	Acceptors	Prevents an unauthorized client to open a large number of connections and keep them open. Because each connection requires a file handle, it consumes resources that are then unavailable to other clients. This timeout limits the amount of time a connection can consume resources without having been authenticated. After the connection is authenticated, you can use resource limit settings to limit resource consumption. The default value is set to **10** seconds. You can set it to any other integer value. You can turn off this option by setting it to 0 or negative integer. After you edit the timeout value, you must restart the broker.
localAddress	Connectors	Specifies which local address the client will use when connecting to the remote address. This is typically used in the Application Server or when running Embedded to control which address is used for outbound connections. If the local-address is not set then the connector will use any local address available.
localPort	Connectors	Specifies which local port the client will use when connecting to the remote address. This is typically used in the Application Server or when running Embedded to control which port is used for outbound connections. If the default is used, which is 0, then the connector will let the system pick up an ephemeral port. Valid ports are 0 to 65535
nioRemotingThreads	Both	When configured to use NIO, the broker will by default use a number of threads equal to three times the number of cores (or hyper-threads) as reported by **Runtime.getRuntime().availableProcessors()** for processing incoming packets. If you want to override this value, you can set the number of threads by specifying this parameter. The default value for this parameter is **-1**, which means use the value derived from **Runtime.getRuntime().availableProcessors()** * 3.
tcpNoDelay	Both	If this is **true** then Nagle's algorithm will be disabled. This is a Java (client) socket option. The default value is **true**.
tcpReceiveBufferSize	Both	Determines the size of the TCP receive buffer in bytes. The default value is **32768**.

Parameter	Use with...	Description
tcpSendBufferSize	Both	Determines the size of the TCP send 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.

Table A.2. Netty HTTP parameters

Parameter	Use with...	Description
httpClientIdleTime	Acceptors	How long a client can be idle before sending an empty HTTP request to keep the connection alive.
httpClientIdleScanPeriod	Acceptors	How often, in milliseconds, to scan for idle clients.
httpEnabled	Acceptors	No longer required. With single port support the broker will now automatically detect if HTTP is being used and configure itself.
httpRequiresSessionId	Both	If **true** the client will wait after the first call to receive a session id. Used when an HTTP connector is connecting to a servlet acceptor. This configuration is not recommended.
httpResponseTime	Acceptors	How long the server can wait before sending an empty HTTP response to keep the connection alive.
httpServerScanPeriod	Acceptors	How often, in milliseconds, to scan for clients needing responses.

Table A.3. Netty TLS/SSL parameters

Parameter	Use with...	Description

Parameter	Use with...	Description
enabledCipherSuites	Both	Comma-separated list of cipher suites used for SSL communication. Specify the most secure cipher suite(s) supported by your client application. If you specify a comma-separated list of cipher suites that are common to both the broker and the client, or you do not specify any cipher suites, the broker and client mutually negotiate a cipher suite to use. If you do not know which cipher suites to specify, you can first establish a broker-client connection with your client running in debug mode to verify the cipher suites that are common to both the broker and the client. Then, configure **enabledCipherSuites** on the broker. The cipher suites available depend on the TLS protocol versions used by the broker and clients. If the default TLS protocol version changes after you upgrade the broker, you might need to select an earlier TLS protocol version to ensure that the broker and the clients can use a common cipher suite. For more information, see **enabledProtocols**.
enabledProtocols	Both	Comma-separated list of TLS protocol versions used for SSL communication. If you don't specify a TLS protocol version, the broker uses the JVM's default version. If the broker uses the JVM's default TLS protocol version and that version changes after you upgrade the broker, the TLS protocol versions used by the broker and clients might be incompatible. While it is recommended that you use the later TLS protocol version, you can specify an earlier version in **enabledProtocols** to interoperate with clients that do not support a newer TLS protocol version.
forceSSLParameters	Connectors	Controls whether any SSL settings that are set as parameters on the connector are used instead of JVM system properties (including both **javax.net.ssl** and AMQ Broker system properties) to configure the SSL context for this connector. Valid values are **true** or **false**. The default value is **false**.

Parameter	Use with...	Description
keyStorePassword	Both	When used on an acceptor this is the password for the server-side keystore. When used on a connector this is the password for the client-side keystore. This is only relevant for a connector if you are using two-way SSL (that is, mutual authentication). Although this value can be configured on the server, it is downloaded and used by the client. If the client needs to use a different password from that set on the server then it can override the server-side setting by either using the customary **javax.net.ssl.keyStorePassword** system property or the ActiveMQ-specific **org.apache.activemq.ssl.keyStorePassword** system property. The ActiveMQ-specific system property is useful if another component on client is already making use of the standard, Java system property.
keyStorePath	Both	When used on an acceptor this is the path to the SSL key store on the server which holds the server's certificates (whether self-signed or signed by an authority). When used on a connector this is the path to the client-side SSL key store which holds the client certificates. This is only relevant for a connector if you are using two-way SSL (that is, mutual authentication). Although this value is configured on the server, it is downloaded and used by the client. If the client needs to use a different path from that set on the server then it can override the server-side setting by either using the customary **javax.net.ssl.keyStore** system property or the ActiveMQ-specific **org.apache.activemq.ssl.keyStore** system property. The ActiveMQ-specific system property is useful if another component on client is already making use of the standard, Java system property.
keyStoreAlias	Both	When used on an acceptor, this is the alias for the key in the key store to present to the client when it connects. When used on a connector, this is the alias for the key in the key store to present to the server when the client connects to it. This is only relevant for a connector when using 2-way SSL, that is, mutual authentication.
needClientAuth	Acceptors	Tells a client connecting to this acceptor that two-way SSL is required. Valid values are **true** or **false**. The default value is **false**.
sslEnabled	Both	Must be **true** to enable SSL. The default value is **false**.

Parameter	Use with...	Description
sslAutoReload	Acceptors	If set to **true**, the broker checks for changes to the SSL configuration, such as the keystore and truststore paths, and automatically reloads the configuration when changes are detected. The default value is **false**. The interval at which the broker checks for changes is determined by the value of the **configuration-file-refresh-period** element. For more information, see Reloading configuration updates.
trustManagerFactoryPlugin	Both	Defines the name of the class that implements **org.apache.activemq.artemis.api.core.TrustManagerFactoryPlugin**. This is a simple interface with a single method that returns a **javax.net.ssl.TrustManagerFactory**. The **TrustManagerFactory** is used when the underlying **javax.net.ssl.SSLContext** is initialized. This allows fine-grained customization of who or what the broker and client trust. The value of **trustManagerFactoryPlugin** takes precedence over all other SSL parameters that apply to the trust manager (that is, **trustAll**, **truststoreProvider**, **truststorePath**, **truststorePassword**, and **crlPath**). You need to place any specified plugin on the Java classpath of the broker. You can use the ***<broker_instance_dir>*/lib** directory, since it is part of the classpath by default.
trustStorePassword	Both	When used on an acceptor this is the password for the server-side trust store. This is only relevant for an acceptor if you are using two-way SSL (that is, mutual authentication). When used on a connector this is the password for the client-side truststore. Although this value can be configured on the server, it is downloaded and used by the client. If the client needs to use a different password from that set on the server then it can override the server-side setting by either using the customary **javax.net.ssl.trustStorePassword** system property or the ActiveMQ-specific **org.apache.activemq.ssl.trustStorePassword** system property. The ActiveMQ-specific system property is useful if another component on client is already making use of the standard, Java system property.

Parameter	Use with...	Description
sniHost	Both	When used on an acceptor, **sniHost** is a regular expression used to match the **server_name** extension on incoming SSL connections (for more information about this extension, see https://tools.ietf.org/html/rfc6066). If the name doesn't match, then the connection to the acceptor is rejected. A WARN message is logged if this happens. If the incoming connection doesn't include the **server_name** extension, then the connection is accepted. When used on a connector, the **sniHost** value is used for the **server_name** extension on the SSL connection.
sslProvider	Both	Used to change the SSL provider between **JDK** and **OPENSSL**. The default is **JDK**. If set to **OPENSSL**, you can add **netty-tcnative** to your classpath to use the natively-installed OpenSSL. This option can be useful if you want to use special ciphersuite-elliptic curve combinations that are supported through OpenSSL but not through the JDK provider.
trustStorePath	Both	When used on an acceptor this is the path to the server-side SSL key store that holds the keys of all the clients that the server trusts. This is only relevant for an acceptor if you are using two-way SSL (that is, mutual authentication). When used on a connector this is the path to the client-side SSL key store which holds the public keys of all the servers that the client trusts. Although this value can be configured on the server, it is downloaded and used by the client. If the client needs to use a different path from that set on the server then it can override the server-side setting by either using the customary **javax.net.ssl.trustStore** system property or the ActiveMQ-specific **org.apache.activemq.ssl.trustStore** system property. The ActiveMQ-specific system property is useful if another component on client is already making use of the standard, Java system property.
useDefaultSslContext	Connector	Allows the connector to use the "default" SSL context (via **SSLContext.getDefault()**), which can be set programmatically by the client (via **SSLContext.setDefault(SSLContext)**). If this parameter is set to **true**, all other SSL-related parameters except for **sslEnabled** are ignored. Valid values are **true** or **false**. The default value is **false**.

Parameter	Use with...	Description
verifyHost	Both	When used on a connector, the CN or Subject Alternative Name values of the server's SSL certificate are compared to the hostname being connected to in order to verify that they match. This is useful for both one-way and two-way SSL. When used on an acceptor, the CN or Subject Alternative Name values of the connecting client's SSL certificate are compared to its hostname to verify that they match. This is useful only for two-way SSL. Valid values are **true** or **false**. The default value is **true** for connectors and **false** for acceptors.
wantClientAuth	Acceptors	Tells a client connecting to this acceptor that two-way SSL is requested, but not required. Valid values are **true** or **false**. The default value is **false**. If the property **needClientAuth** is set to **true**, then that property takes precedence and **wantClientAuth** is ignored.

APPENDIX B. ADDRESS SETTING CONFIGURATION ELEMENTS

The table below lists all of the configuration elements of an **address-setting**. Note that some elements are marked DEPRECATED. Use the suggested replacement to avoid potential issues.

Table B.1. Address setting elements

Name	Description
address-full-policy	Determines what happens when an address configured with a **max-size-bytes** becomes full. The available policies are: **PAGE**: messages sent to a full address will be paged to disk. **DROP**: messages sent to a full address will be silently dropped. **FAIL**: messages sent to a full address will be dropped and the message producers will receive an exception. **BLOCK**: message producers will block when they try and send any further messages. **NOTE** The BLOCK policy works only for the AMQP, OpenWire, and Core Protocol protocols because they feature flow control.
auto-create-addresses	Whether to automatically 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 a queue. The default value is **true**.
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 the parameter 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>**.
auto-create-jms-queues	DEPRECATED: Use **auto-create-queues** instead. Determines whether this broker should automatically create a JMS queue corresponding to the address settings match when a JMS producer or a consumer tries to use such a queue. The default value is **false**.
auto-create-jms-topics	DEPRECATED: Use **auto-create-queues** instead. Determines whether this broker should automatically create a JMS topic corresponding to the address settings match when a JMS producer or a consumer tries to use such a queue. The default value is **false**.

Name	Description
auto-create-queues	Whether to automatically create a queue when a client sends a message to or attempts to consume a message from a queue. The default value is **true**.
auto-delete-addresses	Whether to delete auto-created addresses when the broker no longer has any queues. The default value is **true**.
auto-delete-jms-queues	DEPRECATED: Use **auto-delete-queues** instead. Determines whether AMQ Broker should automatically delete auto-created JMS queues when they have no consumers and no messages. The default value is **false**.
auto-delete-jms-topics	DEPRECATED: Use **auto-delete-queues** instead. Determines whether AMQ Broker should automatically delete auto-created JMS topics when they have no consumers and no messages. The default value is **false**.
auto-delete-queues	Whether to delete auto-created queues when the queue has no consumers and no messages. The default value is **true**.
config-delete-addresses	When the configuration file is reloaded, this setting specifies how to handle an address (and its queues) that has been deleted from the configuration file. You can specify the following values: **OFF** (default) The address is not deleted when the configuration file is reloaded. **FORCE** The address and its queues are deleted when the configuration file is reloaded. If there are any messages in the queues, they are removed also.
config-delete-queues	When the configuration file is reloaded, this setting specifies how to handle queues that have been deleted from the configuration file. You can specify the following values: **OFF** (default) The queue is not deleted when the configuration file is reloaded. **FORCE** The queue is deleted when the configuration file is reloaded. If there are any messages in the queue, they are removed also.
dead-letter-address	The address to which the broker sends dead messages.
dead-letter-queue-prefix	Prefix that the broker applies to the name of an automatically-created dead letter queue. The default value is **DLQ.**
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).
default-address-routing-type	The routing-type used on auto-created addresses. The default value is **MULTICAST**.

Name	Description
default-max-consumers	The maximum number of consumers allowed on this queue at any one time. The default value is **200**.
default-purge-on-no-consumers	Whether to purge the contents of the queue once there are no consumers. The default value is **false**.
default-queue-routing-type	The routing-type used on auto-created queues. The default value is **MULTICAST**.
enable-metrics	Specifies whether a configured metrics plugin such as the Prometheus plugin collects metrics for a matching address or *set* of addresses. The default value is **true**.
expiry-address	The address that will receive expired messages.
expiry-delay	Defines the expiration time in milliseconds that will be used for messages using the default expiration time. The default value is **-1**, which is means no expiration time.
last-value-queue	Whether a queue uses only last values or not. The default value is **false**.
management-browse-page-size	How many messages a management resource can browse. The default value is **200**.
max-delivery-attempts	how many times to attempt to deliver a message before sending to dead letter address. The default is **10**.
max-redelivery-delay	Maximum value for the redelivery-delay, in milliseconds.
max-size-bytes	The maximum memory size for this address, specified in bytes. Used when the **address-full-policy** is **PAGING**, **BLOCK**, or **FAIL**, this value is specified in byte notation such as "K", "Mb", and "GB". The default value is **-1**, which denotes infinite bytes. This parameter is used to protect broker memory by limiting the amount of memory consumed by a particular address space. This setting does not represent the total amount of bytes sent by the client that are currently stored in broker address space. It is an estimate of broker memory utilization. This value can vary depending on runtime conditions and certain workloads. It is recommended that you allocate the maximum amount of memory that can be afforded per address space. Under typical workloads, the broker requires approximately 150% to 200% of the payload size of the outstanding messages in memory.
max-size-bytes-reject-threshold	Used when the **address-full-policy** is **BLOCK**. The maximum size, in bytes, that an address can reach before the broker begins to reject messages. Works in combination with **max-size-bytes** for the AMQP protocol only. The default value is **-1**, which means no limit.

Name	Description
message-counter-history-day-limit	How many days to keep a message counter history for this address. The default value is **0**.
page-size-bytes	The paging size in bytes. Also supports byte notation like **K**, **Mb**, and **GB**. The default value is **10485760** bytes, almost 10.5 MB.
redelivery-delay	The time, in milliseconds, to wait before redelivering a cancelled message. The default value is **0**.
redelivery-delay-multiplier	Multiplier to apply to the redelivery-delay parameter. The default value is **1.0**.
redistribution-delay	Defines how long to wait in milliseconds after the last consumer is closed on a queue before redistributing any messages. The default value is **-1**.
send-to-dla-on-no-route	When set to **true**, a message will be sent to the configured dead letter address if it cannot be routed to any queues. The default value is **false**.
slow-consumer-check-period	How often to check, in seconds, for slow consumers. The default value is **5**.
slow-consumer-policy	Determines what happens when a slow consumer is identified. Valid options are **KILL** or **NOTIFY**. **KILL** kills the consumer's connection, which impacts any client threads using that same connection. **NOTIFY** sends a **CONSUMER_SLOW** management notification to the client. The default value is **NOTIFY**.
slow-consumer-threshold	The minimum rate of message consumption allowed before a consumer is considered slow. Measured in messages-per-second. The default value is **-1**, which is unbounded.

APPENDIX C. CLUSTER CONNECTION CONFIGURATION ELEMENTS

The table below lists all of the configuration elements of a **cluster-connection**.

Table C.1. Cluster connection configuration elements

Name	Description
address	Each cluster connection applies only to addresses that match the value specified in the **address** field. If no address is specified, then all addresses will be load balanced. The **address** field also supports comma separated lists of addresses. Use exclude syntax, **!** to prevent an address from being matched. Below are some example addresses: **jms.eu** Matches all addresses starting with **jms.eu**. **!jms.eu** Matches all addresses except for those starting with **jms.eu** **jms.eu.uk,jms.eu.de** Matches all addresses starting with either **jms.eu.uk** or **jms.eu.de** **jms.eu,!jms.eu.uk** Matches all addresses starting with **jms.eu**, but not those starting with **jms.eu.uk** **NOTE** You should not have multiple cluster connections with overlapping addresses (for example, "europe" and "europe.news"), because the same messages could be distributed between more than one cluster connection, possibly resulting in duplicate deliveries.
call-failover-timeout	Use when a call is made during a failover attempt. The default is **-1**, or no timeout.
call-timeout	When a packet is sent over a cluster connection, and it is a blocking call, **call-timeout** determines how long the broker will wait (in milliseconds) for the reply before throwing an exception. The default is **30000**.
check-period	The interval, in milliseconds, between checks to see if the cluster connection has failed to receive pings from another broker. The default is **30000**.
confirmation-window-size	The size, in bytes, of the window used for sending confirmations from the broker connected to. When the broker receives **confirmation-window-size** bytes, it notifies its client. The default is **1048576**. A value of **-1** means no window.

Name	Description
connector-ref	Identifies the **connector** that will be transmitted to other brokers in the cluster so that they have the correct cluster topology. This parameter is mandatory.
connection-ttl	Determines how long a cluster connection should stay alive if it stops receiving messages from a specific broker in the cluster. The default is **60000**.
discovery-group-ref	Points to a **discovery-group** to be used to communicate with other brokers in the cluster. This element must include the attribute **discovery-group-name**, which must match the **name** attribute of a previously configured **discovery-group**.
initial-connect-attempts	Sets the number of times the system will try to connect a broker in the cluster initially. If the max-retry is achieved, this broker will be considered permanently down, and the system will not route messages to this broker. The default is **-1**, which means infinite retries.
max-hops	Configures the broker to load balance messages to brokers which might be connected to it only indirectly with other brokers as intermediates in a chain. This allows for more complex topologies while still providing message load-balancing. The default value is **1**, which means messages are distributed only to other brokers directly connected to this broker. This parameter is optional.
max-retry-interval	The maximum delay for retries, in milliseconds. The default is **2000**.
message-load-balancing	Determines whether and how messages will be distributed between other brokers in the cluster. Include the **message-load-balancing** element to enable load balancing. The default value is **ON_DEMAND**. You can provide a value as well. Valid values are: OFF 　Disables load balancing. STRICT 　Enable load balancing and forwards messages to all brokers that have a matching queue, whether or not the queue has an active consumer or a matching selector. ON_DEMAND 　Enables load balancing and ensures that messages are forwarded only to brokers that have active consumers with a matching selector. OFF_WITH_REDISTRIBUTION 　Disables load balancing but ensures that messages are forwarded only to brokers that have active consumers with a matching selector when no suitable local consumer is available.

Name	Description
min-large-message-size	If a message size, in bytes, is larger than **min-large-message-size**, it will be split into multiple segments when sent over the network to other cluster members. The default is **102400**.
notification-attempts	Sets how many times the cluster connection should broadcast itself when connecting to the cluster. The default is **2**.
notification-interval	Sets how often, in milliseconds, the cluster connection should broadcast itself when attaching to the cluster. The default is **1000**.
producer-window-size	The size, in bytes, for producer flow control over cluster connection. By default, it is disabled, but you may want to set a value if you are using really large messages in cluster. A value of **-1** means no window.
reconnect-attempts	Sets the number of times the system will try to reconnect to a broker in the cluster. If the max-retry is achieved, this broker will be considered permanently down and the system will stop routing messages to this broker. The default is **-1**, which means infinite retries.
retry-interval	Determines the interval, in milliseconds, between retry attempts. If the cluster connection is created and the target broker has not been started or is booting, then the cluster connections from other brokers will retry connecting to the target until it comes back up. This parameter is optional. The default value is 500 milliseconds.
retry-interval-multiplier	The multiplier used to increase the **retry-interval** after each reconnect attempt. The default is 1.
use-duplicate-detection	Cluster connections use bridges to link the brokers, and bridges can be configured to add a duplicate ID property in each message that is forwarded. If the target broker of the bridge crashes and then recovers, messages might be resent from the source broker. By setting **use-duplicate-detection** to **true**, any duplicate messages will be filtered out and ignored on receipt at the target broker. The default is **true**.

APPENDIX D. COMMAND-LINE TOOLS

AMQ Broker includes a set of command-line interface (CLI) tools so you can manage your messaging journal. The table below lists the name for each tool and its description.

Table D.1. Command-line tools

Tool	Description
exp	Exports the message data using a special and independent XML format.
imp	Imports the journal to a running broker using the output provided by **exp**.
data	Prints reports about journal records and compacts their data.
encode	Shows an internal format of the journal encoded to String.
decode	Imports the internal journal format from encode.

For a full list of commands available for each tool, use the **help** parameter followed by the tool's name. In the example below, the CLI output lists all the commands available to the **data** tool after the user entered the command **./artemis help data**.

```
$ ./artemis help data

NAME
        artemis data - data tools group
        (print|imp|exp|encode|decode|compact) (example ./artemis data print)

SYNOPSIS
        artemis data
        artemis data compact [--broker <brokerConfig>] [--verbose]
              [--paging <paging>] [--journal <journal>]
              [--large-messages <largeMessges>] [--bindings <binding>]
        artemis data decode [--broker <brokerConfig>] [--suffix <suffix>]
              [--verbose] [--paging <paging>] [--prefix <prefix>] [--file-size <size>]
              [--directory <directory>] --input <input> [--journal <journal>]
              [--large-messages <largeMessges>] [--bindings <binding>]
        artemis data encode [--directory <directory>] [--broker <brokerConfig>]
              [--suffix <suffix>] [--verbose] [--paging <paging>] [--prefix <prefix>]
              [--file-size <size>] [--journal <journal>]
              [--large-messages <largeMessges>] [--bindings <binding>]
        artemis data exp [--broker <brokerConfig>] [--verbose]
              [--paging <paging>] [--journal <journal>]
              [--large-messages <largeMessges>] [--bindings <binding>]
        artemis data imp [--host <host>] [--verbose] [--port <port>]
              [--password <password>] [--transaction] --input <input> [--user <user>]
        artemis data print [--broker <brokerConfig>] [--verbose]
              [--paging <paging>] [--journal <journal>]
              [--large-messages <largeMessges>] [--bindings <binding>]

COMMANDS
        With no arguments, Display help information
```

```
    print
        Print data records information (WARNING: don't use while a
        production server is running)


    ...
```

You can use the help at the tool for more information on how to execute each of the tool's commands. For example, the CLI lists more information about the **data print** command after the user enters the **./artemis help data print**.

```
$ ./artemis help data print

NAME
        artemis data print - Print data records information (WARNING: don't use
        while a production server is running)

SYNOPSIS
        artemis data print [--bindings <binding>] [--journal <journal>]
              [--paging <paging>]

OPTIONS
        --bindings <binding>
            The folder used for bindings (default ../data/bindings)

        --journal <journal>
            The folder used for messages journal (default ../data/journal)

        --paging <paging>
            The folder used for paging (default ../data/paging)
```

APPENDIX E. MESSAGING JOURNAL CONFIGURATION ELEMENTS

The table below lists all of the configuration elements related to the AMQ Broker messaging journal.

Table E.1. Journal configuration elements

Name	Description
journal-directory	The directory where the message journal is located. The default value is ***<broker_instance_dir>*/data/journal**. For the best performance, the journal should be located on its own physical volume in order to minimize disk head movement. If the journal is on a volume that is shared with other processes that may be writing other files (for example, bindings journal, database, or transaction coordinator) then the disk head may well be moving rapidly between these files as it writes them, thus drastically reducing performance. When using a SAN, each journal instance should be given its own LUN (logical unit).
create-journal-dir	If set to **true**, the journal directory will be automatically created at the location specified in **journal-directory** if it does not already exist. The default value is **true**.
journal-type	Valid values are **NIO** or **ASYNCIO**. If set to **NIO**, the broker uses Java NIO interface to itsjournal. Set to **ASYNCIO**, and the broker will use the Linux asynchronous IO journal. If you choose **ASYNCIO** but are not running Linux or you do not have libaio installed then the broker will detect this and automatically fall back to using **NIO**.
journal-sync-transactional	If set to **true**, the broker flushes all transaction data to disk on transaction boundaries (that is, commit, prepare, and rollback). The default value is **true**.
journal-sync-non-transactional	If set to **true**, the broker flushes non-transactional message data (sends and acknowledgements) to disk each time. The default value is **true**.
journal-file-size	The size of each journal file in bytes. The default value is **10485760** bytes (10MiB).
journal-min-files	The minimum number of files the broker pre-creates when starting. Files are pre-created only if there is no existing message data. Depending on how much data you expect your queues to contain at steady state, you should tune this number of files to match the total amount of data expected.

Name	Description
journal-pool-files	The system will create as many files as needed; however, when reclaiming files it will shrink back to **journal-pool-files**. The default value is **-1**, meaning it will never delete files on the journal once created. The system cannot grow infinitely, however, as you are still required to use paging for destinations that can grow indefinitely.
journal-max-io	Controls the maximum number of write requests that can be in the IO queue at any one time. If the queue becomes full then writes will block until space is freed up. When using NIO, this value should always be **1**. When using AIO, the default value is **500**. The total max AIO can't be higher than the value set at the OS level (**/proc/sys/fs/aio-max-nr**), which is usually at 65536.
journal-buffer-timeout	Controls the timeout for when the buffer will be flushed. AIO can typically withstand with a higher flush rate than NIO, so the system maintains different default values for both NIO and AIO. The default value for NIO is **3333333** nanoseconds, or 300 times per second, and the default value for AIO is **50000** nanoseconds, or 2000 times per second. **NOTE** By increasing the timeout value, you might be able to increase system throughput at the expense of latency, since the default values are chosen to give a reasonable balance between throughput and latency.
journal-buffer-size	The size of the timed buffer on AIO. The default value is **490KiB**.
journal-compact-min-files	The minimal number of files necessary before the broker compacts the journal. The compacting algorithm will not start until you have at least **journal-compact-min-files**. The default value is **10**. **NOTE** Setting the value to **0** will disable compacting and could be dangerous because the journal could grow indefinitely.
journal-compact-percentage	The threshold to start compacting. Journal data will be compacted if less than **journal-compact-percentage** is determined to be live data. Note also that compacting will not start until you have at least **journal-compact-min-files** data files on the journal. The default value is **30**.

APPENDIX F. ADDITIONAL REPLICATION HIGH AVAILABILITY CONFIGURATION ELEMENTS

The following table lists additional **ha-policy** configuration elements that are not described in the Configuring replication high availability section. These elements have default settings that are sufficient for most common use cases.

Table F.1. Additional configuration elements for replication high availability

Name	Used in	Description
check-for-live-server	Embedded broker coordination	Applies only to brokers configured as master brokers. Specifies whether the original master broker checks the cluster for another live broker using its own server ID when starting up. Set to **true** to fail back to the original master broker and avoid a "split brain" situation in which two brokers become live at the same time. The default value of this property is **false**.
cluster-name	Embedded broker and ZooKeeper coordination	Name of the cluster configuration to use for replication. This setting is only necessary if you configure multiple cluster connections. If configured, the cluster configuration with this name will be used when connecting to the cluster. If unset, the first cluster connection defined in the configuration is used.
initial-replication-sync-timeout	Embedded broker and ZooKeeper coordination	The amount of time the replicating broker will wait upon completion of the initial replication process for the replica to acknowledge that it has received all the necessary data. The default value of this property is **30,000** milliseconds. NOTE: During this interval, any other journal-related operations are blocked.
max-saved-replicated-journals-size	Embedded broker and ZooKeeper coordination	Applies to backup brokers only. Specifies how many backup journal files the backup broker retains. Once this value has been reached, the broker makes space for each new backup journal file by deleting the oldest journal file. The default value of this property is **2**.

APPENDIX G. BROKER PROPERTIES

The following is a list of AMQ Broker properties which can be applied directly to the internal java configuration bean instead of using the XML configuration.

criticalAnalyzerCheckPeriod

Type: long
Default: 0
XML name: critical-analyzer-check-period
Description: The period defaults to half the critical-analyzer-timeout and is calculated at runtime.

pageMaxConcurrentIO

Type: int
Default: 5
XML name: page-max-concurrent-io
Description: The maximum number of concurrent reads allowed during paging.

messageCounterSamplePeriod

Type: long
Default: 10000
XML name: message-counter-sample-period
Description: The sample period (in ms) to use for message counters.

networkCheckNIC

Type: String
Default:
XML name: network-check-nic
Description: The network interface card name to be used to validate the address.

globalMaxSize

Type: long
Default: -1
XML name: global-max-size
Description: Size (in bytes) before all addresses will enter into their Full Policy configured upon messages being produced. Supports byte notation like "K", "Mb", "MiB", "GB", etc.

journalFileSize

Type: int
Default: 10485760
XML name: journal-file-size
Description: The size (in bytes) of each journal file. Supports byte notation, for example: "K", "Mb", "MiB", "GB".

configurationFileRefreshPeriod

Type: long
Default: 5000
XML name: configuration-file-refresh-period
Description: The frequency (in ms) to check the configuration file for modifications.

diskScanPeriod

Type: int
Default: 5000
XML name: disk-scan-period
Description: The frequency, in milliseconds, to scan the disks for full disks.

journalRetentionDirectory

Type: String
Default:
XML name: journal-retention-directory
Description: The directory in which to store journal-retention messages and the retention configuration.

networkCheckPeriod

Type: long
Default: 10000
XML name: network-check-period
Description: The frequency, in milliseconds, at which to check if the network is up.

journalBufferSize_AIO

Type: int
Default: 501760
XML name: journal-buffer-size
Description: The size, in bytes, of the internal buffer on the journal. Supports byte notation, for example, "K", "Mb", "MiB", "GB".

networkCheckURLList

Type: String
Default:
XML name: network-check-URL-list
Description: A comma separated list of URLs to use to validate if the broker should be kept up.

networkCheckTimeout

Type: int
Default: 1000
XML name: network-check-timeout
Description: The timeout, in milliseconds, to be used on the ping.

pageSyncTimeout

Type: int
Default:
XML name: page-sync-timeout
Description: The timeout, in nanoseconds, used to sync pages. The exact default value depend on whether the journal is ASYNCIO or NIO.

journalPoolFiles

Type: int
Default: -1
XML name: journal-pool-files
Description: The number of journal files to pre-create.

criticalAnalyzer

Type: boolean
Default: true
XML name: critical-analyzer
Description: Should analyze response time on critical paths and decide for broker log, shutdown or halt.

readWholePage

Type: boolean
Default: false
XML name: read-whole-page
Description: Specifies whether the whole page is read while getting message after page cache is evicted.

maxDiskUsage

Type: int
Default: 90
XML name: max-disk-usage
Description: The maximum percentage of disk usage before the system blocks or fails clients.

globalMaxMessages

Type: long
Default: -1
XML name: global-max-messages
Description: Number of messages before all addresses will enter into their address full policy configured. It works in conjunction with global-max-size, with the configured address fully policy being executed when either limit is reached.

internalNamingPrefix

Type: String
Default:
XML name: internal-naming-prefix
Description: Artemis uses internal queues and addresses to implement certain behaviors. These queues and addresses are prefixed by default with "$.activemq.internal" to avoid naming clashes with user namespacing. This can be overridden by setting this value to a valid Artemis address.

journalFileOpenTimeout

Type: int
Default: 5
XML name: journal-file-open-timeout
Description: The time, in seconds, to wait when opening a new Journal file before timing out and failing.

journalCompactPercentage

Type: int
Default: 30
XML name: journal-compact-percentage
Description: The percentage of live data on which to consider compacting the journal.

createBindingsDir

Type: boolean
Default: true
XML name: create-bindings-dir

Description: A value of **true** causes the server to create the bindings directory at startup.

suppressSessionNotifications

Type: boolean
Default: false
XML name: suppress-session-notifications
Description: Whether or not to suppress SESSION_CREATED and SESSION_CLOSED notifications. Set to **true** to reduce notification overhead. However, these are required to enforce unique client ID utilization in a cluster for MQTT clients.

journalBufferTimeout_AIO

Type: int
Default:
XML name: journal-buffer-timeout
Description: The timeout,in nanoseconds, used to flush internal buffers on the journal. The exact default value depends on whether the journal is ASYNCIO or NIO.

journalType

Type: JournalType
Default: ASYNCIO
XML name: journal-type
Description: The type of journal to use.

name

Type: String
Default:
XML name: name
Description: Node name. If set, it is used in topology notifications.

networkCheckPingCommand

Type: String
Default:
XML name: network-check-ping-command
Description: The ping command used to ping IPV4 addresses.

temporaryQueueNamespace

Type: String
Default:
XML name: temporary-queue-namespace
Description: The namespace to use for looking up address settings for temporary queues.

pagingDirectory

Type: String
Default: data/paging
XML name: paging-directory
Description: The directory in which to store paged messages.

journalDirectory

Type: String

Default: data/journal
XML name: journal-directory
Description: The directory in which store the journal files.

journalBufferSize_NIO

Type: int
Default: 501760
XML name: journal-buffer-size
Description: The size, in bytes, of the internal buffer on the journal. Supports byte notation, for example: "K", "Mb", "MiB", "GB".

journalDeviceBlockSize

Type: Integer
Default:
XML name: journal-device-block-size
Description: The size, in bytes, used by the device. This is usually translated as fstat/st_blksize and this is a way to bypass the value returned as st_blksize.

nodeManagerLockDirectory

Type: String
Default:
XML name: node-manager-lock-directory
Description: The directory in which to store the node manager lock file.

messageCounterMaxDayHistory

Type: int
Default: 10
XML name: message-counter-max-day-history
Description: The number of days to keep message counter history.

largeMessagesDirectory

Type: String
Default: data/largemessages
XML name: large-messages-directory
Description: The directory in which to store large messages.

networkCheckPing6Command

Type: String
Default:
XML name: network-check-ping6-command
Description: The ping command used to ping IPV6 addresses.

memoryWarningThreshold

Type: int
Default: 25
XML name: memory-warning-threshold
Description: Percentage of available memory at which a warning is generated.

mqttSessionScanInterval

Type: long
Default: 5000
XML name: mqtt-session-scan-interval
Description: The frequency, in milliseconds, at which to scan for expired MQTT sessions.

journalMaxAtticFiles

Type: int
Default:
XML name: journal-max-attic-files
Description:

journalSyncTransactional

Type: boolean
Default: true
XML name: journal-sync-transactional
Description: If set to **true**, waits for transaction data to be synchronized to the journal before returning a response to client.

logJournalWriteRate

Type: boolean
Default: false
XML name: log-journal-write-rate
Description: Specifies whether to log messages related to the journal write-rate.

journalMaxIO_AIO

Type: int
Default:
XML name: journal-max-io
Description: The maximum number of write requests that can be in the AIO queue at any one time. The default is **500** for AIO and **1** for NIO.

messageExpiryScanPeriod

Type: long
Default: 30000
XML name: message-expiry-scan-period
Description: The frequency, in milliseconds, to scan for expired messages.

criticalAnalyzerTimeout

Type: long
Default: 120000
XML name: critical-analyzer-timeout
Description: The default timeout used to analyze timeouts on the critical path.

messageCounterEnabled

Type: boolean
Default: false
XML name: message-counter-enabled
Description: A value of **true** means that message counters are enabled.

journalCompactMinFiles

Type: int
Default: 10
XML name: journal-compact-min-files
Description: The minimum number of data files before the broker starts to compact files.

createJournalDir

Type: boolean
Default: true
XML name: create-journal-dir
Description: A value of **true** means that the journal directory is created.

addressQueueScanPeriod

Type: long
Default: 30000
XML name: address-queue-scan-period
Description: The frequency, in milliseconds, to scan for addresses and queues that need to be deleted.

memoryMeasureInterval

Type: long
Default: -1
XML name: memory-measure-interval
Description: The frequency, in milliseconds, to sample JVM memory. A value of **-1** disables memory sampling.

journalSyncNonTransactional

Type: boolean
Default: true
XML name: journal-sync-non-transactional
Description: If **true**, waits for non transaction data to be synced to the journal before returning a response to the client.

connectionTtlCheckInterval

Type: long
Default: 2000
XML name: connection-ttl-check-interval
Description: The frequency, in milliseconds, to check connections for ttl violations.

rejectEmptyValidatedUser

Type: boolean
Default: false
XML name: reject-empty-validated-user
Description: If **true**, the server does not allow any message that does not have a validated user. In JMS, this is **JMSXUserID**.

journalMaxIO_NIO

Type: int
Default:
XML name: journal-max-io

Description: The maximum number of write requests that can be in the AIO queue at any one time. The default is **500** for AIO and **1** for NIO. Currently, broker properties only support using an integer and measures in bytes.

transactionTimeoutScanPeriod

Type: long
Default: 1000
XML name: transaction-timeout-scan-period
Description: The frequency, in milliseconds, to scan for timeout transactions.

systemPropertyPrefix

Type: String
Default:
XML name: system-property-prefix
Description: The prefix used to parse system properties for the configuration.

transactionTimeout

Type: long
Default: 300000
XML name: transaction-timeout
Description: The duration, in milliseconds, before a transaction can be removed from the resource manager after the create time.

journalLockAcquisitionTimeout

Type: long
Default: -1
XML name: journal-lock-acquisition-timeout
Description: The frequency, in milliseconds, to wait to acquire a file lock on the journal.

journalBufferTimeout_NIO

Type: int
Default:
XML name: journal-buffer-timeout
Description: The timeout, in nanoseconds, used to flush internal buffers on the journal. The exact default value depends on whether the journal is ASYNCIO or NIO.

journalMinFiles

Type: int
Default: 2
XML name: journal-min-files
Description: The number of journal files to pre-create.

G.1. BRIDGECONFIGURATIONS

bridgeConfigurations.<_name_>.retryIntervalMultiplier

Type: double
Default: 1
XML name: retry-interval-multiplier
Description: The multiplier to apply to successive retry intervals.

bridgeConfigurations.*<name>*.maxRetryInterval

Type: long
Default: 2000
XML name: max-retry-interval
Description: The limit to the retry-interval growth, due to retry-interval-multiplier.

bridgeConfigurations.*<name>*.filterString

Type: String
Default:
XML name: filter-string
Description:

bridgeConfigurations.*<name>*.connectionTTL

Type: long
Default: 60000
XML name: connection-ttl
Description: The duration to keep a connection alive if no data is received from the client. The duration should be greater than the ping period.

bridgeConfigurations.*<name>*.confirmationWindowSize

Type: int
Default: 1048576
XML name: confirmation-window-size
Description: The number of bytes received after which the bridge sends a confirmation. Supports byte notation, for example, "K", "Mb", "MiB", "GB".

bridgeConfigurations.*<name>*.staticConnectors

Type: List
Default:
XML name: static-connectors
Description:

bridgeConfigurations.*<name>*.reconnectAttemptsOnSameNode

Type: int
Default:
XML name: reconnect-attempts-on-same-node
Description:

bridgeConfigurations.*<name>*.concurrency

Type: int
Default: 1
XML name: concurrency
Description: The number of concurrent workers. More workers can help increase throughput on high latency networks. The default is **1**.

bridgeConfigurations.*<name>*.transformerConfiguration

Type: TransformerConfiguration
Default:
XML name: transformer-configuration

Description:

bridgeConfigurations.<*name*>.transformerConfiguration.className

Type: String
Default:
XML name: class-name
Description:

bridgeConfigurations.<*name*>.transformerConfiguration.properties

Type: Map
Default:
XML name: property
Description: A KEY/VALUE pair to set on the transformer, for example,
properties.MY_PROPERTY=MY_VALUE

bridgeConfigurations.<*name*>.password

Type: String
Default:
XML name: password
Description: If unspecified, the cluster-password is used.

bridgeConfigurations.<*name*>.queueName

Type: String
Default:
XML name: queue-name
Description: The name of the queue from which this bridge consumes.

bridgeConfigurations.<*name*>.forwardingAddress

Type: String
Default:
XML name: forwarding-address
Description: Address to forward to. If omitted, the original address is used.

bridgeConfigurations.<*name*>.routingType

Type: ComponentConfigurationRoutingType
Default: PASS
XML name: routing-type
Description: How the routing-type on the bridged messages is set.

bridgeConfigurations.<*name*>.name

Type: String
Default:
XML name: name
Description: A unique name for this bridge.

bridgeConfigurations.<*name*>.ha

Type: boolean
Default: false
XML name: ha

Description: Specifies whether this bridge supports fail-over.

bridgeConfigurations.<*name*>.initialConnectAttempts

Type: int
Default: -1
XML name: initial-connect-attempts
Description: The maximum number of initial connection attempts. The default value of **-1** means there is no limit.

bridgeConfigurations.<*name*>.retryInterval

Type: long
Default: 2000
XML name: retry-interval
Description: The interval, in milliseconds, between successive retries.

bridgeConfigurations.<*name*>.producerWindowSize

Type: int
Default: 1048576
XML name: producer-window-size
Description: Producer flow control. Supports byte notation, for example: "K", "Mb", "MiB", "GB".

bridgeConfigurations.<*name*>.clientFailureCheckPeriod

Type: long
Default: 30000
XML name: check-period
Description: The interval, in milliseconds, at which a bridge's client checks if it failed to receive a ping from the server. Specify a value of **-1** to disable this check.

bridgeConfigurations.<*name*>.discoveryGroupName

Type: String
Default:
XML name: discovery-group-ref
Description:

bridgeConfigurations.<*name*>.user

Type: String
Default:
XML name: user
Description: Username. If unspecified, the cluster-user is used.

bridgeConfigurations.<*name*>.useDuplicateDetection

Type: boolean
Default: true
XML name: use-duplicate-detection
Description: Specifies whether duplicate detection headers are inserted in forwarded messages.

bridgeConfigurations.<*name*>.minLargeMessageSize

Type: int
Default: 102400

XML name: min-large-message-size
Description: The size, in bytes, above which a message is considered a large message. Large messages are sent over the network in multiple segments. Supports byte notation, for example, "K", "Mb", "MiB", "GB".

G.2. AMQPCONNECTIONS

AMQPConnections.<*name*>.reconnectAttempts

Type: int
Default: -1
XML name: reconnect-attempts
Description: The number of reconnection attempts after a failure.

AMQPConnections.<*name*>.password

Type: String
Default:
XML name: password
Description: The password used to connect. If not specified, an anonymous connection is attempted.

AMQPConnections.<*name*>.retryInterval

Type: int
Default: 5000
XML name: retry-interval
Description: The interval, in milliseconds, between successive retries.

AMQPConnections.<*name*>.connectionElements

Type: AMQPMirrorBrokerConnectionElement
Default:
XML name: amqp-connection
Description: An AMQP Broker Connection supports 4 types: 1. Mirrors - The broker uses an AMQP connection to another broker and duplicates messages and sends acknowledgments over the wire. 2. Senders - Messages received on specific queues are transferred to another endpoint. 3. Receivers - The broker pulls messages from another endpoint. 4. Peers - The broker creates both senders and receivers on another endpoint that knows how to handle them. This is currently implemented by Apache Qpid Dispatch. Currently, only mirror type is supported.

AMQPConnections.<*name*>.connectionElements.<*name*>.messageAcknowledgments

Type: boolean
Default:
XML name: message-acknowledgments
Description: If **true**, message acknowledgments are mirrored.

AMQPConnections.<*name*>.connectionElements.<*name*>.queueRemoval

Type: boolean
Default:
XML name: queue-removal
Description: Specifies whether the mirror queue deletes events for addresses and queues.

AMQPConnections.<*name*>.connectionElements.<*name*>.addressFilter

Type: String
Default:
XML name: address-filter
Description: Specifies a filter that the mirror uses to determine which events are forwarded towards to the target server based on source address.

AMQPConnections.<*name*>.connectionElements.<*name*>.queueCreation

Type: boolean
Default:
XML name: queue-creation
Description: Specifies whether the mirror queue creates events for addresses and queues.

AMQPConnections.<*name*>.autostart

Type: boolean
Default: true
XML name: auto-start
Description: Specifies whether the broker connection is started when the server is started.

AMQPConnections.<*name*>.user

Type: String
Default:
XML name: user
Description: User name used to connect. If not specified, an anonymous connection is attempted.

AMQPConnections.<*name*>.uri

Type: String
Default:
XML name: uri
Description: The URI of the AMQP connection.

G.3. DIVERTCONFIGURATION

divertConfigurations.<*name*>.transformerConfiguration

Type: TransformerConfiguration
Default:
XML name: transformer-configuration
Description:

divertConfigurations.<*name*>.transformerConfiguration.className

Type: String
Default:
XML name: class-name
Description:

divertConfigurations.<*name*>.transformerConfiguration.properties

Type: Map
Default:
XML name: property
Description: A KEY/VALUE pair to set on the transformer, for example, ...
properties.MY_PROPERTY=MY_VALUE.

divertConfigurations.<*name*>.filterString

Type: String
Default:
XML name: filter-string
Description:

divertConfigurations.<*name*>.routingName

Type: String
Default:
XML name: routing-name
Description: The routing name for the divert.

divertConfigurations.<*name*>.address

Type: String
Default:
XML name: address
Description: The address this divert will divert from.

divertConfigurations.<*name*>.forwardingAddress

Type: String
Default:
XML name: forwarding-address
Description: The forwarding address for the divert.

divertConfigurations.<*name*>.routingType

Type: ComponentConfigurationRoutingType(MULTICAST ANYCAST STRIP PASS)
Default:
XML name: routing-type
Description: How the routing-type on the diverted messages is set.

divertConfigurations.<*name*>.exclusive

Type: boolean
Default: false
XML name: exclusive
Description: Specifies whether this is an exclusive divert.

G.4. ADDRESSSETTINGS

addressSettings.<*address*>.configDeleteDiverts

Type: DeletionPolicy(OFF FORCE)
Default:
XML name: config-delete-addresses
Description:

addressSettings.<*address*>.expiryQueuePrefix

Type: SimpleString
Default:
XML name: expiry-queue-prefix
Description:

addressSettings.<*address*>.defaultConsumerWindowSize

Type: int
Default:
XML name: default-consumer-window-size
Description:

addressSettings.<*address*>.maxReadPageBytes

Type: int
Default:
XML name: max-read-page-bytes
Description:

addressSettings.<*address*>.deadLetterQueuePrefix

Type: SimpleString
Default:
XML name: dead-letter-queue-prefix
Description:

addressSettings.<*address*>.defaultGroupRebalancePauseDispatch

Type: boolean
Default:
XML name: default-group-rebalance-pause-dispatch
Description:

addressSettings.<*address*>.autoCreateAddresses

Type: Boolean
Default:
XML name: auto-create-addresses
Description:

addressSettings.<*address*>.slowConsumerThreshold

Type: long
Default:
XML name: slow-consumer-threshold
Description:

addressSettings.<*address*>.managementMessageAttributeSizeLimit

Type: int
Default:
XML name: management-message-attribute-size-limit
Description:

addressSettings.<*address*>.autoCreateExpiryResources

Type: boolean
Default:
XML name: auto-create-expiry-resources
Description:

addressSettings.<*address*>.pageSizeBytes

Type: int
Default:
XML name: page-size-bytes
Description:

addressSettings.<*address*>.minExpiryDelay

Type: Long
Default:
XML name: min-expiry-delay
Description:

addressSettings.<*address*>.defaultConsumersBeforeDispatch

Type: Integer
Default:
XML name: default-consumers-before-dispatch
Description:

addressSettings.<*address*>.expiryQueueSuffix

Type: SimpleString
Default:
XML name: expiry-queue-suffix
Description:

addressSettings.<*address*>.configDeleteQueues

Type: DeletionPolicy(OFF FORCE)
Default:
XML name: config-delete-queues
Description:

addressSettings.<*address*>.enableIngressTimestamp

Type: boolean
Default:
XML name: enable-ingress-timestamp
Description:

addressSettings.<*address*>.autoDeleteCreatedQueues

Type: Boolean
Default:
XML name: auto-delete-created-queues
Description:

addressSettings.<*address*>.expiryAddress

Type: SimpleString
Default:
XML name: expiry-address
Description:

addressSettings.<*address*>.managementBrowsePageSize

Type: int

Default:
XML name: management-browse-page-size
Description:

addressSettings.<*address*>.autoDeleteQueues

Type: Boolean
Default:
XML name: auto-delete-queues
Description:

addressSettings.<*address*>.retroactiveMessageCount

Type: long
Default:
XML name: retroactive-message-count
Description:

addressSettings.<*address*>.maxExpiryDelay

Type: Long
Default:
XML name: max-expiry-delay
Description:

addressSettings.<*address*>.maxDeliveryAttempts

Type: int
Default:
XML name: max-delivery-attempts
Description:

addressSettings.<*address*>.defaultGroupFirstKey

Type: SimpleString
Default:
XML name: default-group-first-key
Description:

addressSettings.<*address*>.slowConsumerCheckPeriod

Type: long
Default:
XML name: slow-consumer-check-period
Description:

addressSettings.<*address*>.defaultPurgeOnNoConsumers

Type: Boolean
Default:
XML name: default-purge-on-no-consumers
Description:

addressSettings.<*address*>.defaultLastValueKey

Type: SimpleString
Default:

XML name: default-last-value-key
Description:

addressSettings.*<address>*.autoCreateQueues

Type: Boolean
Default:
XML name: auto-create-queues
Description:

addressSettings.*<address>*.defaultExclusiveQueue

Type: Boolean
Default:
XML name: default-exclusive-queue
Description:

addressSettings.*<address>*.defaultMaxConsumers

Type: Integer
Default:
XML name: default-max-consumers
Description:

addressSettings.*<address>*.defaultQueueRoutingType

Type: RoutingType(MULTICAST ANYCAST)
Default:
XML name: default-queue-routing-type
Description:

addressSettings.*<address>*.messageCounterHistoryDayLimit

Type: int
Default:
XML name: message-counter-history-day-limit
Description:

addressSettings.*<address>*.defaultGroupRebalance

Type: boolean
Default:
XML name: default-group-rebalance
Description:

addressSettings.*<address>*.defaultAddressRoutingType

Type: RoutingType(MULTICAST ANYCAST)
Default:
XML name: default-address-routing-type
Description:

addressSettings.*<address>*.maxSizeBytesRejectThreshold

Type: long
Default:
XML name: max-size-bytes-reject-threshold

Description:

addressSettings.<*address*>.pageCacheMaxSize

Type: int
Default:
XML name: page-cache-max-size
Description:

addressSettings.<*address*>.autoCreateDeadLetterResources

Type: boolean
Default:
XML name: auto-create-dead-letter-resources
Description:

addressSettings.<*address*>.maxRedeliveryDelay

Type: long
Default:
XML name: max-redelivery-delay
Description:

addressSettings.<*address*>.configDeleteAddresses

Type: DeletionPolicy
Default:
XML name: config-delete-addresses
Description:

addressSettings.<*address*>.deadLetterAddress

Type: SimpleString
Default:
XML name: dead-letter-address
Description:

addressSettings.<*address*>.autoDeleteQueuesMessageCount

Type: long
Default:
XML name: auto-delete-queues-message-count
Description:

addressSettings.<*address*>.autoDeleteAddresses

Type: Boolean
Default:
XML name: auto-delete-addresses
Description:

addressSettings.<*address*>.addressFullMessagePolicy

Type: AddressFullMessagePolicy
Default:
XML name: address-full-policy
Description:

addressSettings.<*address*>.maxSizeBytes

Type: long
Default:
XML name: max-size-bytes
Description:

addressSettings.<*address*>.defaultDelayBeforeDispatch

Type: Long
Default:
XML name: default-delay-before-dispatch
Description:

addressSettings.<*address*>.redistributionDelay

Type: long
Default:
XML name: redistribution-delay
Description:

addressSettings.<*address*>.maxSizeMessages

Type: long
Default:
XML name: max-size-messages
Description:

addressSettings.<*address*>.redeliveryMultiplier

Type: double
Default:
XML name: redelivery-delay-multiplier
Description:

addressSettings.<*address*>.defaultRingSize

Type: long
Default:
XML name: default-ring-size
Description:

addressSettings.<*address*>.defaultLastValueQueue

Type: boolean
Default:
XML name: default-last-value-queue
Description:

addressSettings.<*address*>.slowConsumerPolicy

Type: SlowConsumerPolicy(KILL NOTIFY)
Default:
XML name: slow-consumer-policy
Description:

addressSettings.<*address*>.redeliveryCollisionAvoidanceFactor

Type: double
Default:
XML name: redelivery-collision-avoidance-factor
Description:

addressSettings.*<address>*.autoDeleteQueuesDelay

Type: long
Default:
XML name: auto-delete-queues-delay
Description:

addressSettings.*<address>*.autoDeleteAddressesDelay

Type: long
Default:
XML name: auto-delete-addresses-delay
Description:

addressSettings.*<address>*.expiryDelay

Type: Long
Default:
XML name: expiry-delay
Description:

addressSettings.*<address>*.enableMetrics

Type: boolean
Default:
XML name: enable-metrics
Description:

addressSettings.*<address>*.sendToDLAOnNoRoute

Type: boolean
Default:
XML name: send-to-d-l-a-on-no-route
Description:

addressSettings.*<address>*.slowConsumerThresholdMeasurementUnit

Type: SlowConsumerThresholdMeasurementUnit(MESSAGES_PER_SECOND
MESSAGES_PER_MINUTE MESSAGES_PER_HOUR MESSAGES_PER_DAY)
Default:
XML name: slow-consumer-threshold-measurement-unit
Description:

addressSettings.*<address>*.redeliveryDelay

Type: long
Default:
XML name: redelivery-delay
Description:

addressSettings.*<address>*.deadLetterQueueSuffix

Type: SimpleString
Default:
XML name: dead-letter-queue-suffix
Description:

addressSettings.<*address*>.defaultNonDestructive

Type: boolean
Default:
XML name: default-non-destructive
Description:

G.5. FEDERATIONCONFIGURATIONS

federationConfigurations.<*name*>.transformerConfigurations

Type: FederationTransformerConfiguration
Default:
XML name: transformer
Description: Optional transformer configuration.

federationConfigurations.<*name*>.transformerConfigurations.<*name*>.transformerConfiguration

Type: TransformerConfiguration
Default:
XML name: transformer
Description: Allows the addition of a custom transformer to amend the message.

federationConfigurations.<*name*>.transformerConfigurations.<*name*>.transformerConfiguration.<*name*>.className

Type: String
Default:
XML name: class-name
Description: The class name of the Transformer implementation.

federationConfigurations.<*name*>.transformerConfigurations.<*name*>.transformerConfiguration.<*name*>.properties

Type: Map
Default:
XML name: property
Description: A KEY/VALUE pair to set on the transformer, for example, … properties.MY_PROPERTY=MY_VALUE.

federationConfigurations.<*name*>.queuePolicies

Type: FederationQueuePolicyConfiguration
Default:
XML name: queue-policy
Description:

federationConfigurations.<*name*>.queuePolicies.<*name*>.priorityAdjustment

Type: Integer
Default:

XML name: priority-adjustment
Description: When a consumer attaches, its priority is used to create the upstream consumer, but with an adjustment so that local consumers are load balanced before remote consumers.

federationConfigurations.<*name*>.queuePolicies.<*name*>.excludes

Type: Matcher
Default:
XML name: exclude
Description: A list of queue matches to exclude.

federationConfigurations.<*name*>.queuePolicies.<*name*>.excludes.<*name*>.queueMatch

Type: String
Default:
XML name: queue-match
Description: A queue match pattern to apply. If none are present, all queues are matched.

federationConfigurations.<*name*>.queuePolicies.<*name*>.transformerRef

Type: String
Default:
XML name: transformer-ref
Description: The ref name for a transformer that you want to configure to transform the message on federation transfer.

federationConfigurations.<*name*>.queuePolicies.<*name*>.includes

Type: Matcher
Default:
XML name: queue-match
Description:

federationConfigurations.<*name*>.queuePolicies.<*name*>.excludes.<*name*>.queueMatch

Type: String
Default:
XML name: queue-match
Description: A queue match pattern to apply. If none are present, all queues are matched.

federationConfigurations.<*name*>.queuePolicies.<*name*>.includeFederated

Type: boolean
Default:
XML name: include-federated
Description: When the value 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.

federationConfigurations.<*name*>.upstreamConfigurations

Type: FederationUpstreamConfiguration
Default:
XML name: upstream
Description:

federationConfigurations.<*name*>.upstreamConfigurations.<*name*>.connectionConfiguration

Type: FederationConnectionConfiguration
Default:
XML name: connection-configuration
Description: The streams connection configuration.

**federationConfigurations.<*name*>.upstreamConfigurations.
<*name*>.connectionConfiguration.priorityAdjustment**

Type: int
Default:
XML name: priority-adjustment
Description:

**federationConfigurations.<*name*>.upstreamConfigurations.
<*name*>.connectionConfiguration.retryIntervalMultiplier**

Type: double
Default: 1
XML name: retry-interval-multiplier
Description: The multiplier to apply to the retry-interval.

**federationConfigurations.<*name*>.upstreamConfigurations.
<*name*>.connectionConfiguration.shareConnection**

Type: boolean
Default: false
XML name: share-connection
Description: If set to **true**, the same connection is shared if there is a downstream and upstream
connection configured for the same broker.

**federationConfigurations.<*name*>.upstreamConfigurations.
<*name*>.connectionConfiguration.maxRetryInterval**

Type: long
Default: 2000
XML name: max-retry-interval
Description: The maximum value for retry-interval.

**federationConfigurations.<*name*>.upstreamConfigurations.
<*name*>.connectionConfiguration.connectionTTL**

Type: long
Default:
XML name: connection-t-t-l
Description:

**federationConfigurations.<*name*>.upstreamConfigurations.
<*name*>.connectionConfiguration.circuitBreakerTimeout**

Type: long
Default: 30000
XML name: circuit-breaker-timeout
Description: Specifies if this connection supports fail-over.

federationConfigurations.<*name*>.upstreamConfigurations. <*name*>.connectionConfiguration.callTimeout

Type: long
Default: 30000
XML name: call-timeout
Description: The duration to wait for a reply.

federationConfigurations.<*name*>.upstreamConfigurations. <*name*>.connectionConfiguration.staticConnectors

Type: List
Default:
XML name: static-connectors
Description: A list of connector references configured via connectors.

federationConfigurations.<*name*>.upstreamConfigurations. <*name*>.connectionConfiguration.reconnectAttempts

Type: int
Default: -1
XML name: reconnect-attempts
Description: The number of reconnection attempts after a failure.

federationConfigurations.<*name*>.upstreamConfigurations. <*name*>.connectionConfiguration.password

Type: String
Default:
XML name: password
Description: A password. If not specified, the federated password is used.

federationConfigurations.<*name*>.upstreamConfigurations. <*name*>.connectionConfiguration.callFailoverTimeout

Type: long
Default: -1
XML name: call-failover-timeout
Description: The duration to wait for a reply during a fail-over. A value of **-1** means there is no limit.

federationConfigurations.<*name*>.upstreamConfigurations.<*name*>.connectionConfiguration.hA

Type: boolean
Default:
XML name: h-a
Description:

federationConfigurations.<*name*>.upstreamConfigurations. <*name*>.connectionConfiguration.initialConnectAttempts

Type: int
Default: -1
XML name: initial-connect-attempts
Description: The number of initial attempts to connect.

federationConfigurations.<*name*>.upstreamConfigurations. <*name*>.connectionConfiguration.retryInterval

Type: long
Default: 500
XML name: retry-interval
Description: The interval, in milliseconds, between successive retries.

**federationConfigurations.<*name*>.upstreamConfigurations.
<*name*>.connectionConfiguration.clientFailureCheckPeriod**

Type: long
Default:
XML name: client-failure-check-period
Description:

**federationConfigurations.<*name*>.upstreamConfigurations.
<*name*>.connectionConfiguration.username**

Type: String
Default:
XML name: username
Description:

federationConfigurations.<*name*>.upstreamConfigurations.<*name*>.policyRefs

Type: Collection
Default:
XML name: policy-refs
Description:

federationConfigurations.<*name*>.upstreamConfigurations.<*name*>.staticConnectors

Type: List
Default:
XML name: static-connectors
Description: A list of connector references configured via connectors.

federationConfigurations.<*name*>.downstreamConfigurations

Type: FederationDownstreamConfiguration
Default:
XML name: downstream
Description:

federationConfigurations.<*name*>.downstreamConfigurations.<*name*>.connectionConfiguration

Type: FederationConnectionConfiguration
Default:
XML name: connection-configuration
Description: The streams connection configuration.

**federationConfigurations.<*name*>.downstreamConfigurations.
<*name*>.connectionConfiguration.priorityAdjustment**

Type: int
Default:
XML name: priority-adjustment
Description:

**federationConfigurations.<*name*>.downstreamConfigurations.
<*name*>.connectionConfiguration.retryIntervalMultiplier**

Type: double
Default: 1
XML name: retry-interval-multiplier
Description: The multiplier to apply to the retry-interval.

**federationConfigurations.<*name*>.downstreamConfigurations.
<*name*>.connectionConfiguration.shareConnection**

Type: boolean
Default: false
XML name: share-connection
Description: If set to **true**, the same connection is shared if there is a downstream and upstream connection configured for the same broker.

**federationConfigurations.<*name*>.downstreamConfigurations.
<*name*>.connectionConfiguration.maxRetryInterval**

Type: long
Default: 2000
XML name: max-retry-interval
Description: The maximum value for the retry-interval.

**federationConfigurations.<*name*>.downstreamConfigurations.
<*name*>.connectionConfiguration.connectionTTL**

Type: long
Default:
XML name: connection-t-t-l
Description:

**federationConfigurations.<*name*>.downstreamConfigurations.
<*name*>.connectionConfiguration.circuitBreakerTimeout**

Type: long
Default: 30000
XML name: circuit-breaker-timeout
Description: Specifies whether this connection supports fail-over.

**federationConfigurations.<*name*>.downstreamConfigurations.
<*name*>.connectionConfiguration.callTimeout**

Type: long
Default: 30000
XML name: call-timeout
Description: The duration to wait for a reply.

**federationConfigurations.<*name*>.downstreamConfigurations.
<*name*>.connectionConfiguration.staticConnectors**

Type: List
Default:
XML name: static-connectors
Description: A list of connector references configured via connectors.

**federationConfigurations.<*name*>.downstreamConfigurations.
<*name*>.connectionConfiguration.reconnectAttempts**

Type: int
Default: -1
XML name: reconnect-attempts
Description: The number of reconnection attempts after a failure.

**federationConfigurations.<*name*>.downstreamConfigurations.
<*name*>.connectionConfiguration.password**

Type: String
Default:
XML name: password
Description: A password. If not specified, the federated password is used.

**federationConfigurations.<*name*>.downstreamConfigurations.
<*name*>.connectionConfiguration.callFailoverTimeout**

Type: long
Default: -1
XML name: call-failover-timeout
Description: The duration to wait for a reply during a fail-over. A value of **-1** means there is no limit.

**federationConfigurations.<*name*>.downstreamConfigurations.
<*name*>.connectionConfiguration.hA**

Type: boolean
Default:
XML name: h-a
Description:

**federationConfigurations.<*name*>.downstreamConfigurations.
<*name*>.connectionConfiguration.initialConnectAttempts**

Type: int
Default: -1
XML name: initial-connect-attempts
Description: The number of initial attempts to connect.

**federationConfigurations.<*name*>.downstreamConfigurations.
<*name*>.connectionConfiguration.retryInterval**

Type: long
Default: 500
XML name: retry-interval
Description: The period, in milliseconds, between successive retries.

**federationConfigurations.<*name*>.downstreamConfigurations.
<*name*>.connectionConfiguration.clientFailureCheckPeriod**

Type: long
Default:
XML name: client-failure-check-period
Description:

**federationConfigurations.<*name*>.downstreamConfigurations.
<*name*>.connectionConfiguration.username**

Type: String
Default:
XML name: username
Description:

federationConfigurations.<*name*>.downstreamConfigurations.<*name*>.policyRefs

Type: Collection
Default:
XML name: policy-refs
Description:

federationConfigurations.<*name*>.downstreamConfigurations.<*name*>.staticConnectors

Type: List
Default:
XML name: static-connectors
Description: A list of connector references configured via connectors.

federationConfigurations.<*name*>.federationPolicys

Type: FederationPolicy
Default:
XML name: policy-set
Description:

federationConfigurations.<*name*>.addressPolicies

Type: FederationAddressPolicyConfiguration
Default:
XML name: address-policy
Description:

federationConfigurations.<*name*>.addressPolicies.<*name*>.autoDeleteMessageCount

Type: Long
Default:
XML name: auto-delete-message-count
Description: The maximum number of messages that can still be in a dynamically-created remote queue before that queue is eligible to be automatically deleted.

federationConfigurations.<*name*>.addressPolicies.<*name*>.enableDivertBindings

Type: Boolean
Default:
XML name: enable-divert-bindings
Description: Setting 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**.

federationConfigurations.<*name*>.addressPolicies.<*name*>.includes.{NAME}.addressMatch

Type: Matcher
Default:

XML name: include
Description:

federationConfigurations.<*name*>.addressPolicies.<*name*>.maxHops

Type: int
Default:
XML name: max-hops
Description: The number of hops that a message can make for it to be federated.

federationConfigurations.<*name*>.addressPolicies.<*name*>.transformerRef

Type: String
Default:
XML name: transformer-ref
Description: The ref name for a transformer that you want to configure to transform the message on federation transfer.

federationConfigurations.<*name*>.addressPolicies.<*name*>.autoDeleteDelay

Type: Long
Default:
XML name: auto-delete-delay
Description: The duration, in milliseconds, after the downstream broker disconnects before the upstream queue can be automatically deleted.

federationConfigurations.<*name*>.addressPolicies.<*name*>.autoDelete

Type: Boolean
Default:
XML name: auto-delete
Description: For address federation, the downstream dynamically creates a durable queue on the upstream address. Specifies if the upstream queue is deleted once the downstream disconnects and the delay and message count parameters are met.

federationConfigurations.<*name*>.addressPolicies.<*name*>.excludes.{NAME}.addressMatch

Type: Matcher
Default:
XML name: include
Description:

G.6. CLUSTERCONFIGURATIONS

clusterConfigurations.<*name*>.retryIntervalMultiplier

Type: double
Default: 1
XML name: retry-interval-multiplier
Description: The multiplier to apply to the retry-interval.

clusterConfigurations.<*name*>.maxRetryInterval

Type: long
Default: 2000
XML name: max-retry-interval

Description: The maximum value for retry-interval.

clusterConfigurations.<*name*>.address

Type: String
Default:
XML name: address
Description: The name of the address to which this cluster connection applies.

clusterConfigurations.<*name*>.maxHops

Type: int
Default: 1
XML name: max-hops
Description: The maximum number of hops to which the cluster topology is propagated.

clusterConfigurations.<*name*>.connectionTTL

Type: long
Default: 60000
XML name: connection-ttl
Description: The duration to keep a connection alive if no data is received from the client.

clusterConfigurations.<*name*>.clusterNotificationInterval

Type: long
Default: 1000
XML name: notification-interval
Description: The interval at which the cluster connection notifies the cluster of its existence.

clusterConfigurations.<*name*>.confirmationWindowSize

Type: int
Default: 1048576
XML name: confirmation-window-size
Description: The size, in bytes, of the window used for confirming data from the server. Supports byte notation, for example, "K", "Mb", "MiB", "GB".

clusterConfigurations.<*name*>.callTimeout

Type: long
Default: 30000
XML name: call-timeout
Description: The duration to wait for a reply.

clusterConfigurations.<*name*>.staticConnectors

Type: List
Default:
XML name: static-connectors
Description: A list of connectors references names.

clusterConfigurations.<*name*>.clusterNotificationAttempts

Type: int
Default: 2
XML name: notification-attempts

Description: The number of times this cluster connection attempts to notify the cluster of its existence.

clusterConfigurations.<*name*>.allowDirectConnectionsOnly

Type: boolean
Default: false
XML name: allow-direct-connections-only
Description: Restricts cluster connections to the listed connector-refs.

clusterConfigurations.<*name*>.reconnectAttempts

Type: int
Default: -1
XML name: reconnect-attempts
Description: The number of reconnection attempts after a failure.

clusterConfigurations.<*name*>.duplicateDetection

Type: boolean
Default: true
XML name: use-duplicate-detection
Description: Specifies if duplicate detection headers are inserted in forwarded messages.

clusterConfigurations.<*name*>.callFailoverTimeout

Type: long
Default: -1
XML name: call-failover-timeout
Description: The duration to wait for a reply during a fail-over. A value of **-1** means there is no limit.

clusterConfigurations.<*name*>.messageLoadBalancingType

Type: MessageLoadBalancingType(OFF STRICT ON_DEMAND OFF_WITH_REDISTRIBUTION)
Default:
XML name: message-load-balancing-type
Description:

clusterConfigurations.<*name*>.initialConnectAttempts

Type: int
Default: -1
XML name: initial-connect-attempts
Description: The number of initial attempts to connect.

clusterConfigurations.<*name*>.connectorName

Type: String
Default:
XML name: connector-ref
Description: The name of the connector reference to use.

clusterConfigurations.<*name*>.retryInterval

Type: long
Default: 500
XML name: retry-interval
Description: The interval, in milliseconds, between successive retries.

clusterConfigurations.<*name*>.producerWindowSize

Type: int
Default: 1048576
XML name: producer-window-size
Description: Producer flow control. Supports byte notation, for example, "K", "Mb", "MiB", "GB".

clusterConfigurations.<*name*>.clientFailureCheckPeriod

Type: long
Default:
XML name: client-failure-check-period
Description:

clusterConfigurations.<*name*>.discoveryGroupName

Type: String
Default:
XML name: discovery-group-name
Description: The name of the discovery group used by this cluster-connection.

clusterConfigurations.<*name*>.minLargeMessageSize

Type: int
Default:
XML name: min-large-message-size
Description: The size, in bytes, above which a message is considered a large message. Large messages are sent over the network in multiple segments. Supports byte notation, for example, "K", "Mb", "MiB", "GB".

G.7. CONNECTIONROUTERS

connectionRouters.<*name*>.cacheConfiguration

Type: CacheConfiguration
Default:
XML name: cache
Description: Controls if the cache entries are persisted and how often a cache removes its entries.

connectionRouters.<*name*>.cacheConfiguration.persisted

Type: boolean
Default: false
XML name: persisted
Description: A value of **true** means that the cache entries are persisted.

connectionRouters.<*name*>.cacheConfiguration.timeout

Type: int
Default: -1
XML name: timeout
Description: The timeout, in milliseconds, before cache entries are removed.

connectionRouters.<*name*>.keyFilter

Type: String
Default:

XML name: key-filter
Description: A filter for the target key.

connectionRouters.<*name*>.keyType

Type: KeyType(CLIENT_ID SNI_HOST SOURCE_IP USER_NAME ROLE_NAME)
Default:
XML name: key-type
Description: The optional target key.

connectionRouters.<*name*>.localTargetFilter

Type: String
Default:
XML name: local-target-filter
Description: A filter to find the local target.

connectionRouters.<*name*>.poolConfiguration

Type: PoolConfiguration
Default:
XML name: pool
Description: The pool configuration.

connectionRouters.<*name*>.poolConfiguration.quorumTimeout

Type: int
Default: 3000
XML name: quorum-timeout
Description: The timeout, in milliseconds, used to get the minimum number of ready targets.

connectionRouters.<*name*>.poolConfiguration.password

Type: String
Default:
XML name: password
Description: The password to access the targets.

connectionRouters.<*name*>.poolConfiguration.localTargetEnabled

Type: boolean
Default: false
XML name: local-target-enabled
Description: A value of **true** means that the local target is enabled.

connectionRouters.<*name*>.poolConfiguration.checkPeriod

Type: int
Default: 5000
XML name: check-period
Description: The period, in milliseconds, used to check if a target is ready.

connectionRouters.<*name*>.poolConfiguration.quorumSize

Type: int
Default: 1
XML name: quorum-size

Description: The minimum number of ready targets.

connectionRouters.<*name*>.poolConfiguration.staticConnectors

Type: List
Default:
XML name: static-connectors
Description: A list of connector references configured via connectors.

connectionRouters.<*name*>.poolConfiguration.discoveryGroupName

Type: String
Default:
XML name: discovery-group-name
Description: The name of a discovery group used by this bridge.

connectionRouters.<*name*>.poolConfiguration.clusterConnection

Type: String
Default:
XML name: cluster-connection
Description: The name of a cluster connection.

connectionRouters.<*name*>.poolConfiguration.username

Type: String
Default:
XML name: username
Description: The username to access the targets.

connectionRouters.<*name*>.policyConfiguration

Type: NamedPropertyConfiguration
Default:
XML name: policy-configuration
Description:

connectionRouters.<*name*>..properties.{PROPERTY}

Type: Properties
Default:
XML name: property
Description: A set of Key value pairs specific to each named property.

Revised on 2024-12-16 13:27:41 UTC