Red Hat AMQ 7.7

Configuring AMQ Broker

For Use with AMQ Broker 7.7

Red Hat AMQ 7.7 Configuring AMQ Broker

For Use with AMQ Broker 7.7

Legal Notice

Abstract

This guide describes how to configure AMQ Broker.

Table of Contents

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 configuration for an acceptor for each supported messaging protocol.

```
<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.

- 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>
   ```

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
```

1.5. DOCUMENT CONVENTIONS

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

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 The **sudo** Command.

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\...**).

CHAPTER 2. NETWORK CONNECTIONS: ACCEPTORS AND CONNECTORS

There are two types of connections used in AMQ Broker: network and In-VM. 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 within the same virtual machine as the broker.

Network connections rely on Netty. Netty is a high-performance, low-level network library that allows network connections to be configured in several different ways: using Java IO or NIO, TCP sockets, SSL/TLS, even 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 within a network connection's configuration determines its type. For example, using **vm** in the URI will create an In-VM connection. In the example below, note that the URI of the **acceptor** starts with **vm**.

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

Using **tcp** in the URI, alternatively, will create a network connection.

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

This chapter will first discuss the two configuration elements specific to network connections, Acceptors and Connectors. Next, configuration steps for TCP, HTTP, and SSL/TLS network connections, as well as In-VM connections, are explained.

2.1. ABOUT ACCEPTORS

One of the most important concepts when discussing network connections in AMQ Broker is the **acceptor**. Acceptors define the way connections are made to the broker. Below is a typical configuration for an **acceptor** that might be in found inside the configuration file ***BROKER_INSTANCE_DIR*/etc/broker.xml**.

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

Note that each **acceptor** is grouped inside an **acceptors** element. There is no upper limit to the number of acceptors you can list per server.

Configuring an Acceptor

You configure an **acceptor** by appending key-value pairs to the query string of the URI defined for the **acceptor**. Use a semicolon (';') to separate multiple key-value pairs, as shown in the following example. It configures an acceptor for SSL/TLS by adding multiple key-value pairs at the end of the URI, starting with **sslEnabled=true**.

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

For details on **connector** configuration parameters, see Acceptor and Connector Configuration Parameters.

2.2. ABOUT CONNECTORS

Whereas acceptors define how a server accepts connections, a **connector** is used by clients to define how they can connect to a server.

Below is a typical **connector** as defined in the *BROKER_INSTANCE_DIR*/**etc/broker.xml** configuration file:

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

Note that connectors are defined inside a **connectors** element. There is no upper limit to the number of connectors per server.

Although connectors are used by clients, they are configured on the server just like acceptors. There are a couple of important reasons why:

- A server itself can act as a client and therefore needs to know how to connect to other servers. For example, when one server is bridged to another or when a server takes part in a cluster.

- A server is often used by JMS clients to look up connection factory instances. In these cases, JNDI needs to know details of the connection factories used to create client connections. The information is provided to the client when a JNDI lookup is performed. See Configuring a Connection on the Client Side for more information.

Configuring a Connector

Like acceptors, connectors have their configuration attached to the query string of their URI. Below is an example of a **connector** that has the **tcpNoDelay** parameter set to **false**, which turns off Nagle's algorithm for this connection.

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

For details on **connector** configuration parameters, see Acceptor and Connector Configuration Parameters.

2.3. 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, remember that a TCP network connection is unencrypted. You may want to consider using an SSL or HTTPS configuration to encrypt messages sent over this connection if encryption is a priority. Refer to Configuring Transport Layer Security for details.

When using a TCP connection, all connections are initiated from the client side. In other words, the server does not initiate any connections to the client, which 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's URI defines the address used for the connection.

Procedure

1. Open the configuration file **BROKER_INSTANCE_DIR/etc/broker.xml**

2. Add or modify the connection and include a URI that uses **tcp** as the protocol. Be sure to include both an IP or hostname as well as a port.

In the example below, an **acceptor** is configured as a TCP connection. A broker configured with this **acceptor** will accept clients making TCP connections to the IP   **10.10.10.1** and port **61617**.

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

You configure a connector to use TCP in much the same way.

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

The **connector** above would be 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**.

For details on available configuration parameters for TCP connections, see Acceptor and Connector Configuration Parameters. Most parameters can be used either with acceptors or connectors, but some only work with acceptors.

2.4. CONFIGURING AN HTTP CONNECTION

HTTP connections tunnel packets over the HTTP protocol and are useful in scenarios where firewalls allow only HTTP traffic. With single port support, AMQ Broker will automatically detect if HTTP is being used, so configuring a network connection for HTTP is the same as configuring a connection for TCP. For a full working example showing how to use HTTP, see the **http-transport** example, located under **INSTALL_DIR/examples/features/standard/**.

Procedure

1. Open the configuration file **BROKER_INSTANCE_DIR/etc/broker.xml**

2. Add or modify the connection and include a URI that uses **tcp** as the protocol. Be sure to include both an IP or hostname as well as a port

In the example below, the broker will accept HTTP communications from clients connecting to port **80** at the IP address **10.10.10.1**. Furthermore, the broker will automatically detect that the HTTP protocol is in use and will communicate with the client accordingly.

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

Configuring a connector for HTTP is again the same as for TCP.

```
<connectors>
```

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

Using the configuration in the example above, a broker will create an outbound HTTP connection to port **80** at the IP address **10.10.10.2**.

An HTTP connection uses the same configuration parameters as TCP, but it also has some of its own. For details on HTTP-related and other configuration parameters, see Acceptor and Connector Configuration Parameters.

2.5. CONFIGURING AN SSL/TLS CONNECTION

You can also configure connections to use SSL/TLS. Refer to Configuring Transport Layer Security for details.

2.6. CONFIGURING AN IN-VM CONNECTION

An In-VM connection can be used when multiple brokers are co-located in the same virtual machine, as part of a high availability solution for example. In-VM connections can also be used by local clients running in the same JVM as the server. For an in-VM connection, the authority part of the URI defines a unique server ID. In fact, no other part of the URI is needed.

Procedure

1. Open the configuration file ***BROKER_INSTANCE_DIR*/etc/broker.xml**

2. Add or modify the connection and include a URI that uses **vm** as the protocol.

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

The example **acceptor** above tells the broker to accept connections from the server with an ID of **0**. The other server must be running in the same virtual machine as the broker.

Configuring a connector as an in-vm connection follows the same syntax.

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

The **connector** in the example above defines how clients establish an in-VM connection to the server with an ID of **0** that resides in the same virtual machine. The client can be be an application or broker.

2.7. CONFIGURING A CONNECTION FROM THE CLIENT SIDE

Connectors are also used indirectly in client applications. You can configure the JMS connection factory directly on the client side without having to define a **connector** on the server side:

```
Map<String, Object> connectionParams = new HashMap<String, Object>();
```

```
connectionParams.put(org.apache.activemq.artemis.core.remoting.impl.netty.TransportConstants.PORT_PROP_NAME, 61617);

TransportConfiguration transportConfiguration =
    new TransportConfiguration(
    "org.apache.activemq.artemis.core.remoting.impl.netty.NettyConnectorFactory",
connectionParams);

ConnectionFactory connectionFactory =
ActiveMQJMSClient.createConnectionFactoryWithoutHA(JMSFactoryType.CF,
transportConfiguration);

Connection jmsConnection = connectionFactory.createConnection();
```

CHAPTER 3. NETWORK CONNECTIONS: PROTOCOLS

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 PROTOCOL

You must associate a protocol with a network connection before you can use it. (See Network Connections: Acceptors and Connectors 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.

```xml
<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 configuration file ***BROKER_INSTANCE_DIR*/etc/broker.xml**

2. Add the following line to the **\<acceptors\>** stanza:

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

Additional parameters in default acceptors

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

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

Acceptor(s)	Parameter	Description
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 control the flow of AMQP messages, see Section 11.2.3, "Blocking AMQP Messages".
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 control the flow of AMQP messages, see Section 11.2.3, "Blocking AMQP Messages".
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 4.13, "Configuring a Prefix to Connect to a Specific Routing Type".
	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 4.13, "Configuring a Prefix to Connect to a Specific Routing Type".

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 configuration file **BROKER_INSTANCE_DIR/etc/broker.xml**

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:

To create this kind of subscription for a multicast-only queue…	Set Terminus Durability to this…
Durable	**UNSETTLED_STATE** or **CONFIGURATION**
Non-durable	**NONE**
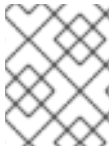
> **NOTE**
>
> The name of a durable queue is composed of the container ID and the link name, for example **my-container-id:my-link-name**.

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 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 configuration file ***BROKER_INSTANCE_DIR*/etc/broker.xml**

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.

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.

The best source of information about the MQTT protocol is in the specification. The MQTT v3.1.1 specification can be downloaded from the OASIS website.

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 configuration file **BROKER_INSTANCE_DIR/etc/broker.xml**

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 examples located under **INSTALL_DIR/examples/protocols/openwire**.

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 configuration file **BROKER_INSTANCE_DIR/etc/broker.xml**

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.

See the **stomp** example located under *INSTALL_DIR*/**examples**/**protocols** for an example of how to configure a broker with STOMP.

3.5.1. Knowing the Limitations When Using STOMP

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 acknowledgements 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 configuration file *BROKER_INSTANCE_DIR*/**etc/broker.xml**

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's Time to Live (TTL)

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 configuration file **BROKER_INSTANCE_DIR/etc/broker.xml**

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's Default Time to Live (TTL)
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 configuration file **BROKER_INSTANCE_DIR/etc/broker.xml**

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.

If you want a STOMP message to map to a ...	The message should....

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. ADDRESSES, QUEUES, AND TOPICS

AMQ Broker has a unique addressing model that is both powerful and flexible and that offers great performance. 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, 0 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. A AMQ Broker address can be configured with two different routing types.

Table 4.1. 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 routing type.
>
> It is possible to define more than one routing type per address, but this typically results in an anti-pattern and is therefore not recommended.
>
> If an address does use both routing types, however, and the client does not show a preference for either one, the broker typically defaults to the **anycast** routing type. The one exception is when the client uses the MQTT protocol. In that case, the default routing type is **multicast**.

4.1. ADDRESS AND QUEUE NAMING REQUIREMENTS

You should 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 it uses, your address and queue names should not include any of the following characters:
  & :: , ? >

- The **#** and * characters are reserved for wildcard expressions. For more information, see the section called "AMQ Broker Wildcard Syntax".

- Address and queue names should not include any spaces.

- To separate words in an address or queue name, use the configured delimiter character (the default is the **.** character). For more information, see the section called "AMQ Broker Wildcard Syntax".

4.2. CONFIGURING 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. Define an **anycast** routing type for an **address** so that its queues receive messages in a point-to-point manner.

When a message is received on an address using **anycast**, AMQ Broker locates the queue associated with the address and routes the message to it. When consumers request to consume from the address, the broker locates the relevant queue and associates this queue with the appropriate consumers. If multiple consumers are connected to the same queue, messages are distributed amongst each consumer equally, providing the consumers are equally able to handle them.

Figure 4.1. Point-to-Point
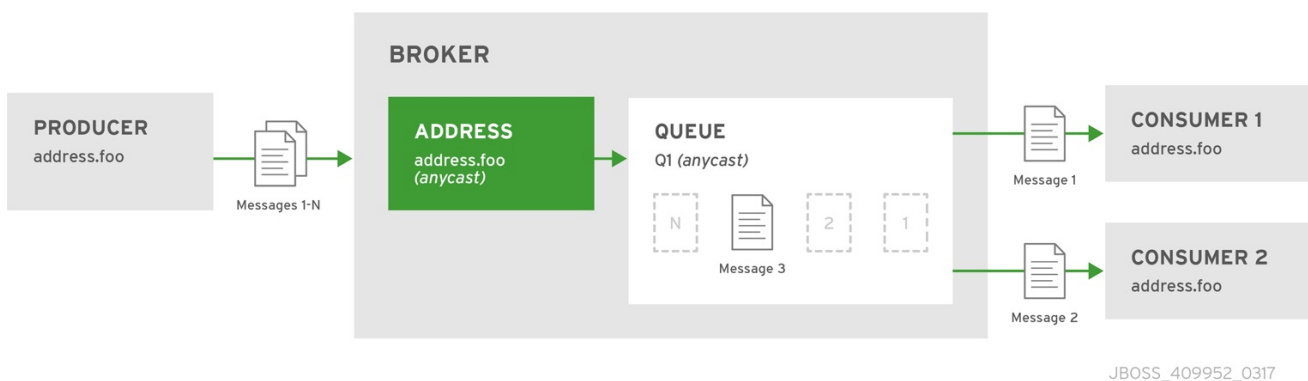
Procedure

1. Open the file **BROKER_INSTANCE_DIR/etc/broker.xml** for editing.

2. Wrap an **anycast** configuration element around the chosen **queue** element of an **address**. Ensure the value of **address name** and **queue name** elements are same.

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

4.3. CONFIGURING 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. When a message is received on an address with a **multicast** routing type, AMQ Broker routes a copy of the message to each

queue. To reduce the overhead of copying, each queue is sent only a reference to the message and not a full copy.

Figure 4.2. Publish-Subscribe
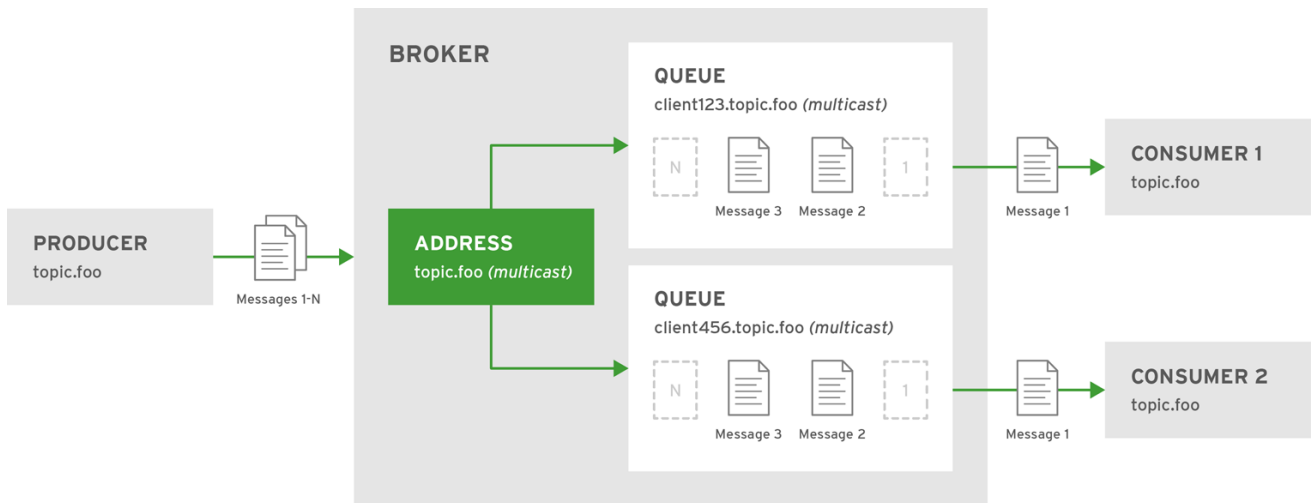
Procedure

1. Open the file ***BROKER_INSTANCE_DIR*/etc/broker.xml** for editing.

2. Add an empty **multicast** configuration element to the chosen address.

   ```
   <configuration ...>
    <core ...>
     ...
      <address name="topic.foo">
       <multicast/>
      </address>
     </core>
   </configuration>
   ```

3. (Optional) Add one 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="topic.foo">
       <multicast>
         <queue name="client123.topic.foo"/>
         <queue name="client456.topic.foo"/>
       </multicast>
      </address>
     </core>
   </configuration>
   ```

4.4. CONFIGURING A POINT-TO-POINT USING TWO QUEUES

You can define more than one queue on an address by using an **anycast** routing type. Messages semt to an **anycast** address are distributed evenly across all associated queues. By using Fully Qualified Queue Names, which are described later, you can have clients connect to a specific queue. If more than one consumer connects to the same queue, AMQ Broker distributes messages between them.

Figure 4.3. Point-to-Point with Two Queues
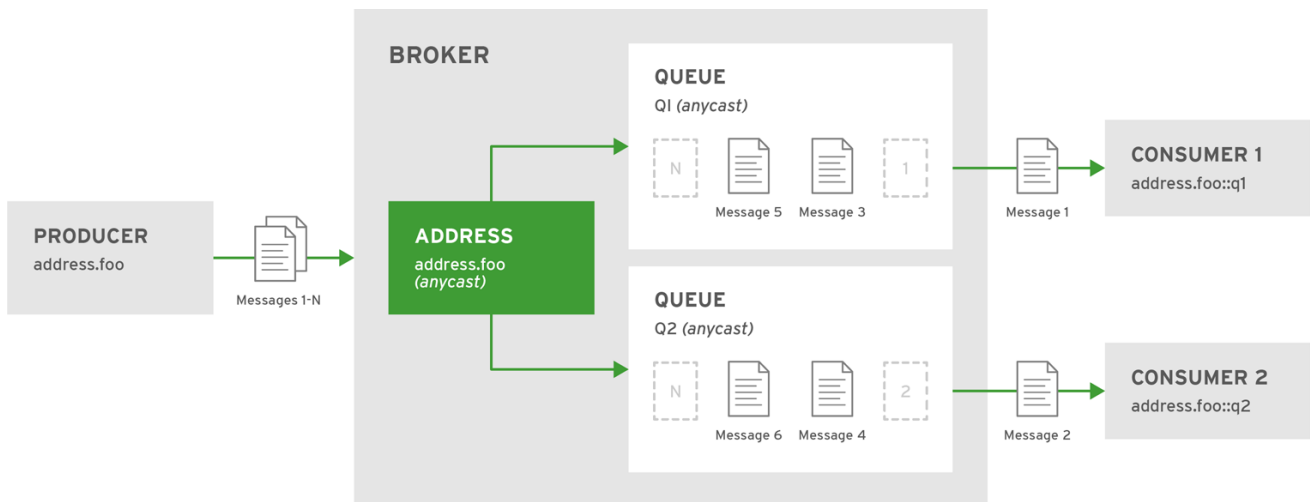
JBOSS_409952_0317

> **NOTE**
>
> This is how AMQ Broker handles load balancing of queues across multiple nodes in a cluster.

Procedure

1. Open the file **BROKER_INSTANCE_DIR/etc/broker.xml** for editing.

2. Wrap an **anycast** configuration element around the **queue** elements in the **address**.

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

4.5. USING POINT-TO-POINT AND PUBLISH-SUBSCRIBE TOGETHER

It is possible to define an address with both point-to-point and publish-subscribe semantics enabled. While not typically recommended, this can be useful when you want, for example, a JMS Queue named **orders** and a JMS topic named **orders**. The different routing types make the addresses appear to be distinct.

Using an example of JMS clients, the messages sent by a JMS queue producer are routed using the

anycast routing type. Messages sent by a JMS topic producer uses the **multicast** routing type. In addition, when a JMS topic consumer attaches, it is attached to its own subscription queue. The JMS queue consumer, however, is attached to the **anycast** queue.

Figure 4.4. Point-to-Point and Publish-Subscribe
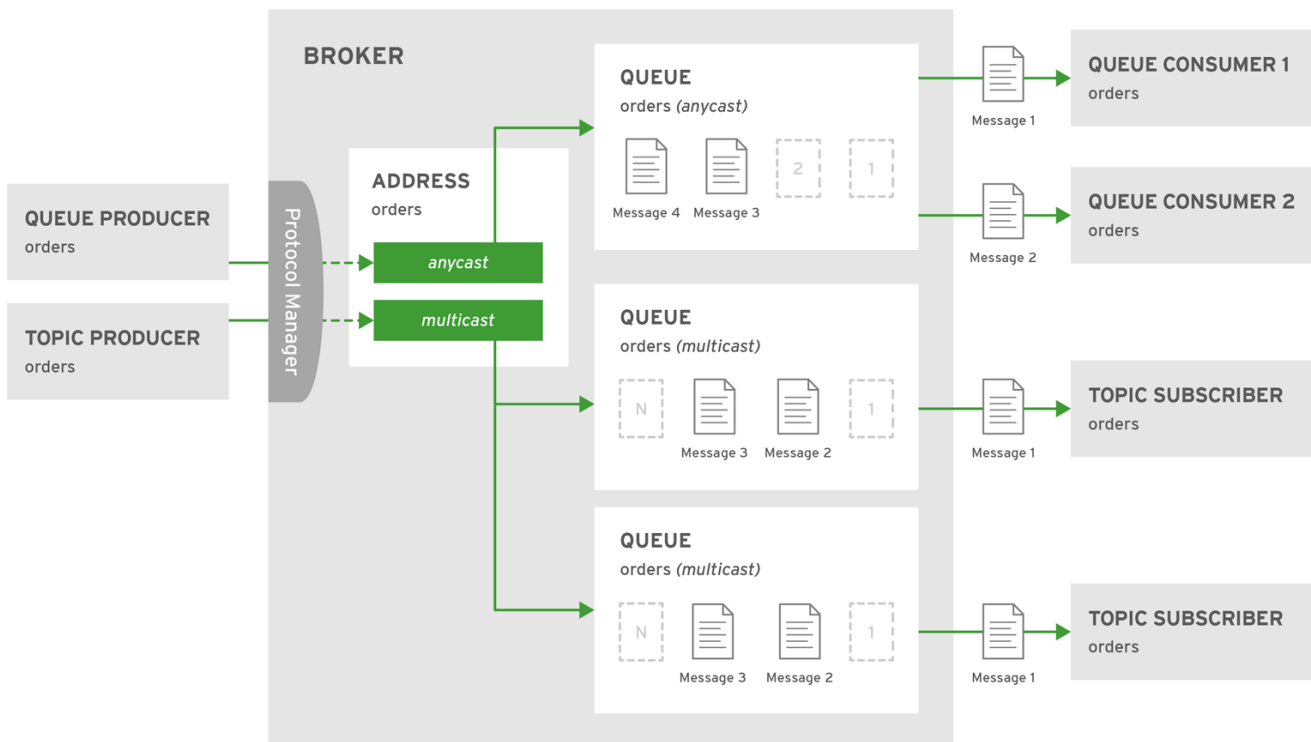
JBOSS_409952_0317

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, check the behavior of each protocol in the sections on protocols.

The XML excerpt below is an example of what the configuration for an address using both **anycast** and **multicast** routing types would look like in **BROKER_INSTANCE_DIR/etc/broker.xml**. Note that subscription queues are typically created on demand, so there is no need to list specific **queue** elements inside the **multicast** routing type.

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

4.6. CONFIGURING SUBSCRIPTION QUEUES

In most cases it is not necessary to pre-create subscription queues because protocol managers create subscription queues automatically when clients first request to subscribe to an address. See 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.

Configuring a Durable Subscription Queue

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

Procedure

1. Open the file ***BROKER_INSTANCE_DIR*/etc/broker.xml** for editing.

2. Add the **durable** configuration element to the chosen **queue** and assign it a value of **true**.

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

Configuring a Non-Shared Durable Subscription

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

Procedure

1. Open the file ***BROKER_INSTANCE_DIR*/etc/broker.xml** for editing.

2. Add the **durable** configuration element to each chosen queue.

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

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

3. Add the **max-consumers** attribute to each chosen **queue** element and assign it a value of **1**.

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

Configuring a Non–Durable Subscription Queue

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

However, if you want to pre-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 attached. In addition, when the last consumer is detached 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 attached to the queue.

Procedure

1. Open the file ***BROKER_INSTANCE_DIR*/etc/broker.xml** for editing.

2. Add the **purge-on-no-consumers** attribute to each chosen **queue** element. Set **purge-on-no-consumers** to **true**.

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

4.7. USING 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 which queues to send messages, or from which queue to receive messages. However,

more advanced use cases might require that the client specify a queue directly. In these situations the client can use a Fully Qualified Queue Name (FQQN), by specifying both the address name and the queue name, separated by a **::**.

Prerequisites

- An address is configured with two or more queues. In the example below the address **foo** has two queues, **q1** and **q2**.

```xml
<configuration ...>
 <core ...>
   ...
   <addresses>
     <address name="foo">
       <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. Remember to use two colons, **::**, to separate the names, as in the example Java code below.

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

4.8. CONFIGURING SHARDED QUEUES

A common pattern for processing of messages across a queue where only partial ordering is required is to use queue sharding. In AMQ Broker this can be achieved by creating an **anycast** address that acts as a single logical queue, but which is backed by many underlying physical queues.

Procedure

1. Open *BROKER_INSTANCE_DIR*/**etc/broker.xml** and add an **address** with the desired name. In the example below the **address** named **sharded** is added to the configuration.

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

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

Using the configuration above, messages sent to **sharded** 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 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. See the chapter on Message Grouping for more information.

4.9. 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 property name is placed in the queue. Through this behavior, last value queues retain only the last values for messages of the same property.

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.
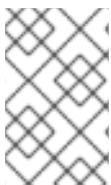
> **NOTE**
>
> The broker delivers messages that are sent to a last value queue without a configured last value property as normal messages. Such messages are not purged from the queue when a new message with a configured last value property arrives.

You can configure last value queues using:

- The **broker.xml** configuration file

- The JMS client

- The Core API

- Address wildcards

For each of the above methods, apart from the Core API, you can specify a custom value for the last value key (also called the last value property), or leave this value unset, meaning that the key is set to the default value instead. The default value for the last value key is **_AMQ_LVQ_NAME**.

For the Core API, you create last value queues using the constant last value key **Message.HDR_LAST_VALUE_NAME** to identify last value messages.

4.9.1. Configuring Last Value Queues Using broker.xml

To specify a custom value for the last value key, include lines in your **broker.xml** configuration file that look like the following:

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

Alternatively, you can configure a last value queue that uses the default last value key name of **_AMQ_LVQ_NAME**. To do this, set the last-value configuration parameter to true in your **broker.xml** configuration file, without specifying a value for the last value key. An example of this configuration is shown below.

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

4.9.2. Configuring Last Value Queues Using the JMS Client

When using the JMS Client to auto-create destinations used by a consumer, you can specify a last value key as part of the address settings. In this case, the auto-created queues are last value queues. An example of this configuration is shown below.

```
Queue queue = session.createQueue("my.destination.name?last-value-key=stock_ticker");
Topic topic = session.createTopic("my.destination.name?last-value-key=stock_ticker");
```

Alternatively, configure a last value queue that uses the default last value key name of **_AMQ_LVQ_NAME**. To do this, set the last-value configuration parameter to true, without specifying a value for the last value key. An example of this configuration is shown below.

```
Queue queue = session.createQueue("my.destination.name?last-value=true");
Topic topic = session.createTopic("my.destination.name?last-value=true");
```

4.9.3. Configuring Last Value Queues Using the Core API

To create a last value queue using the Core API, set the **lastvalue** parameter to **true** when creating a queue. To do this, use the **createQueue** method of the **ClientSession** interface. The syntax for creating a last value queue using this method is shown below.

```
void createQueue(
SimpleString address,
RoutingType routingType,
SimpleString queueName,
SimpleString filter,
Boolean durable,
```

```
Boolean autoCreated,
int maxConsumers,
Boolean purgeOnNoConsumers,
Boolean exclusive,
Boolean lastValue
)
```

In this case, the API uses the constant **Message.HDR_LAST_VALUE_NAME** to identify last value messages delivered to the queue.

Additional Resources

- For more information about using the **createQueue** method of the Core API to create a last value queue, see createQueue.

4.9.4. Configuring Last Value Queues Using Address Wildcards

You can use address wildcards in the **broker.xml** configuration file to configure last value queues for a set of addresses. An example of this configuration is shown below.

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

By default, the value of **default-last-value-key** is **null**.

When using address wildcards, you can also use the default last value key name. To do this, set the **default-last-value-queue** parameter to **true**, without specifying a value for the last value key.

4.9.5. Example of Last Value Queue Behavior

This example shows the configuration and behavior of a last value queue.

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

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

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

Now, suppose that a client sends two messages with the same last value property name to the **prices1** queue, as shown below:

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

```java
TextMessage message = session.createTextMessage("Second message with last value property set");
```

```
message.setStringProperty("stock_ticker", "37.02");
producer.send(message);
```

When two messages with the same last value property name arrive to the last value queue **prices1**, 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 last value property and the second message was received in the queue after the first.

4.9.6. Creating Non-Destructive Consumers

When a consumer attaches 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.

An alternative way to configure a consumer is to configure it as a queue *browser*. In this case, the queue still sends all messages to the consumer. However the browser does not prevent other consumers from receiving the messages. In addition, the messages remain in the queue once the browser has consumed the messages. A consumer configured as a browser is an instance of a non-destructive consumer.

In the case of a last value queue, configuring all consumers as non-destructive consumers ensures that the queue always retains the most up-to-date value for every last value key. The examples in the following sub-sections show how to configure a last value queue to ensure that all consumers are non-destructive.

4.9.6.1. Configuring Non-destructive Consumers Using broker.xml

In the **broker.xml** configuration file, to create a last value queue, set the **non-destructive** parameter to **true**. An example of this configuration is shown below.

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

4.9.6.2. Creating Non-destructive Consumers Using the JMS Client

When using the JMS Client to auto-create destinations used by a consumer, you configure last value queue behavior and non-destructive consumers as part of the address settings. An example of this configuration is shown below.

```
Queue queue = session.createQueue("my.destination.name?last-value-key=stock_ticker&non-destructive=true");
Topic topic = session.createTopic("my.destination.name?last-value-key=stock_ticker&non-destructive=true");
```

4.9.6.3. Configuring Non-destructive Consumers Using Address Wildcards

You can use address wildcards in the **broker.xml** configuration file to configure last value queues for a set of addresses. As part of this configuration, you can also specify that consumers are non destructive, by setting **default-non-destructive** to **true**. An example of this configuration is shown below.

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

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

4.10. 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 is sets an unlimited number of consumers.

Procedure

1. Open *BROKER_INSTANCE_DIR*/etc/broker.xml and add the **max-consumers** attribute to the desired **queue**. In the example below, only **20** consumers can connect to the queue **q3** at the same time.

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

2. (Optional) Create an exclusive consumer by setting **max-consumers** to **1**, as in the example below.

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

3. (Optional) Have an unlimited number of consumers by setting **max-consumers** to **-1**, as in the example below.

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

4.11. EXCLUSIVE QUEUES

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

4.11.1. Configuring Exclusive Queues

You configure exclusive queues in the **broker.xml** configuration file something like this.

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

Or on auto-create when using the JMS Client by using address parameters when creating the destination used by the consumer.

```
Queue queue = session.createQueue("my.destination.name?exclusive=true");
Topic topic = session.createTopic("my.destination.name?exclusive=true");
```

4.11.2. Setting the Exclusive Queue Default

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

The default value for **default-exclusive-queue** is **false**.

4.12. 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.

4.12.1. Configuring Ring Queues

This 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. An example is shown below.

   ```
   <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** parameter to the **queue** element and specify a value. An example is shown below.

   ```
   <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.

4.12.2. Troubleshooting Ring Queue Behavior

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

See:

- Section 4.12.2.1, "In-delivery messages and rollbacks"

- Section 4.12.2.2, "Scheduled messages"

- Section 4.12.2.3, "Paged messages"

4.12.2.1. 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).

4.12.2.2. 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.

4.12.2.3. 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 addressing. To learn more, see Section 4.14, "Configuring Retroactive Addresses".

4.13. CONFIGURING A PREFIX TO CONNECT TO A SPECIFIC ROUTING TYPE

Normally, if a message is received by an address that uses both **anycast** and **multicast**, one of the **anycast** queues receive 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**.

Configuring an Anycast Prefix

- In *BROKER_INSTANCE_DIR*/etc/broker.xml, add the **anycastPrefix** to the URL of the desired **acceptor**. In the example below, the **acceptor** is configured to use **anycast://** for the **anycastPrefix**. Client code can specify **anycast://foo/** if the client needs to send a message to only one of the anycast queues.

```
<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>
```

Configuring a Multicast Prefix

- In ***BROKER_INSTANCE_DIR*/etc/broker.xml**, add the **anycastPrefix** to the URL of the desired **acceptor**. In the example below, the **acceptor** is configured to use **multicast://** for the **multicastPrefix**. Client code can specify **multicast://foo/** if the client needs the message sent to only the multicast queues of the address.

```
<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>
```

4.14. 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.

This 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 4.12, "Configuring Ring Queues".

4.15. PROTOCOL MANAGERS AND ADDRESSES

A protocol manager maps protocol–specific concepts down to the AMQ Broker address model concepts: queues and routing types. For example, when a client sends a 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. The protocol manager therefore 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 4.2. Protocol Manager Actions

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

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

4.16. DISABLING ADVISORY MESSAGES

By default, AMQ 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 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 Console might become cluttered when attempting to display all of the addresses created to send advisory messages. To avoid these situations, use the **supportAdvisory** and **suppressInternalManagementObjects** parameters to manage the advisory messages behavior on the broker side.

- **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 the management service such as JMX registry and AMQ Console, or **false** to not expose them. The default value is **true**.

Use these parameters by editing the *BROKER_INSTANCE_DIR*/**etc**/**broker.xml** configuration file and configure the parameters on openwire acceptors by using URLs. For example:

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

4.17. CONFIGURING ADDRESS SETTINGS

AMQ Broker has several configurable options that control aspects of how and when a message is delivered, how many attempts should be made, and when the message expires. These configuration options all exist within the **<address-setting>** configuration element. You can have AMQ Broker apply a single **<address-setting>** to multiple destinations by using a wildcard syntax.

AMQ Broker Wildcard Syntax

AMQ Broker uses a specific syntax for representing wildcards in security settings, address settings, and when creating consumers.

- A wildcard expression contains words delimited by the character '.'.

- The special characters '#' and '*' also have special meaning and can take the place of a word.

- The character '#' means 'match any sequence of zero or more words'. Use this at the end of your expression.

- The 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** looking to match queues using **my** in their name would not match with a queue named **myqueue**.

When more than one **address-setting** matches a queue, the broker will overlay configurations, using the configuration of the least specific match as the baseline. Literal expressions are more specific than wildcards, and ***** is more specific than **#**. For example, both **my.queue** and **my.*** match the queue **my.queue**. 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.queue address-setting**, which will overwrite any configuration shared with **my.***. Given the configuration below, the queue **my.queue** would have **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.queue">
   <last-value-queue>false</last-value-queue>
</address-setting>
```

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

Table 4.3. Wildcard Examples

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

Configuring Wildcard Syntax

You can customize the syntax used for wildcard addresses by adding configuration to **broker.xml**.

Procedure

- Edit **broker.xml** by adding a **<wildcard-addresses>** section to the configuration, as in the example below.

```
<configuration>
  <core>
    ...
    <wildcard-addresses>  1
      <enabled>true</enabled>  2
      <delimiter>,</delimiter>  3
      <any-words>@</any-words>  4
      <single-word>$</single-word>  5
    </wildcard-addresses>
    ...
  </core>
</configuration>
```

1   Add **wildcard-addresses** beneath the **core** configuration element.

2   Set **enabled** to **true** to tell the broker to use your custom settings.

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

4   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.

5   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.

4.18. CREATING AND DELETING QUEUES AND ADDRESSES AUTOMATICALLY

You can configure AMQ 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.

Automatic creation and deletion of queues and addresses is configured on a per **address-setting** basis. The configuration is applied to any address or queue that is a match for the **address-setting**. For more information about how to use wildcard syntax to match addresses and queues to an **address-setting** see Configuring Address Settings.

IMPORTANT

Disabling the automatic creation of queues and addresses can cause problems and is not recommended. AMQ Broker must be able to auto-create addresses and queues by using the **activemq.#** pattern. For an example see the example address setting.

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

Procedure

Edit the *BROKER_INSTANCE_DIR*/**etc/broker.xml** file and 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.#">          1
    <auto-create-addresses>true</auto-create-addresses>          2
    <auto-delete-addresses>true</auto-delete-addresses>          3
    <auto-create-queues>true</auto-create-queues>          4
    <auto-delete-queues>true</auto-delete-queues>          5
    <default-address-routing-type>ANYCAST</default-address-routing-type>          6
   </address-setting>
  </address-settings>
  ...
 </core>
</configuration>
```

1   The configuration included in this **address-setting** is applied to any address or queue that matches the wildcard **activemq.#**. For more information on using wildcard syntax see AMQ Broker Wildcard Syntax.

2   The broker creates an address that does not exist when a client requests it.

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

4   The broker creates a queue that does not exist when a client requests it.

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

⑥ 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**. See the introduction of this

4.19. 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.

4.19.1. 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 (that is, once every second).

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**.

4.19.2. 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 4.19.1, "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).
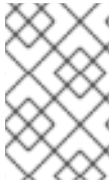
> **NOTE**
>
> Because the expiry address and queue are automatically created, any address settings related to deletion of automatically-created addresses and queues also apply to these expiry resources.

Additional resources

- For more information about address settings used to configure automatic deletion of automatically-created addresses and queues, see Section 4.18, "Creating and Deleting Queues and Addresses Automatically".

4.20. 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

4.20.1. 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 ...>
```

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*.

> **NOTE**
>
> 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 s*et* of addresses.

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 AMQ Broker Wildcard Syntax.

4.20.2. 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 4.20.1, "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).

4.21. 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.

Annotation name	Internal property name	Description

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 16.3, "Filtering AMQP Messages Based on Properties on Annotations" .

CHAPTER 5. USERS AND ROLES

The broker supports a flexible role-based security model for applying security to queues based on their respective addresses. It is important to understand that queues are bound to addresses either one-to-one (for point-to-point style messaging) or many-to-one (for publish-subscribe style messaging). When a message is sent to an address the server looks up the set of queues that are bound to that address and routes the message to that set of queues.

In the default configuration (using **PropertiesLoginModule**), users and their assigned roles are defined in three configuration files:

- **login.config**

- **artemis-users.properties**

- **artemis-roles.properties**

Each of these files is discussed in more detail in the following sections.

The command-line interface allows users and roles to be added to these files via an interactive process.

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

5.1. ENABLING GUEST ACCESS

A user who does not have login credentials, or whose credentials fail validation, can be granted limited access to the broker using a *guest* account.

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

The guest account is configured in the **login.config** file.

Procedure

1. In the **login.config** file, define a name and role for the guest account as follows:

```
activemq {
  org.apache.activemq.artemis.spi.core.security.jaas.GuestLoginModule required
    org.apache.activemq.jaas.guest.user="guest" 1
    org.apache.activemq.jaas.guest.role="guest"; 2
};
```

1   Define the username assigned to anonymous users.

2   Define the role assigned to anonymous users.

The guest login module allows users without credentials (and, depending on how it is configured, possibly also users with invalid credentials) to access the broker. It is implemented by **org.apache.activemq.artemis.spi.core.security.jaas.GuestLoginModule**.

It is common for the guest login module to be used in conjunction with another login module, such as a properties login module. Read more about that use-case in the Section 6.2.5, "Using Multiple Login Modules" section.

5.2. ADDING USERS

When basic username and password validation is required, use the *Properties* login module to define it. This login module checks the user's credentials against a set of local property files.

Users and their corresponding passwords are listed in the **BROKER_INSTANCE_DIR/etc/artemis-users.properties** file. The available roles and the users who are assigned those roles are defined in the **BROKER_INSTANCE_DIR/etc/artemis-roles.properties** file.

Both of these files are referenced in the **BROKER_INSTANCE_DIR/etc/login.config** file.

See the documentation from your Java vendor for more information on JAAS. For example, Oracle has a tutorial on configuring **login.config**.

Example 5.1. **login.config**

```
activemq {  (1)
    org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule required  (2)  (3)
        org.apache.activemq.jaas.properties.user="artemis-users.properties";  (4)
        org.apache.activemq.jaas.properties.role="artemis-roles.properties";
};
```

(1) An alias for a configuration. In this section the alias used is **activemq**. Substitute another in your environment.

(2) The implementation class (**org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule**).

(3) A flag which indicates whether the success of the **LoginModule is `required**, **requisite**, **sufficient**, or **optional**.

(4) A list of configuration options specific to the login module implementation.

Below is an explanation for each of the success states listed in the previous example:

Required

The LoginModule is required to succeed and authentication continues to proceed down the LoginModule list regardless of success or failure.

Requisite

The LoginModule is required to succeed. A failure immediately returns control to the application and authentication does not proceed down the LoginModule list.

Sufficient

The LoginModule 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 **LoginModule** list.

Optional

The LoginModule is not required to succeed. Authentication continues down the **LoginModules** list regardless of success or failure.

More information on these flags and the authentication process is available in the Oracle documentation.

Example 5.2. artemis-users.properties

```
user1=secret ❶
user2=swordfish ❷
user3=myPassword ❸
```

❶   *User1* has a password of *secret*.

❷   *User2* has a password of *swordfish*.

❸   *User3* has a password of *myPassword*.

Example 5.3. artemis-roles.properties

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

❶   *User1* and *user2* belong to the *admin* role.

❷   *User3* belongs to the *developer* role.

> **NOTE**
>
> If necessary, add your security domain alias (in this instance, *activemq*) to the **bootstrap.xml** file as shown below:
>
> ```
> <jaas-security domain="activemq"/>
> ```

5.3. SETTING PERMISSIONS

Permissions are defined against the queues based on their address via the **<security-setting>** element in **broker.xml**. Multiple instances of **<security-setting>** can be defined in **<security-settings>**. An exact match on the address can be used or a wildcard match can be used using the wildcard characters **#** and *****.

Different permissions can be given to the set of queues which match the address. Those permissions are:

Table 5.1. Permissions

To allow users to...	Use this parameter...
Create addresses	**createAddress**

To allow users to...	Use this parameter...
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**

For each permission, a list of roles who are granted that permission is specified. If the user has any of those roles, they are granted that permission for that set of addresses.

5.3.1. Configuring Message Production for a Single Address

To define sending permissions for a single address, a configuration similar to the example shown below is used:

```
<security-settings>
  <security-setting match="queue1">   1
    <permission type="send" roles="producer"/>   2
  </security-setting>
</security-settings>
```

1   Messages sent to this queue get the nominated permissions.

2   The permissions applied to messages in the specified queue.

In the above example, members of the *producer* role have *send* permissions on *queue1*.

5.3.2. Configuring Message Consumption for a Single Address

To define consuming permissions for a single address, a configuration similar to the example shown below is used:

```
<security-settings>
  <security-setting match="queue1">  ❶
    <permission type="consume" roles="consumer"/>  ❷
  </security-setting>
</security-settings>
```

❶ Messages sent to this queue get the nominated permissions.

❷ The permissions applied to messages in the specified queue.

In the above example, members of the *consumer* role have *consume* permissions on *queue1*.

5.3.3. Configuring Complete Access on All Addresses

To allow complete access to addresses and and queues, a configuration similar to the example shown below is used.

```
<security-settings>
  <security-setting match="#">  ❶
    <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>
```

❶ A wildcard setting to apply to all queues.

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

For information about more complex use cases see Configuring Multiple Permissions for Addresses .

5.3.4. Configuring a Queue with a User

The queue is assigned the username of the connecting client when it is auto-created. This is exposed as metadata on the queue. It is exposed by JMX and in the consoleThe queue is assigned the username of the connecting client when it is auto-created. This is exposed as metadata on the queue. It is exposed by JMX and in the console

You can configure a user on a pre-defined queue in **broker.xml**

To define a user for a queue, use a configuration similar to the example shown below:

```
<address name="ExempleQueue">
  <anycast>
```

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

> **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.

5.4. SETTING ROLE BASED ACCESS CONTROL

Role-based access control (RBAC) is used to restrict access to the attributes and methods of MBeans. RBAC enables administrators to grant the correct level of access to all users like web console, management interface, core messages, and so on based on their role. RBAC is configured using the **authorization** element in the *BROKER_INSTANCE_DIR*/**etc/management.xml** configuration file. Within the authorization element, you can configure **Whitelist**, **default-access**, and **role-access** sub-elements.

Prerequisites

You must first set up roles and add users to configure RBAC.

5.4.1. Configuring Whitelist Element for Bypassing the Authentication

A **whitelist** is a set of pre-approved domains or MBeans that do not require user authentication. You can provide a whitelist of domains or list of MBeans or both that must bypass the authentication. For example, you can use **whitelist** element for any MBeans that are needed by the AMQ Management Console to run.

Procedure

1. Open the broker *BROKER_INSTANCE_DIR*/**etc/management.xml** configuration file.

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

   ```
   <whitelist>
       <entry domain="hawtio"/>  1
   </whitelist>
   ```

   1   MBean of this domain will bypass the authentication.

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

3. Start or restart the broker by entering the following command:

   - On Linux: *BROKER_INSTANCE_DIR*/**bin/artemis run**

   - On Windows: *BROKER_INSTANCE_DIR***bin\artemis-service.exe start**

5.4.2. Configuring Authentication Based on Roles

The **role-access** element defines how roles are mapped to particular MBeans and their attributes and methods.

Procedure

1. Open the ***BROKER_INSTANCE_DIR*/etc/management.xml** configuration file.

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

```
<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 by entering the following command:

- 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. Shown below are some examples.

Mapping roles to all queues in a domain

This 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>
```

Mapping roles to a specific queue in a domain

This 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>
```

Mapping roles to all queue names that include a specified prefix

This example shows how to map roles to every queue whose name includes a specified prefix. In this example, a 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>
```

Mapping different roles to different sets of queues

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

CHAPTER 6. SECURITY

This chapter covers the various security options available to administrators, and how they are configured. Administrators can use the information provided in this chapter to tailor the functions of the AMQ Broker security subsystems to their needs.

6.1. ACCESSING THE AMQ MANAGEMENT CONSOLE

Starting with AMQ Broker 7.1.0, you can access the AMQ Management Console only from the local host by default. You must modify the configuration in **BROKER_INSTANCE_DIR**/**etc**/**jolokia-access.xml** to enable remote access. For more information, see Securing AMQ Management Console and AMQ Broker Connections.

6.2. SECURING NETWORK CONNECTIONS

There are two basic use cases for transport layer security (TLS):

- Server-side (or *one-way*); where only the server presents a certificate. This is the most common use case.

- Client-side (or *two-way*); where both the server and the client present certificates. This is sometimes called mutual authentication.

6.2.1. Configuring Server-Side Certificates

One-way TLS is configured in the URL of the relevant **acceptor** in **broker.xml**. Here is a very basic **acceptor** configuration which does not use TLS:

```
<acceptor name="artemis">tcp://0.0.0.0:61616</acceptor>
```

Here is that same **acceptor** configured to use one-way TLS:

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

This **acceptor** uses three additional parameters – **sslEnabled**, **keyStorePath**, and **keyStorePassword**. These, at least, are required to enable one-way TLS.

6.2.2. Configuring Client-Side Certificates

Two-way TLS uses the same **sslEnabled**, **keyStorePath**, and **keyStorePassword** properties as one-way TLS, but it adds **needClientAuth** to tell the client it should present a certificate of its own. For example:

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

This configuration 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 server needs to import the client's certificate into a trust-store and configure the acceptor with **trustStorePath** and **trustStorePassword**. For example:

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

NOTE

Client-side Considerations

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 the Core protocol (a bridge) the TLS parameters are configured on the connector URL much like on the broker's acceptor.

TLS Configuration Details

Below are configuration details to be aware of:

Option	Note
sslEnabled	Must be **true** to enable TLS. Default is **false**.
keyStorePath	**When used on an acceptor:** This is the path to the TLS 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 TLS key store which holds the client certificates. This is only relevant for a **connector** if you are using two-way TLS (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 AMQ-specific **org.apache.activemq.ssl.keyStore** system property. The AMQ-specific system property is useful if another component on client is already making use of the standard, Java system property.

Option	Note
keyStorePassword	**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 TLS (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 AMQ-specific **org.apache.activemq.ssl.keyStorePassword** system property. The AMQ-specific system property is useful if another component on client is already making use of the standard, Java system property.
trustStorePath	**When used on an acceptor:** This is the path to the server-side TLS 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 TLS (mutual authentication). **When used on a connector:** This is the path to the client-side TLS 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 AMQ-specific **org.apache.activemq.ssl.trustStore** system property. The AMQ-specific system property is useful if another component on client is already making use of the standard, Java system property.

Option	Note
trustStorePassword	**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 TLS (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 AMQ-specific **org.apache.activemq.ssl.trustStorePassword** system property. The AMQ-specific system property is useful if another component on client is already making use of the standard, Java system property.
enabledCipherSuites	Whether used on an **acceptor** or **connector** this is a comma-separated list of cipher suites used for TLS communication. The default value is **null** which means the JVM's default is used.
enabledProtocols	Whether used on an **acceptor** or **connector** this is a comma-separated list of protocols used for TLS communication. The default value is **null** which means the JVM's default is used.
needClientAuth	This property is only for an **acceptor**. It indicates to a client connecting to this acceptor that two-way TLS is required. Valid values are **true** or **false**. Default is **false**.

6.2.3. Adding Certificate-based Authentication

The JAAS certificate authentication login module requires TLS to be in use and clients must be configured with their own certificates. In this scenario, authentication is actually performed during the TLS handshake, not directly by the JAAS certificate authentication plug-in.

The role of the plug-in is as follows:

- To further constrain the set of acceptable users, because only the user Distinguished Names (DNs) explicitly listed in the relevant properties file are eligible to be authenticated.

- To associate a list of groups with the received user identity, facilitating integration with authorization.

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

The JAAS certificate login module stores a collection of certificate DNs in a pair of flat files. The files associate a username and a list of group IDs with each Distinguished Name.

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

Prerequisites

- Certificate login configured in **login.config** file.

- A valid **artemis-users.properties** file.

- A valid **artemis-roles.properties** file.

- The Subject DNs from the user certificate(s)

Procedure

1. Obtain the Subject DNs from user certificates

   a. Export the certificate from the KeyStore file into a temporary file. Substitute your required values into the following command:

      ```
      keytool -export -file __FILENAME__ -alias broker-localhost -keystore broker.ks -
      storepass __PASSWORD__
      ```

   b. Print the contents of the exported certificate:

      ```
      keytool -printcert -file __FILENAME__
      ```

      The output is similar to that shown below:

      ```
      Owner: CN=localhost, OU=broker, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
      ```
      1
      ```
      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
      ```

      1   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. Configuring certificate-based authentication

   a. Open the **login.config** file and reference the user and roles properties files.

   b. Open the files declared in the previous step and supply the required information:

Users and their corresponding DNs should be listed in the **artemis-users.properties** file. The available roles and the users who hold those roles are defined in the **artemis-roles.properties** file.

Examples of the syntax of these files is shown below.

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

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

Example Configuration

The following example shows how to configure certificate login module in the **login.config** file:

Example 6.1. login.config

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

❶ Configure the JAAS realm. This example uses a single **org.apache.activemq.artemis.spi.core.security.jaas.TextFileCertificateLoginModule**

❷ Toggle debugging on (**true**) or off (**false**). Default is **false**.

❸ Define the file used to store user data (relative to the directory containing the **login.config** file).

❹ Define the file used to store role data (relative to the directory containing the **login.config** file).

The **artemis-users.properties** file consists of a list of properties with the **user=StringifiedSubjectDN** (where the string encoding is specified by RFC 2253):

Example 6.2. artemis-users.properties

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

❶ The user named **system** is mapped to the **CN=system,O=Progress,C=US** subject DN.

The **artemis-roles.properties** file follows the pattern of **role=user** where **user** can be either a single user or a comma-separated list of users:

Example 6.3. artemis-roles.properties

```
admins=system
users=system,user 1
guests=guest
```

1   Multiple users can be included as a comma-separated entry.

6.2.4. Adding Certificate-based Authentication for AMQP Clients

Use the SASL EXTERNAL mechanism configuration parameter to configure your AQMP client for certificate-based authentication when connecting to a broker.

The broker authenticates the TSL/SSL certificate of your AMQP client in the same way that it authenticates any certificate:

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

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

Prerequisites
Before you can configure your AMQP clients to use certificate-based authentication, you must complete the following tasks:

- Configure a server-side SSL/TLS certificate.

- Configure a client-side SSL/TLS certificate.

- Configure the broker to use certificate-based authentication.

Procedure
To enable your AMQP client to use certificate-based authentication, add configuration parameters to the URI that the client uses to connect to a broker.

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 TrustStore and KeyStore of the client to enable the exchange of TSL/SSL certificates with the broker:

   ```
   amqps://localhost:5500?
   sslEnabled=true&trustStorePath=TRUST_STORE_PATH&trustStorePassword=TRUST_ST
   ORE_PASSWORD&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_ST
ORE_PASSWORD&keyStorePath=KEY_STORE_PATH&keyStorePassword=KEY_STORE_
PASSWORD&saslMechanisms=EXTERNAL
```

Additional Resources

- For more information about certificate-based authentication in AMQ Broker, see Section 6.2.3, "Adding 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.

6.2.5. Using Multiple Login Modules

It is possible to combine login modules to accommodate more complex use cases. The most common reason to combine login modules is to support authentication for both anonymous users and users who submit credentials.

Prerequisites

The prerequisites for different authentication combinations differ based on the methods being implemented. Prerequisites for the most common multiple login scenario are:

- A valid **artemis-users.properties** file

- A valid **artemis-roles.properties** file

- A **login.config** file configured for anonymous access

Procedure

1. Edit the **login.config** file to add entries for the desired authentication modules.

2. Set the parameters in each module entry as required for your environment.

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

   Example 6.4. **bootstrap.xml**

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

Example Configuration

The following examples illustrate the cascading nature of multiple login configurations:

Example 6.5. **login.config**

```
activemq {
   org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule sufficient ❶
      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 ②
            debug=true
            org.apache.activemq.jaas.guest.user="guest"
            org.apache.activemq.jaas.guest.role="restricted";
    };
```

① Password authentications module is activated if the user supplies credentials

② Guest authentication module is activated if the user supplies no credentials or the credentials supplied are incorrect.

The following example shows how to configure a JAAS login entry for the use case where only those users with no credentials are logged in as guests. Note that the order of the login modules is reversed and the flag attached to the properties login module is changed to **requisite**.

Example 6.6. login.config

```
activemq {
    org.apache.activemq.artemis.spi.core.security.jaas.GuestLoginModule sufficient ①
        debug=true
      credentialsInvalidate=true ②
      org.apache.activemq.jaas.guest.user="guest"
      org.apache.activemq.jaas.guest.role="guests";

    org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule requisite ③
        debug=true
        org.apache.activemq.jaas.properties.user="artemis-users.properties"
        org.apache.activemq.jaas.properties.role="artemis-roles.properties";
};
```

① The guest authentication module is activated if no login credentials are supplied.

② The **credentialsInvalidate** option must be set to **true** in the configuration of the guest login module.

③ The password login module is activated if credentials are supplied and the credentials must be valid.

6.2.6. Configure Multiple Security Settings for Address Groups and Sub-groups

Below is an example security block from a **broker.xml** file. The various configuration options based on this example are explained in this section.

```
<security-setting match="globalqueues.europe.#"> ①
    <permission type="createDurableQueue" roles="admin"/> ②
    <permission type="deleteDurableQueue" roles="admin"/> ③
    <permission type="createNonDurableQueue" roles="admin, guest, europe-users"/> ④
    <permission type="deleteNonDurableQueue" roles="admin, guest, europe-users"/> ⑤
```

```
    <permission type="send" roles="admin, europe-users"/> 6
    <permission type="consume" roles="admin, europe-users"/> 7
</security-setting>
```

[1] The '#' character signifies "any sequence of words". Words are delimited by the '.' character. For a full description of the wildcard syntax, see AMQ Broker Wildcard Syntax . The above security block applies to any address that starts with the string *globalqueues.europe.*

[2] [3] Only users who have the **admin** role can create or delete durable queues bound to an address that starts with the string *globalqueues.europe.*

[4] [5] 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.*

[6] [7] Any users with the roles **admin** or **europe-users** can send messages to these addresses or consume messages from queues bound to an address that starts with the string *globalqueues.europe.*

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.

There can be multiple **security-setting** elements in each XML file, or none, depending on requirements. When the **broker.xml** file contains multiple security–setting elements that can apply to a set of addresses, the most specific match takes precedence.

Let us look at an example of that, here's another **security-setting** block:

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

In this **security-setting** block the match **globalqueues.europe.orders.#** is more specific than the previous match 'globalqueues.europe.\#'. So any addresses which match 'globalqueues.europe.orders.\#' will take their security settings **only** from the latter security–setting block.

Note that settings are not inherited from the former block. All the settings will be taken from the more specific matching block, so for the address **globalqueues.europe.orders.plastics** the only permissions that exist are **send** and **consume** for the role **europe-users**. The permissions **createDurableQueue**, **deleteDurableQueue**, **createNonDurableQueue**, **deleteNonDurableQueue** are not inherited from the other security–setting block.

By not inheriting permissions, you can effectively deny permissions in more specific security–setting blocks by simply not specifying them. Otherwise it would not be possible to deny permissions in sub-groups of addresses.

6.2.7. 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. For example, one can limit how many connections a user can create or how many queues a user can create.

6.2.7.1. Configuring Connection and Queue Limits

Here is an example of the XML used to set resource limits:

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

Unlike the **match** from **address-setting**, this **match** does not use any wildcard syntax. It is a simple 1:1 mapping of the limits to a user.

- **max-connections**. Defines how many connections the matched user can make to the broker. The default is **-1**, which means there is no limit.

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

6.3. INTEGRATING WITH LDAP

6.3.1. Using LDAP for Authentication

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**.

Procedure

1. Open the **_BROKER_INSTANCE_DIR_/etc/broker.xml** file and add the following lines:

   ```xml
   <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>
   ```

2. Open the **_BROKER_INSTANCE_DIR_/etc/login.config** file.

3. Locate and edit the appropriate alias block with the appropriate parameters (see the examples included below).

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

NOTE

Apache DS uses the **OID** portion of DN path; however, Microsoft AD does not, and instead uses the **CN** portion.
For example; The DN path **oid=testuser,dc=example,dc=com** would be used in Apache DS, while **cn=testuser,dc=example,dc=com** would be used in Microsoft AD.

Example 6.7. Example Apache DS**login.config** configuration

```
activemq {
  org.apache.activemq.artemis.spi.core.security.jaas.LDAPLoginModule required
    debug=true 1
    initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory 2
    connectionURL="ldap://localhost:10389" 3
    connectionUsername="uid=admin,ou=system" 4
    connectionPassword=secret 5
    connectionProtocol=s 6
    connectionTimeout=5000 7
    authentication=simple 8
    userBase="dc=example,dc=com" 9
    userSearchMatching="(uid={0})" 10
    userSearchSubtree=true 11
    userRoleName= 12
    readTimeout=5000 13
    roleBase="dc=example,dc=com" 14
    roleName=cn 15
    roleSearchMatching="(member={0})" 16
    roleSearchSubtree=true 17
    ;
};
```

Example 6.8. Example Microsoft Active Directory**login.config** Configuration

```
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,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
        ;
    };
```

(1) Toggle debugging on (**true**) or off (**false**). Default is **false**.

(2) The **initialContextFactory** parameter must always be set to **com.sun.jndi.ldap.LdapCtxFactory**

(3) Specify the 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**.

(4) 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.

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

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

(7) 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.

(8) Specify 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).

(9) 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.

(10) Specify an LDAP search filter, which is applied to the subtree selected by **userBase**. See the Search Matching section below for more information.

(11) Specify the search depth for user entries, relative to the node specified by **userBase**. This option is a boolean. A setting of **false** indicates it tries to match one of the child entries of the **userBase** node (maps to **javax.naming.directory.SearchControls.ONELEVEL_SCOPE**), while **true** indicates it tries to match any entry belonging to the subtree of the **userBase** node (maps to **javax.naming.directory.SearchControls.SUBTREE_SCOPE**).

(12) Specify the name of the multi-valued attribute of the user entry that contains a list of role names for the user (where the 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.

13 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

14 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.

15 Specify the 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.

16 Specify an LDAP search filter, which is applied to the subtree selected by **roleBase**. See the Search Matching section below for more information.

17 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**).

Search Matching

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**.

A short introduction to the search filter syntax is available from Oracle's JNDI tutorial

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 username. For example, **jdoe**.

If this option is set to **(member=uid={1})** and the received username 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)**.

6.3.2. Configure 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.

Shown below is an example of the plugin configuration:

```
<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>
```

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 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; http://activemq.apache.org/security.html.

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:

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**.

6.3.3. 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 masked.

Prerequisites

- Ensure that you have modified the **login-config** file to add the required properties as described in Section 6.3.2, "Configure LDAP Authorization".

Procedure

The following procedure explains how to mask the value of the **connectionPassword** 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
   ```

   The encrypted password is displayed on the screen:

   ```
   result: 3a34fd21b82bf2a822fa49a8d8fa115d
   ```

2. Open the *BROKER_INSTANCE_DIR*/etc/login.config file and locate the **connectionPassword**:

   ```
   connectionPassword = PASSWORD
   ```

3. Replace the plain text password with the encrypted value that you created in Step 1:

   ```
   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.

6.4. INTEGRATING WITH KERBEROS

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 Sockets (TLS) to secure your messaging applications. SASL provides user authentication, and TLS provides data integrity.

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. If your operating system is RHEL 7, for example, see the chapter Using Kerberos. A Kerberos Authentication Overview is available for Windows as well.

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

> **NOTE**
>
> 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.

6.4.1. Enabling 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.

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

Procedure

1. Stop the broker.

   a. If the broker is running on Linux:

      ```
      BROKER_INSTANCE_DIR/bin/artemis stop
      ```

   b. If the broker is running on Windows as a service:

      ```
      BROKER_INSTANCE_DIR\bin\artemis-service.exe stop
      ```

2. Open the **broker.xml** configuration file located under *BROKER_INSTANCE_DIR*/etc

3. Add the name-value pair **saslMechanisms=GSSAPI** to the query string of the URL for the **acceptor**, as shown in the following example:

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

   The result is an acceptor that uses the GSSAPI mechanism when authenticating Kerberos credentials.

4. (Optional) The **PLAIN** and **ANONYMOUS** SASL mechanisms are also supported. If you want to use these other mechanisms in addition to **GSSAPI**, add them to the list of **saslMechanisms**. Be sure to separate each value with a comma. In the following example, the name-value pair **saslMechanisms=GSSAPI** is modified to add the value **PLAIN**.

   ```
   <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. If the broker is running on Linux:

      BROKER_INSTANCE_DIR/bin/artemis run

   b. If the broker is running on Windows as a service:

      BROKER_INSTANCE_DIR\bin\artemis-service.exe start

Related Information

See About Acceptors for more information about acceptors.

6.4.2. 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.

Procedure

1. Stop the broker.

   a. If the broker is running on Linux:

      BROKER_INSTANCE_DIR/bin/artemis stop

   b. If the broker is running on Windows as a service:

      BROKER_INSTANCE_DIR\bin\artemis-service.exe stop

2. Open the **login.config** configuration file located under *BROKER_INSTANCE_DIR*/**etc**.

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

   NOTE

   Verify the fully qualified class name of the **Krb5LoginModule** and its available options by referring to the documentation from your Java vendor.

   amqp-sasl-gssapi { **1**

```
        com.sun.security.auth.module.Krb5LoginModule required  2
        isInitiator=false
        storeKey=true
        useKeyTab=true  3
        principal="amqp/my_broker_host@example.com"  4
        debug=true;
    };
```

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

2 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.

3 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.

4 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. If the broker is running on Linux:

      BROKER_INSTANCE_DIR/bin/artemis run

   b. If the broker is running on Windows as a service:

      BROKER_INSTANCE_DIR\bin\artemis-service.exe start

Related Information
See Network Connections and Kerberos, for more information about enabling the GSSAPI mechanism in AMQ Broker.

6.4.2.1. 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**, which was 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>
```

6.4.3. Authorizing Clients with Kerberos Credentials

AMQ Broker is packaged with an implementation of the JAAS **Krb5LoginModule** 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**, which maps the Kerberos-authenticated Peer Principal to an AMQ Broker role.
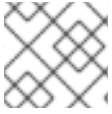
> **NOTE**
>
> The Kerberos Peer Principal does not exist as a broker user, only as a role member.

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. If the broker is running on Linux:

      BROKER_INSTANCE_DIR/bin/artemis stop

   b. If the broker is running on Windows as a service:

      BROKER_INSTANCE_DIR\bin\artemis-service.exe stop

2. Open the **login.config** configuration file located under *BROKER_INSTANCE_DIR*/**etc**.

3. Add configuration for the AMQ Broker **Krb5LoginModule** and the **LDAPLoginModule**.
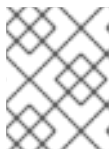
   > **NOTE**
   >
   > Verify the configuration options by referring to the documentation from your LDAP provider.

   ```
   org.apache.activemq.artemis.spi.core.security.jaas.Krb5LoginModule required
       ; ❶
   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
       ;
   ```

> **1** This version of the **Krb5LoginModule** is distributed with AMQ Broker and transforms the Kerberos identity into a broker identity that can be used by other AMQ modules for role

4. Start the broker.

   a. If the broker is running on Linux:

   > *BROKER_INSTANCE_DIR*/bin/artemis run

   b. If the broker is running on Windows as a service:

   > *BROKER_INSTANCE_DIR*\bin\artemis-service.exe start

Related Information
See Network Connections and Kerberos for more information about enabling the GSSAPI mechanism in AMQ Broker.

See Users and Roles for more information about the **PropertiesLoginModule**.

See Integrating with LDAP for more information about the **LDAPLoginModule**.

6.5. 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.

A 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 attributes **keyStorePassword** and **trustStorePassword**.

Example with bootstrap.xml

```
<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 only with the following configuration files.

Supported 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 Console.
>
> **broker.xml**, **bootstrap.xml**, **management.xml**, and **login.config** support passwords that are masked but not hashed.

Procedure

As an example, the following procedure explains how to mask the value of the **cluster-password** configuration element found in the file *BROKER_INSTANCE_DIR*/etc/broker.xml.

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

   ```
   $ BROKER_INSTANCE_DIR/bin/artemis mask PASSWORD
   ```

   The encrypted password is displayed on the screen:

   ```
   result: 3a34fd21b82bf2a822fa49a8d8fa115d
   ```

2. Open the configuration file containing the plain text password you want to mask:

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

3. Replace the plain text password with the encrypted value that you created in Step 1:

   ```
   <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 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.

6.6. 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**.

6.7. DISABLING SECURITY

Security is **enabled** by default. Broker security can be enabled or disabled by setting the **<security-enabled>** parameter in the **<core>** element of the **broker.xml** configuration file.

Procedure

1. Open the **broker.xml** file.

2. Locate the **<security-enabled>** parameter.

3. Edit the entry as needed:

- Set the parameter to **false** to disable security:

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

4. If necessary, change the **security-invalidation-interval** entry (which periodically invalidates secure logins) to a different value (in **ms**). The default is **10000**.

6.8. USING A CUSTOM SECURITY MANAGER

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

However, a system administrator might want more control over the implementation of broker security. In this case, it is 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.ActiveMQSecurityManager3** interface.

6.8.1. 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.ActiveMQSecurityManager3** interface. For example:

   ```
   <broker xmlns="http://activemq.org/schema">

     <security-manager class-name="com.foo.MySecurityManager">
       <property key="myKey1" value="myValue1"/>
       <property key="myKey2" value="myValue2"/>
     </security-manager>

     ...
   </broker>
   ```

Additional Resources

- For more information about the **org.apache.activemq.artemis.spi.core.security.ActiveMQSecurityManager3** interface, see Interface ActiveMQSecurityManager3 in the ActiveMQ Artemis Core JMS API documentation.

6.8.2. Running the Custom Security Manager Example Program

AMQ Broker includes 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

org.apache.activemq.artemis.spi.core.security.ActiveMQJAASSecurityManager (that is, the default security manager).

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

Prerequisites

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

Procedure

1. Navigate to the directory to the directory that contains the custom security manager example.

   ```
   $ cd <install-dir>/examples/features/standard/security-manager
   ```

2. Run the example.

   ```
   $ mvn verify
   ```

   > **NOTE**
   >
   > If you would 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**.

CHAPTER 7. PERSISTING MESSAGES

This chapter describes how persistence works with AMQ Broker and how to configure it.

The broker ships with two persistence options:

1. Journal-based
   The default. A highly performant option that writes messages to journals on the file system.

2. JDBC-based
   Uses the broker's JDBC Store to persist messages to a database of your choice.

Alternatively, you can also configure the broker for zero persistence.

The broker uses a different solution for persisting large messages outside the message journal. See Working with Large Messages for more information. The broker can also be configured to page messages to disk in low memory situations. See Paging Messages for more information.

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

7.1. ABOUT JOURNAL-BASED PERSISTENCE

A broker's 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 uses 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. Journal file sizing therefore is not an exact science.

Other persistence-related features include:

- A sophisticated file garbage collection algorithm that determines whether a particular journal file is still in use. If not, the file can be reclaimed and re-used.

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

- Support for local transactions.

- Support for XA transactions when using AMQ JMS clients.

The majority of the journal is written in Java. However, the interaction with the actual file system is abstracted, so you can use different, pluggable implementations. AMQ Broker ships with two implementations:

- Java NIO.
  Uses the 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.

- Linux Asynchronous IO
  Uses a thin native wrapper to talk to the Linux asynchronous IO library (AIO). 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.

  Using AIO typically provides even better performance than using Java NIO. For instructions on how to install libaio see Using an AIO journal .

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

7.1.1. Using AIO

The Java NIO journal is highly performant, but if you are running the broker using Linux Kernel 2.6 or later, Red Hat recommends using the AIO journal for better persistence performance. 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 **libaio** if it is not already installed.

Procedure

- Use the **yum** command to install **libaio**, as in the example below:

```
yum install libaio
```

7.2. CONFIGURING JOURNAL-BASED PERSISTENCE

Persistence configuration is maintained in the file **BROKER_INSTANCE_DIR/etc/broker.xml**. The broker's default configuration uses journal based persistence and includes the elements 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>
    ...
  </core>
</configuration>
```

persistence-enabled

Specify whether to use 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. The broker falls back to NIO if ASYNCIO is not found.

bindings-directory

File system location of the bindings journal. The default setting is relative to *BROKER_INSTANCE_DIR*.

journal-directory

File system location of the messaging journal. The default setting is relative to *BROKER_INSTANCE_DIR*.

journal-datasync

Specify whether to use **fdatasync** to confirm writes to the disk.

journal-min-files

Number of journal files to 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 clean up.

7.2.1. The Message Journal

The message journal stores all message-related data, including the messages themselves and duplicate ID caches. The files on this journal are prefixed as **activemq-data**. Each file has a **amq** extension and a default size of **10485760** bytes. The location of the message journal is set using the **journal-directory** configuration element. The default value is *BROKER_INSTANCE_DIR*/**data**/**journal**. The default configuration includes other elements related to the messaging journal:

- **journal-min-files**

  The number of journal files to pre-create when the broker starts. The default is **2**.

- **journal-pool-files**

  The number of files to keep after reclaiming un-used files. The default value, **-1,** means that no files are deleted once created by the broker. However, the system cannot grow infinitely, so you are required to use paging for destinations that are unbounded in this way. See the chapter on Paging Messages for more information.

There are several other configuration elements available for the messaging journal. See the appendix for a full list.

7.2.2. The Bindings Journal

The bindings journal is used to store bindings-related data, such as the set of queues deployed on the server 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 has a **bindings** extension and a default size of **1048576** bytes.

Use the following configuration elements in *BROKER_INSTANCE_DIR*/**etc**/**broker.xml** to configure the bindings journal.

- **bindings-directory**

  This is the directory in which the bindings journal lives. The default value is *BROKER_INSTANCE_DIR*/**data**/**bindings**.

- **create-bindings-dir**

  If this is set to **true** then the bindings directory is automatically created at the location specified in **bindings-directory** if it does not already exist. The default value is **true**

7.2.3. 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. Also, any JMS Resources created via the management API is persisted to this journal, but any resources configured via configuration files are not. The JMS Journal is only created if JMS is being used.

The files on this journal are prefixed as **activemq-jms**. Each file has a **jms** extension and and a default size of **1048576** bytes.

The JMS journal shares its configuration with the bindings journal.

7.2.4. Compacting Journal Files

AMQ Broker includes a compaction algorithm that removes dead space from the journal and compresses its data so that it takes up less space on disk. There are two criteria used to determine when to start compaction. After both criteria are met, the compaction process parses the journal and removes all dead records. Consequently, the journal comprises fewer files. The criteria are:

- The number of files created for the journal.

- The percentage of live data in the journal's files.

You configure both criteria in *BROKER_INSTANCE_DIR*/etc/broker.xml.

Procedure

- To configure the criteria for the compaction process, add the following two elements, as in the example below.

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

❶ The minimum number of files created before compaction begins. That is, the compacting algorithm does not start until you have at least **journal-compact-min-files**. The default value is **10**. Setting this to **0** disables compaction, which is dangerous because the journal could grow indefinitely.

❷ The percentage of live data in the journal's files. When less than this percentage is considered live data, compacting begins. Remember that compacting does not begin until you also have at least **journal-compact-min-files** data files on the journal. The default value is **30**.

Compacting Journals Using the CLI
You can also use the command-line interface (CLI) to compact journals.

Procedure

1. As the owner of the ***BROKER_INSTANCE_DIR***, stop the broker. In the example below, the user **amq-broker** was created during the installation of 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. Note that 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
   ```

Related Information

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

7.2.5. Disabling 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 the data has actually made it to disk, so 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.

Procedure

- On Linux, manage your disk's write cache settings using the tools **hdparm** (for IDE disks) or **sdparm** or **sginfo** (for SDSI/SATA disks).

- On Windows, manage the cache setting by right-clicking the disk and clicking **Properties**.

7.3. CONFIGURING JDBC PERSISTENCE

The JDBC persistence store uses a JDBC connection to store messages 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.

NOTE

An administrator might choose to store messaging 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.

Procedure

1. Add the appropriate JDBC client libraries to the broker runtime. You can do this by adding the relevant jars to the *BROKER_INSTANCE_DIR*/**lib** directory.

2. Create a **store** element in your *BROKER_INSTANCE_DIR*/**etc**/**broker.xml** configuration file under the **core** element, as in the example below.

```xml
<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>BINDINGS_TABLE</bindings-table-name>
    <message-table-name>MESSAGE_TABLE</message-table-name>
    <large-message-table-name>LARGE_MESSAGES_TABLE</large-message-table-name>
    <page-store-table-name>PAGE_STORE_TABLE</page-store-table-name>
    <node-manager-store-table-name>NODE_MANAGER_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>
   </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 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 Encrypting Passwords in Configuration Files.

bindings-table-name

Name of the table in which bindings data is stored. Specifying this 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 (that is, 20 seconds). 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. The default value is 2000 milliseconds (that is, 2 seconds).

jdbc-lock-expiration

Time, in milliseconds, that the current JDBC lock is considered active, even if the **jdbc-lock-renew-period** time has elapsed. Setting this property to a value greater than **jdbc-lock-renew-period** ensures that the lock is not immediately lost if the broker that owns the lock experiences an unexpected delay in renewing 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 is 20000 milliseconds (that is, 20 seconds).

jdbc-journal-sync-period

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

7.4. CONFIGURING ZERO PERSISTENCE

In some situations, zero persistence is sometimes required for a messaging system. Configuring the broker to perform zero persistence is straightforward. Set the parameter **persistence-enabled** in *BROKER_INSTANCE_DIR*/etc/broker.xml to **false**.

Note that if you set this parameter to false, then **zero** persistence occurs. That means no bindings data, message data, large message data, duplicate ID caches or paging data is persisted.

CHAPTER 8. PAGING MESSAGES

AMQ Broker transparently supports huge queues containing millions of messages while the server is running with limited memory.

In such a situation it's not possible to store all of the queues in memory at any one time, so AMQ Broker transparently **pages** messages into and out of memory as they are needed, thus allowing massive queues with a low memory footprint.

Paging is done individually per address. AMQ Broker will start paging messages to disk when the size of all messages in memory for an address exceeds a configured maximum size. For more information about addresses, see Addresses, Queues, and Topics.

By default, AMQ Broker does not page messages. You must explicitly configure paging to enable it.

See the **paging** example located under *INSTALL_DIR*/**examples**/**standard**/ for a working example showing how to use paging with AMQ Broker.

8.1. ABOUT PAGE FILES

Messages are stored per address on the file system. Each address has an individual folder where messages are stored in multiple files (page files). Each file will contain messages up to a max configured size (**page-size-bytes**). The system will navigate on the files as needed, and it will remove the page file as soon as all the messages are acknowledged up to that point.

Browsers will read through the page-cursor system.

Consumers with selectors will also navigate through the page-files and ignore messages that don't match the criteria.

> **NOTE**
>
> When you have a queue, and consumers filtering the queue with a very restrictive selector you may get into a situation where you won't be able to read more data from paging until you consume messages from the queue.
>
> Example: in one consumer you make a selector as 'color="red"' but you only have one color red one million messages after blue, you won't be able to consume red until you consume blue ones. This is different to browsing as we will "browse" the entire queue looking for messages and while we "depage" messages while feeding the queue.

8.2. CONFIGURING THE PAGING DIRECTORY LOCATION

To configure the location of the paging directory, add the **paging-directory** configuration element to the broker's main configuration file *BROKER_INSTANCE_DIR*/**etc**/**broker.xml**, as in the example below.

```xml
<configuration ...>
  ...
  <core ...>
    <paging-directory>/somewhere/paging-directory</paging-directory>
    ...
  </core>
</configuration>
```

AMQ Broker will create one directory for each address being paged under the configured location.

8.3. CONFIGURING AN ADDRESS FOR PAGING

Configuration for paging is done at the address level by adding elements to a specific **address-settings**, as in the example below.

```xml
<address-settings>
  <address-setting match="jms.paged.queue">
    <max-size-bytes>104857600</max-size-bytes>
    <page-size-bytes>10485760</page-size-bytes>
    <address-full-policy>PAGE</address-full-policy>
  </address-setting>
</address-settings>
```

In the example above, when messages sent to the address **jms.paged.queue** exceed **104857600** bytes in memory, the broker will begin paging.

> **NOTE**
>
> Paging is done individually per address. If you specify **max-size-bytes** for an address, each matching address does not exceed the maximum size that you specified. It DOES NOT mean that the total overall size of all matching addresses is limited to **max-size-bytes**.

This is the list of available parameters on the address settings.

Table 8.1. Paging Configuration Elements

Element Name	Description	Default
max-size-bytes	The maximum size in memory allowed for the address before the broker enters page mode.	-1 (disabled). When this parameter is disabled, the broker uses **global-max-size** as a memory-usage limit for paging instead. For more information, see Section 8.4, "Configuring a Global Paging Size".
page-size-bytes	The size of each page file used on the paging system.	10MiB (10 * 1024 * 1024 bytes)

Element Name	Description	Default
address-full-policy	Valid values are **PAGE**, **DROP**, **BLOCK**, and **FAIL**. If the value is **PAGE** then further messages will be paged to disk. If the value is **DROP** then further messages will be silently dropped. If the value is **FAIL** then the messages will be dropped and the client message producers will receive an exception. If the value is **BLOCK** then client message producers will block when they try and send further messages.	PAGE
page-max-cache-size	The system will keep up to this number of page files in memory to optimize IO during paging navigation.	5

8.4. CONFIGURING A GLOBAL PAGING SIZE

Sometimes configuring a memory limit per address is not practical, such as when a broker manages many addresses that have different usage patterns. In these situations, use the **global-max-size** parameter to set a global limit to the amount of memory the broker can use before it enters into the page mode configured for the address associated with the incoming message.

The default value for **global-max-size** is half of the maximum memory available to the Java virtual machine (JVM). You can specify your own value for this parameter by configuring it in the **broker.xml** configuration file. The value for **global-max-size** is in bytes, but you can use byte notation ("K", "Mb", "GB", for example) for convenience.

The following procedure shows how to configure the **global-max-size** parameter in the **broker.xml** configuration file.

Configuring the global-max-size parameter

Procedure

1. Stop the broker.

   a. If the broker is running on Linux, run the following command:

      BROKER_INSTANCE_DIR/bin/artemis stop

   b. If the broker is running on Windows as a service, run the following command:

      BROKER_INSTANCE_DIR\bin\artemis-service.exe stop

2. Open the **broker.xml** configuration file located under *BROKER_INSTANCE_DIR*/etc.

3. Add the **global-max-size** parameter to **broker.xml** to limit the amount of memory, in bytes, the broker can use. Note that you can also use byte notation (**K**, **Mb**, **GB**) for the value of **global-max-size**, as shown in the following example.

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

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

In the preceding example, the broker is configured to use a maximum of one gigabyte, **1GB**, of available memory when processing messages. If the configured limit is exceeded, the broker enters the page mode configured for the address associated with the incoming message.

4. Start the broker.

   a. If the broker is running on Linux, run the following command:

      BROKER_INSTANCE_DIR/bin/artemis run

   b. If the broker is running on Windows as a service, run the following command:

      BROKER_INSTANCE_DIR\bin\artemis-service.exe start

Related Information

See Section 8.3, "Configuring an Address for Paging" for information about setting the paging mode for an address.

8.5. LIMITING DISK USAGE WHEN PAGING

You can limit the amount of physical disk the broker uses before it blocks incoming messages rather than pages them. Add the **max-disk-usage** to the **broker.xml** configuration file and provide a value for the percentage of disk space the broker is allowed to use when paging messages. The default value for **max-disk-usage** is **90**, which means the limit is set at **90** percent of disk space.

Configuring the **max-disk-usage**

Procedure

1. Stop the broker.

   a. If the broker is running on Linux, run the following command:

      BROKER_INSTANCE_DIR/bin/artemis stop

   b. If the broker is running on Windows as a service, run the following command:

      BROKER_INSTANCE_DIR\bin\artemis-service.exe stop

2. Open the **broker.xml** configuration file located under *BROKER_INSTANCE_DIR*/etc.

3. Add the **max-disk-usage** configuration element and set a limit to the amount disk space to use when paging messages.

```
<configuration>
 <core>
  ...
   <max-disk-usage>50</max-disk-usage>
```

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

In the preceding example, the broker is limited to using **50** percent of disk space when paging messages. Messages are blocked and no longer paged after **50** percent of the disk is used.

4. Start the broker.

   a. If the broker is running on Linux, run the following command:

      BROKER_INSTANCE_DIR/bin/artemis run

   b. If the broker is running on Windows as a service, run the following command:

      BROKER_INSTANCE_DIR\bin\artemis-service.exe start

8.6. HOW TO DROP MESSAGES

Instead of paging messages when the max size is reached, an address can also be configured to just drop messages when the address is full.

To do this just set the **address-full-policy** to **DROP** in the address settings

8.6.1. Dropping Messages and Throwing an Exception to Producers

Instead of paging messages when the max size is reached, an address can also be configured to drop messages and also throw an exception on the client-side when the address is full.

To do this just set the **address-full-policy** to **FAIL** in the address settings

8.7. HOW TO BLOCK PRODUCERS

Instead of paging messages when the max size is reached, an address can also be configured to block producers from sending further messages when the address is full, thus preventing the memory from being exhausted on the server.

> **NOTE**
>
> Blocking works only if the protocol being used supports it. For example, an AMQP producer will understand a Block packet when it is sent by the broker, but a STOMP producer will not.

When memory is freed up on the server, producers will automatically unblock and be able to continue sending.

To do this just set the **address-full-policy** to **BLOCK** in the address settings.

In the default configuration, all addresses are configured to block producers after 10 MiB of data are in the address.

8.8. CAUTION WITH ADDRESSES WITH MULTICAST QUEUES

When a message is routed to an address that has multicast queues bound to it, for example, a JMS subscription in a Topic, there is only one copy of the message in memory. Each queue handles only a reference to it. Because of this the memory is only freed up after all queues referencing the message have delivered it.

If you have a single lazy subscription, the entire address will suffer IO performance hit as all the queues will have messages being sent through an extra storage on the paging system.

For example:

- An address has 10 queues

- One of the queues does not deliver its messages (maybe because of a slow consumer).

- Messages continually arrive at the address and paging is started.

- The other 9 queues are empty even though messages have been sent.

In this example, all the other 9 queues will be consuming messages from the page system. This may cause performance issues if this is an undesirable state.

CHAPTER 9. 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, AMQP, OpenWire and STOMP protocols.

For the Core 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.

9.1. 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 7.3, "Configuring JDBC Persistence".

9.2. 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).
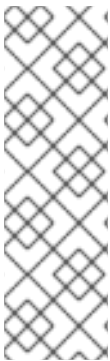
> **NOTE**
>
> - If you set **amqpMinLargeMessageSize** to -1, large message handling for AMQP messages is disabled.
>
> - If the broker receives a persistent AMQP message that does not exceed the value of **amqpMinLargeMessageSize**, but which *does* exceed the size of the messaging journal buffer (specified using the **journal-buffer-size** configuration parameter), the broker converts the message to a large Core message, before storing it in the journal.

9.3. 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.

9.4. 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

- To learn about working with large messages in the AMQ OpenWire 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 in the ***<install-dir>*/examples/features/standard/** directory of your AMQ Broker installation. To learn more about running example programs, see Running an AMQ Broker example program .

CHAPTER 10. 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)  1
```

1 The line in the client code where the connection was instantiated.

Detecting Dead Connections from the Client Side

As long as it is receiving data from the broker, the client considers a connection to be alive. Configure the client to check its connection for failure by providing a value for the **client-failure-check-period** property. The default check period for a network connection is **30000** milliseconds (30 seconds), while the default value for an In-VM connection, is **-1**, which means the client never fails the connection from its side if no data is received.

Typically, you set the check period to be much lower than the value used for the broker's connection time-to-live, which ensures that clients can reconnect in case of a temporary failure.

The examples below show how to set the check period to **10000** milliseconds (10 seconds) using Core JMS clients.

Procedure

- Set the check period for detecting dead connections.

  - If you are using JNDI with your Core JMS client, set the check period within the JNDI context environment, **jndi.properties**, for example, as below.

    ```
    java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory

    connectionFactory.myConnectionFactory=tcp://localhost:61616?
    clientFailureCheckPeriod=10000
    ```

  - If you are not using JNDI set the check period directly by passing a value to **ActiveMQConnectionFactory.setClientFailureCheckPeriod()**.

    ```
    ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
    cf.setClientFailureCheckPeriod(10000);
    ```

10.1. 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 (30 seconds). The default value is **-1**, which allows clients to set their own TTL.

❷ The interval between checks for dead connections is set to **1000** milliseconds (1 second). By default, the checks are done every **2000** milliseconds (2 seconds).

Configuring Time-To-Live on the Client

By default clients can set a TTL for their own connections. The examples below show you how to set the Time-To-Live using Core JMS clients.

Procedure

- Set the Time-To-Live for a Client Connection.

  - If you are using JNDI to instantiate your connection factory, you can specify it in the xml config, using the parameter **connectionTTL**.

    ```
    java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory

    connectionFactory.myConnectionFactory=tcp://localhost:61616?connectionTTL=30000
    ```

  - If you are not using JNDI, the connection TTL is defined by the **ConnectionTTL** attribute on a **ActiveMQConnectionFactory** instance.

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setConnectionTTL(30000);
```

10.2. 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**.

  ```xml
  <configuration>
   <core>
    ...
    <async-connection-execution-enabled>false</async-connection-execution-enabled>
    ...
   </core>
  </configuration>
  ```

10.3. CLOSING CONNECTIONS FROM THE CLIENT SIDE

A client application must close its resources in a controlled manner before it exits to prevent dead connections from occurring. In Java, it is recommended to close connections inside a **finally** block:

```java
Connection jmsConnection = null;
try {
   ConnectionFactory jmsConnectionFactory =
ActiveMQJMSClient.createConnectionFactoryWithoutHA(...);
   jmsConnection = jmsConnectionFactory.createConnection();
   ...use the connection...
}
finally {
   if (jmsConnection != null) {
```

```
      jmsConnection.close();
   }
}
```

CHAPTER 11. FLOW CONTROL

Flow control prevents producers and consumers from becoming overburdened by limiting the flow of data between them. Using AMQ Broker allows you to configure flow control for both consumers and producers.

11.1. CONSUMER FLOW CONTROL

Consumer flow control regulates the flow of data between the broker and the client as the client consumes messages from the broker. AMQ Broker clients buffer messages by default before delivering them to consumers. Without a buffer, the client would first need to request each message from the broker before consuming it. This type of "round-trip" communication is costly. Regulating the flow of data on the client side is important because out of memory issues can result when a consumer cannot process messages quickly enough and the buffer begins to overflow with incoming messages.

11.1.1. Setting the Consumer Window Size

The maximum size of messages held in the client-side buffer is determined by its *window size*. The default size of the window for AMQ Broker clients is 1 MiB, or 1024 * 1024 bytes. The default is fine for most use cases. For other cases, finding the optimal value for the window size might require benchmarking your system. AMQ Broker allows you to set the buffer window size if you need to change the default.

Setting the Window Size
The following examples demonstrate how to set the consumer window size parameter when using a Core JMS client. Each example sets a consumers window size to **300000** bytes.

Procedure

- Set the consumer window size.

  - If the Core JMS Client uses JNDI to instantiate its connection factory, include the **consumerWindowSize** parameter as part of the connection string URL. Store the URL within a JNDI context environment. The example below uses a **jndi.properties** file to store the URL.

    ```
    java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory

    connectionFactory.myConnectionFactory=tcp://localhost:61616?
    consumerWindowSize=300000
    ```

  - If the Core JMS client does not use JNDI to instantiate its connection factory, pass a value to **ActiveMQConnectionFactory.setConsumerWindowSize()**.

    ```
    ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
    cf.setConsumerWindowSize(300000);
    ```

11.1.2. Handling Fast Consumers

Fast consumers can process messages as fast as they consume them. If you are confident that the consumers in your messaging system are that fast, consider setting the window size to **-1**. This setting allows for unbounded message buffering on the client side. Use this setting with caution, however. It can overflow client-side memory if the consumer is not able to process messages as fast as it receives them.

Setting the Window Size for Fast Consumers

Procedure

The examples below show how to set the window size to **-1** when using a Core JMS client that is a fast consumer of messages.

- Set the consumer window size to **-1**.

  - If the Core JMS Client uses JNDI to instantiate its connection factory, include the **consumerWindowSize** parameter as part of the connection string URL. Store the URL within a JNDI context environment. The example below uses a **jndi.properties** file to store the URL.

    ```
    java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory

    connectionFactory.myConnectionFactory=tcp://localhost:61616?consumerWindowSize=-1
    ```

  - If the Core JMS client does not use JNDI to instantiate its connection factory, pass a value to **ActiveMQConnectionFactory.setConsumerWindowSize()**.

    ```
    ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
    cf.setConsumerWindowSize(-1);
    ```

11.1.3. Handling Slow Consumers

Slow consumers take significant time to process each message. In these cases, it is recommended to not buffer messages on the client side. Messages remain on the broker side ready to be consumed by other consumers instead. One benefit of turning off the buffer is that it provides deterministic distribution between multiple consumers on a queue. To handle slow consumers by disabling the client-side buffer, set the window size to **0**.

Setting the Window Size for Slow Consumers

Procedure

The examples below show you how to set the window size to **0** when using the Core JMS client that is a slow consumer of messages.

- Set the consumer window size to **0**.

  - If the Core JMS Client uses JNDI to instantiate its connection factory, include the **consumerWindowSize** parameter as part of the connection string URL. Store the URL within a JNDI context environment. The example below uses a **jndi.properties** file to store the URL.

    ```
    java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory

    connectionFactory.myConnectionFactory=tcp://localhost:61616?consumerWindowSize=0
    ```

  - If the Core JMS client does not use JNDI to instantiate its connection factory, pass a value to **ActiveMQConnectionFactory.setConsumerWindowSize()**.

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setConsumerWindowSize(0);
```

Related Information

See the example **no-consumer-buffering** in *INSTALL_DIR*/**examples/standard** for an example that shows how to configure the broker to prevent consumer buffering when dealing with slow consumers.

11.1.4. Setting the Rate of Consuming Messages

You can regulate the rate at which a consumer can consume messages. Also known as "throttling", regulating the rate of consumption ensures that a consumer never consumes messages at a rate faster than configuration allows.

> **NOTE**
>
> Rate-limited flow control can be used in conjunction with window-based flow control. Rate-limited flow control affects only how many messages a client can consume in a second and not how many messages are in its buffer. With a slow rate limit and a high window-based limit, the internal buffer of the client fills up with messages quickly.

The rate must be a positive integer to enable this functionality and is the maximum desired message consumption rate specified in units of messages per second. Setting the rate to **-1** disables rate-limited flow control. The default value is **-1**.

Setting the Rate of Consuming Messages

Procedure

The examples below use a Core JMS client that limits the rate of consuming messages to **10** messages per second.

- Set the consumer rate.

  - If the Core JMS Client uses JNDI to instantiate its connection factory, include the **consumerMaxRate** parameter as part of the connection string URL. Store the URL within a JNDI context environment. The example below uses a **jndi.properties** file to store the URL.

    ```
    java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory

    java.naming.provider.url=tcp://localhost:61616?consumerMaxRate=10
    ```

  - If the Core JMS client does not use JNDI to instantiate its connection factory, pass the value to **ActiveMQConnectionFactory.setConsumerMaxRate()**.

    ```
    ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
    cf.setConsumerMaxRate(10);
    ```

Related information

See the **consumer-rate-limit** example in *INSTALL_DIR*/**examples/standard** for a working example of how to limit the consumer rate.

11.2. PRODUCER FLOW CONTROL

In a similar way to consumer window-based flow control, AMQ Broker can limit the amount of data sent from a producer to a broker to prevent the broker from being overburdened with too much data. In the case of a producer, the window size determines the amount of bytes that can be in-flight at any one time.

11.2.1. Setting the Producer Window Size

The window size is negotiated between the broker and producer on the basis of credits, one credit for each byte in the window. As messages are sent and credits are used, the producer must request, and be granted, credits from the broker before it can send more messages. The exchange of credits between producer and broker regulates the flow of data between them.

Setting the Window Size

The following examples demonstrate how to set the producer window size to **1024** bytes when using Core JMS clients.

Procedure

- Set the producer window size.

  - If the Core JMS Client uses JNDI to instantiate its connection factory, include the **producerWindowSize** parameter as part of the connection string URL. Store the URL within a JNDI context environment. The example below uses a **jndi.properties** file to store the URL.

    ```
    java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory

    java.naming.provider.url=tcp://localhost:61616?producerWindowSize=1024
    ```

  - If the Core JMS client does not use JNDI to instantiate its connection factory, pass the value to **ActiveMQConnectionFactory.setProducerWindowSize()**.

    ```
    ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
    cf.setProducerWindowSize(1024);
    ```

11.2.2. Blocking Messages

Because more than one producer can be associated with the same address, it is possible for the broker to allocate more credits across all producers than what is actually available. However, you can set a maximum size on any address that prevents the broker from sending more credits than are available.

In the default configuration, a global maximum size of 100Mb is used for each address. When the address is full, the broker writes further messages to the paging journal instead of routing them to the queue. Instead of paging, you can block the sending of more messages on the client side until older messages are consumed. Blocking producer flow control in this way prevents the broker from running out of memory due to producers sending more messages than can be handled at any one time.

In the configuration, blocking producer flow control is managed on a per **address-setting** basis. The configuration applies to all queues registered to an address. In other words, the total memory for all queues bound to that address is capped by the value given for **max-size-bytes**.

NOTE

Blocking is protocol dependent. In AMQ Broker the AMQP, OpenWire, and Core protocols feature producer flow control. The AMQP protocol handles flow control differently, however. See Blocking Flow Control Using AMQP for more information.

Configuring the Maximum Size for an Address

To configure the broker to block messages if they are larger than the set maximum number of bytes, add a new **addres-setting** configuration element to *BROKER_INSTANCE_DIR*/etc/broker.xml.

Procedure

- In the example configuration below, an **address-setting** is set to **BLOCK** producers from sending messages after reaching its maximum size of **300000** bytes.

```
<configuration>
  <core>
    ...
    <address-settings>
      <address-setting match="my.blocking.queue">    1
        <max-size-bytes>300000</max-size-bytes>    2
        <address-full-policy>BLOCK</address-full-policy>    3
      </address-setting>
    </address-settings>
  </core>
</configuration>
```

1 The above configuration applies to any queue referenced by the **my.blocking.queue** address .

2 Sets the maximum size to **300000** bytes. The broker will block producers from sending to the address if the message exceeds **max-size-bytes**. Note that this element supports byte notation such as "K", "Mb", and "GB".

3 Sets the **address-full-policy** to **BLOCK** to enable blocking producer flow control.

11.2.3. Blocking AMQP Messages

As explained earlier in this chapter the Core protocol uses a producer window–size flow control system. In this system, credits represent bytes and are allocated to producers. If a producer wants to send a message, it must wait until it has sufficient credits to accommodate the size of a message before sending it.

AMQP flow control credits are not representative of bytes, however, but instead represent the number of messages a producer is permitted to send, regardless of the message size. It is therefore possible in some scenarios for an AMQP client to significantly exceed the **max-size-bytes** of an address.

To manage this situation, add the element **max-size-bytes-reject-threshold** to the **address-setting** to specify an upper bound on an address size in bytes. Once this upper bound is reached, the broker rejects AMQP messages. By default, **max-size-bytes-reject-threshold** is set to **-1**, or no limit.

Configuring the Broker to Block AMQP Messages

To configure the broker to block AMQP messages if they are larger than the set maximum number of bytes, add a new **addres-setting** configuration element to *BROKER_INSTANCE_DIR*/etc/broker.xml.

Procedure

- The example configuration below applies a maximum size of **300000** bytes to any AMQP message routed to the **my.amqp.blocking.queue** address.

```
<configuration>
 <core>
   ...
    <address-settings>
      ...
       <address-setting match="my.amqp.blocking.queue">  1
          <max-size-bytes-reject-threshold>300000</max-size-bytes-reject-threshold>  2
       </address-setting>
     </address-settings>
    </core>
</configuration>
```

1 The above configuration applies to any queue referenced by the **my.amqp.blocking.queue** address.

2 The broker is configured to reject AMQP messages sent to queues matching this address if they are larger than the **max-size-bytes-reject-threshold** of **300000** bytes. Note that this element *does not* support byte notation such as **K**, **Mb**, and **GB**.

Addtional resources

- For more information about how to configure credits for AMQP producers, see *Chapter 3, Network Connections: Protocols*.

11.2.4. Setting the Rate of Sending Messages

AMQ Broker can also limit the rate a producer can emit messages. The producer rate is specified in units of messages per second. Setting it to **-1**, the default, disables rate-limited flow control.

Setting the Rate of Sending Messages
The examples below demonstrate how to set the rate of sending messages when the producer is using a Core JMS client. Each example sets the maximum rate of sending messages to **10** per second.

Procedure

- Set the rate that a producer can send messages.

  - If the Core JMS Client uses JNDI to instantiate its connection factory, include the **producerMaxRate** parameter as part of the connection string URL. Store the URL within a JNDI context environment. The example below uses a **jndi.properties** file to store the URL.

    ```
    java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory

    java.naming.provider.url=tcp://localhost:61616?producerMaxRate=10
    ```

  - If the Core JMS client does not use JNDI to instantiate its connection factory, pass the value to **ActiveMQConnectionFactory.setProducerMaxRate()**.

```
ConnectionFactory cf =  ActiveMQJMSClient.createConnectionFactory(...)
cf.setProducerMaxRate(10);
```

Related Information

See the **producer-rate-limit** example in *INSTALL_DIR*/**examples**/**standard** for a working example of how to limit a the rate of sending messages.

CHAPTER 12. MESSAGE GROUPING

Message groups are sets of messages that have the following characteristics:

- Messages in a message group share the same group ID, that is, they have same group identifier property. For JMS messages, the property is **JMSXGroupID**.

- Messages in a message group are always consumed by the same consumer, even if there are many consumers on a queue. Another consumer is chosen to receive a message group if the original consumer closes.

Message groups are useful when you want all messages for a certain value of the property to be processed serially by the same consumer. For example, you may want orders for any particular stock purchase to be processed serially by the same consumer. To do this you could create a pool of consumers, then set the stock name as the value of the message property. This ensures that all messages for a particular stock are always processed by the same consumer.

> **NOTE**
>
> Grouped messages might impact the concurrent processing of non-grouped messages due to the underlying FIFO semantics of a queue. For example, if there is a chunk of 100 grouped messages at the head of a queue followed by 1,000 non-grouped messages, all the grouped messages are sent to the appropriate client before any of the non-grouped messages are consumed. The functional impact in this scenario is a temporary suspension of concurrent message processing while all the grouped messages are processed. Keep this potential performance bottleneck in mind when determining the size of your message groups. Consider whether to isolate your grouped messages from your non-grouped messages.

12.1. CLIENT-SIDE MESSAGE GROUPING

The examples below show how to use message grouping using Core JMS clients.

Procedure

- Set the group ID.

  - If you are using JNDI to establish a JMS connection factory for your JMS client, add the **groupID** parameter and supply a value. All messages sent using this connection factory have the property **JMSXGroupID** set to the specified value.

    ```
    java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory

    connectionFactory.myConnectionFactory=tcp://localhost:61616?groupID=MyGroup
    ```

  - If you are not using JNDI, set the **JMSXGroupID** property using the **setStringProperty()** method.

    ```
    Message message = new TextMessage();
    message.setStringProperty("JMSXGroupID", "MyGroup");
    producer.send(message);
    ```

Related Information

See **mesagge-group** and **message-group2** under *INSTALL_DIR***/examples/features/standard** for working examples of how message groups are configured and used.

12.2. AUTOMATIC MESSAGE GROUPING

Instead of supplying a group ID yourself, you can have the ID automatically generated for you. Messages grouped in this way are still processed serially by a single consumer.

Procedure

The examples below show how to enable message grouping using Core JMS clients.

- Enable automatic generation of the group ID.

  - If you are using a JNDI context environment to instantiate your JMS connection factory, add the **autogroup=true** name-value pair to the query string of the connection URL.

    ```
    java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory

    connectionFactory.myConnectionFactory=tcp://localhost:61616?autoGroup=true
    ```

  - If you are not using JNDI, set **autogroup** to **true** on the **ActiveMQConnectonFactory**.

    ```
    ActiveMQConnectionFactory cf =
    ActiveMQJMSClient.createConnectionFactoryWithoutHA(...);
    cf.setAutoGroup(true);
    ```

CHAPTER 13. DUPLICATE MESSAGE DETECTION

AMQ Broker includes automatic duplicate message detection, which filters out any duplicate messages it receives so you do not have to code your own duplicate detection logic.

Without duplicate detection, a client cannot determine whether a message it sent was successful whenever the target broker or the connection to it fails. For example, if the broker or connection fails *before* the message was received and processed by the broker, the message never arrives at its address, and the client does not receive a response from the broker due to the failure. On the other hand, if the broker or connection failed *after* a message was received and processed by the broker, the message is routed correctly, but the client still does not receive a response.

Moreover, using a transaction to determine success does not help in these cases. If the broker or connection fails while the transaction commit is being processed, for example, the client is still unable to determine whether it successfully sent the message.

If the client resends the last message in an effort to correct the assumed failure, the result could be a duplicate message being sent to the address, which could negatively impact your system. Sending a duplicate message could mean that a purchase order is fulfilled twice, for example. Fortunately, {AMQ Broker} provides automatic duplicate messages detection as a way to prevent these kind of issues from happening.

13.1. USING THE DUPLICATE ID MESSAGE PROPERTY

To enable duplicate message detection provide a unique value for the message property **_AMQ_DUPL_ID**. When a broker receives a message, it checks if **_AMQ_DUPL_ID** has a value. If it does, the broker then checks in its memory cache to see if it has already received a message with that value. If a message with the same value is found, the incoming message is ignored.

Procedure

The examples below illustrate how to set the duplicate detection property using a Core JMS Client. Note that for convenience, the clients use the value of the constant **org.apache.activemq.artemis.api.core.Message.HDR_DUPLICATE_DETECTION_ID** for the name of the duplicate ID property, **_AMQ_DUPL_ID**.

- Set the value for **_AMQ_DUPL_ID** to a unique **String**.

  ```
  Message jmsMessage = session.createMessage();
  String myUniqueID = "This is my unique id";
  message.setStringProperty(HDR_DUPLICATE_DETECTION_ID.toString(), myUniqueID);
  ```

13.2. CONFIGURING THE DUPLICATE ID CACHE

The broker maintains caches of received values of the **_AMQ_DUPL_ID** property. Each address has its own distinct cache. The cache is circular and fixed. New entries replace the oldest ones as cache space demands.

> **NOTE**
>
> Be sure to size the cache appropriately. If a previous message arrived more than **id-cache-size** messages before the arrival of a new message with the same **_AMQ_DUPL_ID**, the broker cannot detect the duplicate. This results in both messages being processed by the broker.

Procedure

The example configuration below illustrates how to configure the ID cache by adding elements to **BROKER_INSTANCE_DIR/etc/broker.xml**.

```
<configuration>
 <core>
   ...
   <id-cache-size>5000</id-cache-size>      1
   <persist-id-cache>false</persist-id-cache>      2
 </core>
</configuration>
```

1   The maximum size of the cache is configured by the parameter **id-cache-size**. The default value is **20000** entries. In the example above, the cache size is set to **5000** entries.

2   Set **persist-id-cache** to **true** to have each ID persisted to disk as they are received. The default value is **true**. In the example above, persistence is disabled by setting the value to **false**.

13.3. DUPLICATE DETECTION AND TRANSACTIONS

Using duplicate detection to move messages between brokers can give you the same once and only once delivery guarantees as using an XA transaction to consume messages, but with less overhead and much easier configuration than using XA.

If you are sending messages in a transaction, you do not have to set **_AMQ_DUPL_ID** for every message in the transaction, but only in one of them. If the broker detects a duplicate message for any message in the transaction, it ignores the entire transaction.

13.4. DUPLICATE DETECTION AND CLUSTER CONNECTIONS

You can configure cluster connections to insert a duplicate ID for each message they move across the cluster.

Procedure

- Add the element **use-duplicate-detection** to the configuration of the desired cluster connection found in **BROKER_INSTANCE_DIR/etc/broker.xml**. Note that the default value for this parameter is **true**. In the example below, the element is added to the configuration for the cluster connection **my-cluster**.

  ```
  <configuration>
   <core>
     ...
     <cluster-connection>
       <cluster-connection name="my-cluster"> 2
         <use-duplicate-detection>true</use-duplicate-detection>
         ...
       </cluster-connection>
       ...
     </cluster-connections>
   </core>
  </configuration>
  ```

Related Information

For more information about broker clusters, see Section 17.2, "Creating a broker cluster".

CHAPTER 14. 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.

14.1. 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 the 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;
  }
}
```

14.2. 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>
```

14.3. INTERCEPTORS ON THE CLIENT SIDE

Clients can use interceptors to intercept packets either sent by the client to the server or by the server to the client. As in the case of a broker-side interceptor, if it returns **false**, no other interceptors are called and the client does not process the packet further. This process happens transparently to the client except when an outgoing packet is sent in a **blocking** fashion. In those cases, an **ActiveMQException** is thrown to the caller because blocking sends provides reliability. The **ActiveMQException** thrown contains the name of the interceptor that returned false.

As on the server, the client interceptor classes and their dependencies must be added to the Java classpath of the client to be properly instantiated and invoked.

CHAPTER 15. 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.

15.1. 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.

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 JMS 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.

15.2. 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.

15.2.1. 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.

15.2.2. 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 Divert Example (external).

CHAPTER 16. 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.

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.

16.1. 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>
```

16.2. 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**.

```xml
<configuration>
  <core>
    ...
    <addresses>
      <address name="myQueue">
        <anycast>
          <queue name="myQueue">
            <filter string="convert_string_expressions='age > 18'"/>
          </queue>
        </anycast>
      </address>
    </addresses>
  </core>
</configuration>
```

16.3. 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.
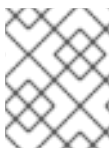
> **NOTE**
>
> The properties that the broker applies are *internal* properties These properties are are not exposed to clients for regular use, but **can** be specified by a client in a filter.

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 4.21, "Annotations and Properties on Expired or Undelivered AMQP Messages".

CHAPTER 17. 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.

17.1. UNDERSTANDING BROKER CLUSTERS

Before creating a broker cluster, you should understand some important clustering concepts.

17.1.1. 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. The exact order depends on the order in which the brokers started.

Figure 17.1. Message load balancing
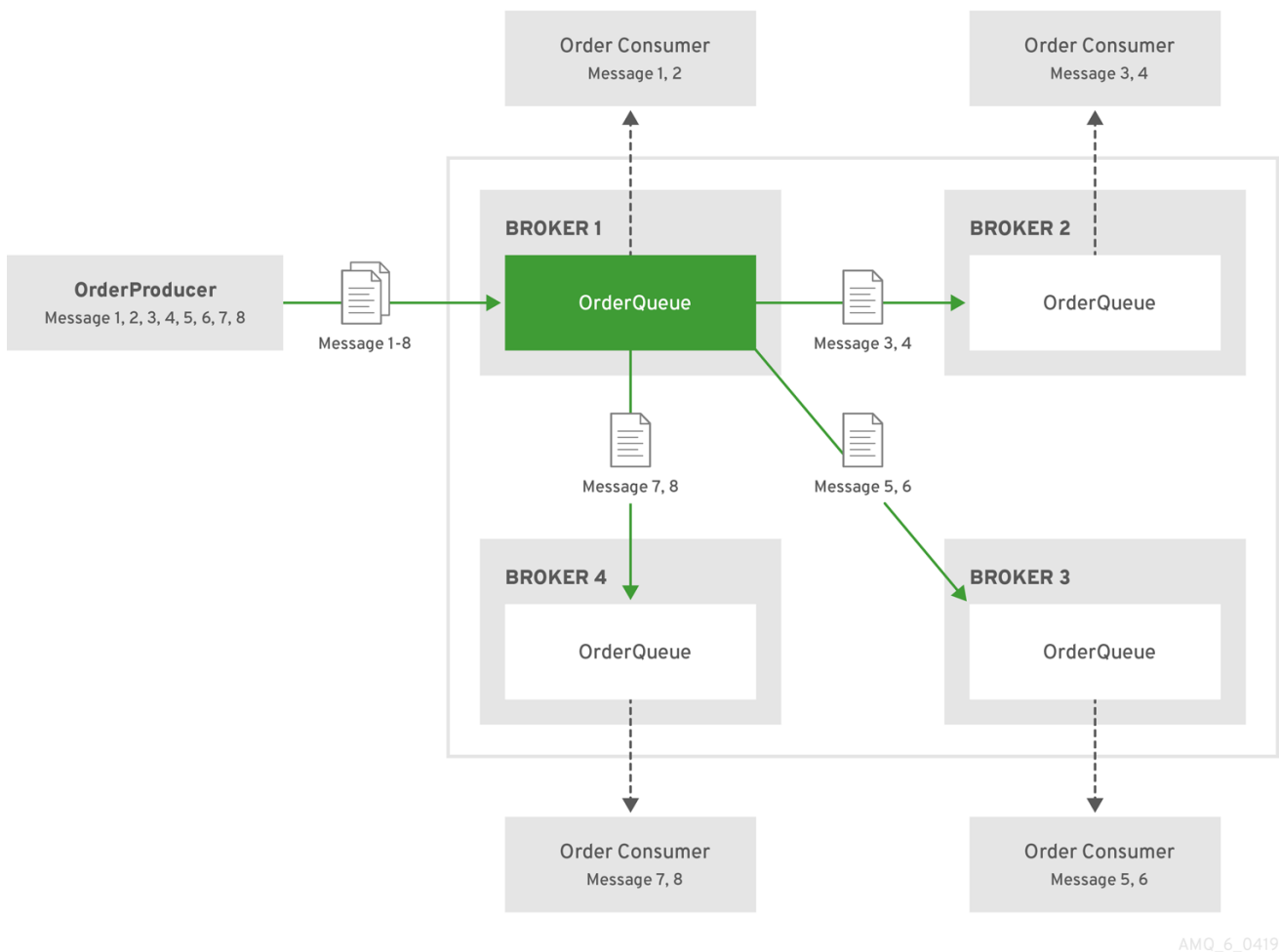
Without message load balancing, the messages sent to **Broker1** would stay on **Broker1** and only **OrderConsumer1** would be able to consume them.

While AMQ Broker automatically load balances messages by default, you can configure the cluster to only load balance messages to brokers that have a matching consumer. You can also configure message redistribution 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 17.4.2, "Configuring message redistribution".

17.1.2. 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.

17.1.3. 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.

17.1.4. 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 17.2. Symmetric cluster
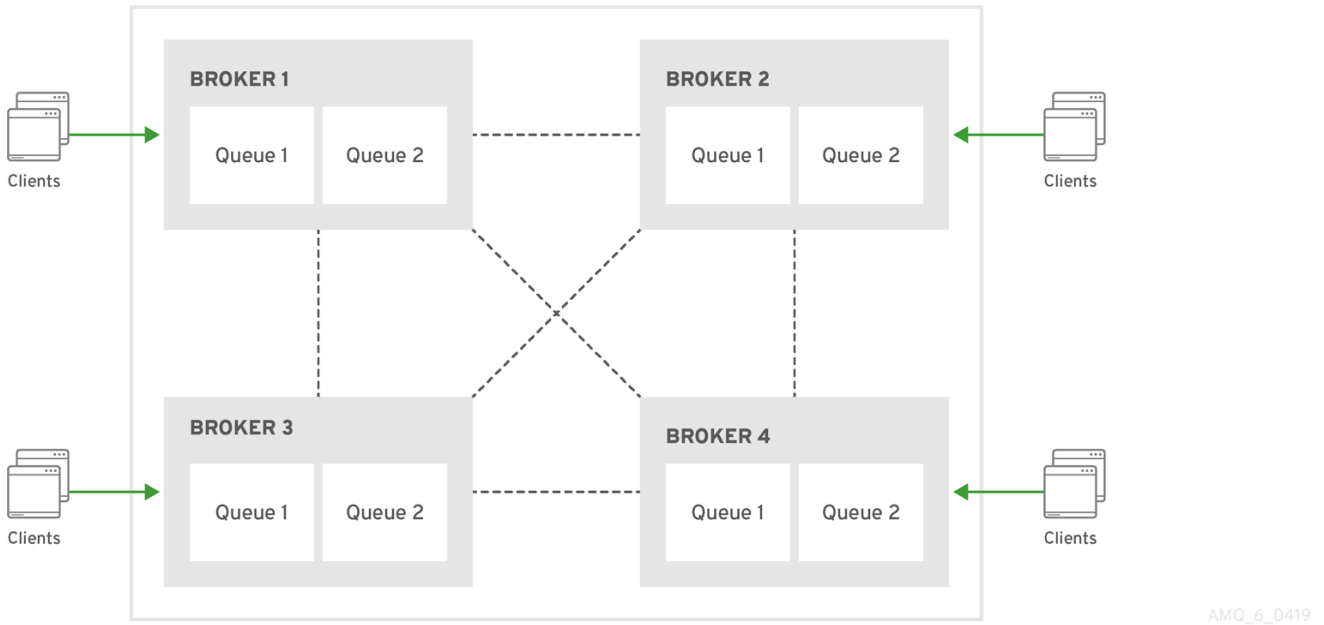
Each broker in a symmetric cluster is aware of all of the queues that exist on every other broker in the cluster and the consumers that are listening on those queues. Therefore, symmetric clusters are able to load balance and redistribute messages more optimally than a chain cluster.

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 17.3. Chain cluster
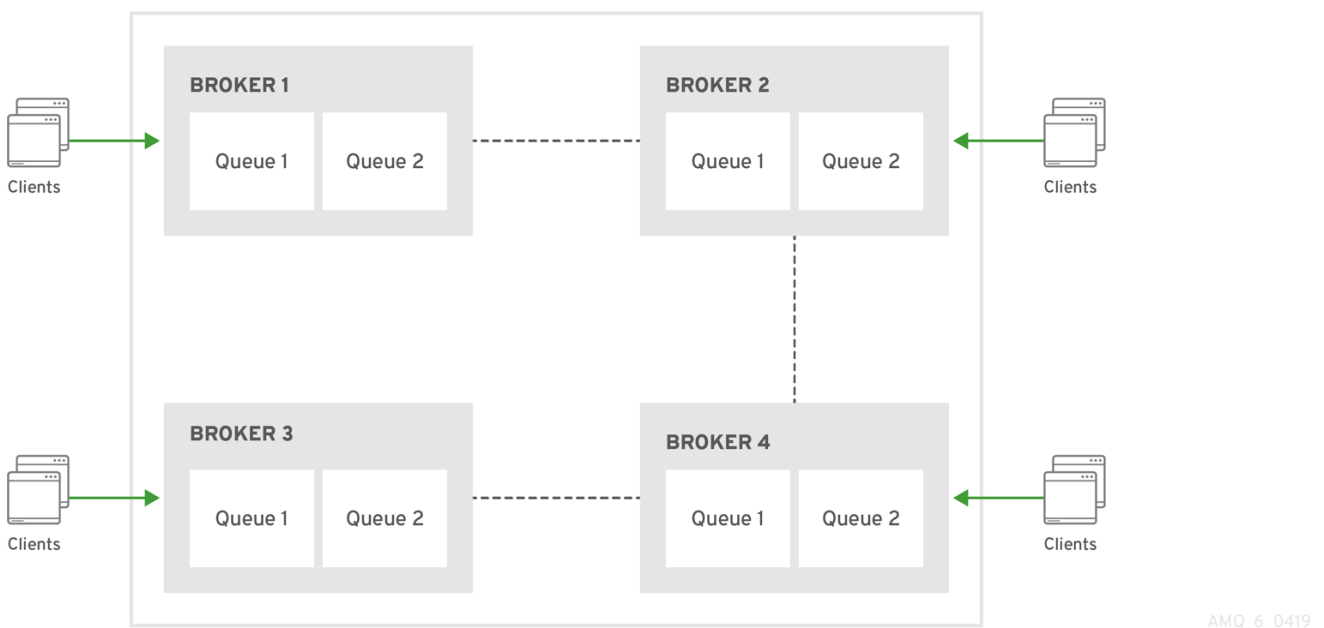
Chain clusters are more difficult to set up than symmetric clusters, but can be useful when brokers are

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.

17.1.5. 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.

17.1.6. 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.
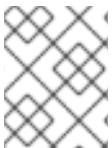
> **NOTE**
>
> After creating the cluster, you can adjust the size by adding and removing brokers. You can add and remove brokers without losing any messages.

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 17.1.4, "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 17.3, "Implementing high availability".

17.2. 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 17.1.6, "Cluster sizing considerations".

17.2.1. 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 2.2, "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.
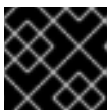
   > **IMPORTANT**
   >
   > The cluster password must be the same on every broker in the cluster.

   ```
   <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** AMQ Broker example program .

17.2.2. 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 (2 seconds).

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:

   ```xml
   <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.

```xml
<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** AMQ Broker example program .

17.2.3. 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** AMQ Broker example program.

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. 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:

```
<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 (2 seconds).

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** AMQ Broker example program .

17.3. IMPLEMENTING HIGH AVAILABILITY

After creating a broker cluster, you can improve its reliability by implementing high availability (HA). With HA, the broker cluster can continue to function even if one or more brokers go offline.

Implementing HA involves several steps:

1. 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 .

2. 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

3. 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 .

17.3.1. 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.

17.3.1.1. 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.

17.3.1.2. 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.

MASTER BROKER   SHARED FILE SYSTEM   SLAVE BROKER

Journal
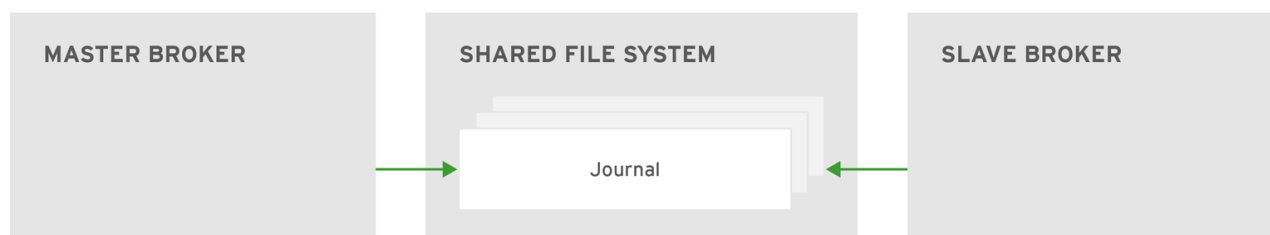
JBOSS_409952_0317

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.

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.
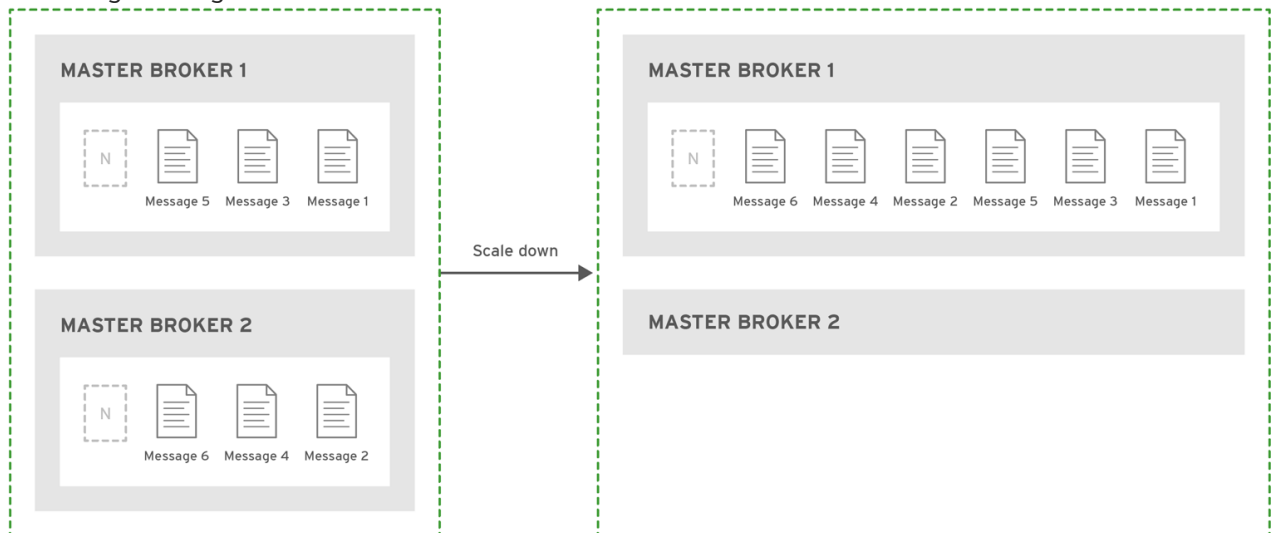
Additional resources

- For more information about the persistent message data that is shared between brokers in a live-backup group, see Section 7.1, "About Journal-based Persistence".

17.3.1.3. Replication policy limitations

Network isolation (sometimes called "split brain") is a limitation of the replication high availability (HA) policy. You should understand how it occurs, and how to avoid it.

Network isolation 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. Specifically, if the backup broker can still connect to more than half of the live brokers in the cluster, it also becomes active. 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 avoid network isolation, consider the following:

- To **eliminate** any possibility of network isolation, use the **shared store** HA policy.

- If you do use the replication HA policy, you can reduce (but not eliminate) the chance of encountering network isolation 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.

- Suppose that both brokers in a live-backup pair were previously shut down, but are now available to be restarted. In this case, to avoid message loss, you need to 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.

17.3.2. 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.

17.3.2.1. 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.
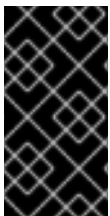
> **IMPORTANT**
>
> It is important to use reasonable values when you configure the **timeo** and **retrans** options. A default **timeo** wait time of 600 deciseconds (60 seconds) combined with a **retrans** value of 5 retries can result in a five-minute wait for AMQ Broker to detect an NFS disconnection.

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.

17.3.2.2. 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 Configuring JDBC Persistence.

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>BINDINGS_TABLE</bindings-table-name>
            <message-table-name>MESSAGE_TABLE</message-table-name>
            <large-message-table-name>LARGE_MESSAGES_TABLE</large-message-
      table-name>
            <page-store-table-name>PAGE_STORE_TABLE</page-store-table-name>
            <node-manager-store-table-name>NODE_MANAGER_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.

17.3.3. 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 network isolation in which a live broker and its backup become live at the same time.

Replication requires **at least three live-backup pairs** to lessen (but not eliminate) the risk of network isolation. Using at least three live-backup broker pairs enables your cluster to use *quorum voting* to avoid having two live brokers.

The sections that follow explain how quorum voting works and how to configure replication HA for a broker cluster with at least three live-backup pairs.

> **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.

17.3.3.1. About quorum voting

In the event that a live broker and its backup experience an interrupted replication connection, you can configure a process called *quorum voting* to mitigate against network isolation (or "split brain") issues. During network isolation, a live broker and its backup can become active at the same time.

The following table describes the two types of quorum voting that AMQ Broker uses.

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, when you use the replication HA policy, 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 network isolation. 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.

17.3.3.2. Configuring a broker cluster for replication high availability

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.

Replication requires at least three live-backup pairs to lessen (but not eliminate) the risk of network isolation.

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 17, *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. To form a live–backup group, the live and backup brokers must be configured with the same group name.

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, *Replication High Availability Configuration Elements*.

d. Open the backup broker's **<broker-instance-dir>/etc/broker.xml** configuration file.

e. Configure the backup (that is, slave) broker to use replication for its HA policy.

```xml
<configuration>
  <core>
    ...
    <ha-policy>
      <replication>
        <slave>
          <allow-failback>true</allow-failback>
          <restart-backup>true</restart-backup>
          <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.

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**.

group-name

The group name of the live broker to which this backup should connect. A backup broker connects only to a live broker that shares the same group name.

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, *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 example programs.

- For more information about node IDs, see Understanding node IDs .

17.3.4. 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.

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>\n <scale-down>\n <connectors>\n <connector-ref>broker1-connector</connector-ref>\n </connectors>\n </scale-down>\n</live-only>```
Any broker in the cluster	Specify the broker cluster's discovery group. ```<live-only>\n <scale-down>\n <discovery-group-ref discovery-group-name="my-discovery-group"/>\n </scale-down>\n</live-only>```
A broker in a particular broker group	Specify a broker group. ```<live-only>\n <scale-down>\n <group-name>my-group-name</group-name>\n </scale-down>\n</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 programs.

17.3.5. 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 17.4. Colocated live and backup brokers

JBOSS_ 409952_0317
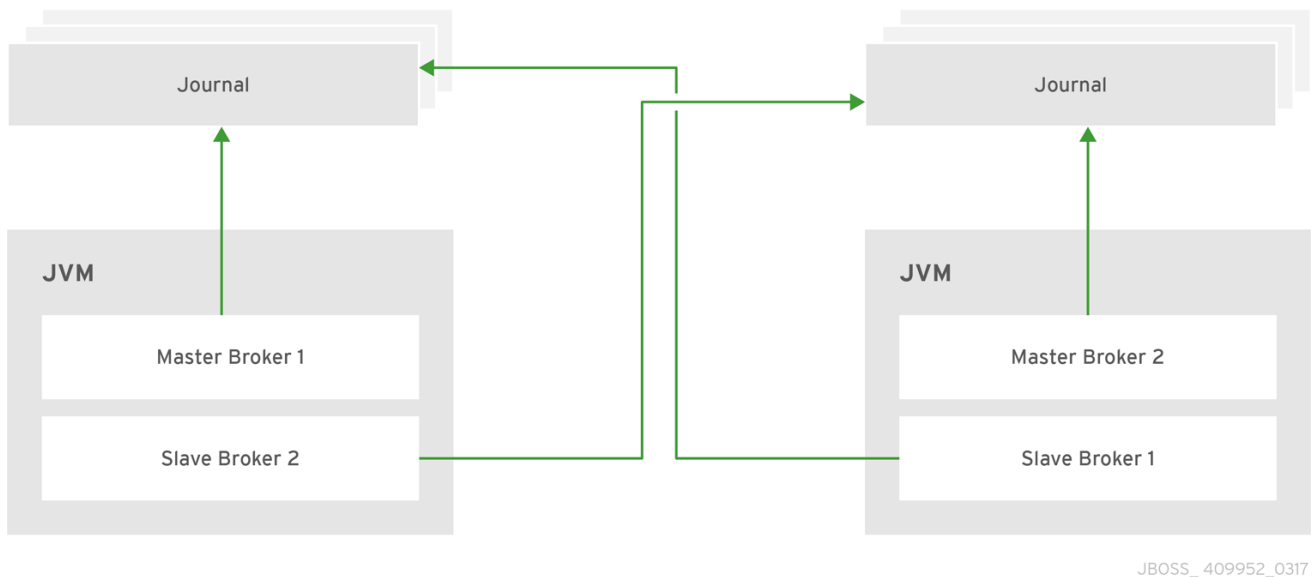
You can use colocation with either shared store or replication as the high availability (HA) policy. The new backup broker inherits its configuration from the live broker that creates it. The name of the backup is set to **colocated_backup_n** where **n** is the number of backups the live broker has 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 example programs.

17.3.6. 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.
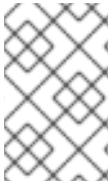
> **NOTE**
>
> In the event of transient network problems, AMQ Broker automatically reattaches connections to the same broker. This is similar to failover, except that the client reconnects to the same broker.

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 .

17.4. ENABLING MESSAGE REDISTRIBUTION

If your broker cluster uses on-demand message load balancing, 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

17.4.1. Understanding message redistribution

Broker clusters use load balancing to distribute the message load across the cluster. When configuring load balancing in the cluster connection, if you set **message-load-balancing** to **ON_DEMAND**, 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.

17.4.1.1. Limitations of message redistribution with message filters

Message redistribution *does not* support the use of filters (also know as *selectors*) by consumers. A common use case for consumers with filters is a request–reply pattern using a correlation ID. For example, consider the following scenario:

1. You have a cluster of two brokers, **brokerA** and **brokerB**. Each broker is configured with **redistribution-delay** set to **0** and **message-load-balancing** set to **ON_DEMAND**.

2. **brokerA** and **brokerB** each has a queue named **myQueue**.

3. Based on a request, a producer sends a message that is routed to queue **myQueue** on **brokerA**. The message has a correlation ID property named **myCorrelID**, with a value of **10**.

4. A consumer connects to queue **myQueue** on **brokerA** with a filter of **myCorrelID=5**. This filter does not match the correlation ID value of the message.

5. Another consumer connects to queue **myQueue** on **brokerB** with a filter of **myCorrelID=10**. This filter matches the correlation ID value of the message.
   In this case, although the filter of the consumer on **brokerB** matches the message, the message is *not* redistributed from **brokerA** to **brokerB** because a consumer for the queue **myQueue** exists on **brokerA**.

In the preceding scenario, you can ensure that the intended client receives the message by creating the consumers *before* the request is sent to the producer. The message is immediately routed to the consumer with a filter matching the correlation ID of the message. Redistribution is not required.

Additional resources

- For more information about cluster load balancing, see Section 17.1.1, "How broker clusters balance message load".

17.4.2. Configuring message redistribution

This procedure shows how to configure message 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 (5 seconds) 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 the section called "AMQ Broker Wildcard Syntax" .

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** AMQ Broker example program .

17.5. 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.

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 (5 seconds). 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 (5 seconds). 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 **clustered-grouping** AMQ Broker example program .

17.6. CONNECTING CLIENTS TO A BROKER CLUSTER

You can use the AMQ 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 .

CHAPTER 18. CONFIGURING A MULTI-SITE, FAULT-TOLERANT MESSAGING SYSTEM

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 the Red Hat Ceph Storage 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* .

The following sections explain how to protect your messaging system from data center outages. These sections provide information about:

- 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 .

18.1. 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?

18.2. 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
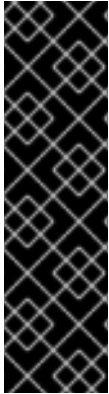
> **IMPORTANT**
>
> You can run the OSD, MON, MGR, and MDS nodes on either the same or separate physical or virtual machines. However, to ensure fault tolerance within your Red Hat Ceph Storage cluster, it is good practice to distribute each of these types of nodes across distinct data centers. In particular, 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

18.3. 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
   ```
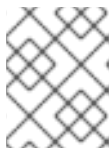
   > **NOTE**
   >
   > In the preceding command, if your storage cluster uses solid–state drives (SSD), specify **ssd** instead of **hdd** (hard disk drives).

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.
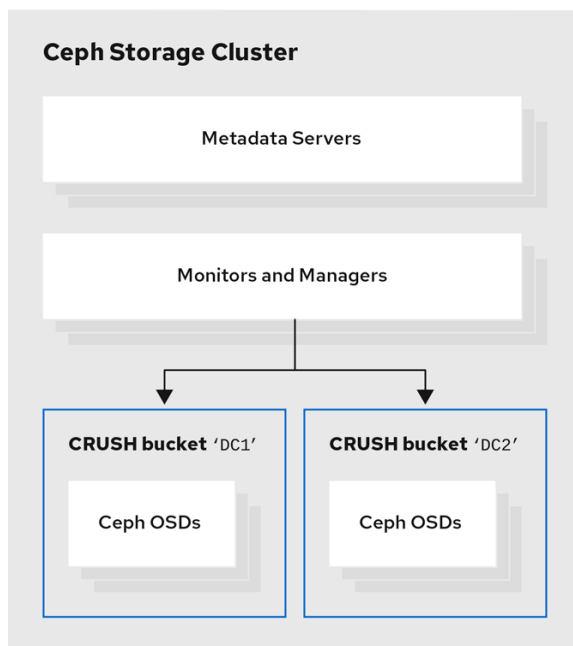
The following figure shows a possible layout of the first data center, including your broker servers. Specifically, the data center hosts:

- 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.
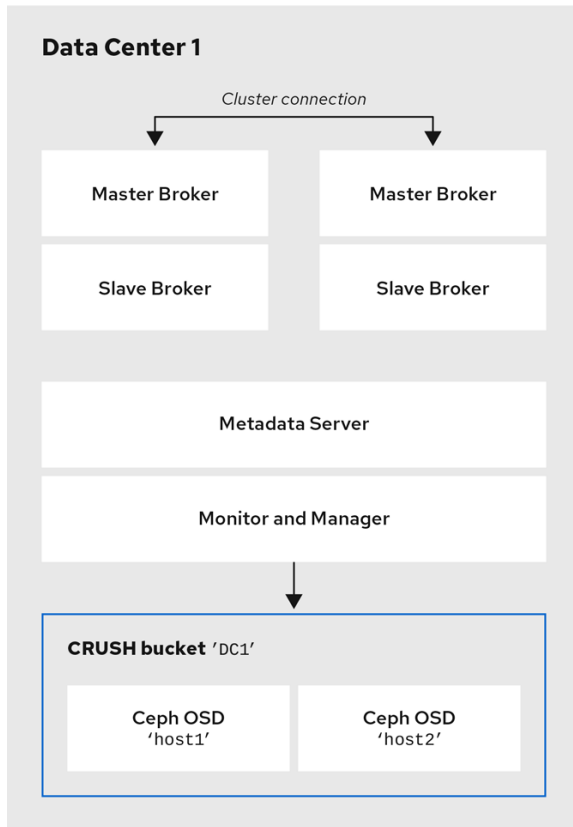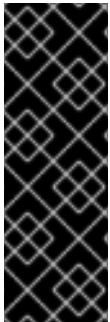
> **IMPORTANT**
>
> You can run the OSD, MON, MGR, and MDS nodes on either the same or separate physical or virtual machines. However, to ensure fault tolerance within your Red Hat Ceph Storage cluster, it is good practice to distribute each of these types of nodes across distinct data centers. In particular, 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.
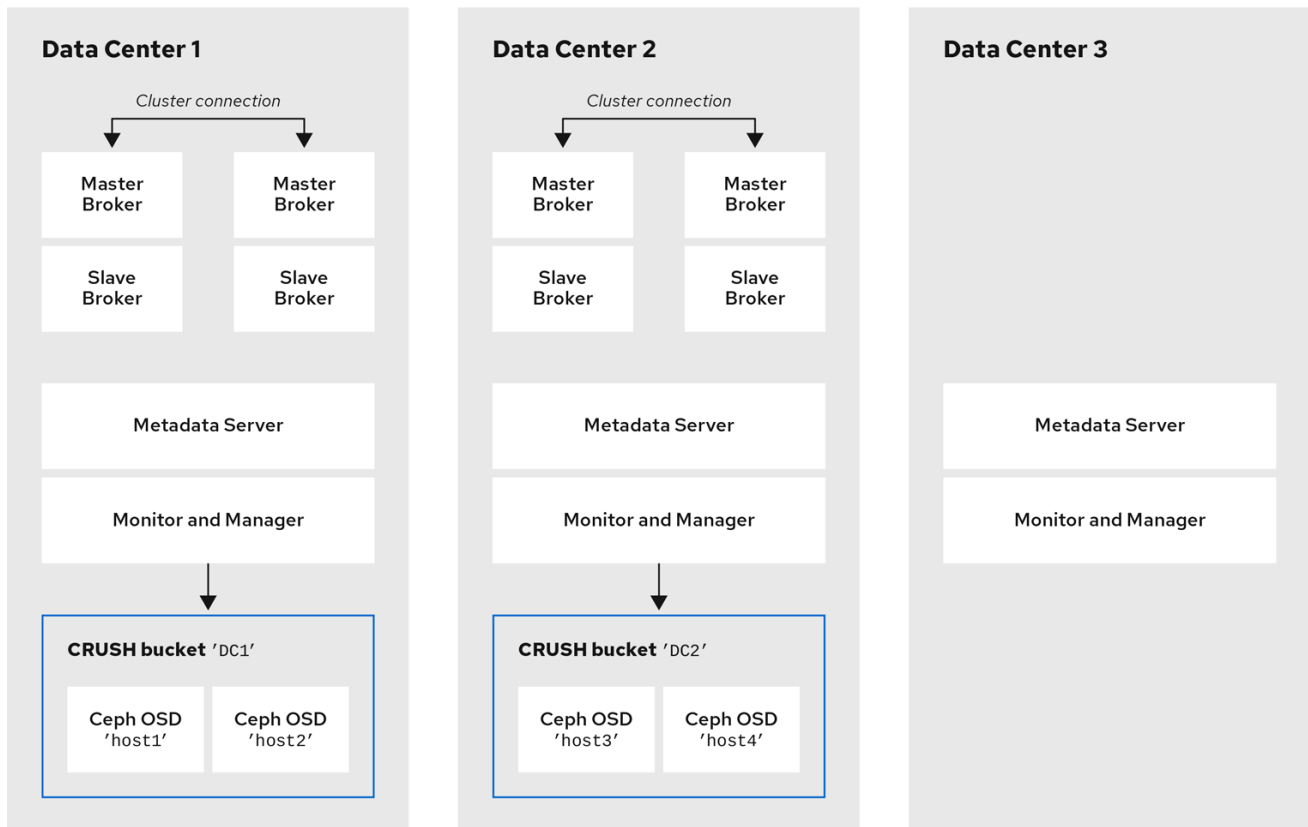
NOTE

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.
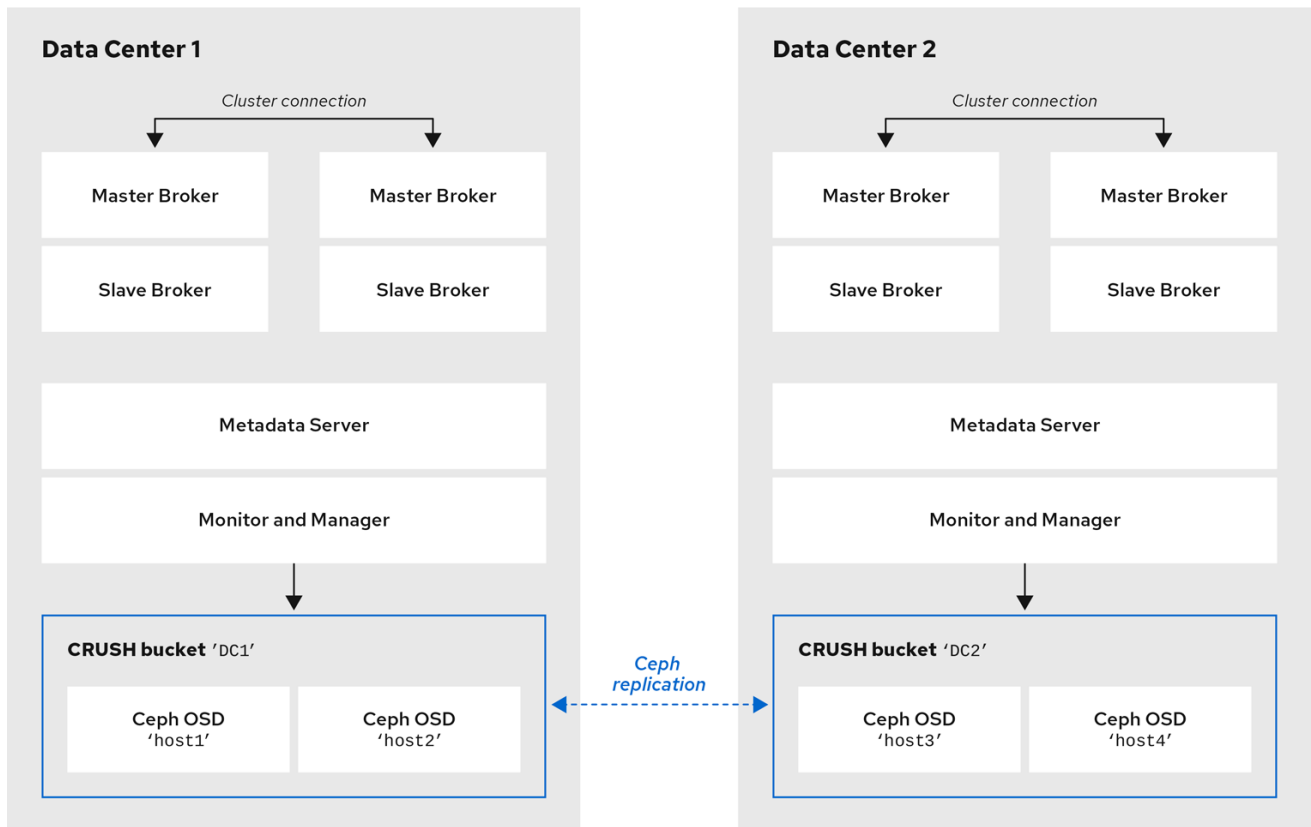
Additional resources

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.

18.4. 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.

18.5. 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

18.5.1. 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.
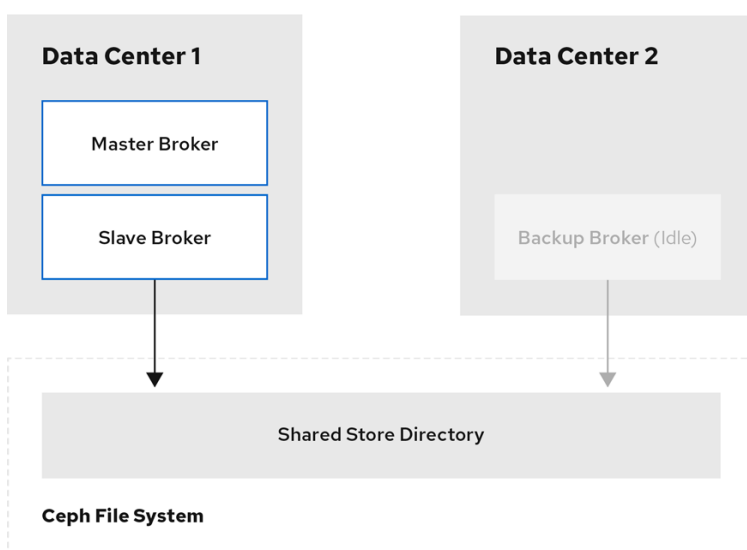
Additional resources

- To learn how to create additional broker instances, see Creating a standalone broker .

- For information about configuring broker network connections, see Network Connections: Acceptors and Connectors.

18.5.2. 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 .

18.5.3. 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 .

18.6. 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.
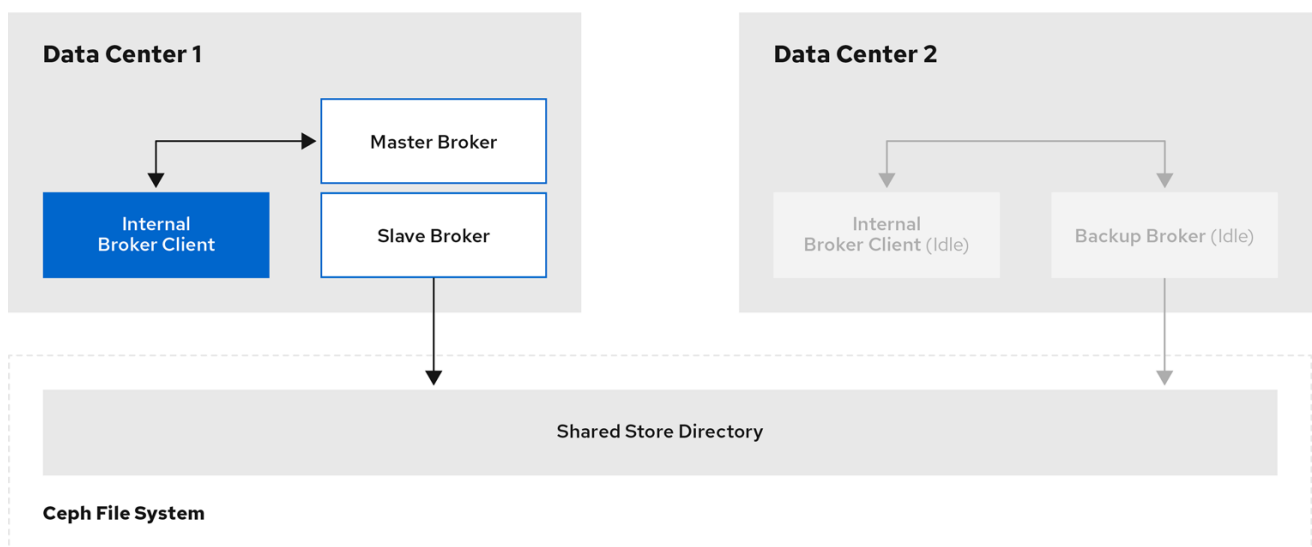
An external client application is one running on a machine located outside the broker data center. The following figure shows this topology.
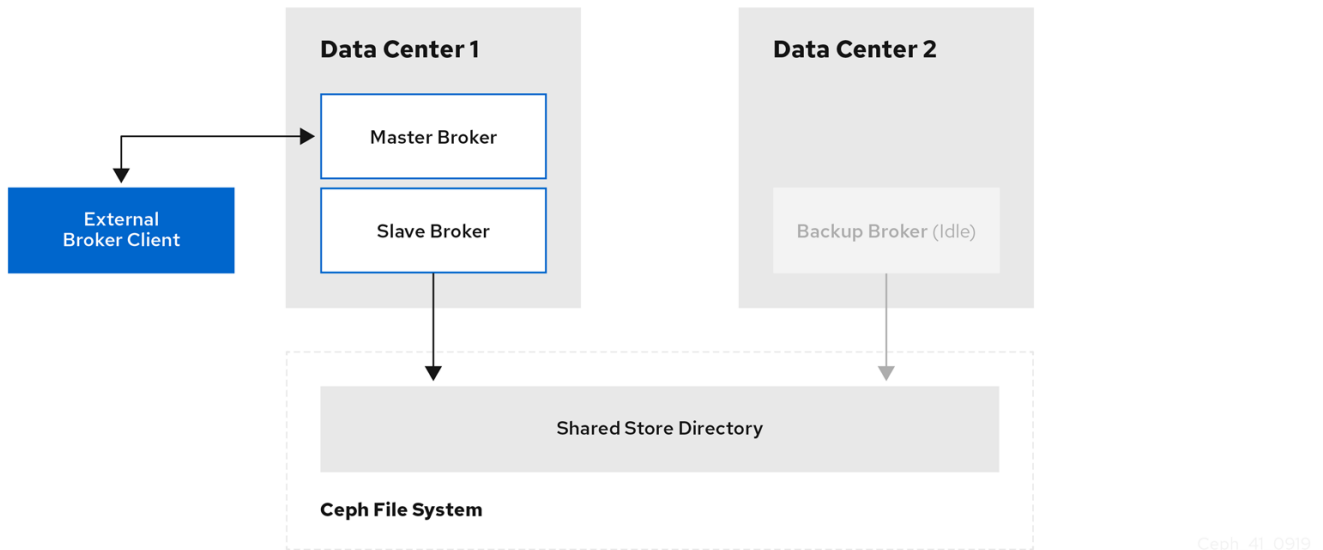
The following sub-sections describe show examples of configuring your internal and external client applications to connect to a backup broker in another data center in the event of a data center outage.

18.6.1. 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 and client network connections, see:

- Network Connections: Acceptors and Connectors .

- Configuring a Connection from the Client Side .

18.6.2. 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 AMQP 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 AMQP 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=foo&failover.maxReconnectAttempts=20
  ```

Additional resources

For more information about configuring failover on:

- The AMQ Core Protocol JMS client, see Reconnect and failover.

- The AMQP JMS client, see Failover options.

- Other supported clients, consult the client-specific documentation in the AMQ Clients section of Product Documentation for Red Hat AMQ 7.7 .

18.7. 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](#).

18.8. 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.
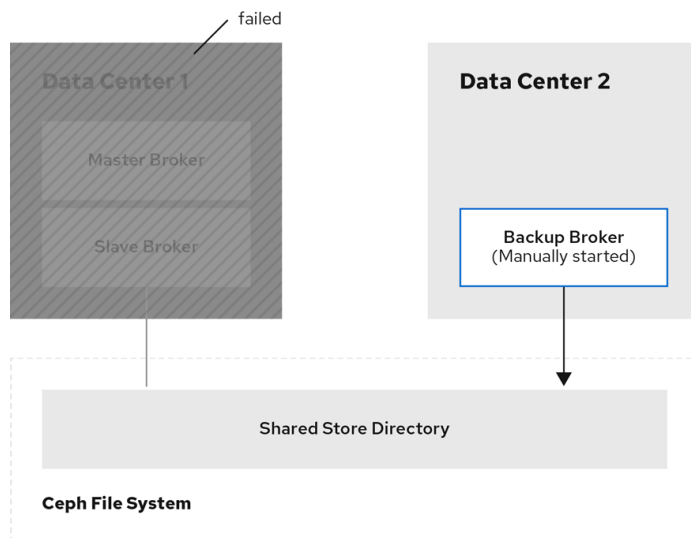
2. Reestablish client connections.

   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.
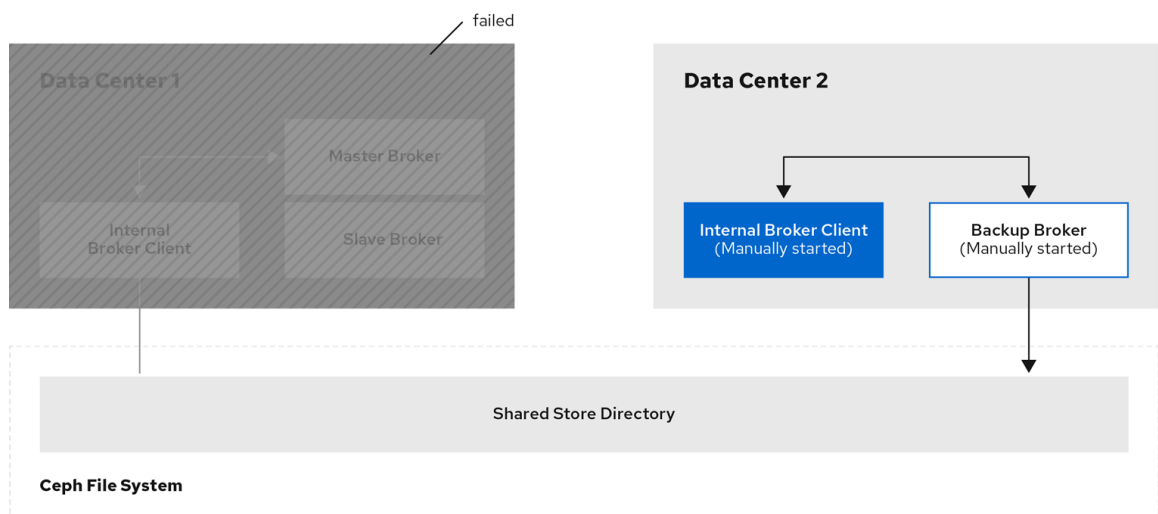
   b. If you have an external client, manually connect the external client to the new active broker or observe that the clients automatically fails over to the new active broker, based on its configuration. For more information, see Configuring external clients.
      The following figure shows the new topology.
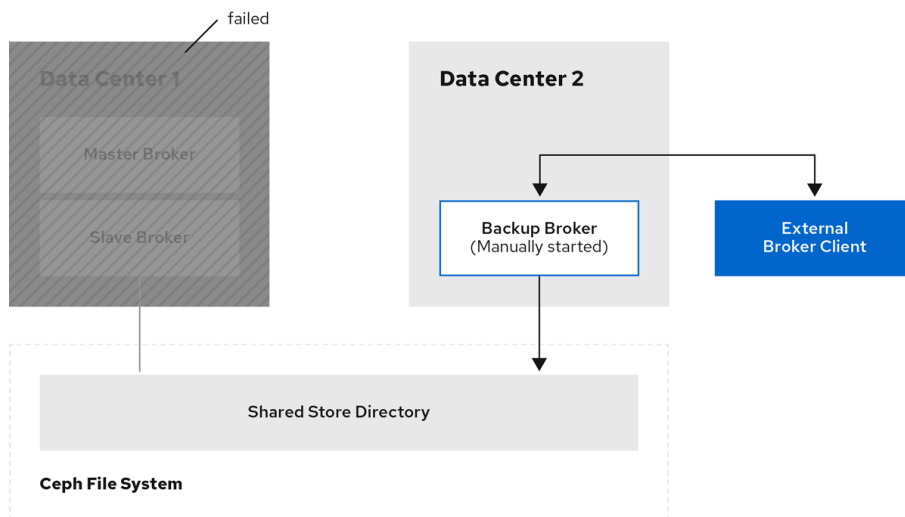
18.9. RESTARTING A PREVIOUSLY FAILED DATA CENTER

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.

18.9.1. 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%.

18.9.2. 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.

18.9.3. 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.

18.9.3.1. 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 and client network connections, see:

- Network Connections: Acceptors and Connectors

- Configuring a Connection from the Client Side

18.9.3.2. 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 19. LOGGING

AMQ Broker uses the JBoss Logging framework to do its logging and is configurable via the **BROKER_INSTANCE_DIR/etc/logging.properties** configuration file. This configuration file is a list of key-value pairs.

You specify loggers in your broker configuration by including them in the **loggers** key of the **logging.properties** configuration file, as shown below.

```
loggers=org.eclipse.jetty,org.jboss.logging,org.apache.activemq.artemis.core.server,org.apache.activem
q.artemis.utils,org.apache.activemq.artemis.journal,org.apache.activemq.artemis.jms.server,org.apache
activemq.artemis.integration.bootstrap,org.apache.activemq.audit.base,org.apache.activemq.audit.mess
age,org.apache.activemq.audit.resource
```

The loggers available in AMQ Broker are shown in the following table.

Logger	Description
org.jboss.logging	The root logger. Logs any calls not handled by the other broker loggers.
org.apache.activemq.artemis.core.server	Logs the broker core
org.apache.activemq.artemis.utils	Logs utility calls
org.apache.activemq.artemis.journal	Logs Journal 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

There are also two default logging handlers configured in the **logger.handlers** key, as shown below.

```
logger.handlers=FILE,CONSOLE
```

logger.handlers=FILE

The logger outputs log entries to a file.

logger.handlers=CONSOLE

The logger outputs log entries to the AMQ Broker management console.

19.1. CHANGING THE LOGGING LEVEL

The default logging level for all loggers is **INFO** and is configured on the root logger, as shown below.

```
logger.level=INFO
```

You can configure the logging level for all other loggers individually, as shown below.

```
logger.org.apache.activemq.artemis.core.server.level=INFO
logger.org.apache.activemq.artemis.journal.level=INFO
logger.org.apache.activemq.artemis.utils.level=INFO
logger.org.apache.activemq.artemis.jms.level=INFO
logger.org.apache.activemq.artemis.integration.bootstrap.level=INFO
logger.org.apache.activemq.audit.base.level=INFO
logger.org.apache.activemq.audit.message.level=INFO
logger.org.apache.activemq.audit.resource.level=INFO
```

The available logging levels are described in the following table. The logging levels are listed in ascending order, from the least detailed to the most.

Level	Description
FATAL	Use the FATAL logging level for events that indicate a critical service failure. If a service issues a FATAL error, it is completely unable to execute requests of any kind.
ERROR	Use the ERROR logging level for events that indicate a disruption in a request or the ability to service a request. A service should have *some* capacity to continue to service requests in the presence of this level of error.
WARN	Use the WARN logging level for events that might indicate a non-critical service error. Resumable errors, or minor breaches in request expectations meet this description. The distinction between WARN and ERROR is one for an application developer to make. A simple criterion for making this distinction is whether the error would require a user to seek technical support. If an error would require technical support, set the logging level to ERROR. Otherwise, set the level to WARN.
INFO	Use the INFO logging level for service lifecycle events and other crucial related information. INFO-level messages for a given service category should clearly indicate what state the service is in.

Level	Description
DEBUG	Use the DEBUG logging level for log messages that convey extra information for lifecycle events. Use this logging level for developer-oriented information or in-depth information required for technical support. When the DEBUG logging level is enabled, the JBoss server log **should not** grow proportionally with the number of server requests. DEBUG- and INFO-level messages for a given service category should clearly indicate what state the service is in, as well as what broker resources it is using; ports, interfaces, log files, and so on.
TRACE	Use the TRACE logging level for log messages that are directly associated with request activity. Such messages should not be submitted to a logger unless the logger category priority threshold indicates that the message will be rendered. Use the **Logger.isTraceEnabled()** method to determine whether the category priority threshold is enabled. TRACE-level logging enables deep probing of the broker behavior when necessary. When TRACE logging level is enabled, the number of messages in the JBoss sever log grows to at least $a * N$, where N is the number of requests received by the broker, and a is some constant. The server log might grow to some power of N, depending on the request-handling layer being traced.

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.

- The logging level specified for the root logger determines the most detailed logging level for **all** loggers, even if other loggers have more detailed logging levels specified in their configurations. For example, suppose **org.apache.activemq.artemis.utils** has a specified logging of **DEBUG**, while the root logger, **org.jboss.logging**, has a specified logging level of **WARN**. In this case, both loggers use a logging level of **WARN**.

19.2. 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, each audit logger is disabled (that is, the logging level is set to **ERROR**, which is a not a valid logging level for the audit loggers). To enable one of the audit loggers, set the logging level to **INFO**. For example:

```
logger.org.apache.activemq.audit.base.level=INFO
```
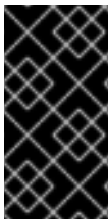
> **IMPORTANT**
>
> The message audit logger runs on a performance-intensive path on the broker. Enabling the logger might negatively affect the performance of the broker, particularly if the broker is running under a high messaging load. Use of the message audit logger is not recommended on messaging systems where high throughput is required.

19.3. CONFIGURING CONSOLE LOGGING

You can configure console logging using the following keys.

```
handler.CONSOLE=org.jboss.logmanager.handlers.ConsoleHandler
handler.CONSOLE.properties=autoFlush
handler.CONSOLE.level=DEBUG
handler.CONSOLE.autoFlush=true
handler.CONSOLE.formatter=PATTERN
```
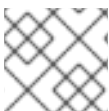
> **NOTE**
>
> **handler.CONSOLE** refers to the name given in the **logger.handlers** key.

The console logging configuration options are described in the following table.

Property	Description
name	Handler name
encoding	Character encoding used by the handler
level	Logging level, specifying the message levels logged. Message levels lower than this value are discarded.
formatter	Defines a formatter. See Section 19.5, "Configuring the logging format".

Property	Description
autoflush	Species whether to automatically flush the log after each write
target	Target of the console handler. The value can either be SYSTEM_OUT or SYSTEM_ERR.

19.4. CONFIGURING FILE LOGGING

You can configure file logging using the following keys.

```
handler.FILE=org.jboss.logmanager.handlers.PeriodicRotatingFileHandler
handler.FILE.level=DEBUG
handler.FILE.properties=suffix,append,autoFlush,fileName
handler.FILE.suffix=.yyyy-MM-dd
handler.FILE.append=true
handler.FILE.autoFlush=true
handler.FILE.fileName=${artemis.instance}/log/artemis.log
handler.FILE.formatter=PATTERN
```

NOTE

handler.FILE refers to the name given in the **logger.handlers** key.

The file logging configuration options are described in the following table.

Property	Description
name	Handler name
encoding	Character encoding used by the handler
level	Logging level, specifying the message levels logged. Message levels lower than this value are discarded.
formatter	Defines a formatter. See Section 19.5, "Configuring the logging format".
autoflush	Species whether to automatically flush the log after each write
append	Specifies whether to append to the target file
file	File description, consisting of the path and optional relative to path.

19.5. CONFIGURING THE LOGGING FORMAT

The formatter describes how log messages should be displayed. The following is the default configuration.

```
formatter.PATTERN=org.jboss.logmanager.formatters.PatternFormatter
formatter.PATTERN.properties=pattern
formatter.PATTERN.pattern=%d{HH:mm:ss,SSS} %-5p [%c] %s%E%n
```

In the preceding configuration, **%s** is the message and **%E** is the exception, if one exists.

The format is the same as the Log4J format. A full description can be found [here](#).

19.6. CLIENT OR EMBEDDED SERVER LOGGING

If you want to enable logging on a client, you need to include the JBoss logging JARs in your client's class path. If you are using Maven, add the following dependencies:

```xml
<dependency>
  <groupId>org.jboss.logmanager</groupId>
  <artifactId>jboss-logmanager</artifactId>
  <version>1.5.3.Final</version>
</dependency>
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>artemis-core-client</artifactId>
  <version>1.0.0.Final</version>
</dependency>
```

There are two properties that you need to set when starting your Java program. The first is to set the Log Manager to use the JBoss Log Manager. This is done by setting the `-Djava.util.logging.manager` property. For example:

```
-Djava.util.logging.manager=org.jboss.logmanager.LogManager
```

The second is to set the location of the **logging.properties** file to use. This is done by setting the **-Dlogging.configuration** property with a valid URL. For example:

```
-Dlogging.configuration=file:///home/user/projects/myProject/logging.properties
```

The following is a typical **logging.properties** file for a client:

```
# Root logger option
loggers=org.jboss.logging,org.apache.activemq.artemis.core.server,org.apache.activemq.artemis.utils,or
g.apache.activemq.artemis.journal,org.apache.activemq.artemis.jms,org.apache.activemq.artemis.ra

# Root logger level
logger.level=INFO
# ActiveMQ Artemis logger levels
logger.org.apache.activemq.artemis.core.server.level=INFO
logger.org.apache.activemq.artemis.utils.level=INFO
logger.org.apache.activemq.artemis.jms.level=DEBUG
```

```
# Root logger handlers
logger.handlers=FILE,CONSOLE

# Console handler configuration
handler.CONSOLE=org.jboss.logmanager.handlers.ConsoleHandler
handler.CONSOLE.properties=autoFlush
handler.CONSOLE.level=FINE
handler.CONSOLE.autoFlush=true
handler.CONSOLE.formatter=PATTERN

# File handler configuration
handler.FILE=org.jboss.logmanager.handlers.FileHandler
handler.FILE.level=FINE
handler.FILE.properties=autoFlush,fileName
handler.FILE.autoFlush=true
handler.FILE.fileName=activemq.log
handler.FILE.formatter=PATTERN

# Formatter pattern configuration
formatter.PATTERN=org.jboss.logmanager.formatters.PatternFormatter
formatter.PATTERN.properties=pattern
formatter.PATTERN.pattern=%d{HH:mm:ss,SSS} %-5p [%c] %s%E%n
```

19.7. 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 will be 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.

19.7.1. 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 then place the **.jar** file under the regular class path of your embedded application.

19.7.2. 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 will be 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>
```

19.7.3. 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());
```

19.7.4. 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.

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 have changed 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.

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 Network Connections: Acceptors and Connectors 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 will allow. 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.
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
enabledCipherSuites	Both	Comma-separated list of cipher suites used for SSL communication. The default value is empty which means the JVM's default will be used.

Parameter	Use with...	Description
enabledProtocols	Both	Comma-separated list of protocols used for SSL communication. The default value is empty which means the JVM's default will be used.
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**.
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.
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
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.
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.

Parameter	Use with...	Description
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**.
verifyHost	Both	When used on an acceptor, the CN of the connecting client's SSL certificate is compared to its hostname to verify that they match. This is useful only for two-way SSL. When used on a connector, the CN of the server's SSL certificate is compared to its hostname to verify that they match. This is useful for both one-way and two-way SSL. Valid values are **true** or **false**. The default value is **false**.

Parameter	Use with...	Description
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 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-max-cache-size	The number of page files to keep in memory to optimize I/O during paging navigation. The default value is **5**.
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
	    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
	    Ensures that messages are forwarded only to brokers that have active consumers or a matching selector.

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**.

Name	Description
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.

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. Address Setting 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. REPLICATION HIGH AVAILABILITY CONFIGURATION ELEMENTS

The following tables list the valid **ha-policy** configuration elements when using a replication HA policy.

Table F.1. Configuration Elements Available when Using Replication High Availability

Name	Description
check-for-live-server	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	Name of the cluster configuration to use for replication. This setting is only necessary if you configure multiple cluster connections. If configured, the 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.
group-name	If set, backup brokers will only pair with live brokers that have a matching value for **group-name**.
initial-replication-sync-timeout	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, or 30 seconds. **NOTE** During this interval, any other journal-related operations are blocked.
max-saved-replicated-journals-size	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**.
allow-failback	Applies to backup brokers only. Determines whether the backup broker resumes its original role when another broker such as the live broker makes a request to take its place. The default value of this property is **true**.
restart-backup	Applies to backup brokers only. Determines whether the backup broker automatically restarts after it fails back to another broker. The default value of this property is **true**.

Revised on 2020-09-01 15:25:47 UTC