Red Hat AMQ 7.2

Using AMQ Broker

For Use with AMQ Broker 7.2

Red Hat AMQ 7.2 Using AMQ Broker

For Use with AMQ Broker 7.2

Legal Notice

Abstract

This guide describes how to install, configure, monitor, and manage the broker.

Table of Contents

CHAPTER 1. OVERVIEW

AMQ Broker is a full-featured, message-oriented middleware broker. It offers specialized queueing behaviors, message persistence, and manageability. Core messaging is provided by Apache ActiveMQ Artemis with support for different messaging styles such as publish-subscribe, point-to-point, and store-and-forward. It supports multiple protocols and client languages, freeing you to use many if not all of your application assets. Lastly, AMQ Broker is supported to work with Red Hat JBoss Enterprise Application Platform.

AMQ Broker is based on Apache ActiveMQ Artemis.

1.1. KEY FEATURES

- Clustering and high availability options

- Fast, native-IO persistence

- Supports local transactions

- Supports XA transactions when using AMQ Core Protocol JMS and AMQ OpenWire JMS clients

- Written in Java for broad platform support

- Multiple management interfaces: AMQ Console, Management APIs, and JMX

1.2. SUPPORTED STANDARDS AND PROTOCOLS

- Supports multiple wire protocols

  - AMQP 1.0

  - Artemis Core Protocol

  - HornetQ Core Protocol

  - MQTT

  - OpenWire (Used by A-MQ 6 clients)

  - STOMP

- Supports JMS 2.0

> **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.

1.3. SUPPORTED CONFIGURATIONS

Refer to the article "Red Hat AMQ 7 Supported Configurations" on the Red Hat Customer Portal for current information regarding AMQ Broker supported configurations.

1.4. 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\...`).

1.5. KEY CONCEPTS

Messaging brokers allow you to loosely couple heterogeneous systems together, while typically providing reliability, transactions, and many other features.

Unlike systems based on a Remote Procedure Call (RPC) pattern, messaging systems primarily use an asynchronous message-passing pattern with no tight relationship between requests and responses. Most messaging systems also support a request-response mode, but this is not a primary feature of messaging systems.

Designing systems to be asynchronous from end to end allows you to really take advantage of your hardware resources, minimizing the number of threads blocking on IO operations, and to use your network bandwidth to its full capacity. With an RPC approach you have to wait for a response for each request you make so are limited by the network round-trip time, or **latency**, of your network. With an asynchronous system you can pipeline flows of messages in different directions, so you are limited by the network **bandwidth**, not the latency. This typically allows you to create much higher performance applications.

Messaging systems decouple the senders of messages from the consumers of messages. The senders and consumers of messages are completely independent and know nothing of each other. This allows you to create flexible, loosely coupled systems.

CHAPTER 2. INSTALLATION

2.1. PREREQUISITES

AMQ Broker requires the following components.

- JRE 8 (for running AMQ Broker)

- JDK 8 (for running the examples)

- Maven 3.2 (for running the examples)

Note that the broker runtime requires only a JRE. However, running the included examples requires a full JDK as well as Maven.

If you are installing AMQ Broker on a supported version of Red Hat Enterprise Linux, you can use the **yum** command to install any needed pre-requisites. For example, the command below installs OpenJDK 8 and Maven.

```
$ sudo yum install java-1.8.0-openjdk maven
```

You can also download supported versions of a JDK and Maven from their respective websites, OpenJDK and Apache Maven for example. Consult Supported Configurations to ensure you are using a supported version of Java.

2.2. DOWNLOADING AN AMQ BROKER ARCHIVE

A platform-independent, archived distribution of AMQ Broker is available for download from the Red Hat Customer Portal. See Using Your Subscription for more information on how to access the customer portal using your Red Hat subscription. You can download a copy of the distribution by following the steps below.

Procedure

1. Open a browser and log in to the Red Hat Customer Portal at https://access.redhat.com/downloads/. The **Product Downloads** page is displayed.

2. Find the category titled **JBoss Integration and Automation** containing a link to **Red Hat AMQ Broker**.

3. Click the hyperlink **Red Hat AMQ Broker**. The **Software Downloads** page is displayed.

4. Select the desired AMQ Broker version from the **Version** drop-down menu.

5. On the **Releases** tab, click the **Download** link for the specific AMQ Broker file you want to download.

2.3. INSTALLING ON LINUX

Prerequisites

You must first download the installation archive from the Red Hat Customer Portal.

Procedure

1. Create a new user named **amq-broker** and provide it a password.

   ```
   sudo useradd amq-broker
   sudo passwd amq-broker
   ```

2. Create the directory **/opt/redhat/amq-broker** and make the new **amq-broker** user and group the owners of it.

   ```
   sudo mkdir /opt/redhat
   sudo mkdir /opt/redhat/amq-broker
   sudo chown -R amq-broker:amq-broker /opt/redhat/amq-broker
   ```

3. Change the owner of the archive to the new user.

   ```
   sudo chown amq-broker:amq-broker amq-broker-7.x.x-bin.zip
   ```

4. Move the installation archive to the directory you just created.

   ```
   sudo mv amq-broker-7.x.x-bin.zip /opt/redhat/amq-broker
   ```

5. As the new user **amq-broker**, extract the contents with a single **unzip** command.

   ```
   su - amq-broker
   cd /opt/redhat/amq-broker
   unzip amq-broker-7.x.x-bin.zip
   ```

A directory named something similar to **AMQ7-7.2-redhat-1** will be created. In the documentation this location is referred to as *INSTALL_DIR*.

2.4. INSTALLING ON WINDOWS

Prerequisites

You must first download the zip version of the archive from the Red Hat Customer Portal.

Procedure

1. Use Windows Explorer to create the directory folder **\redhat\amq-broker** on the desired letter drive, **C:** for example.

2. Use Windows Explorer to move the installation archive to the new directory and extract it.

3. Extract the file contents into a directory by right-clicking on the zip file and choosing **Extract All**.

A directory named something similar to **AMQ7-x.x-redhat-1** will be created. In the documentation this location is referred to as *INSTALL_DIR*.

2.5. ARCHIVE CONTENTS

The directory created by extracting the archive is the top-level directory for the AMQ Broker installation. This directory is referred to as *INSTALL_DIR* and includes a number of important directories noted in the table below.

Table 2.1. Contents of *INSTALL_DIR*

If you want to find…	Look here…
API documentation	*INSTALL_DIR*/web/api
Binaries and scripts needed to run AMQ Broker	*INSTALL_DIR*/bin
Configuration files	*INSTALL_DIR*/etc
JMS and Java EE examples	*INSTALL_DIR*/examples
Jars and libraries needed to run AMQ Broker	*INSTALL_DIR*/lib
XML Schemas used to validate AMQ Broker configuration files	*INSTALL_DIR*/schema
Web context loaded when AMQ Broker runs.	*INSTALL_DIR*/web

CHAPTER 3. UPGRADING YOUR BROKER

3.1. ABOUT UPGRADES

Red Hat releases new versions of AMQ Broker to the Customer Portal. Update your brokers to the newest version to ensure that you have the latest enhancements and fixes. In general, Red Hat releases a new version of AMQ Broker in one of three ways:

Major Release

A major upgrade or migration is required when an application is transitioned from one major release to the next, for example, from AMQ Broker 6 to AMQ Broker 7. This type of upgrade is not addressed in this guide. For instructions on how to upgrade from previous releases of AMQ Broker, see Migrating to Red Hat AMQ 7.

Minor Release

AMQ Broker periodically provides minor releases, which are updates that include new features, as well as bug and security fixes. If you plan to upgrade from one AMQ Broker minor release to another, for example, from AMQ Broker 7.0 to AMQ Broker 7.1, code changes should not be required for applications that do not use private, unsupported, or tech preview components.

Micro Release

AMQ Broker also periodically provides micro releases that contain minor enhancements and fixes. Micro releases increment the minor release version by the last digit, for example from 7.0.1 to 7.0.2. A micro release should not require code changes, however, some releases may require configuration changes.

3.2. UPGRADING A BROKER INSTANCE FROM 7.0.X TO 7.0.Y

The procedure for upgrading AMQ Broker from one version of 7.0 to another is similar to the one for installation: you download an archive from the Customer Portal and then extract it. The following subsections describe how to upgrade a 7.0.x broker for different operating systems.

- Upgrading from 7.0.x to 7.0.y on Linux

- Upgrading from 7.0.x to 7.0.y on Windows

3.2.1. Upgrading from 7.0.x to 7.0.y on Linux

The name of the archive that you download could differ from what is used in the following examples.

Prerequisites

- Before upgrading AMQ Broker, review the release notes for the target release.
  The release notes describe important enhancements, known issues, and changes to behavior in the target release.

  For more information, see the AMQ Broker 7.0 Release Notes.

Procedure

1. Download the desired archive from the Red Hat Customer Portal by following the instructions provided in Downloading an AMQ Broker Archive.

2. Change the owner of the archive to the same user that owns the AMQ Broker installation to be upgraded.

```
sudo chown amq-broker:amq-broker jboss-amq-7.x.x.redhat-1.zip
```

3. Move the archive to the directory created during the original installation of AMQ Broker. In the following example, the directory **/opt/redhat** is used.

```
sudo mv jboss-amq-7.x.x.redhat-1.zip /opt/redhat
```

4. As the directory owner, extract the contents of the compressed archive. The archive is kept in a compressed format. In the following example, the user **amq-broker** extracts the archive by using the unzip command.

```
su - amq-broker
cd /opt/redhat
unzip jboss-amq-7.x.x.redhat-1.zip
```

5. Stop the broker if it is running.

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

6. Back up the instance directory of the broker by copying it to the home directory of the current user.

```
cp -r BROKER_INSTANCE_DIR ~/
```

7. (Optional) Note the current version of the broker. After the broker stops, a line similar to the one below is displayed at the end of its log file, which can be found at **BROKER_INSTANCE_DIR/log/artemis.log**.

```
INFO  [org.apache.activemq.artemis.core.server] AMQ221002: Apache
ActiveMQ Artemis Message Broker version 2.0.0.amq-700005-redhat-1
[4782d50d-47a2-11e7-a160-9801a793ea45] stopped, uptime 28 minutes
```

8. Edit the **BROKER_INSTANCE_DIR/etc/artemis.profile** configuration file to set the **ARTEMIS_HOME** property to the new directory created when the archive was extracted.

```
ARTEMIS_HOME='/opt/redhat/jboss-amq-7.x.x-redhat-1'
```

9. Restart the broker by entering the following command:

```
BROKER_INSTANCE_DIR/bin/artemis run
```

10. (Optional) Confirm that the broker is running and that the version has changed. After starting the broker, open the log file **BROKER_INSTANCE_DIR/log/artemis.log** and find two lines similar to the ones below. Note the new version number that appears in the log after the broker is live.

```
INFO  [org.apache.activemq.artemis.core.server] AMQ221007: Server is
now live
...
```

```
INFO  [org.apache.activemq.artemis.core.server] AMQ221001: Apache
ActiveMQ Artemis Message Broker version 2.1.0.amq-700005-redhat-1
[0.0.0.0, nodeID=4782d50d-47a2-11e7-a160-9801a793ea45]
```

3.2.2. Upgrading from 7.0.x to 7.0.y on Windows

Prerequisites

- Before upgrading AMQ Broker, review the release notes for the target release.
  The release notes describe important enhancements, known issues, and changes to behavior in the target release.

  For more information, see the AMQ Broker 7.0 Release Notes.

Procedure

1. Download the desired archive from the Red Hat Customer Portal by following the instructions provided in Downloading an AMQ Broker Archive.

2. Use a file manager to move the archive to the folder you created during the last installation of AMQ Broker.

3. Extract the file contents into the directory by right-clicking on the zip file and choosing **Extract All**.

4. Stop the broker if it is running by entering the following command.

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

5. Back up the broker by using a file manager.

   a. Right click on the *BROKER_INSTANCE_DIR* folder and select **Copy**.

   b. Right click in the same window and select **Paste**.

6. (Optional) Note the current version of the broker. After the broker stops, a line similar to the one below is displayed at the end of its log file, which can be found at **BROKER_INSTANCE_DIR\log\artemis.log**.

   ```
   INFO  [org.apache.activemq.artemis.core.server] AMQ221002: Apache
   ActiveMQ Artemis Message Broker version 2.0.0.amq-700005-redhat-1
   [4782d50d-47a2-11e7-a160-9801a793ea45] stopped, uptime 28 minutes
   ```

7. Edit the **BROKER_INSTANCE_DIR\etc\artemis.profile** configuration file to set the **ARTEMIS_HOME** property to the new directory created when the archive was extracted.

   ```
   ARTEMIS_HOME=NEW_INSTALL_DIR
   ```

8. Restart the broker entering the following command:

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

9. (Optional) Confirm that the broker is running and that the version has changed. After starting the broker, open the log file **BROKER_INSTANCE_DIR\log\artemis.log** and find two lines

similar to the ones below. Note the new version number that appears in the log after the broker is live.

```
INFO  [org.apache.activemq.artemis.core.server] AMQ221007: Server is
now live
...
INFO  [org.apache.activemq.artemis.core.server] AMQ221001: Apache
ActiveMQ Artemis Message Broker version 2.1.0.amq-700005-redhat-1
[0.0.0.0, nodeID=4782d50d-47a2-11e7-a160-9801a793ea45]
```

3.3. UPGRADING A BROKER INSTANCE FROM 7.0.X TO 7.1.0

AMQ Broker 7.1.0 includes configuration files and settings that were not included with previous versions. Upgrading a broker instance from 7.0.x to 7.1.0 requires adding these new files and settings to your existing 7.0.x broker instances. The following subsections describe how to upgrade a 7.0.x broker instance to 7.1.0 for different operating systems.

> **IMPORTANT**
>
> Starting with AMQ Broker 7.1.0, you can access the AMQ 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 Console and AMQ Broker Connections.

- Upgrading from 7.0.x to 7.1.0 on Linux

- Upgrading from 7.0.x to 7.1.0 on Windows

3.3.1. Upgrading from 7.0.x to 7.1.0 on Linux

Before you can upgrade a 7.0.x broker, you need to install Red Hat AMQ Broker 7.1.0 and create a temporary broker instance. This will generate the 7.1.0 configuration files required to upgrade a 7.0.x broker.

Prerequisites

- Before upgrading AMQ Broker, review the release notes for the target release.
  The release notes describe important enhancements, known issues, and changes to behavior in the target release.

  For more information, see the AMQ Broker 7.1 Release Notes.

- Before upgrading your 7.0.x brokers, you must first install version 7.1.
  For steps on installing 7.1 on Linux, see Section 2.3, "Installing on Linux".

Procedure

1. If it is running, stop the 7.0.x broker you want to upgrade:

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

2. Back up the instance directory of the broker by copying it to the home directory of the current user.

```
cp -r BROKER_INSTANCE_DIR ~/
```

3. Open the file **artemis.profile** in the ***BROKER_INSTANCE_DIR/etc/*** directory of the 7.0.x broker.

   a. Update the **ARTEMIS_HOME** property so that its value refers to the installation directory for AMQ Broker 7.1.0:

   ```
   ARTEMIS_HOME="7.1.0_INSTALL_DIR"
   ```

   b. On the line below the one you updated, add the property **ARTEMIS_INSTANCE_URI** and assign it a value that refers to the 7.0.x broker instance directory:

   ```
   ARTEMIS_INSTANCE_URI="file://7.0.x_BROKER_INSTANCE_DIR"
   ```

   c. Update the **JAVA_ARGS** property by adding the **jolokia.policyLocation** parameter and assigning it the following value:

   ```
   -Djolokia.policyLocation=${ARTEMIS_INSTANCE_URI}/etc/jolokia-
   access.xml
   ```

4. Create a 7.1.0 broker instance. The creation procedure generates the configuration files required to upgrade from 7.0.x to 7.1.0. In the following example, note that the instance is created in the directory **upgrade_tmp**:

   ```
   $ 7.1.0_INSTALL_DIR/bin/artemis create --allow-anonymous --user
   admin --password admin upgrade_tmp
   ```

5. Copy configuration files from the **etc** directory of the temporary 7.1.0 instance into the ***BROKER_INSTANCE_DIR/etc/*** directory of the 7.0.x broker.

   a. Copy the **management.xml** file:

   ```
   $ cp TEMPORARY_7.1.0_BROKER_INSTANCE_DIR/etc/management.xml
   7.0_BROKER_INSTANCE_DIR/etc/
   ```

   b. Copy the **jolokia-access.xml** file:

   ```
   $ cp TEMPORARY_7.1.0_BROKER_INSTANCE_DIR/etc/jolokia-access.xml
   7.0_BROKER_INSTANCE_DIR/etc/
   ```

6. Open up the **bootstrap.xml** file in the ***BROKER_INSTANCE_DIR/etc/*** directory of the 7.0.x broker.

   a. Comment out or delete the following two lines:

   ```
   <app url="jolokia" war="jolokia.war"/>
   <app url="hawtio" war="hawtio-no-slf4j.war"/>
   ```

   b. Add the following to replace the two lines removed in the previous step:

   ```
   <app url="console" war="console.war"/>
   ```

7. Start the broker that you upgraded:

```
$ BROKER_INSTANCE_DIR/bin/artemis run
```

Additional Resources

For more information about creating an instance of the broker, see Section 4.1, "Creating a Broker Instance".

3.3.2. Upgrading from 7.0.x to 7.1.0 on Windows

Before you can upgrade a 7.0.x broker, you need to install Red Hat AMQ Broker 7.1.0 and create a temporary broker instance. This will generate the 7.1.0 configuration files required to upgrade a 7.0.x broker.

Prerequisites

- Before upgrading AMQ Broker, review the release notes for the target release.
  The release notes describe important enhancements, known issues, and changes to behavior in the target release.

  For more information, see the AMQ Broker 7.1 Release Notes.

- Before upgrading your 7.0.x brokers, you must first install version 7.1.
  For steps on installing 7.1 on Windows, see Section 2.4, "Installing on Windows".

Procedure

1. If it is running, stop the 7.0.x broker you want to upgrade:

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

2. Back up the instance directory of the broker by using a file manager.

   a. Right click on the *BROKER_INSTANCE_DIR* folder and select **Copy**.

   b. Right click in the same window and select **Paste**.

3. Open the file **artemis.profile** in the *BROKER_INSTANCE_DIR/etc/* directory of the 7.0.x broker.

   a. Update the **ARTEMIS_HOME** property so that its value refers to the installation directory for AMQ Broker 7.1.0:

```
ARTEMIS_HOME="7.1.0_INSTALL_DIR"
```

   b. On the line below the one you updated, add the property **ARTEMIS_INSTANCE_URI** and assign it a value that refers to the 7.0.x broker instance directory:

```
ARTEMIS_INSTANCE_URI="file://7.0.x_BROKER_INSTANCE_DIR"
```

   c. Update the **JAVA_ARGS** property by adding the **jolokia.policyLocation** parameter and assigning it the following value:

```
-Djolokia.policyLocation=${ARTEMIS_INSTANCE_URI}/etc/jolokia-
access.xml
```

4. Create a 7.1.0 broker instance. The creation procedure generates the configuration files required to upgrade from 7.0.x to 7.1.0. In the following example, note that the instance is created in the directory **upgrade_tmp**:

```
> 7.1.0_INSTALL_DIR/bin/artemis create --allow-anonymous --user
admin --password admin upgrade_tmp
```

5. Copy configuration files from the **etc** directory of the temporary 7.1.0 instance into the **BROKER_INSTANCE_DIR/etc/** directory of the 7.0.x broker.

   a. Copy the **management.xml** file:

   ```
   > cp TEMPORARY_7.1.0_BROKER_INSTANCE_DIR/etc/management.xml
   7.0_BROKER_INSTANCE_DIR/etc/
   ```

   b. Copy the **jolokia-access.xml** file:

   ```
   > cp TEMPORARY_7.1.0_BROKER_INSTANCE_DIR/etc/jolokia-access.xml
   7.0_BROKER_INSTANCE_DIR/etc/
   ```

6. Open up the **bootstrap.xml** file in the **BROKER_INSTANCE_DIR/etc/** directory of the 7.0.x broker.

   a. Comment out or delete the following two lines:

   ```
   <app url="jolokia" war="jolokia.war"/>
   <app url="hawtio" war="hawtio-no-slf4j.war"/>
   ```

   b. Add the following to replace the two lines removed in the previous step:

   ```
   <app url="console" war="console.war"/>
   ```

7. Start the broker that you upgraded:

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

Additional Resources

For more information about creating an instance of the broker, see Section 4.1, "Creating a Broker Instance".

3.4. UPGRADING A BROKER INSTANCE FROM 7.1.X TO 7.2.0

AMQ Broker 7.2.0 includes configuration files and settings that were not included with 7.0.x versions. If you are running 7.0.x instances, you must first upgrade those broker instances from 7.0.x to 7.1.0 before upgrading to 7.2.0. The following subsections describe how to upgrade a 7.1.x broker instance to 7.2.0 for different operating systems.
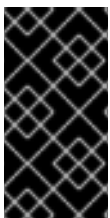
IMPORTANT

Starting with AMQ Broker 7.1.0, you can access the AMQ 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 Console and AMQ Broker Connections.

- Upgrading from 7.1.x to 7.2.0 on Linux

- Upgrading from 7.1.x to 7.2.0 on Windows

3.4.1. Upgrading from 7.1.x to 7.2.0 on Linux

NOTE

The name of the archive that you download could differ from what is used in the following examples.

Procedure

1. Download the desired archive from the Red Hat Customer Portal by following the instructions provided in Downloading an AMQ Broker Archive.

2. Change the owner of the archive to the same user that owns the AMQ Broker installation to be upgraded.

   ```
   sudo chown amq-broker:amq-broker amq-7.x.x.redhat-1.zip
   ```

3. Move the archive to the directory created during the original installation of AMQ Broker. In the following example, the directory **/opt/redhat** is used.

   ```
   sudo mv amq-7.x.x.redhat-1.zip /opt/redhat
   ```

4. As the directory owner, extract the contents of the compressed archive. In the following example, the user **amq-broker** extracts the archive by using the unzip command.

   ```
   su - amq-broker
   cd /opt/redhat
   unzip jboss-amq-7.x.x.redhat-1.zip
   ```

5. Stop the broker if it is running.

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

6. Back up the instance directory of the broker by copying it to the home directory of the current user.

   ```
   cp -r BROKER_INSTANCE_DIR ~/
   ```

7. (Optional) Note the current version of the broker. After the broker stops, a line similar to the one below is displayed at the end of its log file, which can be found at **BROKER_INSTANCE_DIR/log/artemis.log**.

   ```
   INFO  [org.apache.activemq.artemis.core.server] AMQ221001: Apache
   ActiveMQ Artemis Message Broker version 2.5.0.amq-720001-redhat-1
   [0.0.0.0, nodeID=554cce00-63d9-11e8-9808-54ee759954c4]
   ```

8. Edit the **BROKER_INSTANCE_DIR/etc/artemis.profile** configuration file to set the **ARTEMIS_HOME** property to the new directory created when the archive was extracted.

```
ARTEMIS_HOME='/opt/redhat/amq-7.x.x-redhat-1'
```

9. Restart the broker by entering the following command:

```
BROKER_INSTANCE_DIR/bin/artemis run
```

10. (Optional) Confirm that the broker is running and that the version has changed. After starting the broker, open the log file **BROKER_INSTANCE_DIR/log/artemis.log** and find two lines similar to the ones below. Note the new version number that appears in the log after the broker is live.

```
INFO  [org.apache.activemq.artemis.core.server] AMQ221007: Server is
now live
...
INFO  [org.apache.activemq.artemis.core.server] AMQ221001: Apache
ActiveMQ Artemis Message Broker version 2.5.0.amq-720001-redhat-1
[0.0.0.0, nodeID=554cce00-63d9-11e8-9808-54ee759954c4]
```

Additional Resources

- For more information about creating an instance of the broker, see Section 4.1, "Creating a Broker Instance".

- You can now store a broker instance's configuration files and data in any custom directory, including locations outside of the broker instance's directory. In the **BROKER_INSTANCE_DIR/etc/artemis.profile** file, update the **ARTEMIS_INSTANCE_ETC_URI** property by specifying the location of the custom directory after creating the broker instance. Previously, these configuration files and data could only be stored in the **etc/** and **data/** directories within the broker instance's directory.

3.4.2. Upgrading from 7.1.x to 7.2.0 on Windows

Procedure

1. Download the desired archive from the Red Hat Customer Portal by following the instructions provided in Downloading an AMQ Broker Archive.

2. Use a file manager to move the archive to the folder you created during the last installation of AMQ Broker.

3. Extract the file contents into the directory by right-clicking on the zip file and choosing **Extract All**.

4. Stop the broker if it is running by entering the following command.

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

5. Back up the broker by using a file manager.

   a. Right click on the *BROKER_INSTANCE_DIR* folder and select **Copy**.

   b. Right click in the same window and select **Paste**.

6. (Optional) Note the current version of the broker. After the broker stops, a line similar to the one below is displayed at the end of its log file, which can be found at **BROKER_INSTANCE_DIR\log\artemis.log**.

   ```
   INFO  [org.apache.activemq.artemis.core.server] AMQ221002: Apache
   ActiveMQ Artemis Message Broker version 2.0.0.amq-700005-redhat-1
   [4782d50d-47a2-11e7-a160-9801a793ea45] stopped, uptime 28 minutes
   ```

7. Edit the **BROKER_INSTANCE_DIR\etc\artemis.profile** configuration file to set the **ARTEMIS_HOME** property to the new directory created when the archive was extracted.

   ```
   ARTEMIS_HOME=NEW_INSTALL_DIR
   ```

8. Restart the broker entering the following command:

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

9. (Optional) Confirm that the broker is running and that the version has changed. After starting the broker, open the log file **BROKER_INSTANCE_DIR\log\artemis.log** and find two lines similar to the ones below. Note the new version number that appears in the log after the broker is live.

   ```
   INFO  [org.apache.activemq.artemis.core.server] AMQ221007: Server is
   now live
   ...
   INFO  [org.apache.activemq.artemis.core.server] AMQ221001: Apache
   ActiveMQ Artemis Message Broker version 2.5.0.amq-720001-redhat-1
   [0.0.0.0, nodeID=554cce00-63d9-11e8-9808-54ee759954c4]
   ```

Additional Resources

- For more information about creating an instance of the broker, see Section 4.1, "Creating a Broker Instance".

- You can now store a broker instance's configuration files and data in any custom directory, including locations outside of the broker instance's directory. In the **BROKER_INSTANCE_DIR\etc\artemis.profile** file, update the **ARTEMIS_INSTANCE_ETC_URI** property by specifying the location of the custom directory after creating the broker instance. Previously, these configuration files and data could only be stored in the **\etc** and **\data** directories within the broker instance's directory.

CHAPTER 4. GETTING STARTED

This chapter introduces you to AMQ Broker by showing you how to perform the following basic but important tasks:

1. Create a broker instance.

2. Start a broker instance.

3. Stop a broker instance.

4. Run the examples included with the installation of AMQ Broker.

4.1. CREATING A BROKER INSTANCE

A broker instance is a directory containing all the configuration and runtime data, such as logs and data files. The runtime data is associated with a unique broker process.

Creating a Broker Instance on Linux

Procedure

1. Create a directory location for the broker instance, for example **/var/opt/amq-broker**, and assign the user you created during installation as its owner:

```
$ sudo mkdir /var/opt/amq-broker
$ sudo chown -R amq-broker:amq-broker /var/opt/amq-broker
```

2. Move to the new directory and use the **artemis create** command to create a broker. For example, to create the broker instance **mybroker** under the **/var/opt/amq-broker** directory, run the following commands. Note that in the following example the user that was created during installation is the one to run the **create** command:

```
$ su - amq-broker
$ cd /var/opt/amq-broker
$ INSTALL_DIR/bin/artemis create mybroker
```

Creating a Broker Instance on Windows

Procedure

1. Using Windows Explorer create a new folder to serve as the location for the broker instance.

2. Open a Windows command prompt from the location you just created and use the **artemis.cmd create** command to create a broker. For example, to create the broker instance **mybroker**, run the following command:

```
> INSTALL_DIR\bin\artemis.cmd create mybroker
```

Responding to Questions Asked During the Creation Process
The create command will ask a series of questions to configure the broker instance. It will prompt for mandatory property value only. For the full list of properties available when creating an instance, run the **artemis help create** command.

The following example shows the command-line interactive process when creating a broker instance.

```
Creating ActiveMQ Artemis instance at: /var/opt/amq-broker/mybroker

--user: is mandatory with this configuration:
Please provide the default username:
<username>

--password: is mandatory with this configuration:
Please provide the default password:
<password>

--role: is mandatory with this configuration:
Please provide the default role:
amq

--allow-anonymous | --require-login: is mandatory with this configuration:
Allow anonymous access? (Y/N):
Y

Auto tuning journal ...
done! Your system can make 19.23 writes per millisecond, your journal-
buffer-timeout will be 52000

You can now start the broker by executing:

"/var/opt/amq-broker/mybroker/bin/artemis" run

Or you can run the broker in the background using:

"/var/opt/amq-broker/mybroker/bin/artemis-service" start
```

Broker Instance Directory Contents

The process creates a top level directory and several subdirectories containing configuration and runtime data. In the preceding examples, the top level directory is named **mybroker**. The documentation refers to the top level location as the *BROKER_INSTANCE_DIR*. The table below details its structure and contents.

Table 4.1. Contents of a broker instance directory

If you want to find...	Look here...
Scripts that launch and manage the instance	*BROKER_INSTANCE_DIR*/bin
Configuration files	*BROKER_INSTANCE_DIR*/etc
Storage for persistent messages	*BROKER_INSTANCE_DIR*/data
Log files	*BROKER_INSTANCE_DIR*/log
Temporary files	*BROKER_INSTANCE_DIR*/tmp

4.2. STARTING A BROKER INSTANCE
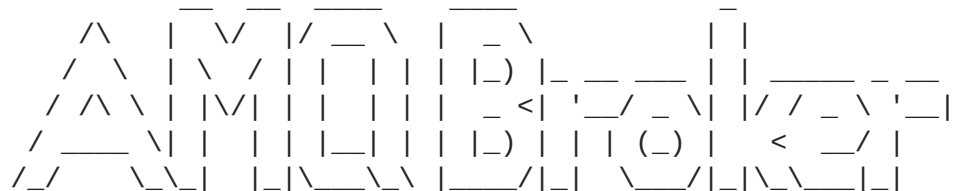
After the broker is created, use the **artemis** script in the ***BROKER_INSTANCE_DIR*/bin** directory to start it. For instance, to start a broker created at **/var/opt/amq-broker/mybroker**, use the following command as the user you created during installation. In the following example, the user **amq-broker** starts the broker:

```
$ su - amq-broker
$ /var/opt/amq-broker/mybroker/bin/artemis run
```

The **artemis** command has other other options available. Provide the **help** parameter to learn more:

```
$ /var/opt/amq-broker/mybroker/bin/artemis help
```

When started, the broker will produce output similar to the following:

```
         __  __  ____  ____    ____
    /\   |  \/  |/ __ \ |  _ \         | |
   /  \  | \  / | |  | | |_) |_ __ ___ | | ____ _ __
  / /\ \ | |\/| | |  | |  _ <| '__/ _ \| |/ / _ \ '__|
 / ____ \| |  | | |__| | |_) | | | (_) |   < __/ |
/_/    \_\_|  |_|\___\_\ |____/|_|  \___/|_|\_\___|_|

Red Hat JBoss AMQ 7.1.0.GA

10:53:43,959 INFO  [org.apache.activemq.artemis.integration.bootstrap]
AMQ101000: Starting ActiveMQ Artemis Server
10:53:44,076 INFO  [org.apache.activemq.artemis.core.server] AMQ221000:
live Message Broker is starting with configuration Broker Configuration
(clustered=false,journalDirectory=./data/journal,bindingsDirectory=./data/
bindings,largeMessagesDirectory=./data/large-
messages,pagingDirectory=./data/paging)
10:53:44,099 INFO  [org.apache.activemq.artemis.core.server] AMQ221012:
Using AIO Journal
10:53:44,158 INFO  [org.apache.activemq.artemis.core.server] AMQ221057:
Global Max Size is being adjusted to 1/2 of the JVM max size (-Xmx). being
defined as 1,073,741,824
10:53:44,176 INFO  [org.apache.activemq.artemis.core.server] AMQ221043:
Protocol module found: [artemis-server]. Adding protocol support for: CORE
10:53:44,176 INFO  [org.apache.activemq.artemis.core.server] AMQ221043:
Protocol module found: [artemis-amqp-protocol]. Adding protocol support
for: AMQP
10:53:44,177 INFO  [org.apache.activemq.artemis.core.server] AMQ221043:
Protocol module found: [artemis-hornetq-protocol]. Adding protocol support
for: HORNETQ
10:53:44,177 INFO  [org.apache.activemq.artemis.core.server] AMQ221043:
Protocol module found: [artemis-mqtt-protocol]. Adding protocol support
for: MQTT
10:53:44,177 INFO  [org.apache.activemq.artemis.core.server] AMQ221043:
Protocol module found: [artemis-openwire-protocol]. Adding protocol
support for: OPENWIRE
10:53:44,177 INFO  [org.apache.activemq.artemis.core.server] AMQ221043:
Protocol module found: [artemis-stomp-protocol]. Adding protocol support
for: STOMP
10:53:44,250 INFO  [org.apache.activemq.artemis.core.server] AMQ221034:
```

```
Waiting indefinitely to obtain live lock
10:53:44,250 INFO  [org.apache.activemq.artemis.core.server] AMQ221035:
Live Server Obtained live lock
10:53:44,348 INFO  [org.apache.activemq.artemis.core.server] AMQ221003:
Deploying queue DLQ on address DLQ
10:53:44,458 INFO  [org.apache.activemq.artemis.core.server] AMQ221003:
Deploying queue ExpiryQueue on address ExpiryQueue
10:53:44,767 INFO  [org.apache.activemq.artemis.core.server] AMQ221020:
Started EPOLL Acceptor at 0.0.0.0:61616 for protocols
[CORE,MQTT,AMQP,STOMP,HORNETQ,OPENWIRE]
10:53:44,772 INFO  [org.apache.activemq.artemis.core.server] AMQ221020:
Started EPOLL Acceptor at 0.0.0.0:5445 for protocols [HORNETQ,STOMP]
10:53:44,775 INFO  [org.apache.activemq.artemis.core.server] AMQ221020:
Started EPOLL Acceptor at 0.0.0.0:5672 for protocols [AMQP]
10:53:44,778 INFO  [org.apache.activemq.artemis.core.server] AMQ221020:
Started EPOLL Acceptor at 0.0.0.0:1883 for protocols [MQTT]
10:53:44,780 INFO  [org.apache.activemq.artemis.core.server] AMQ221020:
Started EPOLL Acceptor at 0.0.0.0:61613 for protocols [STOMP]
10:53:44,787 INFO  [org.apache.activemq.artemis.core.server] AMQ221007:
Server is now live
10:53:44,787 INFO  [org.apache.activemq.artemis.core.server] AMQ221001:
Apache ActiveMQ Artemis Message Broker version 2.4.0.amq-710001-redhat-1
[0.0.0.0, nodeID=036f2ead-df5d-11e7-a02f-0800274e493d]
10:53:45,041 INFO  [io.hawt.branding.plugin.PluginContextListener]
Initialized hawtio-redhat-fuse-branding plugin
10:53:45,107 INFO
[org.apache.activemq.hawtio.plugin.PluginContextListener] Initialized
artemis-plugin plugin
10:53:45,552 INFO  [io.hawt.system.ConfigManager] Configuration will be
discovered via system properties
10:53:45,556 INFO  [io.hawt.jmx.JmxTreeWatcher] Welcome to hawtio
1.4.0.redhat-630310 : http://hawt.io/ : Don't cha wish your console was
hawt like me? ;-)
10:53:45,558 INFO  [io.hawt.jmx.UploadManager] Using file upload
directory: /var/opt/amq-broker/broker-7.1/tmp/uploads
10:53:45,572 INFO  [io.hawt.web.AuthenticationFilter] Starting hawtio
authentication filter, JAAS realm: "activemq" authorized role(s): "amq"
role principal classes:
"org.apache.activemq.artemis.spi.core.security.jaas.RolePrincipal"
10:53:45,599 INFO  [io.hawt.web.JolokiaConfiguredAgentServlet] Jolokia
overridden property: [key=policyLocation, value=file:/var/opt/amq-
broker/broker-7.1//etc/jolokia-access.xml]
10:53:45,624 INFO  [io.hawt.web.RBACMBeanInvoker] Using MBean
[hawtio:type=security,area=jmx,rank=0,name=HawtioDummyJMXSecurity] for
role based access control
10:53:45,782 INFO  [io.hawt.system.ProxyWhitelist] Initial proxy
whitelist: [localhost, 127.0.0.1, 192.168.99.101, rhel7-1, 10.0.3.15]
10:53:46,040 INFO  [org.apache.activemq.artemis] AMQ241001: HTTP Server
started at http://localhost:8161
10:53:46,041 INFO  [org.apache.activemq.artemis] AMQ241002: Artemis
Jolokia REST API available at http://localhost:8161/console/jolokia
10:53:46,041 INFO  [org.apache.activemq.artemis] AMQ241004: Artemis
Console available at http://localhost:8161/console
```

4.2.1. Starting a Broker as a Linux Service

If the broker is installed on Linux, you can run it as a service.

Procedure

1. Create a new **amq-broker.service** file in the **/etc/systemd/system/** directory.

2. Copy the following text into the file.
   Modify the path and user fields according to the information provided during the broker instance creation. In the example below, the user **amq-broker** starts the broker service installed under the **/var/opt/amq-broker/mybroker/** directory.

   ```
   [Unit]
   Description=AMQ Broker
   After=syslog.target network.target

   [Service]
   ExecStart=/var/opt/amq-broker/mybroker/bin/artemis run
   Restart=on-failure
   User=amq-broker
   Group=amq-broker

   # A workaround for Java signal handling
   SuccessExitStatus=143

   [Install]
   WantedBy=multi-user.target
   ```

3. Open a terminal.

4. Enable the broker service using the following command:

   ```
   sudo systemctl enable amq-broker
   ```

5. Run the broker service using the following command:

   ```
   sudo systemctl start amq-broker
   ```

4.2.2. Starting a Broker as a Windows Service

If the broker is installed on Windows, you can run it as a service.

Procedure

The following steps require that you open a command prompt to enter the commands.

1. Install the broker as a service with the following command:

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

2. Start the service by using the following command:

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

3. (Optional) Uninstall the service:

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

4.3. STOPPING A BROKER INSTANCE

You can stop the broker manually by issuing the **stop** command. Immediately after the command is entered, all connections to the broker are forcefully stopped and the shutdown process begins.

Procedure

1. Stop the broker.

   a. If you are running the broker on Linux, issue the following command:

   ```
   BROKER_INSTANCE_DIR\bin\artemis stop
   ```

   b. If you are running the broker on Windows as a service, issue the following command:

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

4.3.1. Stopping a Broker Instance Gracefully

A manual shutdown forcefully disconnects all clients after a stop command is entered. As an alternative, configure the broker to shutdown gracefully by using the **graceful-shutdown-enabled** configuration element.

When **graceful-shutdown-enabled** is set to **true**, no new client connections are allowed after a stop command is entered. However, existing connections are allowed to close on the client-side before the shutdown process is started. The default value for **graceful-shutdown-enabled** is **false**.

Use the **graceful-shutdown-timeout** configuration element to set a length of time, in milliseconds, for clients to disconnect before connections are forcefully closed from the broker side. After all connections are closed, the shutdown process is started. One advantage of using **graceful-shutdown-timeout** is that it prevents client connections from delaying a shutdown. The default value for **graceful-shutdown-timeout** is **-1**, meaning the broker waits indefinitely for clients to disconnect.

The following procedure demonstrates how to configure a graceful shutdown that uses a timeout.

Procedure

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

2. Add the **graceful-shutdown-enabled** configuration element and set the value to **true**.

   ```xml
   <configuration>
       <core>
           ...
           <graceful-shutdown-enabled>
               true
           </graceful-shutdown-enabled>
           ...
       </core>
   </configuration>
   ```

3. Add the **graceful-shutdown-timeout** configuration element and set a value for the timeout in milliseconds. In the following example, client connections are forcefully closed 30 seconds (**30000** milliseconds) after the stop command is issued.

```
<configuration>
    <core>
        ...
        <graceful-shutdown-enabled>
            true
        </graceful-shutdown-enabled>
        <graceful-shutdown-timeout>
            30000
        </graceful-shutdown-timeout>
        ...
    </core>
</configuration>
```

4.4. RUNNING THE INCLUDED EXAMPLES

AMQ Broker ships with many example programs that highlight various basic and advanced features of the product. They can be found under **INSTALL_DIR/examples**.

The examples require a supported release of the Java Development Kit (JDK) and Maven version 3.2 or greater to build and run. Consult Supported Configurations for details about JDK support. The next section shows you how to install and configure Maven, enabling you to run the examples.

Downloading Maven and the AMQ Repository
Running the examples requires that both Maven and the Red Hat AMQ Broker Maven repository are installed locally. The steps below show you how to download, install, and configure both. For more information about Maven refer to the appendix.

Downloading and Installing Maven

1. Go to Apache Maven Project - Download Maven and download the latest distribution for your operating system.

2. Install Maven for your operating system.
   For more information, see Installing Apache Maven.

Downloading and Installing the AMQ Maven Repository

1. Open a browser and log in to the Red Hat Customer Portal at https://access.redhat.com.

2. Click **Downloads**.

3. Find **Red Hat AMQ** in the **Product Downloads** list.

4. Select **Download Latest Broker** from the drop-down list located on the same row to go to the **Software Downloads** page for AMQ Broker.

5. From the **Software Downloads** page, select the correct AMQ Broker version from the **Version** drop-down menu.

6. Click the **Download** link for the Maven Repository to download a .zip file containing the repository.

7. Unzip the file on the local file system into a directory of your choosing.
   This creates a new directory, which contains the Maven repository in a subdirectory named
   maven-repository/.
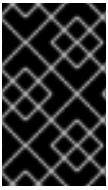
> **IMPORTANT**
>
> If you want to continue to use an older local repository, you must configure it separately in
> the Maven **settings.xml** configuration file. Each local repository must be configured
> within its own **<repository>** element.
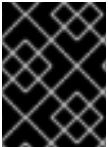
> **IMPORTANT**
>
> Remove the cached **repository/** subdirectory located under the **.m2/** directory before
> attempting to use a new Maven repository.

Configuring the Maven Settings File

1. Locate the Maven **settings.xml** file for your operating system. It is usually located in the
   ${user.home}/.m2/ directory.

   - For Linux or Mac, this is **~/.m2/**

   - For Windows, this is **\Documents and Settings\.m2** or **\Users\.m2**

2. (Optional) If you do not find a **settings.xml** file in **${user.home}/.m2/**, there is a default
   version located in the **conf/** directory of your Maven installation. Copy the default
   settings.xml file into the **${user.home}/.m2/** directory.

3. Copy the following XML into the **<profiles>** element of the **settings.xml** file.

```xml
<!-- Configure the JBoss AMQ Maven repository -->
<profile>
  <id>jboss-amq-maven-repository</id>
  <repositories>
    <repository>
      <id>jboss-amq-maven-repository</id>
      <url>file://JBOSS_AMQ_REPOSITORY_PATH</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
    </repository>
  </repositories>
  <pluginRepositories>
    <pluginRepository>
      <id>jboss-amq-maven-repository</id>
      <url>file://JBOSS_AMQ_REPOSITORY_PATH</url>
      <releases>
        <enabled>true</enabled>
      </releases>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
```

```
      </pluginRepository>
    </pluginRepositories>
  </profile>
```

4. Replace *JBOSS_AMQ_REPOSITORY_PATH* within the **url** element with the location of the maven repository on the local file system. If you followed the installation steps above, the location should end with **/maven-repository**.

5. Copy the following XML into the **<activeProfiles>** element of the **settings.xml** file.

```
<activeProfile>jboss-amq-maven-repository</activeProfile>
```

6. (Optional) If you copied the default **settings.xml** from your Maven installation, uncomment the **<active-profiles>** section if it was commented out by default.

> **IMPORTANT**
>
> If your Maven repository contains outdated artifacts, you may encounter one of the following Maven error messages when you build or deploy your project:
>
> - Missing artifact ARTIFACT_NAME
>
> - [ERROR] Failed to execute goal on project PROJECT_NAME; Could not resolve dependencies for PROJECT_NAME
>
> To resolve the issue, delete the cached version of your local repository to force a download of the latest Maven artifacts. The cached repository is located here: **${user.home}/.m2/repository/**

Several of the examples use UDP clustering, which may not work in your environment when you first attempt to run them. On Linux, you can issue a command to provide the appropriate environment. The following command redirects any traffic directed to 224.0.0.0 to the loopback interface, allowing examples that use UDP clustering to run successfully:

```
sudo route add -net 224.0.0.0 netmask 240.0.0.0 dev lo
```

Running the `queue` Example to Test Your Installation

To run any example, change directories to the desired example and type **mvn clean verify**. You can also use the profile **-P examples** to run multiple examples under any example directory tree. For details on each example, read the **readme.html** file in each example directory. An excellent example to use as a "sanity test" of your broker installation is the **queue** example. The example creates a producer to send a message to the broker and a consumer that receives the same message.

Procedure

- To build and run the example, navigate to the *INSTALL_DIR***/examples/features/standard/queue** directory:

  ```
  $ cd INSTALL_DIR/examples/features/standard/queue
  ```

- Run the **mvn clean verify** command:

  ```
  $ mvn clean verify
  ```

Maven will start the broker and run the example. When completed the output should look something like the following example. Note that the example may take a long time to run the first time since Maven will download any missing dependencies required to run the example.

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------
---------
[INFO] Building ActiveMQ Artemis JMS Queue Example 1.2.0.amq-700
[INFO] ------------------------------------------------------------------
---------
  ...
server-out:11:30:45,534 INFO  [org.apache.activemq.artemis.core.server]
AMQ221001: Apache ActiveMQ Artemis Message Broker version 1.2.0.amq-700
[0.0.0.0, nodeID=e748cdbd-232c-11e6-b145-54ee7531eccb]
[INFO] Server started
[INFO]
[INFO] --- artemis-maven-plugin:1.2.0.amq-700:runClient (runClient) @
queue ---
Sent message: This is a text message
server-out:11:30:46,545 INFO  [org.apache.activemq.artemis.core.server]
AMQ221003: Trying to deploy queue jms.queue.exampleQueue
Received message: This is a text message
[INFO]
[INFO] --- artemis-maven-plugin:1.2.0.amq-700:cli (stop) @ queue ---
[INFO] ------------------------------------------------------------------
---------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------
---------
[INFO] Total time: 6.368 s
[INFO] Finished at: 2016-05-26T11:30:47+01:00
[INFO] Final Memory: 44M/553M
[INFO] ------------------------------------------------------------------
---------
```

CHAPTER 5. CONFIGURATION

In the previous chapter you saw how to install AMQ Broker and how to start a broker instance. You should understand how the broker instance you created is configured and what the important aspects of the default configuration are.

You configure brokers by editing plain text and XML files. Changing a broker's configuration typically involves opening the appropriate configuration file, locating the proper location for the relevant configuration within the XML schema, and then adding or removing XML elements and attributes.

5.1. CONFIGURATION FILES AND LOCATIONS

All of a broker's configuration files are found in the directory **BROKER_INSTANCE_DIR/etc**. The table below lists each file and describes what they do.

Table 5.1. Broker Configuration Files

File(s)	Description
broker.xml	The main configuration file. 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. Use it to change the location of **broker.xml**, configure the web server, and set some security settings.
logging.properties	Use this file to set logging properties for the broker instance.
artemis.profile	Sets environment variables used while the broker instance is running.
login.config artemis-users.properties artemis-roles.properties	Security-related files. Use these files to set up authentication for user access to the broker instance.

The schema for the broker's XML configuration files are found under the product installation at **INSTALL_DIR/schema**. Consult the XSD files, especially **artemis-configuration.xsd**, for the schema definitions of the XML configuration elements used in this guide as well as a description of each element.

5.2. DEFAULT CONFIGURATION

The following sections identify important aspects of the default configuration found in **broker.xml** after first installing a broker instance.

5.2.1. Reloading Configuration Updates

You can configure the broker to perform a periodic check for changes in the configuration files and, if any are found, reload the configuration to activate them.

The check interval is configured using the **configuration-file-refresh-period** setting in **broker.xml**. The default is 5000 milliseconds.

After this function is activated, the broker automatically checks for, and reloads, changes in 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.

- **config-delete-addresses** - If this is set to **FORCE**, then the address (and its queues) that was deleted from the configuration file will be deleted when the configuration file is reloaded. If there are any messages in the queues, they are removed also.

- **config-delete-queues** - If this is set to **FORCE**, then the queue that was deleted from the configuration file will be deleted when the configuration file is reloaded. If there are any messages in the queue, they are removed also.

Destinations can be explicitly removed by using CLI or management operations.

Security Settings

SSL/TLS key 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. For most circumstances, other administrative operations exist to remove clients when appropriate.

5.2.2. Acceptors

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. Be default AMQ Broker includes configuration for several acceptors.

Default Acceptor Configuration in broker.xml

```xml
<configuration>
  <core>
  ...
  <acceptors>
   <!-- Default ActiveMQ Artemis Acceptor.  Multi-protocol adapter.
Currently supports ActiveMQ Artemis Core, OpenWire, STOMP, AMQP, MQTT, and
HornetQ Core. -->
   <!-- performance tests have shown that openWire performs best with
these buffer sizes -->
   <acceptor name="artemis">tcp://0.0.0.0:61616?
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576</acceptor>

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

   <!-- STOMP Acceptor. -->
   <acceptor name="stomp">tcp://0.0.0.0:61613?protocols=STOMP</acceptor>

   <!-- HornetQ Compatibility Acceptor.  Enables HornetQ Core and STOMP
for legacy HornetQ clients. -->
   <acceptor name="hornetq">tcp://0.0.0.0:5445?
```

```
protocols=HORNETQ,STOMP</acceptor>

   <!-- MQTT Acceptor -->
   <acceptor name="mqtt">tcp://0.0.0.0:1883?protocols=MQTT</acceptor>
  </acceptors>
  ...
 </core>
</configuration>
```

See Networking Connections for details on acceptors and connectors configuration. For more information on the protocols supported by the broker, refer to Protocols.

5.2.3. Addresses and Queues

The default configuration defines two queues. The first one handles messages that arrive with no known destination and is known as a Dead Letter Queue (DLQ). The second queue holds messages that have lived past their expiration and therefore should not be routed to their original destination and is known as an Expiry Queue.

Default JMS Queues in broker.xml

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

The default configuration also uses the concept of an address setting to establish a default, or catch all, set of configuration that is applied to any created queue or topic.

Default Address Setting in broker.xml

```
<configuration>
 <core>
  ...
  <address-settings>
   <!--default for catch all-->
   <address-setting match="#">
      <dead-letter-address>jms.queue.DLQ</dead-letter-address>
      <expiry-address>jms.queue.ExpiryQueue</expiry-address>
      <redelivery-delay>0</redelivery-delay>
      <max-size-bytes>10485760</max-size-bytes>
      <message-counter-history-day-limit>10</message-counter-history-day-
```

```
limit>
      <address-full-policy>BLOCK</address-full-policy>
   </address-setting>
  </address-settings>
 </core>
</configuration>
```

Note how the address-setting uses a wildcard syntax to dictate which queues and addresses are to have the configuration applied. In this case the single # symbol tells AMQ Broker to apply the configuration to all queues and topics.

5.2.4. Security

AMQ Broker contains a flexible role-based security model for applying security to queues, based on their addresses. Also, just like an address setting, you can use a wildcard syntax, as does the default configuration.

Default Security Configuration in broker.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>
```

Note all the types of permission that can be used as part of a security setting. For more details on permission types and other security topics see the Security chapter.

5.2.5. Message Persistence

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. As an alternative to using a journal, you can configure persistence to use a JDBC connection to write to a database table. A third option is to disable persistence completely.

Default Persistence Configuration in broker.xml

```
<configuration>
  <core>
```

```
    ...
    <persistence-enabled>true</persistence-enabled>
    <!-- this could be ASYNCIO or NIO-->
    <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>-1</journal-pool-files>
    ...
  </core>
</configuration>
```

When **persistence-enabled** is set to **true** the journal persists to the directories specified.

The two valid options for **journal-type** are:

ASYNCIO

Use this option to maintain journals by using **libaio**. This option is recommended if you are running the broker with Linux Kernel 2.6 or later.
If the **libaio** package is found when creating the broker instance, then the **ASYNCIO** option is set by default.

NIO

Use this option to maintain journals by using Java. This option is automatically enabled when **journal-type** is set to **ASYNCIO**, but the **libaio** package cannot be found.
If the **libaio** package is *not* found when creating the broker instance, then the **NIO** option is set by default.

For more details about the available options and how to configure them, see Configuring Persistence.

5.3. MODULARIZING THE BROKER CONFIGURATION FILE

AMQ Broker supports XML inclusions in the **broker.xml** configuration file, which enables you to break **broker.xml** into separate files. For example, if you have multiple brokers that share the same set of addresses and address settings, you can define the addresses in a separate file, and then include that file in each broker's **broker.xml** configuration file.

You must create a separate XML file for each section to include it in the **broker.xml** file. For example, you must create separate files for **address-settings** and **security-settings** sections. If you include two sections in a single XML file and start the broker, you will get an error something similar to **The markup in the document following the root element must be well-formed.**.

> **NOTE**
>
> Add the following line inside the **<configuration>** tag at the beginning of the **broker.xml** file if it is not listed.
>
> ```
> xmlns:xi="http://www.w3.org/2001/XInclude"
> ```

Procedure

- Add the **include** tag and the file name with full path in the **broker.xml** file to include the XML configuration from another file:

  ```
  <xi:include href="/opt/my-broker-config/my-addresses.xml"/>
  ```

  In the above example, the **my-address-settings.xml** file is sourced into the **broker.xml** file.

You must ensure XML elements in separated files must be namespaced correctly. For example, if **address-settings** element was separated, it must have the element namespace defined as below:

```
<address-settings xmlns="urn:activemq:core">
```

NOTE

The broker reloads configuration updates based on the frequency set in the **configuration-file-refresh-period** setting in the **broker.xml** file. For more information, see Reloading Configuration Updates.

Additional Resources

- **Validating XML Schema**: You can use any XML validator program to validate the **broker.xml** file against the Schema. If you use **xmllint**, use the **--xinclude** flag when running the command. For more information about **XInclude**, see https://www.w3.org/TR/xinclude/.

- For information about the multiple configuration options that can be added to the **broker.xml** file, see https://activemq.apache.org/artemis/docs/latest/configuration-index.html.

CHAPTER 6. 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.

6.1. 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.

6.2. 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.

6.3. 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.

6.4. 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.

6.5. CONFIGURING AN SSL/TLS CONNECTION

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

6.6. 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.

6.7. 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.NettyConnectorFactor
y", connectionParams);

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

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

CHAPTER 7. 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.

7.1. 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.

Default acceptors in `broker.xml`

```
<configuration>
  <core>
    ...
    <acceptors>
      <!-- Default ActiveMQ Artemis Acceptor.  Multi-protocol adapter.
Currently supports ActiveMQ Artemis Core, OpenWire, STOMP, AMQP, MQTT, and
HornetQ Core. -->
      <acceptor name="artemis">tcp://0.0.0.0:61616?
tcpSendBufferSize=1048576;tcpReceiveBufferSize=1048576</acceptor>

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

      <!-- STOMP Acceptor. -->
      <acceptor name="stomp">tcp://0.0.0.0:61613?
protocols=STOMP</acceptor>

      <!-- HornetQ Compatibility Acceptor.  Enables HornetQ Core and STOMP
for legacy HornetQ clients. -->
      <acceptor name="hornetq">tcp://0.0.0.0:5445?
protocols=HORNETQ,STOMP</acceptor>

      <!-- MQTT Acceptor -->
```

```
            <acceptor name="mqtt">tcp://0.0.0.0:1883?protocols=MQTT</acceptor>
        </acceptors>
        ...
    </core>
</configuration>
```

> **NOTE**
>
> The only requirement to enable a protocol is to add the **protocols** parameter to the URI query string. 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>
```

7.2. 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.

7.2.1. 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.

7.2.2. Configuring AMQP Security

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

7.3. 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.

7.4. 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**.

7.5. 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.

7.5.1. 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).

7.5.2. 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**.

7.5.3. 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.

7.5.4. 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….
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.

7.5.5. 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 8. 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.

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 8.1. 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**.

8.1. 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".

8.2. 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 8.1. Point-to-Point
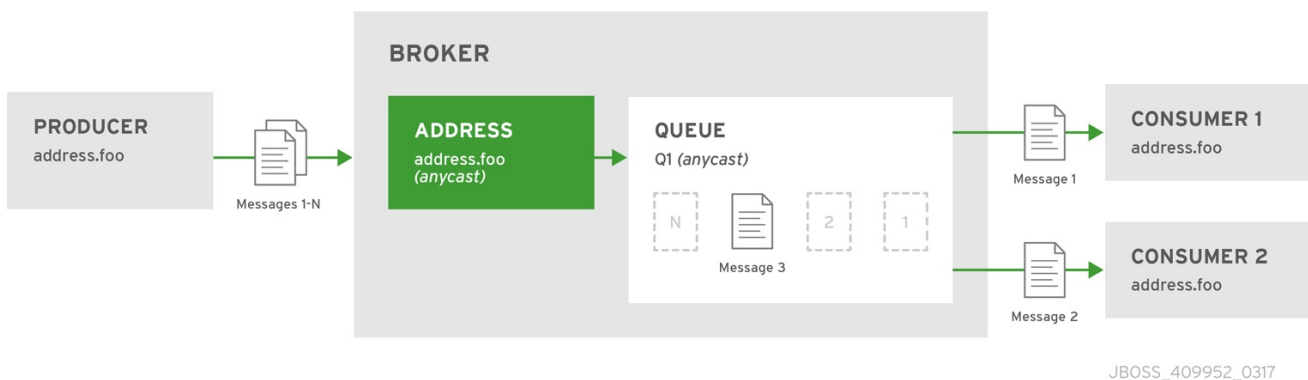
JBOSS_409952_0317

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>
```

8.3. 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 8.2. Publish-Subscribe
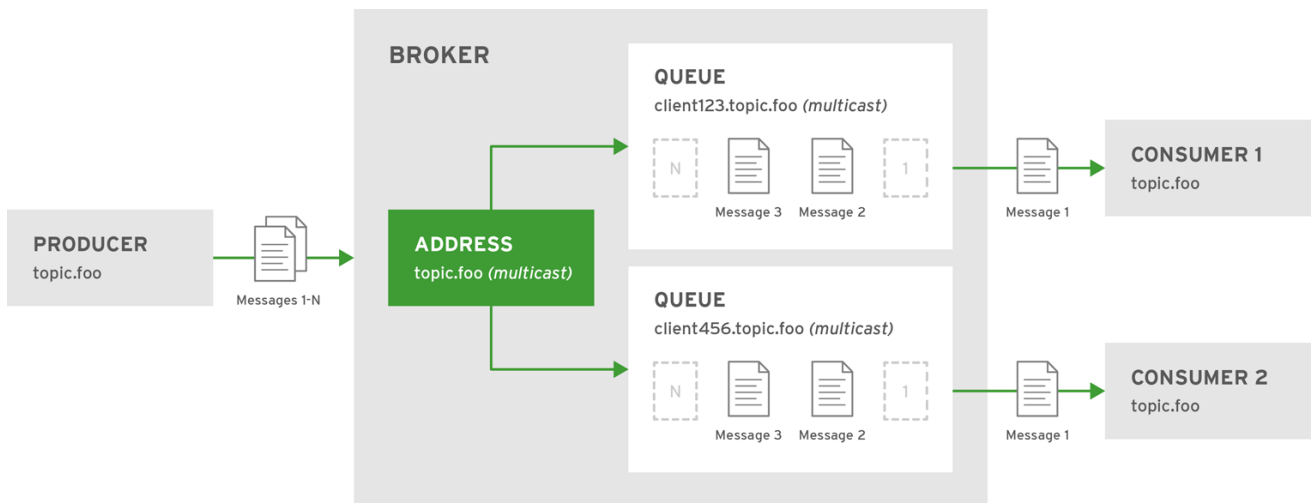
JBOSS_409952_0317

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>
```

8.4. 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 8.3. Point-to-Point with Two Queues
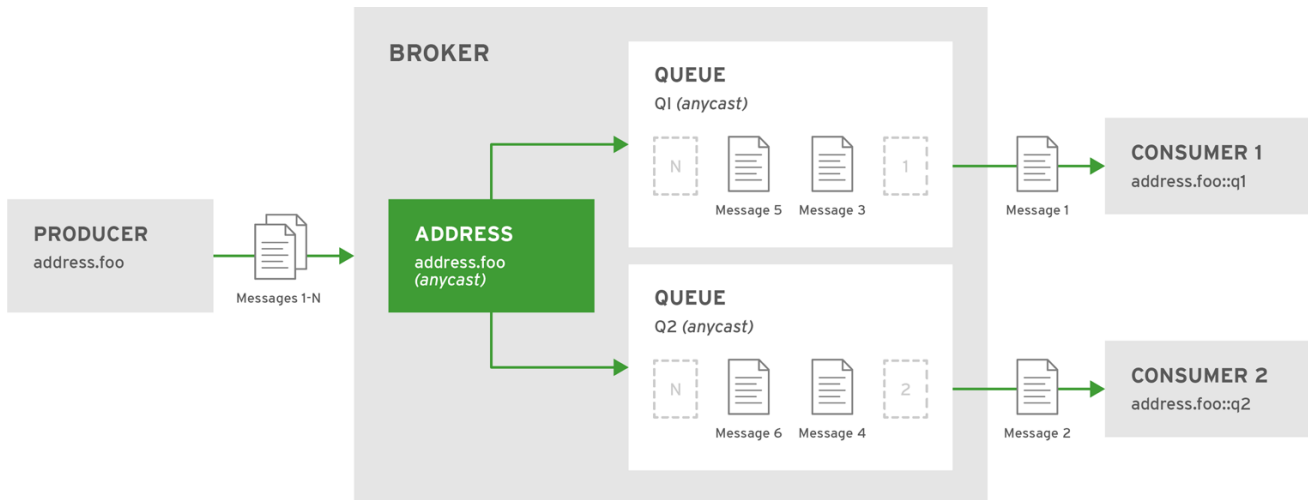
JBOSS_409952_0317
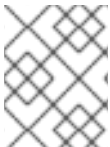
> **NOTE**
>
> This is how AMQ Broker handles load balancing of queues across multiple nodes in a cluster.

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>
   ```

8.5. 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 8.4. Point-to-Point and Publish-Subscribe

JBOSS_409952_0317
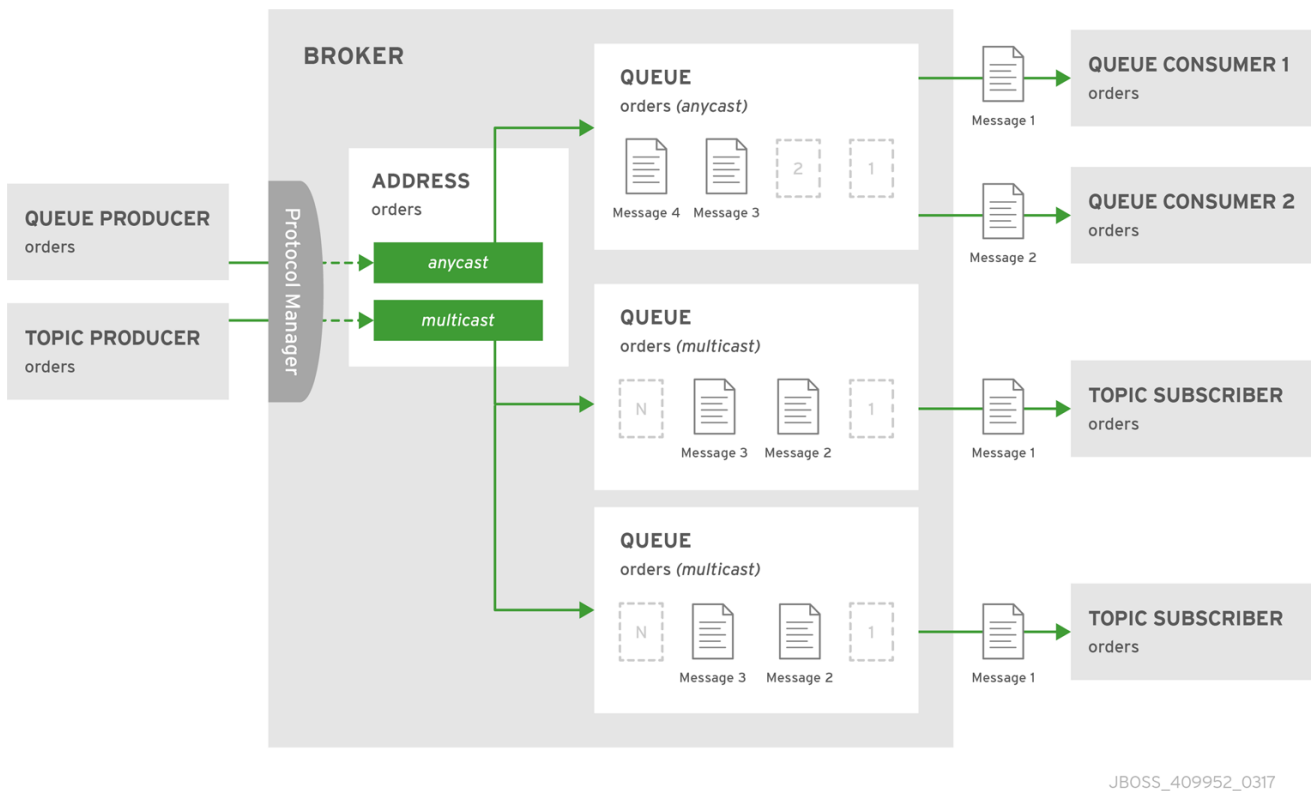
NOTE

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>
```

8.6. 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>
```

8.7. 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 `::`.
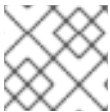
NOTE

Only message consumers can use FQQN. Do not use FQQN with message producers.

Prerequisites

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

```
<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.

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

8.8. 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.

```
<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.

8.9. LIMITING THE NUMBER OF CONSUMERS CONNECTED TO A QUEUE

You can limit the number of consumers connected to for a particular queue by using the **max-consumers** attribute. Create an exclusive consumer by setting **max-consumers** flag can be set 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>
```

8.10. 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.

8.10.1. 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");
```

8.10.2. 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**.

8.11. 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>
```

8.12. 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 8.2. Protocol Manager Actions

If the queue is of this type…	The typical action for a protocol manager is to…

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

8.13. 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;suppressInternalManagem
entObjects=false</acceptor>
```

8.14. 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 8.3. 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**.

Example	Description
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.

8.15. 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 chapter for more information about routing types.

CHAPTER 9. 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.

9.1. 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 10.2.5, "Using Multiple Login Modules" section.

9.2. 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 9.1. `login.config`**
>
> ```
> activemq {  ❶
>
> org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule
> required  ❷  ❸
>         org.apache.activemq.jaas.properties.user="artemis-
> users.properties";  ❹
>         org.apache.activemq.jaas.properties.role="artemis-
> roles.properties";
> };
> ```
>
> ❶ An alias for a configuration. In this section the alias used is **activemq**. Substitute another in your environment.
>
> ❷ The implementation class (**org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule**).
>
> ❸ A flag which indicates whether the success of the **LoginModule is `required**, **requisite**, **sufficient**, or **optional**.
>
> ❹ 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 9.2. `artemis-users.properties`

```
user1=secret    1
user2=swordfish    2
user3=myPassword    3
```

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

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

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

Example 9.3. `artemis-roles.properties`

```
admin=user1,user2    1
developer=user3    2
```

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

2   *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"/>
> ```

9.3. 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 9.1. Permissions

To allow users to…	Use this parameter…
Create addresses	`createAddress`
Delete addresses	`deleteAddress`
Create a durable queue under matching addresses	`createDurableQueue`
Delete a durable queue under matching addresses	`deleteDurableQueue`
Create a non-durable queue under matching addresses	`createNonDurableQueue`
Delete a non-durable queue under matching addresses	`deleteNonDurableQueue`
Send a message to matching addresses	`send`
Consume a message from a queue bound to matching addresses	`consume`
Invoke management operations by sending management messages to the management address	`manage`
Browse a queue bound to the matching address	`browse`

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.

9.3.1. 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">  ❶
        <permission type="send" roles="producer"/>  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 *producer* role have *send* permissions on *queue1*.

9.3.2. 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:

```xml
<security-settings>
    <security-setting match="queue1">    1
        <permission type="consume" roles="consumer"/>    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 *consumer* role have *consume* permissions on *queue1*.

9.3.3. 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.

```xml
<security-settings>
    <security-setting match="#">    1
        <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>
```

1   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.

9.3.4. 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.

9.4. 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.

9.4.1. 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 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"/>   ❶
   </whitelist>
   ```

   ❶   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**

9.4.2. Configuring Authentication Based on Roles

The **role-access** method 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">  ❶
           <access method="list*" roles="view,update,amq"/>  ❷
           <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>
   ```

   ❶ A match will be applied to any MBean attribute that has the domain name org.apache.activemq.apache.

   ❷ Specified roles can invoke the listed methods.
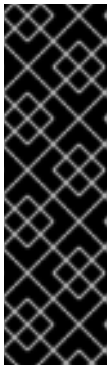
   > **IMPORTANT**
   >
   > === You must ensure that the order of the individual lines in the configuration is as per the template. Any change in the line indentation leads to changes in the semantics of the assigned privileges. For example, if you move the line **<access method="*" roles="amq,guest"/>** inside the **<role-access>** tag from last position to the first, it changes the semantics of applied privileges. If used as the first line, it means *grant access to everything to these roles with the exception of following specific cases*. If used as the last line, it means *grant access to everything to these roles* (default). ===

   Here, the specific tasks like **list***, **get***, **set***, **is** and ***** are specified using the access method. The invoked method is matched against the methods listed in the configuration. The user is assigned the roles given for the best matching method. For example, if you try the invoke a method called **listMessages** on an MBean with the **org.apache.activemq.artemis** domain, then it would match the access with the method of **list**. You can also explicitly configure this by using the full method name like the following:

   <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. For example,

```
<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>
```

Access to MBean attributes are converted to method calls, so these are controlled using the **list***, **set***, **get***, and **is*** syntax. The * (wildcard) syntax is used as a catch-all for every other method that is not listed in the configuration.

> **NOTE**
>
> The **default-access** element is mainly the catch-all for every method call that is not handled using the **role-access** configuration. The **default-access** and **role-access** have the same **match** element semantics.

CHAPTER 10. 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.

10.1. ACCESSING THE AMQ CONSOLE

Starting with AMQ Broker 7.1.0, you can access the AMQ 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 Console and AMQ Broker Connections.

10.2. 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.

10.2.1. 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.

10.2.2. 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;trustStorePath=../etc/client.truststore;trustStorePas
sword=5678!</acceptor>
```

NOTE

AMQ Broker supports multiple protocols, and each protocol and platform has different ways to specify TLS parameters. However, in the case of a client using 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`.

10.2.3. 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  ❶
      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
      ```

      ❶ 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 10.1. `login.config`

```
activemq {

org.apache.activemq.artemis.spi.core.security.jaas.TextFileCertificateLo
ginModule 1
        debug=true 2
        org.apache.activemq.jaas.textfiledn.user="artemis-
users.properties" 3
        org.apache.activemq.jaas.textfiledn.role="artemis-
roles.properties"; 4
};
```

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

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

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

4 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 10.2. `artemis-users.properties`

```
system=CN=system,O=Progress,C=US 1
user=CN=humble user,O=Progress,C=US
guest=CN=anon,O=Progress,C=DE
```

1   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 10.3. `artemis-roles.properties`**
>
> ```
> admins=system
> users=system,user
> guests=guest
> ```
> **1**

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

10.2.4. 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=T
RUST_STORE_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=T
RUST_STORE_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 10.2.3, "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.

10.2.5. 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 10.4. bootstrap.xml

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

Example Configuration

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

> **Example 10.5. `login.config`**
>
> ```
> activemq {
>
> org.apache.activemq.artemis.spi.core.security.jaas.PropertiesLoginModule
> sufficient    1
>         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 10.6. `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.

10.2.6. 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 a user and what roles they have is handled by the security manager. AMQ Broker ships with a user manager that reads user credentials from a file on disk, and can also plug into JAAS or JBoss Application Server 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.

10.2.7. 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.

10.2.7.1. Configuring Connection and Queue Limits

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

```
<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.

10.3. INTEGRATING WITH LDAP

10.3.1. 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:

```
<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 10.7. 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
    authentication=simple 7
    userBase="dc=example,dc=com" 8
    userSearchMatching="(uid={0})" 9
    userSearchSubtree=true 10
    userRoleName= 11
    roleBase="dc=example,dc=com" 12
    roleName=cn 13
    roleSearchMatching="(member={0})" 14
    roleSearchSubtree=true 15
    ;
};
```

Example 10.8. 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
    authentication=simple
    userBase="dc=example,dc=com"
    userSearchMatching="(CN={0})"
    userSearchSubtree=true
    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 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).

8 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.

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

10

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**

11 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.

12 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.

13 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.

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

15 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)**.

10.3.2. Configure LDAP Authorization

The **LegacyLDAPSecuritySettingPlugin** security-setting-plugin will read the security information that was previously handled by **LDAPAuthorizationMap** and the **cachedLDAPAuthorizationMap** in Apache A-MQ 6 and turn it into corresponding **security settings** where possible.

The security implementations of the two brokers do not match perfectly so some translation must occur to achieve near equivalent functionality.

Here is an example of the plugin's configuration:

```
<security-setting-plugin class-
name="org.apache.activemq.artemis.core.server.impl.LegacyLDAPSecuritySetti
ngPlugin"> 1
 <setting name="initialContextFactory"
value="com.sun.jndi.ldap.LdapCtxFactory"/> 2
 <setting name="connectionURL" value="ldap://localhost:1024"/> 3
 <setting name="connectionUsername" value="uid=admin,ou=system"/> 4
 <setting name="connectionPassword" value="secret"/> 5
 <setting name="connectionProtocol" value="s"/> 6
 <setting name="authentication" value="simple"/> 7
</security-setting-plugin>
```

1   **class-name**. The implementation is **org.apache.activemq.artemis.core.server.impl.LegacyLDAPSecuritySettingPlugin**.

2   **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).

3   **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 DN of a particular node in the directory tree. For example, **ldap://ldapserver:10389/ou=system**. The default is **ldap://localhost:1024**.

4   **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.

5

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

6 **connectionProtocol** - any value is supported but is effectively unused. In the future, this option may allow one 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.

7 **authentication**. Specifies the authentication method used when binding to the LDAP server. Can take either of the values, **simple** (username and password, the default value) or **none** (anonymous). Note: Simple Authentication and Security Layer (SASL) authentication is currently not supported.

Other possible settings not shown in the example above 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** (that is, the default value).

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. 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

Whether or not to enable a listener that will automatically receive updates made in the LDAP server and update the broker's authorization configuration in real-time. The default value is **true**.

The name of the queue or topic defined in LDAP will serve as the "match" for the security-setting, the permission value will be mapped from the A-MQ 6 type to the AMQ 7 type, and the role will be mapped as-is. Since the name of the queue or topic coming from LDAP will server as the "match" for the security-setting the security-setting may not be applied as expected to JMS destinations since AMQ 7 always prefixes JMS destinations with "jms.queue." or "jms.topic." as necessary.

A-MQ 6 only has three permission types - **read**, **write**, and **admin**. These permission types are described on the ActiveMQ website; http://activemq.apache.org/security.html.

However, as described previously, AMQ 7 has 10 permission types:

- **createAddress**

- **deleteAddress**

- **createDurableQueue**

- **deleteDurableQueue**

- **createNonDurableQueue**

- **deleteNonDurableQueue**

- **send**

- **consume**

- **manage**

- **browse**

The list below shows how the old types are mapped to the new types:

- **read** - **consume**, **browse**

- **write** - **send**

- **admin** - **createDurableQueue**, **deleteDurableQueue**, **createNonDurableQueue**, **deleteNonDurableQueue**, **createAddress**, **deleteAddress**

As mentioned, there are a few places where a translation was performed to achieve some equivalence:

- This mapping does not include the AMQ 7 **manage** permission type since there is no type analogous for that in A-MQ 6.

- The **admin** permission in A-MQ 6 relates to whether or not the broker will auto-create a destination if it does not exist and the user sends a message to it. AMQ 7 automatically allows the automatic creation of a destination if the user has permission to send message to it. Therefore, the plugin will map the **admin** permission to the four aforementioned permissions in AMQ 7.

10.3.3. 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 10.3.2, "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 Chapter 5, *Configuration*.

10.4. 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.

10.4.1. 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.

10.4.2. 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.

10.4.2.1. 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=alternativ
e-sasl-gssapi`
</acceptor>
```

10.4.3. 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
       ;  1
   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 mapping.

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**.

10.5. 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**
>
> When a user is created upon broker creation, the **artemis-users.properties** file 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 masked password does not allow access to the AMQ Console. You can use both *masked* and *hashed* password in other configuration files except the **artemis-users.properties** file.

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 Chapter 5, *Configuration*.

10.6. 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**.

CHAPTER 11. 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.

11.1. 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.

11.1.1. 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
```

11.2. 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.

```
<configuration>
  <core>
    ...
    <persistence-enabled>true</persistence-enabled>         1
    <journal-type>ASYNCIO</journal-type>        2
    <bindings-directory>./data/bindings</bindings-directory>      3
    <journal-directory>./data/journal</journal-directory>      4
    <journal-datasync>true</journal-datasync>       5
    <journal-min-files>2</journal-min-files>       6
    <journal-pool-files>-1</journal-pool-files>      7
    ...
  </core>
</configuration>
```

1   Set to **true** to use the file based journal for persistence.

2   The type of journal to use. If set to ASYNCIO, the broker first attempts to use AIO and falls back to NIO if ASYNCIO is not found.

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

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

5 Set to **true** to use **fdatasync** to confirm writes to the disk.

6 The number of journal files to pre-create when the broker starts.

7 The number of files to keep after reclaiming un-used files. The default value, -1, means that no files are deleted during clean up.

11.2.1. 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.

11.2.2. 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**

11.2.3. 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.

11.2.4. 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.

```xml
<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.

11.2.5. 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**.

11.3. 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.

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

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.

```
<configuration>
  <core>
    <store>
      <database-store>
        <jdbc-connection-url>jdbc:derby:data/derby/database-store;create=true</jdbc-connection-url>  1
        <bindings-table-name>BINDINGS_TABLE</bindings-table-name>
  2
        <message-table-name>MESSAGE_TABLE</message-table-name>  3
        <large-message-table-name>LARGE_MESSAGES_TABLE</large-message-table-name>  4
        <jdbc-driver-class-name>org.apache.derby.jdbc.EmbeddedDriver</jdbc-driver-class-name>
  5
      </database-store>
    </store>
  </core>
</configuration>
```

1 **jdbc-connection-url** is the full JDBC connection URL for your database server. The connection url should include all configuration parameters and database name.

2 **bindings-table-name** is the name of the table in which the bindings data is stored. Specifying table name allows users to share single database amongst multiple servers, without interference.

3 **message-table-name** is the name of the table in which the bindings data is stored.

4 **large-message-table-name** is the name of the table in which messages and related data is persisted.

5 **jdbc-driver-class-name** is the fully qualified class name of the desired database Driver.

11.4. 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 12. 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.

12.1. 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.

12.2. 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.

```
<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.

12.3. 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.

```
<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 12.1. 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)
page-size-bytes	The size of each page file used on the paging system.	10MiB (10 * 1024 * 1024 bytes)
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

12.4. 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** configuration element 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 value for **global-max-size** is in bytes, but you can use byte notation ("K", "Mb", "GB", for example) for convenience. The default value for **global-max-size** is **-1**, which means no limit.

Configuring the `global-max-size`

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** configuration element 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 12.3, "Configuring an Address for Paging" for information about setting the paging mode for an address.

12.5. 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
      ```

12.6. 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

12.6.1. 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

12.7. 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.
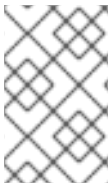
> **NOTE**
>
> Blocking works only if the protocol being used supports it. For example, an AMQP producer will understand a Block packet when it is sent by the broker, but a STOMP producer will not.

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.

12.8. 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 13. WORKING WITH LARGE MESSAGES

You can configure AMQ Broker to store large messages on a physical disk or in a database table. Handling large messages in this way avoids the memory overhead that occurs when storing several large messages in memory.

AMQ Broker can persist large messages even if the client and broker are running with limited memory. The maximum size of a large message depends only on the amount of space available for your physical disk or database table.

> **NOTE**
>
> Large message support is available for the AMQP, Core, and OpenWire protocols. Additionally, the STOMP protocol provides its own method for handling large messages. See "Handling Large Messages with STOMP" for more information.

> **NOTE**
>
> If you persist large messages on a disk, it is recommended that the large messages directory be located on a different volume than the one used to persist the message journal or the paging directory.

13.1. PREPARING BROKERS TO STORE LARGE MESSAGES

Large messages are stored on physical disk or database table. You must configure the broker to specify where large messages are stored.

Procedure

- Add configuration to **BROKER_INSTANCE_DIR/etc/broker.xml** that references the storage location for large messages.

  - If you are storing large messages on disk, add the **large-messages-directory** configuration element and provide the file system location, as shown in the following example:

    ```
    <configuration>
      <core>
        ...
        <large-messages-directory>/path/to/large-messages</large-messages-directory> 1
        ...
      </core>
    </configuration>
    ```

    1 The default value for the **large-messages-directory** configuration element is **BROKER_INSTANCE_DIR/data/largemessages**

  - If you are storing large messages in a database table, add the name of the table to your **database-store**, as shown in the following example:

    ```
    <store>
      <database-store>
    ```

```
      ...
      <large-message-table>MY_TABLE</large-message-table>  1
    </database-store>
  </store>
```

1   The default value for the **large-message-table** configuration element is
         LARGE_MESSAGE_TABLE.

> **NOTE**
>
> **Handling large AMQP messages**: The broker treats AMQP messages that
> are larger than the size of either the **journal-buffer-size** or **journal-
> file-size** as large messages. For more information about the journal, see
> Messaging Journal Configuration Elements.

Additional Resources

See the **large-message** example found under **BROKER_INSTANCE_DIR/examples/standard/** for
a working example showing how to work with large messages.

For more information about configuring a **data-store** see Configuring JDBC Persistence.

13.2. PREPARING AMQ CORE PROTOCOL JMS CLIENTS TO SEND LARGE MESSAGES

You prepare client connections to handle large messages by setting a value for the property
minLargeMessageSize. The value can be provided as a parameter in the connection URL, or it can be
set by using a supported client API. Any message larger than **minLargeMessageSize** is considered a
large message.

> **NOTE**
>
> AMQ Broker messages are encoded using two bytes per character. Therefore, if the
> message data is filled with ASCII characters (which are one byte in size), the size of the
> resulting message would roughly double. When setting the value of
> **minLargeMessageSize**, remember that encoding can increase message size. The
> default value for minLargeMessageSize` is 100KiB.

Procedure

- Set the minimum size for large messages.

  - If you are using JNDI to instantiate your connection factory, set the size in a
    jndi.properties file by using the parameter **minLargeMessageSize**.

    ```
    java.naming.factory.initial=org.apache.activemq.artemis.jndi.Acti
    veMQInitialContextFactory
    connectionFactory.myConnectionFactory=tcp://localhost:61616?
    minLargeMessageSize=250000
    ```

  - If you are not using JNDI, set the size using the method
    ActiveMQConnectionFactory.setMinLargeMessageSize().

```
ConnectionFactory cf =
ActiveMQJMSClient.createConnectionFactory(...)
cf.setMinLargeMessageSize(250000);
```

13.3. PREPARING OPENWIRE CLIENTS TO SEND LARGE MESSAGES

> **NOTE**
>
> Configuration options added to the connection URI used by an AMQ OpenWire JMS client must include the prefix **wireFormat.** to take effect. Options missing this prefix are ignored.

Procedure

- Set the minimum size for large messages.

  - If you are using JNDI to instantiate your connection factory, set the size in a **jndi.properties** file by using the parameter **minLargeMessageSize**. You must add the prefix **wireFormat.** to the parameter for it to take effect.

    ```
    java.naming.factory.initial=org.apache.activemq.artemis.jndi.Acti
    veMQInitialContextFactory
    connectionFactory.myConnectionFactory=tcp://localhost:61616?
    wireFormat.minLargeMessageSize=250000
    ```

  - If you are not using JNDI, set the size using the method **ActiveMQConnectionFactory.setMinLargeMessageSize()**.

    ```
    ConnectionFactory cf =
    ActiveMQJMSClient.createConnectionFactory(...)
    cf.setMinLargeMessageSize(250000);
    ```

13.4. SENDING LARGE MESSAGES

AMQ Broker supports Java-based **InputStreams** for sending large messages. The most common use case is to send files stored on your disk, but you could also send the data as JDBC Blobs or JSON objects recovered from **HTTPRequests**.

> **NOTE**
>
> When using JMS, streaming large messages is supported only when using **StreamMessage** and **BytesMessage**.

Procedure

- To send a large message, set the **JMS_AMQ_InputStream** property to mark the message as streamed:

  ```
  BytesMessage message = session.createBytesMessage();
  FileInputStream fileInputStream = new FileInputStream(fileInput);
  BufferedInputStream bufferedInput = new
  BufferedInputStream(fileInputStream);
  ```

```
message.setObjectProperty("JMS_AMQ_InputStream", bufferedInput);
...
```

13.5. RECEIVING LARGE MESSAGES

The AMQ Broker Core JMS API has a method for synchronously receiving a streamed message. The methods block further processing until the input stream is completely received.

Procedure

- To receive a large message, set the **JMS_AMQ_SaveStream** on the message object:

```
BytesMessage messageReceived =
(BytesMessage)messageConsumer.receive(120000);
File outputFile = new File("large_message_received.dat");
FileOutputStream fileOutputStream = new
FileOutputStream(outputFile);
BufferedOutputStream bufferedOutput = new
BufferedOutputStream(fileOutputStream);

// This will block until the entire content is saved on disk
messageReceived.setObjectProperty("JMS_AMQ_SaveStream",
bufferedOutput);
```

Receiving a Large Message Asynchronously
The Core JMS API also has a method for asynchronously receiving a streamed message. The method does not block processing by a consumer while it receives the input stream.

Procedure

- To receive a large message asynchronously, set the **JMS_AMQ_OutputStream** parameter on the message object:

```
BytesMessage messageReceived =
(BytesMessage)messageConsumer.receive(120000);
File outputFile = new File("large_message_received.dat");
FileOutputStream fileOutputStream = new
FileOutputStream(outputFile);
BufferedOutputStream bufferedOutput = new
BufferedOutputStream(fileOutputStream);

// This will not block until the entire content is saved on disk
messageReceived.setObjectProperty("JMS_AMQ_OutputStream",
bufferedOutput);
```

13.6. LARGE MESSAGES AND JAVA CLIENTS

There are a two recommended options available to Java developers who are writing clients that use large messages.

One option is to use an instance of **InputStream** and **OutputStream**. For example, a **FileInputStream** could be used to send a message taken from a large file on a physical disk. A **FileOutputStream** could 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:

```
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
}
```

13.7. COMPRESSING LARGE MESSAGES

You can enable clients to compress large messages before sending them. The ZIP algorithm is used to compress the message body as the message is sent to the broker.

> **NOTE**
>
> If the compressed size of a large message is less than the value of **minLargeMessageSize**, the message is sent as a regular message. Therefore, it is not written to the broker's large-message data directory.

- If you use a Core JMS client and JNDI, use the JNDI context environment to enable message compression:

  ```
  java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveM
  QInitialContextFactory
  connectionFactory.myConnectionFactory=tcp://localhost:61616?
  compressLargeMessages=true
  ```

- Add the **minLargeMessageSize** parameter to the connection factory URL to set the minimum size requirement for messages to be compressed. In the following example, messages are compressed only when they exceed 250 kilobytes in size.

  ```
  connectionFactory.myConnectionFactory=tcp://localhost:61616?
  compressLargeMessages=true&minLargeMessageSize=250kb
  ```

13.8. HANDLING LARGE MESSAGES WITH STOMP

STOMP clients might send large bodies of frames, which can exceed the size of the broker's internal buffer, causing unexpected errors.

To prevent this situation from occurring, set the acceptor's **stompMinLargeMessageSize** parameter to the desired size. Proper sizing is affected by system resources such as the amount of disk space available, as well as the size of the messages. It is recommended that you run performance tests using several values for **stompMinLargeMessageSize** to determine an appropriate size.

The broker checks the size of the body of each STOMP frame coming from connections established with this acceptor. If the size of the body is equal to or greater than the value of **stompMinLargeMessageSize**, the message is persisted as a large message.

Procedure

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

2. Add the **stompMinLargeMessageSize** parameter and its desired value to an existing or new **acceptor**, as shown in the following example:

```
<acceptors>
   <acceptor name="stomp-acceptor">tcp://localhost:61613?
protocols=STOMP;stompMinLargeMessageSize=10240</acceptor>
   ...
</acceptors>
```

In the preceding example, the broker is configured to accept STOMP messages on port **61613**. If the acceptor receives a STOMP frame with a body larger than or equal to **10240** bytes the broker will persist it as a large message.

When a large message is delivered to a STOMP consumer, the broker automatically converts it 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.

The default value of **stompMinLargeMessageSize** is 102400 bytes.

CHAPTER 14. 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.Acti
    veMQInitialContextFactory
    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);
```

14.1. 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>  1
  <connection-ttl-check-interval>1000</connection-ttl-check-
interval>  2
  ...
 </core>
</configuration>
```

1   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.

2   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.Acti
    veMQInitialContextFactory
    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);
```

14.2. DISABLING ASYNCHRONOUS CONNECTION EXECUTION

Most packets received on the broker side are executed on the **remoting** thread. These packets represent short-running operations and are always executed on the **remoting** thread for performance reasons. However, some packet types are executed using a thread pool instead of the **remoting** thread, which adds a little network latency.

The packet types that use the thread pool are implemented within the Java classes listed below. The classes are all found in the package **org.apache.actiinvemq.artemis.core.protocol.core.impl.wireformat**.

- RollbackMessage

- SessionCloseMessage

- SessionCommitMessage

- SessionXACommitMessage

- SessionXAPrepareMessage

- SessionXARollbackMessage

Procedure

- To disable asynchronous connection execution, add the **async-connection-execution-enabled** configuration element to *BROKER_INSTANCE_DIR*/etc/broker.xml and set it to **false**, as in the example below. The default value is **true**.

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

14.3. 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:

```
Connection jmsConnection = null;
```

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

CHAPTER 15. 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.

15.1. 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.

15.1.1. 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.Acti
    veMQInitialContextFactory
    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);
    ```

15.1.2. 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.Acti
    veMQInitialContextFactory
    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);
    ```

15.1.3. 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.Acti
    veMQInitialContextFactory
    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.

15.1.4. 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.
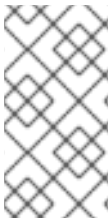
> **NOTE**
>
> Rate-limited flow control can be used in conjunction with window-based flow control. Rate-limited flow control affects only how many messages a client can consume in a second and not how many messages are in its buffer. With a slow rate limit and a high window-based limit, the internal 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.Acti
veMQInitialContextFactory
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.

15.2. 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.

15.2.1. 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.Acti
    veMQInitialContextFactory
    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);
    ```

15.2.2. 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.

15.2.3. 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**.

15.2.4. 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.Acti
veMQInitialContextFactory
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 16. 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.

16.1. 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.Acti
    veMQInitialContextFactory
    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.

16.2. 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.Acti
    veMQInitialContextFactory
    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);
    ```

16.3. CLUSTERED MESSAGE GROUPING

Using message groups in a cluster is a complex undertaking. Messages with a particular group ID can arrive on any broker, so each broker needs to know which group IDs are bound to which consumer on which broker. Each clustered broker therefore uses a grouping handler to manage the complexity of routing of grouped messages. There are two types of grouping handlers: Local and Remote. Each cluster should choose one broker to have a local grouping handler. All the other brokers use remote handlers.

Specifically, a route is initially proposed by the broker that received the message. A suitable route follows the normal clustered routing conditions, round robin for example. If the proposal is accepted by the grouping handlers, the broker routes messages to this queue from that point on. If the route is rejected, an alternative is offered, and the broker again routes to that queue indefinitely. Other brokers also route to the queue chosen at proposal time. Once the message arrives at the queue, the message is pinned to a consumer on that queue.

Configuration for a grouping handler includes three important elements, as shown in the example below.

```
<grouping-handler name="my-grouping-handler">
   <type>LOCAL</type>     1
   <address>jms</address>     2
   <timeout>100000</timeout>     3
</grouping-handler>
```

1   The **type** element determines whether the broker uses a **LOCAL** or **REMOTE** handler.

2   The **address** element references a pre-existing cluster connection. See Configuring a Cluster Connection in the chapter on clustering for more information.

3   The **timeout** element sets how long to wait for a decision to be made, an exception is thrown during the send if this timeout is reached, ensuring strict message ordering. The default is 5000 milliseconds, or 5 seconds.

Procedure

You configure clustered brokers to use grouping handlers by adding configuration to the **BROKER_INSTACE_DIR/etc/broker.xml** file.

1. Choose a broker from the cluster to be the one to use a local handler and add the following configuration.

```xml
<grouping-handler name="my-grouping-handler">
    <type>LOCAL</type>
    <address>jms</address>
    <timeout>10000</timeout>
</grouping-handler>
```

2. Configure the other brokers to use a remote handler.

```xml
<grouping-handler name="my-grouping-handler">
    <type>REMOTE</type>
    <address>jms</address>
    <timeout>5000</timeout>
</grouping-handler>
```

Related Information

See **clustered-grouping** under **INSTALL_DIR/examples/features/clustered** for a working example of how message groups are used and configured within a cluster of brokers.

Clustered Message Grouping Best Practices
Some best practices should be followed when using clustered grouping:

- There is a single point of failure when using only one local handler. To avoid this, create a slave broker as backup, using the same configuration for the local handler as the master.

- Distribute consumers evenly across brokers. This is only an issue if you are creating and closing consumers regularly. Since messages are always routed to the same queue, removing a consumer from a queue may leave it with no consumers.

- Use durable queues when possible. If you remove a queue after a message group is bound to it, other brokers may still try to route messages to it. Avoid this situation by making sure that the queue is deleted by the session that sent the messages. Deleting the queue in this way forces a new routing proposal. Alternatively, you could start using a different group ID.

- When using a timeout, set the value for the remote handlers to at least half of the value of the local handler.

CHAPTER 17. 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.

17.1. 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);
  ```

17.2. 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**.

17.3. 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.

17.4. 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 on cluster connections and how to configure them, see Configuring a Cluster Connection in the chapter on clustering.

CHAPTER 18. 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.

18.1. 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.StompFrameIntercepto r** interface.

```
package com.example;

import
org.apache.activemq.artemis.core.protocol.stomp.StompFrameInterce
ptor;
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;
  }
}
```

18.2. 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.

      ```
      <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.

      ```
      <configuration>
        <core>
          ...
          <remoting-outgoing-interceptors>
            <class-name>org.example.MyOutgoingInterceptor</class-name>
          </remoting-outgoing-interceptors>
        </core>
      </configuration>
      ```

18.3. 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 19. 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 size of a message in bytes. The value is an integer.

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.

19.1. 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**.

```xml
<configuration>
  <core>
    ...
    <addresses>
        <address name="myQueue">
            <anycast>
                <queue name="myQueue">
                    <filter string="NEWS='technology'"/>
                </queue>
            </anycast>
        </address>
    </addresses>
  </core>
</configuration>
```

19.2. 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.

The JMS specification also states that hyphens cannot be used as property identifiers, making them unavailable for use in filters. However, this constraint can be overcome by using the **hyphenated_props:** prefix.

Configuring a Filter to Convert a String to a Number
To convert String properties to a numeric type, add the prefix **convert_string_expressions:** to the value of the **filter**.

Procedure

- Edit *BROKER_INSTANCE_DIR***/etc/broker.xml** by applying the prefix **convert_string_expressions:** to the desired **filter**. The example below edits the **filter** value from **age > 18** to **convert_string_expressions:age > 18**.

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

Enabling a Filter to Use Hyphens
To enable the use of hyphens when filtering JMS properties, add the prefix **hyphenated_props:** to the value of the **filter**.

Procedure

- Edit *BROKER_INSTANCE_DIR***/etc/broker.xml** by applying the prefix **hyphenated_props:** to the desired **filter**. In the example below, a **filter** is edited so that it can select for the hyphenated property **foo-bar**.

```
<configuration>
  <core>
    ...
    <addresses>
        <address name="myQueue">
            <anycast>
                <queue name="myQueue">
```

```
                <filter string="hyphenated_props='foo-bar = 0'"/>
            </queue>
        </anycast>
      </address>
    </addresses>
  </core>
</configuration>
```

CHAPTER 20. CLUSTERING

With AMQ Broker, clustering allows brokers to be grouped together and share the message processing load. Each active broker in the cluster manages its own messages and handles its own connections. A cluster provides a number of benefits:

- Brokers can be connected together in many different topologies.

- Client connections can be balanced across the cluster.

- Messages can be redistributed to avoid broker starvation.

- Clients and brokers can connect to the cluster with minimal information about it.

High Availability (HA) is another benefit of clustering that groups brokers into master-slave pairs. One feature of HA is that it enables client requests to fail over to the slave broker in case the master loses connectivity to the network. See the chapter on HA and failover for more information.

You configure a cluster inside the broker's **_BROKER_INSTANCE_DIR_/etc/broker.xml** configuration file. There are three major configuration elements related to clustering:

discovery-group

Determines how the broker discovers other members in the cluster. Discovery can be dynamic and use either UDP or JGroups to find other brokers on the network that are members of the same cluster. (JGroups is based on IP multicast, although TCP can also be used as transport.) Alternatively, discovery can be static and use a configured list of connections to specific brokers.

broadcast-group

Determines how the broker transmits cluster-related information, such as enrollment, to other brokers in the cluster. A **broadcast-group** can use either UDP or JGroups, but the choice must match its **discovery-group** counterpart. For example, if the **broadcast-group** is configured to use UDP, the **discovery-group** must also use UDP, including the same multicast address.

cluster-connection

Details about the broker's connection to other brokers in the cluster. Uses a **discovery-group** to make the initial connection to each broker in the cluster.
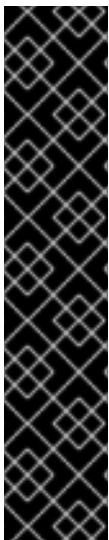
> **IMPORTANT**
>
> After a cluster broker has been configured, it is common to copy the configuration to other brokers to produce a symmetric cluster. However, when copying the broker files, do **not** copy any of the following directories from one broker to another:
>
> - **bindings**
>
> - **journal**
>
> - **large-messages**
>
> When a broker is started for the first time and initializes its journal files, it also persists a special identifier to the **journal** directory. This id **must** be unique among brokers in the cluster, or the cluster will not form properly.
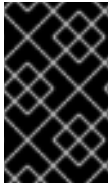
IMPORTANT

The sequence of the elements in the cluster connection configuration has to be in a specific order. You must adhere to the order as shown in the schema file *INSTALL_DIR*/schema/artemis-configuration.xsd.

20.1. ABOUT BROADCAST GROUPS

A broker uses a broadcast group to push information about its cluster-related connection to other potential cluster members on the network. The connection information is captured in the configuration as a **connector**. See Networking Connections: Acceptors and Connectors for details on connectors. There can be many broadcast groups per broker. If the broker has a backup, the backup's connection information will be broadcast too. A **broadcast-group** can use either UDP or JGroups, but the choice must match its **discovery-group** counterpart.

Prerequisites

- Configuring a **broadcast-group** requires a **connector**, which defines connection information for each broker. The configuration for the connector will be sent to other brokers in the cluster during discovery. The default configuration includes several connectors already, or you can create a new one. See About Connectors for more information.

Procedure

- Add **broadcast-group** configuration to *BROKER_INSTANCE_DIR*/etc/broker.xml. Below is an example **broadcast-group** that uses UDP, followed by a description of each configuration element. You typically use the default broadcast **group-address** and **group-port** values, but you can specify any of the following elements to suit your environment.

```
<configuration>
  <core>
    ...
    <broadcast-groups>
      <broadcast-group name="my-broadcast-group">    1
        <local-bind-address>172.16.9.3</local-bind-address>    2
        <local-bind-port>5432</local-bind-port>    3
        <group-address>231.7.7.7</group-address>    4
        <group-port>9876</group-port>    5
        <broadcast-period>2000</broadcast-period>    6
        <connector-ref>netty-connector</connector-ref>    7
      </broadcast-group>
    </broadcast-groups>
  </core>
</configuration>
```

1 You must use the **name** attribute to give the broadcast group a unique name.

2 The **local-bind-address** is the address to which the UDP socket is bound. If you have multiple network interfaces on your broker, you should specify which one you wish to use for broadcasts. If this property is not specified, the socket will be bound to an IP address chosen by the OS. This is a UDP specific attribute.

3

If you want to specify a local port to which the datagram socket is bound, you can specify it here. In most cases, you would just use the default value of **-1**, which signifies that an anonymous port

4   The **group-address** is the multicast address to which the data will be broadcast. It is a class D IP address in the range **224.0.0.0** to **239.255.255.255**, inclusive. The address **224.0.0.0** is reserved and is not available for use. This parameter is mandatory. This is a UDP specific attribute.

5   The **group-port** is the UDP port number used for broadcasting. This parameter is mandatory. This is a UDP specific attribute.

6   The **broadcast-period** is the interval in milliseconds between consecutive broadcasts. This parameter is optional; the default value is **2000** milliseconds.

7   The **connector-ref** declares a reference to a previously configured connector that is transmitted by the broadcast (see Network Connections: Acceptors and Connectors for more information).

Related Information

See the **clustered-queue** example under **INSTALL_DIR/examples/features/clustered** for a working example of a **broadcast-group** that uses UDP.

20.1.1. Configuring a Broadcast Group to Use JGroups

Sometimes using UDP is not an option due to constraints put on the network. In these situations, you can configure the broadcast group to communicate using JGroups as an alternative to UDP.

Prerequisites

JGroups communication requires a separate configuration file. See **clustered-jgroups** under **INSTALL_DIR/examples/features/clustered** for an example JGroups configuration file.

Procedure

To create a broadcast group that uses JGroups to communicate, add configuration to **BROKER_INSTANCE_DIR/etc/broker.xml** configuration file. Below is an example **broadcast-group** that uses JGroups, followed by a description of each configuration element.

```
<configuration>
  <core>
    ...
    <broadcast-groups>
      <broadcast-group name="my-broadcast-group">  1
        <jgroups-file>test-jgroups-file_ping.xml</jgroups-file>  2
        <jgroups-channel>activemq_broadcast_channel</jgroups-channel>

3
        <broadcast-period>2000</broadcast-period>  4
       <connector-ref>netty-connector</connector-ref>  5
      </broadcast-group>
    </broadcast-groups>
  </core>
</configuration>
```

To use JGroups to broadcast, you must specify the following:

❶ You must use the **name** attribute to give the broadcast group a unique name.

❷ 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.

❸ The name of the JGroups channel to connect to for broadcasting.

❹ The period in milliseconds between consecutive broadcasts. This parameter is optional; the default value is **2000** milliseconds.

❺ The **connector-ref** declares a reference to a previously configured connector that is transmitted by the broadcast (see Network Connections: Acceptors and Connectors for more information.

Related Information

See **clustered-jgroups** under **INSTALL_DIR/examples/features/clustered** for a working example of a **broadcast-group** that uses JGroups.

20.2. ABOUT DISCOVERY GROUPS

While the broadcast group defines how cluster-related information is transmitted, a discovery group defines how connector information is received. Discovery groups maintain a list of connectors—one entry for each broker. As it receives broadcasts from a broker, it updates its entry. If it has not received a broadcast from a broker for a length of time, it will remove the entry.

Discovery groups are typically used in two places:

- By cluster connections so they know how to obtain an initial connection to download the topology.

- By messaging clients so they know how to obtain an initial connection to download the topology.

Although a discovery group always accepts broadcasts, its current list of available live and backup brokers is used only when an initial connection is made. After that point, broker discovery is performed over the normal broker connections.

20.2.1. Configuring a Discovery Group to Use UDP

Define discovery groups in the **BROKER_INSTANCE_DIR/etc/broker.xml** configuration file. You can define multiple discovery groups.

Procedure

To create a discovery group that uses UDP to communicate, add configuration to **BROKER_INSTANCE_DIR/etc/broker.xml** configuration file. Below is an example, followed by a description of each configuration element.

```
<configuration>
  <core>
    ...
    <discovery-groups>
      <discovery-group name="my-discovery-group">  ❶
        <local-bind-address>172.16.9.7</local-bind-address>  ❷
        <group-address>231.7.7.7</group-address>  ❸
```

```
            <group-port>9876</group-port>  4
            <refresh-timeout>10000</refresh-timeout>  5
         </discovery-group>
      </discovery-groups>
      ...
   </core>
</configuration>
```

1 You must use the **name** attribute to give the discovery group a unique name.

2 Use **local-bind-address** to specify that this discovery group only listens to a specific interface. This is a UDP specific attribute.

3 The **group-address** is the multicast IP address of the group on which to listen. It should match the **group-address** in the broadcast group from which you want to listen. This parameter is mandatory. This is a UDP specific attribute.

4 The **group-port** is UDP port of the multicast group. It should match the **group-port** in the broadcast group from which you wish to listen. This parameter is mandatory. This is a UDP specific attribute.

5 The **refresh-timeout** is the period the discovery group waits after receiving the last broadcast from a particular broker before removing that broker's connector pair entry from its list. You would normally set this to a value significantly higher than the **broadcast-period** on the broadcast group, or - due to a slight difference in timing - brokers might intermittently disappear from the list even though they are still broadcasting. This parameter is optional. The default value is **10000** milliseconds (10 seconds).

Related Information

See the **clustered-queue** example under **INSTALL_DIR/examples/features/clustered** for a working example of a **discovery-group** that uses UDP.

20.2.2. Configuring a Discovery Group to Use JGroups

Sometimes using UDP is not an option due to constraints put on the network. In these situations, you can configure the broadcast group to communicate using JGroups as an alternative to UDP. Define discovery groups in the **BROKER_INSTANCE_DIR/etc/broker.xml** configuration file. You can define multiple discovery groups.

Prerequisites

JGroups communication requires a separate configuration file. See **clustered-jgroups** under **INSTALL_DIR/examples/features/clustered** for an example JGroups configuration file.

Procedure

To create a discovery group that uses JGroups, add configuration to **BROKER_INSTANCE_DIR/etc/broker.xml**. Below is an example, followed by a description of each configuration element. To receive broadcasts from JGroups channels, you must specify each of the elements listed.

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

```
        <discovery-groups>
          <discovery-group name="my-discovery-group"> 1
              <jgroups-file>test-jgroups-file_ping.xml</jgroups-file> 2
              <jgroups-channel>activemq_broadcast_channel</jgroups-channel> 3
              <refresh-timeout>10000</refresh-timeout> 4
          </discovery-group>
        </discovery-groups>
        ...
      </core>
  </configuration>
```

1 You must use the **name** attribute to give the discovery group a unique name.

2 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.

3 The name of the JGroups channel to connect to for broadcasting.

4 The **refresh-timeout** is the period the discovery group waits after receiving the last broadcast from a particular broker before removing that broker's connector pair entry from its list. You would normally set this to a value significantly higher than the **broadcast-period** on the broadcast group, or - due to a slight difference in timing - brokers might intermittently disappear from the list even though they are still broadcasting. This parameter is optional. The default value is **10000** milliseconds (10 seconds).

Related Information

See **clustered-jgroups** under **INSTALL_DIR/examples/features/clustered** for a working example of a **discovery-group** that uses JGroups.

20.3. ABOUT CLUSTER CONNECTIONS

Cluster connections group brokers into clusters so that messages can be load balanced between the brokers in the cluster. You can configure multiple cluster connections for each broker.

Each cluster connection only applies to addresses that match its assigned address. The assigned address can be any value, and you can have many cluster connections with different addresses, simultaneously balancing messages for those addresses, potentially to different clusters of brokers. By having multiple cluster connections on different addresses, a single broker can effectively take part in multiple clusters simultaneously.

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**

However, you should not to have multiple cluster connections with overlapping addresses, for example, "europe" and "europe.news", because the same messages could then be distributed between more than one cluster connection, possibly resulting in duplicate deliveries.

20.3.1. Configuring a Cluster Connection

Configure a cluster connections by adding a **cluster-connection** element to **BROKER_INSTANCE_DIR/etc/broker.xml**.

Prerequisites

1. A **connector** for other brokers in the cluster to use when communicating to this broker. The configuration for the connector will be sent to other brokers in the cluster during discovery. The default configuration includes several connectors already, or you can create a new one. See About Connectors for more information.

2. A **discovery-group** for this broker to use while making initial connections to the cluster. See About Discovery Groups in this chapter for more information.

Procedure

To create a cluster connection, add configuration to **BROKER_INSTANCE_DIR/etc/broker.xml**. Below is an example, followed by a description of each configuration element. The example below is the minimal configuration required for a cluster connection.

```
<configuration>
  <core>
    ...
    <cluster-connections>  1
      <cluster-connection name="my-cluster">  2
        <connector-ref>netty-connector</connector-ref>  3
        <discovery-group-ref discovery-group-name="my-discovery-group"/>
  4
        ...
      </cluster-connection>
      ...
    </cluster-connections>
  </core>
</configuration>
```

1 You place each **cluster-connection** element under the parent element, **cluster-connections**.

2 A **cluster-connection** must be given a unique name.

3 The **connector-ref** points to a previously configured **connector** that will have its information broadcast to the cluster. This enables other brokers in the cluster to connect to this broker.

4 The **discovery-group-ref** points to a previously configured **discovery-group** that this broker will use to locate other members of the cluster. Alternatively, you can use a static list of brokers.

Related Information

See the table in the appendix for a full list of configuration elements and a description

20.3.2. Specifying a Static List of Cluster Members

Rather than use dynamic discovery to form a cluster, you can configure a static list of connectors that restricts the cluster to a limited set of brokers. Use this static discovery method to form non-symmetrical clusters such as chain clusters or ring clusters.

Prerequisites

A configured cluster connection. See Configuring a Cluster Connection for details on creating a **cluster-connection** within the configuration.

Procedure

To specify members of a cluster explicitly, add configuration to **BROKER_INSTANCE_DIR/etc/broker.xml** as in the following steps.

1. Create a **static-connectors** element within the relevant **cluster-connection**.

```xml
<configuration>
  <core>
    ...
    <cluster-connections>
      <cluster-connection name="my-cluster">
        ...
        <static-connectors></static-connectors>
        ...
      </cluster-connection>
    </cluster-connections>
  </core>
</configuration>
```

2. Add a **connector-ref** element that names the connector to use when creating connections to other brokers in the cluster.
   In the example below, there is a set of two brokers of which one will always be available. If there are other brokers in the cluster, they will be discovered by one of these connectors when an initial connection is made.

```xml
<configuration>
  <core>
    ...
    <cluster-connections>
      <cluster-connection name="my-cluster">
        ...
        <static-connectors>
          <connector-ref>server1-connector</connector-ref>
          <connector-ref>server2-connector</connector-ref>
        </static-connectors>
        ...
      </cluster-connection>
    </cluster-connections>
  </core>
</configuration>
```

Related Information

See `clustered-static-discovery` under *INSTALL_DIR*`/examples/features/clustered` for a working example that uses static discovery.

20.3.3. Configuring a Client to Use Dynamic Discovery

You can configure a Red Hat AMQ Core JMS client to discover a list of brokers when attempting to establish a connection.

Configuring Dynamic Discovery Using JMS
If you are using JNDI on the client to look up your JMS connection factory instances, you can specify these parameters in the JNDI context environment. Typically the parameters are defined in a file named `jndi.properties`. The host and part in the URL for the connection factory should match the `group-address` and `group-port` from the corresponding `broadcast-group` inside broker's `broker.xml` configuration file. Below is an example of a `jndi.properties` file configured to connect to a broker's discovery group.

```
java.naming.factory.initial = ActiveMQInitialContextFactory
connectionFactory.myConnectionFactory=udp://231.7.7.7:9876
```

When this connection factory is downloaded from JNDI by a client application and JMS connections are created from it, those connections will be load-balanced across the list of servers that the discovery group maintains by listening on the multicast address specified in the broker's discovery group configuration.

As an alternative to using JNDI, you can use specify the discovery group parameters directly in your Java code when creating the JMS connection factory. The code below provides an example of how to do this.

```
final String groupAddress = "231.7.7.7";
final int groupPort = 9876;

DiscoveryGroupConfiguration discoveryGroupConfiguration = new
DiscoveryGroupConfiguration();
UDPBroadcastEndpointFactory udpBroadcastEndpointFactory = new
UDPBroadcastEndpointFactory();
udpBroadcastEndpointFactory.setGroupAddress(groupAddress).setGroupPort(gro
upPort);
discoveryGroupConfiguration.setBroadcastEndpointFactory(udpBroadcastEndpoi
ntFactory);

ConnectionFactory jmsConnectionFactory =
ActiveMQJMSClient.createConnectionFactoryWithHA(discoveryGroupConfiguratio
n, JMSFactoryType.CF);

Connection jmsConnection1 = jmsConnectionFactory.createConnection();
Connection jmsConnection2 = jmsConnectionFactory.createConnection();
```

The refresh timeout can be set directly on the `DiscoveryGroupConfiguration` by using the setter method `setRefreshTimeout()`. The default value is 10000 milliseconds.

On first usage, the connection factory will make sure it waits this long since creation before creating the first connection. The default wait time is 10000 milliseconds, but you can change it by passing a new value to `DiscoveryGroupConfiguration.setDiscoveryInitialWaitTimeout()`.

20.3.4. Configuring a Client to Use Static Discovery

Sometimes it may be impossible to use UDP on the network you are using. In this case you can configure a connection with an initial list if possible servers. The list can be just one broker that you know will always be available, or a list of brokers where at least one will be available.

This does not mean that you have to know where all your servers are going to be hosted, you can configure these servers to use the reliable servers to connect to. After they are connected, their connection details will be propagated via the server the client.

Both Red Hat AMQ Core JMS and Java EE JMS clients can use a static list to discover brokers.

Configuring Static Discovery
If you are using JNDI on the client to look up your JMS connection factory instances, you can specify these parameters in the JNDI context environment. Typically the parameters are defined in a file named **jndi.properties**. Below is an example **jndi.properties** file that provides a static list of brokers instead of using dynamic discovery.

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQIniti
alContextFactory
connectionFactory.myConnectionFactory=
(tcp://myhost:61616,tcp://myhost2:61616)
```

When the above connection factory is used by a client, its connections will be load-balanced across the list of brokers defined within the parentheses **()**.

If you are instantiating the JMS connection factory directly, you can specify the connector list explicitly when creating the JMS connection factory, as in the example below.

```
HashMap<String, Object> map = new HashMap<String, Object>();
map.put("host", "myhost");
map.put("port", "61616");
TransportConfiguration broker1 = new
TransportConfiguration(NettyConnectorFactory.class.getName(), map);

HashMap<String, Object> map2 = new HashMap<String, Object>();
map2.put("host", "myhost2");
map2.put("port", "61617");
TransportConfiguration broker2 = new
TransportConfiguration(NettyConnectorFactory.class.getName(), map2);

ActiveMQConnectionFactory cf =
ActiveMQJMSClient.createConnectionFactoryWithHA(JMSFactoryType.CF,
broker1, broker2);
```

20.4. ENABLING MESSAGE REDISTRIBUTION

Cluster connections allow brokers to load balance their messages. For example, consider a cluster of four brokers A, B, C, and D. Each broker contains configuration for a cluster connection for a queue named **OrderQueue**. When a client connects to broker A, the broker will forward messages sent to **OrderQueue** to brokers B, C, and D. The broker distributes the messages in a round-robin fashion. The exact order depends on the order in which the brokers started. Also, you can configure a cluster connection to load balance messages only to other brokers that have a matching consumer.

You can configure the broker to automatically redistribute messages from queues that do not have any consumers to queues that do have consumers.

If message load balancing is **OFF** or **ON_DEMAND**, messages are not moved to queues that do not have consumers to consume them. However, if a matching consumer on a queue closes after the messages have been sent to the queue, the messages will stay in the queue without being consumed. This scenario is called *starvation*, and message redistribution can be used to move these messages to queues with matching consumers.

By default, message redistribution is disabled, but you can enable it for an address, and can configure the redistribution delay to define how often the messages should be redistributed.

Prerequisites

A configured cluster connection. See Configuring a Cluster Connection for details on creating a **cluster-connection** within the configuration.

Procedure

To enable message redistribution, add the following configuration to the **BROKER_INSTANCE_DIR/etc/broker.xml** configuration file:

1. Verify that the **cluster-connection** has its **message-load-balancing** element set to **ON_DEMAND**. Add this element if it does not exist.

   ```
   <configuration>
     <core>
       ...
       <cluster-connections>
         <cluster-connection name="my-cluster">
           ...
           <message-load-balancing>ON_DEMAND</message-load-balancing>
           ...
         </cluster-connection>
       </cluster-connections>
     </core>
   </configuration>
   ```

2. Enable message redistribution for the relevant queue by configuring an **address-setting** for it.

   a. First, add the parent element, **address-settings**, if it does not exist. This element contains all **address-setting** elements created for the broker.

      ```
      <configuration>
        <core>
          ...
          <address-settings></address-settings>
          ...
        </core>
      </configuration>
      ```

   b. Next add an **address-setting** for the queue. The value for the **match** attribute must a match for the name of the queue. In the example below, an address setting is created for the queue named **my.queue**. You can use the broker wildcard syntax instead of a literal match.

```
<configuration>
  <core>
    ...
    <address-settings>
        <address-setting match="my.queue"></address-setting>
    </address-settings>
    ...
  </core>
</configuration>
```

c. Finally add the **redistribution-delay** element beneath the **address-setting**. In
most cases, you should set a delay before redistributing, because it is common for a
consumer to close but another one to be quickly created on the same queue. A value of **0**
means the messages will be immediately redistributed. A value of **-1**, which is the default,
signifies that messages will never be redistributed.

```
<configuration>
  <core>
    ...
    <address-settings>
        <address-setting match="my.queue">
            <redistribution-delay>0</redistribution-delay>
        </address-setting>
    </address-settings>
    ...
  </core>
</configuration>
```

Related Information

See **queue-message-redistribution** under **INSTALL_DIR/examples/features/clustered**
for a working example of message redistribution.

20.5. CHANGING THE DEFAULT CLUSTER USER AND PASSWORD

When creating connections between brokers of a cluster to form a cluster connection, the broker uses a
cluster user and cluster password. You should change these values from their default, or remote clients
will be able to make connections to the broker using the default values. The broker will also detect the
default credentials when it starts, and display a warning.

Procedure

To change the cluster user and password, modify the **BROKER_INSTANCE_DIR/etc/broker.xml**
configuration file as in the steps below.

1. Under **core**, add the **cluster-user** and **cluster-password** elements.

```
<configuration>
  <core>
    ...
    <cluster-user></cluster-user>
    <cluster-password></cluster-password>
    ...
  </core>
</configuration>
```

2. Add the desired values for **cluster-user** and **cluster-password**.

```
<configuration>
  <core>
    ...
    <cluster-user>cluster_user</cluster-user>
    <cluster-password>cluster_user_password</cluster-password>
    ...
  </core>
</configuration>
```

20.6. USING CLIENT-SIDE LOAD BALANCING

Client-side load balancing distributes client sessions created from a single connection factory to more than one broker. By connecting to different brokers, load-balanced sessions share their workloads across the cluster.

Different strategies, or policies, determine the way load-balanced sessions are distributed to brokers. As a convenience, AMQ Broker includes four Java classes, found in the package **org.apache.activemq.artemis.api.core.client.loadbalance**, that implement four common load balancing policies.

Table 20.1. Load Balancing Policies Included with AMQ Broker

If you want this kind of load balancing policy…	Use this class…
Round Robin. The default policy. The first connection goes to a randomly chosen broker. Subsequent connections go to the next broker in sequence. For example, brokers can be chosen in any of the following sequences: • A, B, C, D, A, … • B, C, D, A, B, … • C, D, A, B, C, … • D, A, B, C, D, …	RoundRobinConnectionLoadBalancingPolicy
Random. Each connection goes to a randomly chosen broker.	RandomConnectionLoadBalancingPolicy
Random Sticky. The first connection goes to a randomly chosen broker. Subsequent connections go to the same broker.	RandomStickyConnectionLoadBalancingPolicy
First Element. Connections always go to the first broker in the cluster.	FirstElementConnectionLoadBalancingPolicy

To use client-side load balancing, configure a connection factory with the *fully qualified name* of the class that implements the policy you want to use. It is recommended that you use one of the four classes from

the package **org.apache.activemq.artemis.api.core.client.loadbalance** that are included with AMQ Broker. However, you can use your own policy by developing a Java class that implements the interface **org.apache.activemq.artemis.api.core.client.loadbalance.ConnectionLoadBalancingPolicy**. You have two choices for how to configure a connection factory to use client-side load balancing:

- As part of your JNDI context environment (recommended).

- As part of your AMQ Core Protocol JMS client code.

20.6.1. Setting the Load Balancing Policy By Using JNDI

You can use a JNDI context environment to configure a connection factory with a load balancing policy. Using JNDI is the recommended method because the context environment can be maintained in a text file that is external to your source code. Consequently, you do not need to recompile your client application after making changes to your load balancing policy.

Procedure

1. Open the file named **jndi.properties** that contains configuration for the connection factory used by your clients.

2. Append a name-value pair to the URL used by the connection factory when connecting to the broker. For the name, specify **connectionLoadBalancingPolicyClassName**, and for the value specify the *fully qualified name* of the class implementing the load balancing policy that you want to use:

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

connection.myConnectionFactory=tcp://localhost:61616?
connectionLoadBalancingPolicyClassName=org.apache.activemq.artemis.a
pi.core.client.loadbalance.RandomConnectionLoadBalancingPolicy
```

20.6.2. Setting the Load Balancing Policy Programmatically

You can use the AMQ Core Protocol JMS client API to programmatically set the client-side load balancing policy used by a connection factory. A disadvantage to using the API is that you must recompile your client code before any configuration changes can take effect. It is recommended that you use JNDI to configure client-side load balancing instead. For more information, see Section 20.6.1, "Setting the Load Balancing Policy By Using JNDI".

Procedure

1. Open the source file that contains the instance of **ActiveMQConnectionFactory** used by your AMQ Core Protocol JMS clients.

2. Add a method call to **setConnectionLoadBalancingPolicyClassName()** *before* the client code attempts a connection to the cluster. The method parameter must be the *fully qualified name* of the class implementing the load balancing policy that you want to use:

```
ConnectionFactory jmsConnectionFactory =
ActiveMQJMSClient.createConnectionFactoryWithHA(...)
```

```
jmsConnectionFactory.setConnectionLoadBalancingPolicyClassName("org.
apache.activemq.artemis.api.core.client.loadbalance.RandomConnection
LoadBalancingPolicy");
```

20.7. CONFIGURING CLUSTER CONNECTIONS FOR USE IN VARIOUS TOPOLOGIES

Broker clusters can be connected together in many different topologies. However, symmetric and chain clusters are the most common. You can also scale clusters up and down without message loss.

20.7.1. Symmetric Clusters

With a symmetric cluster, every broker in the cluster is connected to every other broker in the cluster. This means that every broker in the cluster is no more than one hop away from every other broker.

To form a symmetric cluster, every broker in the cluster defines a cluster connection with the attribute **max-hops** set to **1**. Typically, the cluster connection will use broker discovery in order to know what other brokers in the cluster it should connect to, although it is possible to explicitly define each target broker too in the cluster connection if, for example, UDP is not available on your network.

With a symmetric cluster, each broker knows about all the queues that exist on all of the other brokers, and what consumers they have. With this knowledge, it can determine how to load balance and redistribute messages around the brokers.

20.7.2. Chain Clusters

With 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.

An example of a chain cluster would be a three broker chain consisting of brokers A, B and C. broker A is hosted in one network and has many producer clients connected to it sending order messages. Due to corporate policy, the order consumer clients need to be hosted in a different network, and that network is only accessible via a third network. In this setup, broker B acts as a mediator with no producers or consumers on it. Any messages arriving on broker A will be forwarded to broker B, which in turn forwards them to broker C where they can be consumed. broker A does not need to directly connect to C, but all of the brokers can still act as a part of the cluster.

To set up a cluster in this way, broker A would define a cluster connection that connects to broker B, and broker B would define a cluster connection that connects to broker C. In this case, the cluster connections only need to be in one direction, because messages are moving from broker A→B→C and never from C→B→A.

For this topology, you would set **max-hops** to **2**. With a value of **2**, the knowledge of what queues and consumers that exist on broker C would be propagated from broker C to broker B to broker A. broker A would then know to distribute messages to broker B when they arrive, even though broker B has no consumers itself. It would know that a further hop away is broker C, which does have consumers.

20.7.3. Scaling Clusters

If the size of a cluster changes frequently in your environment, you can scale it up or down with no message loss (even for non-durable messages).

You can scale up clusters by adding brokers, in which case there is no risk of message loss. Similarly, you can scale clusters down by removing brokers. However, to prevent message loss, you must first configure the broker to send its messages to another broker in the cluster.

To scale down a cluster:

1. On the broker you want to scale down, open the **BROKER_INSTANCE_DIR/etc/broker.xml** file and set **enabled** to true.

2. If necessary, set **group-name** to the name of the group that contains the broker to which you want the messages to be sent.
   If cluster brokers are grouped together with different **group-name** values, be careful how you set this parameter. If all of the brokers in a single group are shut down, then the messages from that broker or group will be lost.

3. If the broker is using multiple cluster connections, then set **scale-down-clustername** to identify the name of the **cluster-connection** which should be used for scaling down.

4. Shut down the broker gracefully using the **BROKER_INSTANCE_DIR/artemis stop** command. The broker finds another broker in the cluster and sends all of its messages (both durable and non-durable) to that broker. The messages are processed in order and go to the back of the respective queues on the other broker (just as if the messages were sent from an external client for the first time).

CHAPTER 21. HIGH AVAILABILITY

AMQ Broker enables you to connect brokers as *master and slave*, where each master broker can have one or more slave brokers. Normally, only master brokers actively serve client requests. However, if a master broker and its clients are no longer able to communicate, for example, due to a power outage or similar failure on the master broker, a slave broker takes over for the master. This is referred to as *failover*. When a failover occurs, the slave broker is activated, clients reconnect to the slave broker and resume handling requests.

You can choose from two main high-availability (HA) policies. Each has different benefits and trade-offs:

- *Replication* continuously synchronizes the data from the master broker to the slave broker over the network.

- *Shared-store* uses a shared file system to store journal data for both master and slave brokers.

You can collocate master and slave brokers within the same Java runtime when using either the replication or shared-store policy.

There is also a third policy, called HA *live-only*, that provides a limited amount of HA when scaling down master brokers. This policy is useful during controlled shutdowns.

> **NOTE**
>
> Only persistent message data survives a failover. Any non persistent message data is not be available after failover. See Persisting Messages for more information about how to persist your messages.

21.1. REPLICATION AND HIGH AVAILABILITY

When using replication as the HA policy for your cluster, all data synchronization occurs over the network. A slave broker first needs to synchronize all existing data from the master broker before becoming capable of replacing it. The time it takes for this to happen depends on the amount of data to be synchronized and the connection speed. In general, synchronization occurs in parallel with current network traffic, so this does not cause any blocking on clients. However, synchronization blocks journal-related operations. The maximum length of time that this exchange blocks journal-related operations is controlled by the **initial-replication-sync-timeout** configuration element. There is a critical moment at the end of this process where the master broker must complete the synchronization and ensure the slave broker acknowledges the completion. After the initial synchronization phase is complete and the slave broker has all the current data from the master, then the master synchronizes data across the network as it receives it, to ensure that no data is lost if the master broker fails.

Figure 21.1. Replicated Store for High Availability
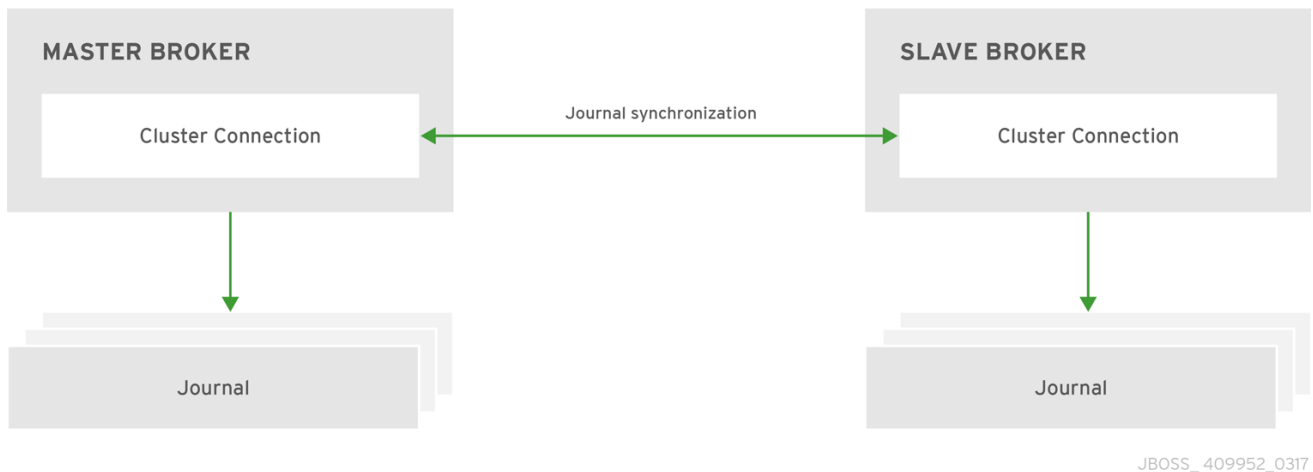
The advantage of using replication is that it doesn't require a shared file system that is accessible by both the master and slave broker. The disadvantage is that performance can be reduced while data is replicated between a master broker and a slave broker.

The replicating master and slave pair must be part of a cluster. The cluster connection also defines how a slave broker identifies a master broker to which it can connect. See Clusters for details on how to configure a cluster connection.

> **IMPORTANT**
>
> Using replication across data centers is not currently recommended. In addition, your cluster should contain three or more master-slave pairs if you are using replication. This is to avoid the scenario sometimes referred to as "split-brain."

Within a cluster using data replication, there are two ways that a slave broker locates a master broker:

- *Connect to a group*. A slave broker can be configured to connect only to a master broker that shares the same broker `group-name`.

- *Connect to any live*. The behavior if `group-name` is not configured. Slave brokers are free to connect to any master broker.

The slave searches for any master broker that it is configured to connect to. It then tries to replicate with each master broker in turn until it finds a master broker that has no current slave configured. If no master broker is available it waits until the cluster topology changes and repeats the process.

> **NOTE**
>
> The slave broker does not know whether any data it might have is up to date, so it really cannot decide to activate automatically. To activate a replicating slave broker using the data it has, the administrator must change its configuration to make it a master broker.

When the master broker stops or crashes, its replicating slave becomes active and take over its duties. Specifically, the slave becomes active when it loses connection to its master broker. This can be problematic because a connection loss might be due to a temporary network problem. In order to address this issue, the slave tries to determine whether it can connect to the other brokers in the cluster. If it can connect to more than half the brokers, it becomes active. If it can connect to fewer than half, the slave does not become active but tries to reconnect with the master. This avoids a split-brain situation.

21.1.1. Configuring Replication

You configure brokers for replication by editing the **broker.xml** configuration file. The default configuration values cover most use cases, making it easy to start using replication. You can also supply your own configuration for these values when needed, however. The appendix includes a table of the configuration elements you can add to **broker.xml** when using replication HA.

Prerequisites

Master and slave brokers must form a cluster and use a **cluster-connection** to communicate. See Clustering for more information on cluster connections.

Procedure

Configure a cluster of brokers to use the replication HA policy by modifying the main configuration file, **BROKER_INSTANCE_DIR/etc/broker.xml**.

1. Configure the master broker to use replication for its HA policy.

   ```
   <configuration>
     <core>
       ...
       <ha-policy>
          <replication>
              <master/>
          </replication>
       </ha-policy>
       ...
     </core>
   </configuration>
   ```

2. Configure the slave brokers in the same way, but use the **slave** element instead of **master** to denote their role in the cluster.

   ```
   <configuration>
     <core>
       ...
       <ha-policy>
          <replication>
              <slave/>
          </replication>
       </ha-policy>
       ...
     </core>
   </configuration>
   ```

Related Information

- See the appendix for a table of the configuration elements available when configuring master and slave brokers for replication.

- For working examples demonstrating replication HA see the example Maven projects located under **INSTALL_DIR/examples/features/ha**.

21.1.2. Failing Back to the Master Broker

After a master broker has failed and a slave has taken over its duties, you might want to restart the master broker and have clients fail back to it.

In replication HA mode, you can configure a master broker so that at startup it searches the cluster for another broker using the same cluster node ID. If it finds one, the master attempts to synchronize its data with it. Once the data is synchronized, the master requests that the other broker shut down. The master broker then resumes its active role within the cluster.

Prerequisites

Configuring a master broker to fail back as described above requires the replication HA policy.

Procedure

To configure brokers to fail back to the original master, edit the **BROKER_INSTANCE_DIR/etc/broker.xml** configuration file for the master and slave brokers as follows.

1. Add the **check-for-live-server** element and set its value to **true** to tell this broker to check if a slave has assumed the role of master.

```
<configuration>
  <core>
    ...
    <ha-policy>
       <replication>
          <master>
             <check-for-live-server>true</check-for-live-server>
             ...
          </master>
       </replication>
    </ha-policy>
    ...
  </core>
</configuration>
```

2. Add the **allow-failback** element to the slave broker(s) and set its value to **true** so that the slave fails back to the original master.

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

> **⚠ WARNING**
>
> Be aware that if you restart a master broker after failover has occurred, then the value for **check-for-live-server** must be set to **true**. Otherwise, the master broker restarts and process the same messages that the slave has already handled, causing duplicates.

21.1.3. Grouping Master and Slave Brokers

You can specify a group of master brokers that a slave broker can connect to. This is done by adding a **group-name** configuration element to *BROKER_INSTANCE_DIR/etc/broker.xml*. A slave broker connects only to a master broker that shares the same **group-name**.

As an example of using **group-name**, suppose you have five master brokers and six slave brokers. You could divide the brokers into two groups.

- Master brokers **master1**, **master2**, and **master3** use a **group-name** of **fish**, while **master4** and **master5** use **bird**.

- Slave brokers **slave1**, **slave2**, **slave3**, and **slave4** use **fish** for their **group-name**, and **slave5** and **slave6** use **bird**.

After joining the cluster, each slave with a **group-name** of **fish** searches for a master broker also assigned to **fish**. Since there is one slave too many, the group has one spare slave that remains un-paired. Meanwhile, each slave assigned to **bird** pairs with one of the master brokers in their group, **master4** or **master5**.

Prerequisites

Grouping brokers into HA groups requires you configure the brokers to use the replication HA policy.

Configuring a Broker Cluster to Use Groups
Configure a cluster of brokers to form groups of master and slave brokers by modifying the main configuration file, *BROKER_INSTANCE_DIR/etc/broker.xml*.

Procedure

1. Configure the master broker to use the chosen **group-name** by adding it beneath the **master** configuration element. In the example below the master broker is assigned the group name **fish**.

```xml
<configuration>
  <core>
    ...
    <ha-policy>
      <replication>
        <master>
          <group-name>fish</group-name>
          ...
        </master>
      </replication>
```

```
      </ha-policy>
      ...
   </core>
</configuration>
```

2. Configure the slave broker(s) in the same way, by adding the **group-name** element under **slave**.

```
<configuration>
  <core>
    ...
    <ha-policy>
      <replication>
        <slave>
          <group-name>fish</group-name>
          ...
        </slave>
      </replication>
    </ha-policy>
    ...
  </core>
</configuration>
```

21.2. SHARED-STORE AND HIGH AVAILABILITY

When using a shared-store high availability policy, both master and slave brokers access a single directory on a shared file system that is accessible by both the master and slave nodes. Typically, this is some kind of Storage Area Network (SAN), which uses a dedicated high-performance network to provide block-level storage. In general, Network Attached Storage (NAS) is not recommended in scenarios where optimal throughput is required. Depending on the network's configuration, NAS performance can be slower when compared to SAN.

If a master broker encounters a failure, a slave broker loads the data from the shared file system and takes over for the failed master. Clients can then reconnect to the slave broker and resume handling requests.

The advantage of using shared-store high availability is that there is no performance overhead incurred for replicating data between a master broker and a slave broker. Instead, a master and a slave broker read and write data to and from the shared store independently.

The disadvantage of shared-store replication is that it requires a shared file system. And when the slave broker takes over, it needs to load the journal from the shared-store, which can take some time, depending on the amount of data.

Figure 21.2. shared-store for High Availability

JBOSS_409952_0317

21.2.1. Configuring a shared-store

Prerequisites

1. For master-slave pairs to operate properly using shared-store, both brokers must use the *same* location to store persistent message data. See Persistence for more information.

2. Master and slave brokers must be part of cluster and use a **cluster-connection** to communicate. See Clustering for more information on cluster connections.

Procedure

Configure a cluster of brokers to use the shared-store HA policy by modifying the main configuration file, *BROKER_INSTANCE_DIR*/etc/broker.xml.

1. Configure the master broker to use shared-store for its HA policy:

```xml
<configuration>
  <core>
    ...
    <ha-policy>
       <shared-store>
          <master/>
       </shared-store>
    </ha-policy>
    ...
  </core>
</configuration>
```

2. Enable failover on the master by adding the **failover-on-shutdown** element and setting its value to **true**, as in the example below.

```xml
<configuration>
  <core>
    ...
    <ha-policy>
       <shared-store>
          <master>
             <failover-on-shutdown>true</failover-on-shutdown>
          </master>
       </shared-store>
    </ha-policy>
    ...
  </core>
</configuration>
```

3. Configure each slave broker using the **slave** element to denote their role in the cluster.

```xml
<configuration>
  <core>
    ...
    <ha-policy>
       <shared-store>
          <slave/>
       </shared-store>
```

```
      </ha-policy>
      ...
    </core>
</configuration>
```

4. Configure *one* slave broker to take over client requests if the master broker is shut down by adding the **failover-on-shutdown** element, as in the example below.

```
<configuration>
  <core>
    ...
    <ha-policy>
      <shared-store>
        <slave>
          <failover-on-shutdown>true</failover-on-shutdown>
        </slave>
      </shared-store>
    </ha-policy>
    ...
  </core>
</configuration>
```

Related Information

- The following table lists the valid configuration elements when using a shared-store policy. Note that there are only two elements and that they have default values. Therefore, you do not need to include these elements unless you want to change their default value.

Table 21.1. Configuration Elements Available when Using shared-store

Name	Description
failover-on-shutdown	For a master broker, determines whether to fail over to a slave if it is shut down. For a slave broker, this element determines whether it becomes the master in case the master is shut down. The default value for this element is **false**.
allow-failback	Whether a slave broker automatically restarts and resume its original role after the master broker returns to the cluster after disconnecting and requests to resume its role. The default is **true**.

21.2.2. Failing Back to the Master Broker

After a master broker has failed and a slave has taken over its duties, you might want to restart the master broker and have clients fail back to it. When using a shared-store, you only need to restart the original master broker and kill the slave broker.

Alternatively, you can set **allow-failback** to **true** on the slave configuration, which forces a slave that has become master to automatically stop.

Procedure

- In each slave broker, add the **allow-failback** configuration element and set its value to **true**, as in the example below.

```
<configuration>
  <core>
    ...
    <ha-policy>
      <shared-store>
        <slave>
          <allow-failback>true</allow-failback>
          ...
        </slave>
      </shared-store>
    </ha-policy>
    ...
  </core>
</configuration>
```

21.3. COLOCATING SLAVE BROKERS

It is also possible to colocate slave brokers in the same JVM as a master broker. A master broker can be configured to request another master to start a slave broker that resides in its Java Virtual Machine. You can colocate slave brokers using either shared-store or replication as your HA policy. The new slave broker inherits its configuration from the master broker creating it. The name of the slave is set to **colocated_backup_n** where **n** is the number of backups the master broker has created.

The slave inherits configuration for its connectors and acceptors from the master broker creating it. However, AMQ Broker applies a default port offset of 100 for each. For example, if the master contains configuration for a connection that uses port 61616, the first slave created uses port 61716, the second uses 61816, and so on.

> **NOTE**
>
> For In-VM connectors and acceptors the ID has **colocated_backup_n** appended, where **n** is the slave broker number.

Directories for the journal, large messages, and paging are set according to the HA strategy 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.

Figure 21.3. Co-located Master and Slave Brokers
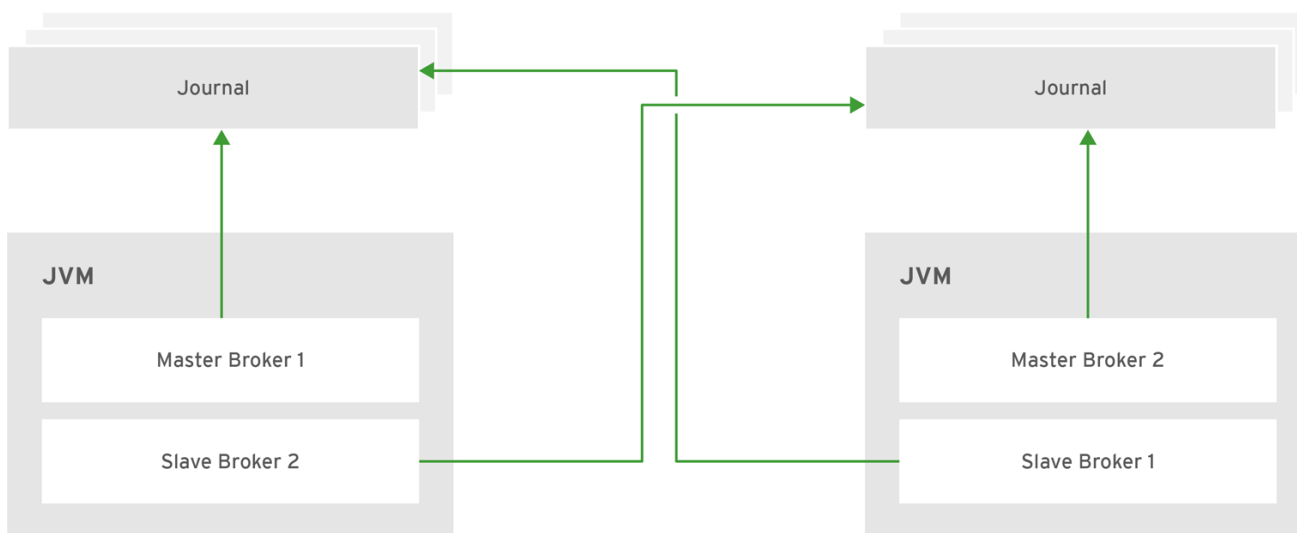
JBOSS_ 409952_0317

21.3.1. Configuring Colocated Slaves

A master broker can also be configured to allow requests from backups and also how many backups a master broker can start. This way you can evenly distribute backups around the cluster. This is configured under the **ha-policy** element in the ***BROKER_INSTANCE_DIR*/etc/broker.xml** file.

Prerequisites

You must configure a master broker to use either replication or shared-store as its HA policy.

Procedure

- After choosing an HA policy, add configuration for the colocation of master and slave broker. The example below uses each of the configuration options available and gives a description for each after the example. Some elements have a default value and therefore do not need to be explicitly added to the configuration unless you want to use your own value. Note that this example uses **replication** but you can use a **shared-store** for your **ha-policy** as well.

```
<configuration>
  <core>
    ...
    <ha-policy>
      <replication>
        <colocated>    1
          <request-backup>true</request-backup>    2
          <max-backups>1</max-backups>    3
          <backup-request-retries>-1</backup-request-retries>    4
          <backup-request-retry-interval>5000</backup-request-
retry-interval/>    5
          <backup-port-offset>150</backup-port-offset>    6
          <master>
            ...    7
          </master>
          <slave>
            ...    8
```

```
            </slave>
          </colocated>
        <replication>
      </ha-policy>
     </core>
  </configuration>
```

1 You add the **colocated** element directly underneath the choice of **ha-policy**. In the example above, **colocated** appears after **replication**. The rest of the configuration falls under this element.

2 Use **request-backup** to determine whether this broker requests a slave on another broker in the cluster. The default is **false**.

3 Use **max-backups** to determine how many backups a master broker can create. Set to **0** to stop this live broker from accepting backup requests from other live brokers. The default is **1**.

4 Setting **backup-request-retries** defines how many times the master broker tries to request a slave. The default is **-1**, which means unlimited tries.

5 The broker waits this long in milliseconds before retrying a request for a slave broker. The default value for **backup-request-retry-interval** is **5000**, or 5 seconds.

6 The port offset to use for the connectors and acceptors for a new slave broker. The default is **100**.

7 The master broker is configured according to the **ha-policy** you chose, **replication** or **shared-store**.

8 Like the master, the slave broker adheres to the configuration of the chosen **ha-policy**.

Related Information

- For working examples that demonstrate colocation see the colocation example Maven projects located under *INSTALL_DIR*/examples/features/ha.

21.3.2. Excluding Connectors

Sometimes some of the connectors you configure are for external brokers and should be excluded from the offset. For instance, you might have a connector used by the cluster connection to do quorum voting for a replicated slave broker. Use the **excludes** element to identify connectors you do not want offset.

Prerequisites

You must configure a broker for colocation before modifying the configuration to exclude connectors.

Procedure

1. Modify *BROKER_INSTANCE_DIR*/etc/broker.xml by adding the **excludes** configuration element, as in the example below.

```
<configuration>
  <core>
```

```
    ...
    <ha-policy>
       <replication>
          <colocated>
             <excludes>
             </excludes>
             ...
          <colocated>
       </replication>
    </ha-policy>
  </core>
</configuration>
```

2. Add a **connector-ref** element for each connector you want to exclude. In the example below, the connector with the name **remote-connector** is excluded from the connectors inherited by the slave.

```
<configuration>
  <core>
    ...
    <ha-policy>
       <replication>
          <colocated>
             <excludes>
                <connector-ref>remote-connector</connector-ref>
             </excludes>
             ...
          <colocated>
       </replication>
    </ha-policy>
  </core>
</configuration>
```

21.4. USING A LIVE-ONLY POLICY FOR SCALING DOWN BROKERS

You can configure brokers to *scale down* as an alternative to using a replication or shared-store HA policy. When configured for scale down, a master broker copies its messages and transaction state to another master broker before shutting down. The advantage of scale down is that you do not need full backups to provide some form of HA. However, scaling down handles only cases where a broker stops gracefully. It is not made to handle an unexpected failure gracefully.

Another disadvantage is that it is possible to lose message ordering when scaling down. This happens because the messages in the broker that is scaling down are appended to the end of the queues of the other broker. For example, two master brokers have ten messages distributed evenly between them. If one of the brokers scales down, the messages sent to the other broker are added to queue after the ones already there. Consequently, after Broker 2 scales down, the order of the messages in Broker 1 would be 1, 3, 5, 7, 9, 2, 4, 6, 8, 10.

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 these should be high enough to deal with the time needed to scale down.

Figure 21.4. Scaling Down Master Brokers
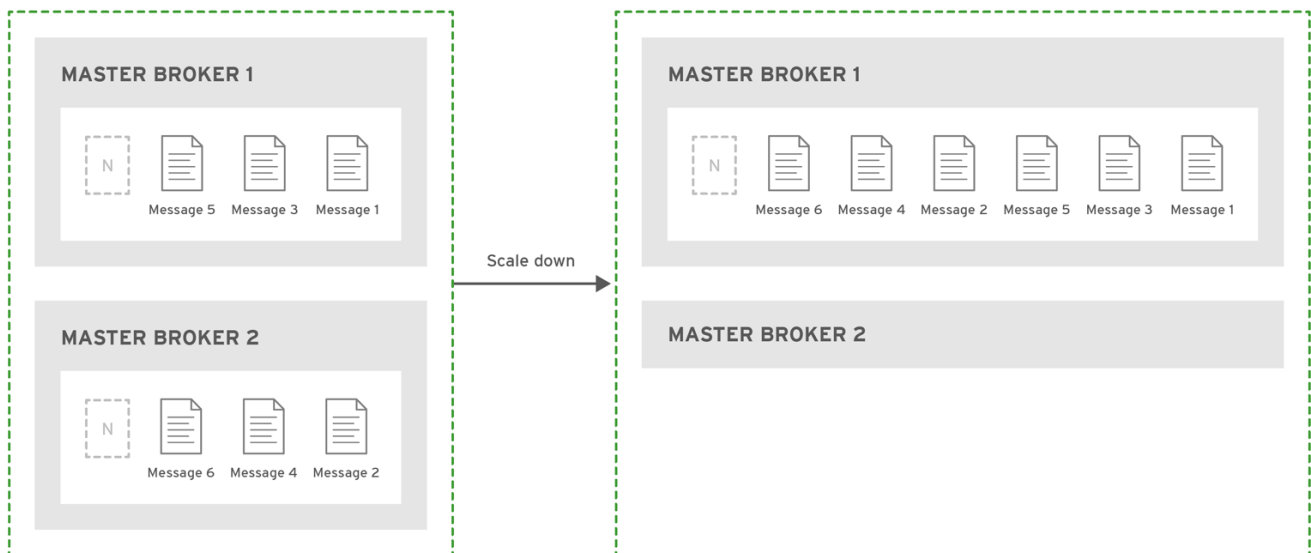
JBOSS_ 409952_0317

21.4.1. Using a Specific Connector when Scaling Down

You can configure a broker to use a specific connector to scale down. If a connector is not specified, the broker uses the first In-VM connector appearing in the configuration.

Prerequisites

Using a static list of brokers during scale down requires that you configure a **connector** to the broker that receives the state of the broker scaling down. See About Connectors for more information.

Procedure

- Configure scale down to a specific broker by adding a **connector-ref** element under the configuration for the **scale-down** in *BROKER_INSTANCE_DIR*/etc/broker.xml, as in the example below.

```xml
<configuration>
  <core>
    ...
    <ha-policy>
      <live-only>
        <scale-down>
          <connectors>
            <connector-ref>server1-connector</connector-ref>
          </connectors>
        </scale-down>
      </live-only>
    </ha-policy>
    ...
  </core>
</configuration>
```

Related Information

- For a working example of scaling down using a static connector that demonstrate colocation see the **scale-down** example Maven project located under
  ***INSTALL_DIR*/examples/features/ha**.

21.4.2. Using Dynamic Discovery

You can use dynamic discovery when configuring the cluster for scale down. Instead of scaling down to a specific broker by using a connector, brokers instead use a discovery group and find another broker dynamically.

Prerequisites

Using dynamic discovery during scale down requires that you configure a **discovery-group**. See About Discovery Groups for more information.

Procedure

- Configure scale down to use a discovery group by adding a **discovery-group-ref** element under the configuration for the **scale-down** in ***BROKER_INSTANCE_DIR*/etc/broker.xml**, as in the example below. Note that **discovery-group-ref** uses the attribute **discovery-group-name** to hold the name of the discovery group to use.

  ```xml
  <configuration>
    <core>
      ...
      <ha-policy>
        <live-only>
            <scale-down>
                <discovery-group-ref discovery-group-name="my-discovery-group"/>
            </scale-down>
        </live-only>
      </ha-policy>
      ...
    </core>
  </configuration>
  ```

21.4.3. Using Broker Groups

It is also possible to configure brokers to scale down only to brokers that are configured with the same group.

Procedure

- Configure scale down for a group of brokers by adding a **group-name** element, and a value for the desired group name, in ***BROKER_INSTANCE_DIR*/etc/broker.xml**.
  In the example below, only brokers that belong to the group **my-group-name** are scaled down.

  ```xml
  <configuration>
    <core>
      ...
      <ha-policy>
        <live-only>
            <scale-down>
  ```

```
            <group-name>my-group-name</group-name>
         </scale-down>
      </live-only>
   </ha-policy>
   ...
 </core>
</configuration>
```

21.4.4. Using Slave Brokers

You can mix scale down with HA and use master and slave brokers. In such a configuration, a slave immediately scales down to another master broker instead of becoming active itself.

Procedure

Edit the master's **broker.xml** to colocate a slave broker that is configured for scale down. Configuration using replication for its HA policy would look like the example below.

```
<configuration>
  <core>
   ...
   <ha-policy>
      <replication>
         <colocated>
            <backup-request-retries>44</backup-request-retries>
            <backup-request-retry-interval>33</backup-request-retry-
interval>
            <max-backups>3</max-backups>
            <request-backup>false</request-backup>
            <backup-port-offset>33</backup-port-offset>
            <master>
               <group-name>purple</group-name>
               <check-for-live-server>true</check-for-live-server>
               <cluster-name>abcdefg</cluster-name>
            </master>
            <slave>
               <group-name>tiddles</group-name>
               <max-saved-replicated-journals-size>22</max-saved-
replicated-journals-size>
               <cluster-name>33rrrrr</cluster-name>
               <restart-backup>false</restart-backup>
               <scale-down>
                  <!--a grouping of servers that can be scaled down to--
>
                  <group-name>boo!</group-name>
                  <!--either a discovery group-->
                  <discovery-group-ref discovery-group-name="wahey"/>
               </scale-down>
            </slave>
         </colocated>
      </replication>
   </ha-policy>
   ...
 </core>
</configuration>
```

CHAPTER 22. CLIENT FAILOVER

AMQ Broker 7.2 defines two types of client failover, each of which is covered in its own section later in this chapter: *automatic client failover* and *application-level client failover*. The broker also provides 100% transparent automatic reattachment of connections to the same broker, as in the case of transient network problems, for example. This is similar to failover, except the client is reconnecting to the same broker.

During failover, if the client has consumers on any non persistent or temporary queues, those queues are automatically re-created during failover on the slave broker, since the slave broker does not have any knowledge of non persistent queues.

22.1. AUTOMATIC CLIENT FAILOVER

A client can receive information about all master and slave brokers, so that in the event of a connection failure, it can reconnect to the slave broker. The slave broker then automatically re-creates any sessions and consumers that existed on each connection before failover. This feature saves you from having to hand-code manual reconnection logic in your applications.

When a session is re-created on the slave, it does not have any knowledge of messages already sent or acknowledged. Any in-flight sends or acknowledgements at the time of failover might also be lost. However, even without 100% transparent failover, it is simple to guarantee *once and only once* delivery, even in the case of failure, by using a combination of duplicate detection and retrying of transactions.

Clients detect connection failure when they have not received packets from the broker within a configurable period of time. See Detecting Dead Connections for more information.

You have a number of methods to configure clients to receive information about master and slave. One option is to configure clients to connect to a specific broker and then receive information about the other brokers in the cluster. See Configuring a Client to Use Static Discovery for more information. The most common way, however, is to use *broker discovery*. For details on how to configure broker discovery, see Configuring a Client to Use Dynamic Discovery.

Also, you can configure the client by adding parameters to the query string of the URL used to connect to the broker, as in the example below.

```
connectionFactory.ConnectionFactory=tcp://localhost:61616?
ha=true&reconnectAttempts=3
```

Procedure

To configure your clients for failover through the use of a query string, ensure the following components of the URL are set properly.

1. The **host:port** portion of the URL should point to a master broker that is properly configured with a backup. This host and port is used only for the initial connection. The **host:port** value has nothing to do with the actual connection failover between a live and a backup server. In the example above, **localhost:61616** is used for the **host:port**.

2. (Optional) To use more than one broker as a possible initial connection, group the **host:port** entries as in the following example:

```
connectionFactory.ConnectionFactory=
(tcp://host1:port,tcp://host2:port)?ha=true&reconnectAttempts=3
```

3. Include the name-value pair **ha=true** as part of the query string to ensure the client receives information about each master and slave broker in the cluster.

4. Include the name-value pair **reconnectAttempts=n**, where **n** is an integer greater than **0**. This parameter sets the number of times the client attempts to reconnect to a broker.

> **NOTE**
>
> Failover occurs only if **ha=true** and **reconnectAttempts** is greater than **0**. Also, the client must make an initial connection to the master broker in order to receive information about other brokers. If the initial connection fails, the client can only retry to establish it. See Failing Over During the Initial Connection for more information.

22.1.1. Failing Over During the Initial Connection

Because the client does not receive information about every broker until after the first connection to the HA cluster, there is a window of time where the client can connect only to the broker included in the connection URL. Therefore, if a failure happens during this initial connection, the client cannot failover to other master brokers, but can only try to re-establish the initial connection. Clients can be configured for set number of reconnection attempts. Once the number of attempts has been made an exception is thrown.

Setting the Number of Reconnection Attempts

Procedure

The examples below shows how to set the number of reconnection attempts to **3** using the AMQ JMS client. The default value is **0**, that is, try only once.

- Set the number of reconnection attempts by passing a value to **ServerLocator.setInitialConnectAttempts()**.

```
ConnectionFactory cf =
ActiveMQJMSClient.createConnectionFactory(...)
cf.setInitialConnectAttempts(3);
```

Setting a Global Number of Reconnection Attempts
Alternatively, you can apply a global value for the maximum number of reconnection attempts within the broker's configuration. The maximum is applied to all client connections.

Procedure

- Edit *BROKER_INSTANCE_DIR*/etc/broker.xml by adding the **initial-connect-attempts** configuration element and providing a value for the time-to-live, as in the example below.

```
<configuration>
 <core>
  ...
  <initial-connect-attempts>3</initial-connect-attempts>  ❶
  ...
 </core>
</configuration>
```

1 All clients connecting to the broker are allowed a maximum of three attempts to reconnect. The default is **-1**, which allows clients unlimited attempts.

22.1.2. Handling Blocking Calls During Failover

When failover occurs and the client is waiting for a response from the broker to continue its execution, the newly created session does not have any knowledge of the call that was in progress. The initial call might otherwise hang forever, waiting for a response that never comes. To prevent this, the broker is designed to unblock any blocking calls that were in progress at the time of failover by making them throw an exception. Client code can catch these exceptions and retry any operations if desired.

When using AMQ JMS clients, if the unblocked method is a call to **commit()** or **prepare()**, the transaction is automatically rolled back and the broker throws an exception.

22.1.3. Handling Failover with Transactions

When using AMQ JMS clients, if the session is transactional and messages have already been sent or acknowledged in the current transaction, the broker cannot be sure that those messages or their acknowledgements were lost during the failover. Consequently, the transaction is marked for rollback only. Any subsequent attempt to commit it throws an **javax.jms.TransactionRolledBackException**.

> **WARNING**
>
> The caveat to this rule is when XA is used. If a two-phase commit is used and **prepare()** has already been called, rolling back could cause a **HeuristicMixedException**. Because of this, the commit throws an **XAException.XA_RETRY** exception, which informs the Transaction Manager it should retry the commit at some later point. If the original commit has not occurred, it still exists and can be committed. If the commit does not exist, it is assumed to have been committed, although the transaction manager might log a warning. A side effect of this exception is that any nonpersistent messages are lost. To avoid such losses, always use persistent messages when using XA. This is not an issue with acknowledgements since they are flushed to the broker before **prepare()** is called.

The AMQ JMS client code must catch the exception and perform any necessary client side rollback. There is no need to roll back the session, however, because it was already rolled back. The user can then retry the transactional operations again on the same session.

If failover occurs when a commit call is being executed, the broker unblocks the call to prevent the AMQ JMS client from waiting indefinitely for a response. Consequently, the client cannot determine whether the transaction commit was actually processed on the master broker before failure occurred.

To remedy this, the AMQ JMS client can enable duplicate detection in the transaction, and retry the transaction operations again after the call is unblocked. If the transaction was successfully committed on the master broker before failover, duplicate detection ensures that any durable messages present in the transaction when it is retried are ignored on the broker side. This prevents messages from being sent more than once.

If the session is non transactional, messages or acknowledgements can be lost in case of failover. If you want to provide *once and only once* delivery guarantees for non transacted sessions, enable duplicate detection and catch unblock exceptions.

22.1.4. Getting Notified of Connection Failure

JMS provides a standard mechanism for getting notified asynchronously of connection failure: `java.jms.ExceptionListener`.

Any **ExceptionListener** or **SessionFailureListener** instance is always called by the broker if a connection failure occurs, whether the connection was successfully failed over, reconnected, or reattached. You can find out if a reconnect or a reattach has happened by examining the **failedOver** flag passed in on the **connectionFailed** on **SessionFailureListener**. Alternatively, you can inspect the error code of the **javax.jms.JMSException**, which can be one of the following:

Table 22.1. JMSException Error Codes

Error code	Description
FAILOVER	Failover has occurred and the broker has successfully reattached or reconnected
DISCONNECT	No failover has occurred and the broker is disconnected

22.2. APPLICATION-LEVEL FAILOVER

In some cases you might not want automatic client failover, but prefer to code your own reconnection logic in a failure handler instead. This is known as *application-level* failover, since the failover is handled at the application level.

To implement application-level failover when using JMS, set an **ExceptionListener** class on the JMS connection. The **ExceptionListener** is called by the broker in the event that a connection failure is detected. In your **ExceptionListener**, you should close your old JMS connections. You might also want to look up new connection factory instances from JNDI and create new connections.

CHAPTER 23. 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.

There are six loggers available, which are configured by the **loggers** key.

```
loggers=org.jboss.logging,org.apache.activemq.artemis.core.server,org.apac
he.activemq.artemis.utils,org.apache.activemq.artemis.journal,org.apache.a
ctivemq.artemis.jms.server,org.apache.activemq.artemis.integration.bootstr
ap
```

Table 23.1. Loggers

Logger	Description
org.jboss.logging	Logs any calls not handled by the Brokers 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

By default there are two loggers configured by default by the **logger.handlers** key.

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

As the names suggest these log to the console and to a file.

23.1. CHANGING THE LOGGING LEVEL

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

```
logger.level=INFO
```

All other loggers specified can be configured individually via the logger name.

```
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
```

NOTE

The root logger configuration will always be the finest logging logged even if the other logs have a finer logging configuration.

Table 23.2. Available Logging Levels

Level	Description
FATAL	Use the FATAL level priority for events that indicate a critical service failure. If a service issues a FATAL error it is completely unable to service requests of any kind.
ERROR	Use the ERROR level priority 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 ERRORs.
WARN	Use the WARN level priority for events that may indicate a non-critical service error. Resumable errors, or minor breaches in request expectations fall into this category. The distinction between WARN and ERROR may be hard to discern and so it is up to the developer to judge. The simplest criterion is would this failure result in a user support call. If it would use ERROR. If it would not use WARN.
INFO	Use the INFO level priority for service life-cycle events and other crucial related information. Looking at the INFO messages for a given service category should tell you exactly what state the service is in.
DEBUG	Use the DEBUG level priority for log messages that convey extra information regarding life-cycle events. Developer or in depth information required for support is the basis for this priority. The important point is that when the DEBUG level priority is enabled, the JBoss server log should not grow proportionally with the number of server requests. Looking at the DEBUG and INFO messages for a given service category should tell you exactly what state the service is in, as well as what server resources it is using: ports, interfaces, log files, and so on.

Level	Description
TRACE	Use TRACE the level priority for log messages that are directly associated with activity that corresponds requests. Further, 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 if the category priority threshold is enabled. The point of the TRACE priority is to allow for deep probing of the JBoss server behavior when necessary. When the TRACE level priority is enabled, you can expect the number of messages in the JBoss server log to grow at least a x N, where N is the number of requests received by the server, a some constant. The server log may well grow as power of N depending on the request-handling layer being traced.

23.2. CONFIGURING CONSOLE LOGGING

Console Logging can be configured via 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.

Table 23.3. Available Console Configuration

Property	Description
name	The handler's name.
encoding	The character encoding used by this Handler.
level	The log level specifying which message levels will be logged by this. Message levels lower than this value will be discarded.
formatter	Defines a formatter. See Section 23.4, "Configuring the Logging Format".
autoflush	Automatically flush after each write.

Property	Description
target	Defines the target of the console handler. The value can either be SYSTEM_OUT or SYSTEM_ERR.

23.3. CONFIGURING FILE LOGGING

File Logging can be configured via 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.

Table 23.4. Available Console Configuration

Property	Description
name	The handler's name.
encoding	The character encoding used by this Handler.
level	The log level specifying which message levels will be logged by this. Message levels lower than this value will be discarded.
formatter	Defines a formatter. See Section 23.4, "Configuring the Logging Format".
autoflush	Automatically flush after each write.
append	Specify whether to append to the target file.
file	The file description consisting of the path and optional relative to path.

23.4. CONFIGURING THE LOGGING FORMAT

The formatter describes how log messages should be shown. 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
```

Where **%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.

23.5. CLIENT OR EMBEDDED SERVER LOGGING

Firstly, if you want to enable logging on the client side you need to include the JBoss logging JARs in your client's class path. If you are using Maven, add the following dependencies:

```
<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 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.proper
ties
```

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.apac
he.activemq.artemis.utils,org.apache.activemq.artemis.journal,org.apache.a
ctivemq.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
```

23.6. AMQ BROKER PLUG-INS SUPPORT

AMQ supports custom plug-ins. You can use plug-ins to log information about many different types of events that would otherwise only be available through debug logs. Multiple plug-ins can be registered, tied, and executed together. The plug-ins will be executed based on the order of the registration, that is, the first plug-in registered is always executed first.

You can create custom plug-ins and implement them using the **ActiveMQServerPlugin** interface. This interface ensures that the plug-in 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.

23.6.1. Adding Plug-ins to the Classpath

Add the custom created broker plug-ins to the broker runtime by adding the relevant jars to the **BROKER_INSTANCE_DIR/lib** directory.

If you are using an embedded system then place the jar under the regular classpath of your embedded application.

23.6.2. Registering a Plug-in

You must register a plug-in by adding the **broker-plugins** element in the **broker.xml** file. You can specify the plug-in configuration value using the **property** child elements. These properties will be read and passed into the plug-in's init (Map<String, String>) operation after the plug-in 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>
```

23.6.3. Registering a Plug-in Programmatically

To register a plug-in programmatically, use the **registerBrokerPlugin()** method and pass in a new instance of your plug-in. The example below shows the registration of the **UserPlugin** plugin:

```
Configuration config = new ConfigurationImpl();

config.registerBrokerPlugin(new UserPlugin());
```

23.6.4. Logging Specific Events

By default, AMQ broker provides the **LoggingActiveMQServerPlugin** plug-in to log specific broker events. The **LoggingActiveMQServerPlug-in** plug-in is commented out by default and does not log any information. The following table provides information about the plug-in properties and its description. Set the 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** 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.LoggingActiveMQS
erverPlugin">
            <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>
```

After changing the configuration parameters inside the **<broker-plugins>** section of the
broker.xml file, you must restart the broker to reload the configuration updates. These configuration
changes will not be reloaded by 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 has occurred. For example, it logs
beforeCreateConsumer() and **afterCreateConsumer()**. If the log Level is set to **DEBUG**, it logs
more information for a notification when available.

CHAPTER 24. MANAGEMENT

AMQ Broker provides both a graphical as well as a programming interface to help you manage your brokers.

24.1. USING AMQ CONSOLE

If you prefer to use a graphic interface to manage AMQ, you can use AMQ Console. AMQ Console is a web console included in the AMQ Broker installation, and it enables you to use a web browser to manage AMQ Broker and AMQ Interconnect.

For more information, see Using AMQ Console.

24.2. USING THE MANAGEMENT API

AMQ Broker 7.2 has an extensive management API that allows a user to modify a broker's configuration, create new resources (for example, addresses and queues), inspect these resources (for example, how many messages are currently held in a queue), and interact with them (for example, to remove messages from a queue). Using the management API, clients can also manage the broker and subscribe to management notifications.

There are two ways to manage the broker:

1. Using JMX — JMX is the standard way to manage Java applications

2. Using the JMS API — management operations are sent to the broker using JMS messages and the AMQ JMS client

Although there are two different ways to manage the broker, each API supports the same functionality. If it is possible to manage a resource using JMX it is also possible to achieve the same result by using JMS messages and the AMQ JMS client.

This choice depends on your particular requirements, application settings, and environment.

Regardless of the way you invoke management operations, the management API is the same.

For each managed resource, there exists a Java interface describing what can be invoked for this type of resource.

The broker exposes its managed resources in the `org.apache.activemq.artemis.api.core.management` package.

The way to invoke management operations depends on whether JMX messages or JMS messages and the AMQ JMS client is used.

> **NOTE**
>
> A few management operations require a `filter` parameter to choose which messages are affected by the operation. Passing `null` or an empty string means that the management operation will be performed on *all messages*.

24.2.1. Managing the Broker

Listing, creating, deploying, and destroying queues

A list of deployed queues can be retrieved using the **getQueueNames()** method.

Queues can be created or destroyed using the management operations **createQueue()**, **deployQueue()**, or **destroyQueue()** on the **ActiveMQServerControl** (with the **ObjectName org.apache.activemq.artemis:broker="*BROKER_NAME*"** or the resource name **server**).

createQueue will fail if the queue already exists while **deployQueue** will do nothing.

Pausing and resuming queues

The **QueueControl** can pause and resume the underlying queue. When a queue is paused, it will receive messages but will not deliver them. When it is resumed, it will begin delivering the queued messages, if any.

Listing and closing remote connections

- Retrieve a client's remote addresses by using **listRemoteAddresses()**. It is also possible to close the connections associated with a remote address using the **closeConnectionsForAddress()** method.

- Alternatively, list connection IDs using **listConnectionIDs()** and list all the sessions for a given connection ID using **listSessions()**.

Managing transactions

In case of a broker crash, when the broker restarts, some transactions might require manual intervention. Use the the following methods to help resolve issues you encounter.

- List the transactions which are in the prepared states (the transactions are represented as opaque Base64 Strings) using the **listPreparedTransactions()** method lists.

- Commit or rollback a given prepared transaction using **commitPreparedTransaction()** or **rollbackPreparedTransaction()** to resolve heuristic transactions.

- List heuristically completed transactions using the **listHeuristicCommittedTransactions()** and **listHeuristicRolledBackTransactions** methods.

Enabling and resetting message counters

- Enable and disable message counters using the **enableMessageCounters()** or **disableMessageCounters()** method.

- Reset message counters by using the **resetAllMessageCounters()** and **resetAllMessageCounterHistories()** methods.

Retrieving broker configuration and attributes

The **ActiveMQServerControl** exposes the broker's configuration through all its attributes (for example, **getVersion()** method to retrieve the broker's version, and so on).

Listing, creating, and destroying Core Bridge and diverts

- List deployed Core Bridge and diverts using the **getBridgeNames()** and **getDivertNames()** methods respectively.

- Create or destroy using bridges and diverts using **createBridge()** and **destroyBridge()** or **createDivert()** and **destroyDivert()** on the **ActiveMQServerControl** (with the **ObjectName**

`org.apache.activemq.artemis:broker="`*BROKER_NAME*`"` or the resource name `server`).

Stopping the broker and forcing failover to occur with any currently attached clients

Use the `forceFailover()` on the `ActiveMQServerControl` (with the `ObjectName org.apache.activemq.artemis:broker="`*BROKER_NAME*`"` or the resource name `server`)
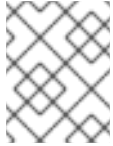
> **NOTE**
>
> Since this method actually stops the broker you will probably receive some sort of error depending on which management service you use to call it.

24.2.2. Managing Addresses

Manage addresses using the `AddressControl` class (with the `ObjectName org.apache.activemq.artemis:broker="`*BROKER_NAME*`",component=addresses,address="`*ADDRESS_NAME*`"` or the resource name `address.`*ADDRESS_NAME*).

- Modify roles and permissions for an address using the `addRole()` or `removeRole()` methods. You can list all the roles associated with the queue with the `getRoles()` method.

24.2.3. Managing Queues

The bulk of the core management API deals with queues. The `QueueControl` class defines the queue management operations (with the `ObjectName org.apache.activemq.artemis:broker="`*BROKER_NAME*`",component=addresses,address="`*BOUND_ADDRESS*`",subcomponent=queues,routing-type="`*ROUTING_TYPE*`",queue="`*QUEUE_NAME*`"` or the resource name `queue.`*QUEUE_NAME*).

Most of the management operations on queues take either a single message ID (for example, to remove a single message) or a filter (for example, to expire all messages with a given property.)

Expiring, sending to a dead letter address, and moving messages

- Expire messages from a queue using the `expireMessages()` method. If an expiry address is defined, messages will be sent to it, otherwise they are discarded. The queue's expiry address can be set with the `setExpiryAddress()` method.

- Send messages to a dead letter address with the `sendMessagesToDeadLetterAddress()` method. It returns the number of messages which are sent to the dead letter address. If a dead letter address is not defined, messages are removed from the queue and discarded. The queue's dead letter address can be set with the `setDeadLetterAddress()` method.

- Move messages from one queue to another by using the `moveMessages()` method.

Listing and removing messages

- List messages from a queue using the `listMessages()` method. It will return an array of `Map`, one `Map` for each message.

- Remove messages from a queue using the `removeMessages()` method, which returns a

boolean for the single message ID variant or the number of removed messages for the filter variant. This method takes a **filter** argument to remove only filtered messages. Setting the filter to an empty string will in effect remove all messages.

- Counting messages
  The number of messages in a queue is returned by the **getMessageCount()** method. Alternatively, the **countMessages()** will return the number of messages in the queue which match a given filter.

- Changing message priority
  The message priority can be changed by using the **changeMessagesPriority()** method which returns a **boolean** for the single message ID variant or the number of updated messages for the filter variant.

- Message counters
  Message counters can be listed for a queue with the **listMessageCounter()** and **listMessageCounterHistory()** methods (see the Message Counters section). The message counters can also be reset for a single queue using the **resetMessageCounter()** method.

- Retrieving the queue attributes
  The **QueueControl** exposes queue settings through its attributes (for example, **getFilter()** to retrieve the queue's filter if it was created with one, **isDurable()** to know whether the queue is durable, and so on).

- Pausing and resuming queues
  The **QueueControl** can pause and resume the underlying queue. When a queue is paused, it will receive messages but will not deliver them. When it is resumed, it will begin delivering the queued messages, if any.

24.2.3.1. Managing Other Resources

You can start and stop the broker's remote resources (acceptors, diverts, bridges, and so on) so that a broker can be taken offline for a given period of time without stopping it completely (for example, if other management operations must be performed, such as resolving heuristic transactions). These resources are:

- Acceptors
  Start or stop an acceptor using the **start()** or. **stop()** method on the **AcceptorControl** class (with the **ObjectName org.apache.activemq.artemis:broker="BROKER_NAME",component=acceptors,name="ACCEPTOR_NAME"** or the resource name **acceptor.ADDRESS_NAME**). Acceptor parameters can be retrieved using the **AcceptorControl** attributes. See Network Connections:Acceptors and Connectors for more information about Acceptors.

- Diverts
  Start or stop a divert using the **start()** or **stop()** method on the **DivertControl** class (with the **ObjectName org.apache.activemq.artemis:broker="BROKER_NAME",component=diverts,name="DIVERT_NAME"** or the resource name **divert.DIVERT_NAME**). Divert parameters can be retrieved using the **DivertControl** attributes.

- Bridges

Start or stop a bridge using the **start()** (resp. **stop()**) method on the **BridgeControl** class (with the **ObjectName org.apache.activemq.artemis:broker="*BROKER_NAME*",component=bridge,name= "*BRIDGE_NAME*"** or the resource name **bridge.*BRIDGE_NAME***). Bridge parameters can be retrieved using the **BridgeControl** attributes. See Clustering for more information.

- Broadcast groups
  Start or stop a broadcast group using the **start()** or **stop()** method on the **BroadcastGroupControl** class (with the **ObjectName org.apache.activemq.artemis:broker="*BROKER_NAME*",component=broadcast-group,name="*BROADCAST_GROUP_NAME*"** or the resource name **broadcastgroup.*BROADCAST_GROUP_NAME***). Broadcast group parameters can be retrieved using the **BroadcastGroupControl** attributes. See Clustering for more information.

- Discovery groups
  Start or stop a discovery group using the **start()** or **stop()** method on the **DiscoveryGroupControl** class (with the **ObjectName org.apache.activemq.artemis:broker="*BROKER_NAME*",component=discovery-group,name="*DISCOVERY_GROUP_NAME*"** or the resource name **discovery.*DISCOVERY_GROUP_NAME***). Discovery groups parameters can be retrieved using the **DiscoveryGroupControl** attributes. See Clustering for more information.

- Cluster connections
  Start or stop a cluster connection using the **start()** or **stop()** method on the **ClusterConnectionControl** class (with the **ObjectName org.apache.activemq.artemis:broker="*BROKER_NAME*",component=cluster-connection,name="*CLUSTER_CONNECTION_NAME*"** or the resource name **clusterconnection.*CLUSTER_CONNECTION_NAME***). Cluster connection parameters can be retrieved using the **ClusterConnectionControl** attributes. See Clustering for more information.

24.2.4. Managing the Broker Using JMX

The broker can be managed using JMX. The management API is exposed by the broker using MBeans interfaces. The broker registers its resources with the domain **org.apache.activemq**.

For example, the **ObjectName** to manage a queue named **exampleQueue** is:

```
org.apache.activemq.artemis:broker="__BROKER_NAME__",component=addresses,a
ddress="exampleQueue",subcomponent=queues,routingtype="anycast",queue="exa
mpleQueue"
```

and the MBean is:

```
org.apache.activemq.artemis.api.management.QueueControl
```

The MBean's **ObjectName** is built using the helper class **org.apache.activemq.artemis.api.core.management.ObjectNameBuilder**. You can also use **jconsole** to find the **ObjectName** of the MBeans you want to manage.

Managing the broker using JMX is identical to management of any Java applications using JMX. It can be done by reflection or by creating proxies of the MBeans.

24.2.4.1. Configuring JMX Management

By default, JMX is enabled to manage the broker. It can be disabled by setting **jmx-management-enabled** to **false** in **broker.xml**:

```
<jmx-management-enabled>false</jmx-management-enabled>
```

If JMX is enabled, the broker can be managed locally using **jconsole**.

> **NOTE**
>
> Remote connections to JMX are not enabled by default for security reasons. Refer to Oracle's Java Management Guide to configure the broker for remote management. System properties must be set in the **artemis**, or **artemis.cmd** for Windows installations, shell script located under *INSTALL_DIR/bin*.

By default, the broker uses the JMX domain "org.apache.activemq.artemis". To manage several brokers from the *same* MBeanServer, the JMX domain can be configured for each individual broker by setting **jmx-domain** in **broker.xml**:

```
<jmx-domain>my.org.apache.activemq</jmx-domain>
```

24.2.4.2. MBeanServer Configuration

When the broker is run in standalone mode, it uses the Java Virtual Machine's **Platform MBeanServer** to register its MBeans. By default Jolokia is also deployed to allow access to the MBean server using REST.

24.2.4.3. Exposing JMX Using Jolokia

The default Broker configuration ships with the Jolokia http agent deployed as a web application. Jolokia is a remote JMX over HTTP bridge that exposes MBeans. For more information see the Jolokia documentation.
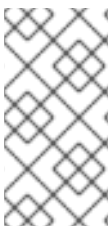
> **NOTE**
>
> To use Jolokia, the user must belong to the role defined by the **hawtio.role** system property in the *BROKER_INSTANCE_DIR/etc/artemis.profile* configuration file. By default, this role is **amq**. For more information about assigning a user to a role, see Section 9.2, "Adding Users".

Example 24.1. Using Jolokia to Query the Broker's Version

This example uses a Jolokia REST URL to find the version of a broker.

```
$ curl
http://admin:admin@localhost:8161/console/jolokia/read/org.apache.active
mq.artemis:broker=\"0.0.0.0\"/Version
{"request":
{"mbean":"org.apache.activemq.artemis:broker=\"0.0.0.0\"","attribute":"V
ersion","type":"read"},"value":"2.4.0.amq-710002-redhat-
1","timestamp":1527105236,"status":200}
```

24.2.5. Managing the Broker Using JMS Messages and the AMQ JMS Client

The management queue is a special queue and needs to be instantiated directly by the client:

```
Queue managementQueue =
ActiveMQJMSClient.createQueue("activemq.management");
```

To invoke management operations using JMS messages and the AMQ JMS client:

1. Create a **QueueRequestor** to send messages to the management address and receive replies.

2. Create a **Message**.

3. Use the helper class **org.apache.activemq.artemis.api.jms.management.JMSManagementHelper** to fill the message with the management properties.

4. Send the message using the **QueueRequestor**.

5. Use the helper class **org.apache.activemq.artemis.api.jms.management.JMSManagementHelper** to retrieve the operation result from the management reply.

For example, to view the number of messages in the JMS queue **exampleQueue**:

```
Queue managementQueue =
ActiveMQJMSClient.createQueue("activemq.management");

QueueSession session = ...
QueueRequestor requestor = new QueueRequestor(session, managementQueue);
connection.start();
Message message = session.createMessage();
JMSManagementHelper.putAttribute(message, "queue.exampleQueue",
"messageCount");
Message reply = requestor.request(message);
int count = (Integer)JMSManagementHelper.getResult(reply);
System.out.println("There are " + count + " messages in exampleQueue");
```

24.2.5.1. Configuring Broker Management Using JMS Messages and the AMQ JMS Client

The management address to send management messages is configured in the broker.xml file: <management-address>queue.activemq.management</management-address>

By default, the address is **queue.activemq.management**. The management address requires a special user permission type, **manage**, to be able to receive and handle management messages. This permission type is specified in the broker.xml file:

<security-setting-match="queue.activemq.management"> <permission-type="manage" roles="admin"/> </security-setting>

24.2.6. Management Notifications

The broker sends notifications to inform listeners of events such as the creation of new resources, security violations, and other events.

There are two ways to receive these notifications:

* JMX notifications

* JMS messages

24.2.6.1. JMX Notifications

If JMX is enabled (see Configuring JMX Management), JMX notifications can be received by subscribing to **ObjectName org.apache.activemq.artemis:broker="*BROKER_NAME*"**.

24.2.6.2. Notification Types and Headers

Below is a list of all the different kinds of notifications as well as which headers are on the messages. Every notification has a **_AMQ_NotifType** (value noted in parentheses) and **_AMQ_NotifTimestamp** header. The timestamp is the unformatted result of a call to **java.lang.System.currentTimeMillis()**.

* **BINDING_ADDED** (0)

  ```
  `_AMQ_Binding_Type`, `_AMQ_Address`, `_AMQ_ClusterName`,
  `_AMQ_RoutingName`, `_AMQ_Binding_ID`, `_AMQ_Distance`,
  `_AMQ_FilterString`
  ```

* **BINDING_REMOVED** (1)

  ```
  `_AMQ_Address`, `_AMQ_ClusterName`, `_AMQ_RoutingName`,
  `_AMQ_Binding_ID`, `_AMQ_Distance`, `_AMQ_FilterString`
  ```

* **CONSUMER_CREATED** (2)

  ```
  `_AMQ_Address`, `_AMQ_ClusterName`, `_AMQ_RoutingName`,
  `_AMQ_Distance`, `_AMQ_ConsumerCount`, `_AMQ_User`,
  `_AMQ_RemoteAddress`, `_AMQ_SessionName`, `_AMQ_FilterString`
  ```

* **CONSUMER_CLOSED** (3)

  ```
  `_AMQ_Address`, `_AMQ_ClusterName`, `_AMQ_RoutingName`,
  `_AMQ_Distance`, `_AMQ_ConsumerCount`, `_AMQ_User`,
  `_AMQ_RemoteAddress`, `_AMQ_SessionName`, `_AMQ_FilterString`
  ```

* **SECURITY_AUTHENTICATION_VIOLATION** (6)

  ```
  `_AMQ_User`
  ```

* **SECURITY_PERMISSION_VIOLATION** (7)

  ```
  `_AMQ_Address`, `_AMQ_CheckType`, `_AMQ_User`
  ```

* **DISCOVERY_GROUP_STARTED** (8)

`name`

- **DISCOVERY_GROUP_STOPPED** (9)

  `name`

- **BROADCAST_GROUP_STARTED** (10)

  `name`

- **BROADCAST_GROUP_STOPPED** (11)

  `name`

- **BRIDGE_STARTED** (12)

  `name`

- **BRIDGE_STOPPED** (13)

  `name`

- **CLUSTER_CONNECTION_STARTED** (14)

  `name`

- **CLUSTER_CONNECTION_STOPPED** (15)

  `name`

- **ACCEPTOR_STARTED** (16)

  `factory`, `id`

- **ACCEPTOR_STOPPED** (17)

  `factory`, `id`

- **PROPOSAL** (18)

  `_JBM_ProposalGroupId`, `_JBM_ProposalValue`, `_AMQ_Binding_Type`, `_AMQ_Address`, `_AMQ_Distance`

- **PROPOSAL_RESPONSE** (19)

  `_JBM_ProposalGroupId`, `_JBM_ProposalValue`, `_JBM_ProposalAltValue`, `_AMQ_Binding_Type`, `_AMQ_Address`, `_AMQ_Distance`

- **CONSUMER_SLOW** (21)

```
`_AMQ_Address`, `_AMQ_ConsumerCount`, `_AMQ_RemoteAddress`,
`_AMQ_ConnectionName`, `_AMQ_ConsumerName`, `_AMQ_SessionName`
```

24.2.7. Message Counters

Message counters can be used to obtain information on queues over time as the broker keeps a history on queue metrics.

They can be used to show trends on queues. For example, using the management API, it would be possible to query the number of messages in a queue at regular intervals. However, this would not be enough to know if the queue is used: the number of messages can remain constant because nobody is sending or receiving messages from the queue or because there are as many messages sent to the queue than messages consumed from it. The number of messages in the queue remains the same in both cases but its use is widely different.

Message counters gives additional information about the queues:

- **count**

  The *total* number of messages added to the queue since the broker was started

- **countDelta**

  The number of messages added to the queue *since the last message counter update*

- **messageCount**

  The *current* number of messages in the queue

- **messageCountDelta**

  The *overall* number of messages added/removed from the queue *since the last message counter update*. For example, if **messageCountDelta** is equal to **-10** this means that overall 10 messages have been removed from the queue (for example, 2 messages were added and 12 were removed)

- **lastAddTimestamp**

  The timestamp of the last time a message was added to the queue

- **udpateTimestamp**

  The timestamp of the last message counter update

  These attributes can be used to determine other meaningful data as well. For example, to know specifically how many messages were consumed from the queue since the last update simply subtract the **messageCountDelta** from **countDelta**.

24.2.7.1. Configuring Message Counters

By default, message counters are disabled as it might have a small negative effect on memory.

To enable message counters, you can set it to **true** in **broker.xml**:

```xml
<message-counter-enabled>true</message-counter-enabled>
```

Message counters keeps a history of the queue metrics (10 days by default) and samples all the queues at regular interval (10 seconds by default). If message counters are enabled, these values should be configured to suit your messaging use case in **broker.xml**:

```
<!-- keep history for a week -->
<message-counter-max-day-history>7</message-counter-max-day-history>
<!-- sample the queues every minute (60000ms) -->
<message-counter-sample-period>60000</message-counter-sample-period>
```

Message counters can be retrieved using the Management API. For example, to retrieve message counters on a JMS queue using JMX:

```
// retrieve a connection to the brokers MBeanServer
MBeanServerConnection mbsc = ...
JMSQueueControlMBean queueControl =
(JMSQueueControl)MBeanServerInvocationHandler.newProxyInstance(mbsc,
    on,
    JMSQueueControl.class,
    false);
// message counters are retrieved as a JSON String
String counters = queueControl.listMessageCounter();
// use the MessageCounterInfo helper class to manipulate message counters
more easily
MessageCounterInfo messageCounter = MessageCounterInfo.fromJSON(counters);
System.out.format("%s message(s) in the queue (since last sample: %s)\n",
messageCounter.getMessageCount(),
messageCounter.getMessageCountDelta());
```

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. A valid Red Hat subscription is required.

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.
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 2-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 2-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 2-way SSL is required. Valid values are `true` or `false`. Default is `false`.
sslEnabled	Both	Must be `true` to enable SSL. Default is `false`.

Parameter	Use with…	Description
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 2-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.
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 2-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.
verifyHost	Both	When used on an acceptor the CN of the connecting client's SSL certificate will be compared to its hostname to verify they match. This is useful only for 2-way SSL. When used on a connector the CN of the server's SSL certificate will be compared to its hostname to verify they match. This is useful for both 1-way and 2-way SSL. Valid values are `true` or `false`. Default is `false`.

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-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**.
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**.

Name	Description
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 **false**.

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.
default-address-routing-type	The routing-type used on auto-created addresses. The default value is **MULTICAST**.
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**.

Name	Description
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.
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**.

Name	Description
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. This parameter is mandatory.
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.
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.

Name	Description
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**.
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.

Name	Description
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 to master brokers only. Determines whether the master checks the cluster for another master server using its own server ID when starting up. Set to `true` if you want to fail back to the original master broker. The default 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, slave brokers will only pair with master brokers with the matching `group-name`.
initial-replication-sync-timeout	The amount of time the replicating broker will wait at the completion of the initial replication process for the replica to acknowledge it has received all the necessary data. The default is `30,000` milliseconds, or 30 seconds. **NOTE** During this interval any journal related operations will be blocked.
max-saved-replicated-journals-size	Applies to slave brokers only. Specifies how many times a slave broker can restart after moving its files on start. Once there are this number of slave journal files the broker will stop permanently after if fails back. The default is `2`.
allow-failback	Applies to slave brokers only. Determines whether the slave broker will automatically restart and resume its original role when another broker places a request to take over its place. The default is `true`.

APPENDIX G. USING YOUR SUBSCRIPTION

AMQ is provided through a software subscription. To manage your subscriptions, access your account at the Red Hat Customer Portal.

Accessing your account

1. Go to access.redhat.com.

2. If you do not already have an account, create one.

3. Log in to your account.

Activating a subscription

1. Go to access.redhat.com.

2. Navigate to **My Subscriptions**.

3. Navigate to **Activate a subscription** and enter your 16-digit activation number.

Downloading zip and tar files

To access zip or tar files, use the customer portal to find the relevant files for download. If you are using RPM packages, this step is not required.

1. Open a browser and log in to the Red Hat Customer Portal **Product Downloads** page at access.redhat.com/downloads.

2. Locate the **Red Hat AMQ** entries in the **JBOSS INTEGRATION AND AUTOMATION** category.

3. Select the desired AMQ product. The **Software Downloads** page opens.

4. Click the **Download** link for your component.

Registering your system for packages

To install RPM packages on Red Hat Enterprise Linux, your system must be registered. If you are using zip or tar files, this step is not required.

1. Go to access.redhat.com.

2. Navigate to **Registration Assistant**.

3. Select your OS version and continue to the next page.

4. Use the listed command in your system terminal to complete the registration.

To learn more see How to Register and Subscribe a System to the Red Hat Customer Portal.

APPENDIX H. ABOUT APACHE MAVEN

Apache Maven is a distributed build automation tool used in Java application development to create, manage, and build software projects. Maven uses standard configuration files called Project Object Model, or POM, files to define projects and manage the build process. POMs describe the module and component dependencies, build order, and targets for the resulting project packaging and output using an XML file. This ensures that the project is built in a correct and uniform manner.

Maven achieves this by using a repository. A Maven repository stores Java libraries, plug-ins, and other build artifacts. The default public repository is the Maven 2 Central Repository, but repositories can be private and internal within a company with a goal to share common artifacts among development teams. Repositories are also available from third-parties. AMQ includes a Maven repository that contains tested and supported versions of the AMQ 7 Java packages and dependencies.

For more information about Maven, see Welcome to Apache Maven.

For more information about Maven repositories, see Apache Maven Project - Introduction to Repositories.

H.1. ABOUT THE MAVEN POM FILE

The Project Object Model, or POM, file is a configuration file used by Maven to build projects. It is an XML file that contains information about the project and how to build it, including the location of the source, test, and target directories, the project dependencies, plug-in repositories, and goals it can execute. It can also include additional details about the project including the version, description, developers, mailing list, license, and more. Maven favors "convention over configuration". A **pom.xml** file requires minimal configuration and will default all other values.

The schema for the **pom.xml** file can be found at http://maven.apache.org/maven-v4_0_0.xsd.

For more information about POM files, see the Apache Maven Project POM Reference.

Minimum Requirements of a Maven POM File
The minimum requirements of a **pom.xml** file are as follows:

- project root

- modelVersion

- groupId - the id of the project's group

- artifactId - the id of the artifact (project)

- version - the version of the artifact under the specified group

Example: Sample pom.xml File

A basic **pom.xml** file might look like this:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jboss.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

H.2. ABOUT THE MAVEN SETTINGS FILE

The Maven `settings.xml` file contains user-specific configuration information for Maven. It contains information that must not be distributed with the `pom.xml` file, such as developer identity, proxy information, local repository location, and other settings specific to a user.

There are two locations where the `settings.xml` can be found:

- **In the Maven installation:** The settings file can be found in the `$M2_HOME/conf/` directory. These settings are referred to as `global` settings. The default Maven settings file is a template that can be copied and used as a starting point for the user settings file.

- **In the user's installation:** The settings file can be found in the `${user.home}/.m2/` directory. If both the Maven and user `settings.xml` files exist, the contents are merged. Where there are overlaps, the user's `settings.xml` file takes precedence.

Example: Maven Settings File

```xml
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <profiles>
    <!-- Configure the JBoss AMQ Maven repository -->
    <profile>
      <id>jboss-amq-maven-repository</id>
      <repositories>
        <repository>
          <id>jboss-amq</id>
          <url>file:///path/to/repo/</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>jboss-amq-maven-plugin-repository</id>
          <url>file://path/to/repo</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>
  <activeProfiles>
    <!-- Optionally, make the repository active by default -->
    <activeProfile>jboss-amq-maven-repository</activeProfile>
```

```
    </activeProfiles>
</settings>
```

The schema for the `settings.xml` file can be found at http://maven.apache.org/xsd/settings-1.0.0.xsd.

Revised on 2018-11-15 13:01:39 UTC