Red Hat AMQ 7.5

Using the AMQ Core Protocol JMS Client

For Use with AMQ Clients 2.6

Last Updated: 2020-02-26

Red Hat AMQ 7.5 Using the AMQ Core Protocol JMS Client

For Use with AMQ Clients 2.6

Legal Notice

Abstract

This guide describes how to install and configure the client, run hands-on examples, and use your client with other AMQ components.

Table of Contents

CHAPTER 1. OVERVIEW 3
1.1. KEY FEATURES 3
1.2. SUPPORTED STANDARDS AND PROTOCOLS 3
1.3. SUPPORTED CONFIGURATIONS 3
1.4. TERMS AND CONCEPTS 3
1.5. DOCUMENT CONVENTIONS 4

CHAPTER 2. INSTALLATION 5
2.1. USING THE RED HAT MAVEN REPOSITORY 5
2.2. INSTALLING A LOCAL MAVEN REPOSITORY 5
2.3. INSTALLING THE ZIP FILE 5

CHAPTER 3. GETTING STARTED 7
3.1. PREREQUISITES 7
3.2. PREPARING THE BROKER 7
3.3. RUNNING YOUR FIRST EXAMPLE 7

CHAPTER 4. RECONNECT AND FAILOVER 9
4.1. AUTOMATIC CLIENT FAILOVER 9
4.1.1. Failing over during the initial connection 10
Setting the number of reconnection attempts 10
Setting a global number of reconnection attempts 10
4.1.2. Handling blocking calls during failover 10
4.1.3. Handling failover with transactions 11
4.1.4. Getting notified of connection failure 11
4.2. APPLICATION-LEVEL FAILOVER 12
4.3. DETECTING DEAD CONNECTIONS 12
Detecting dead connections from the client side 12
4.4. CONNECTION TIME-TO-LIVE 13
Configuring time-to-live on the broker 13
Configuring time-to-live on the client 14
4.5. DISABLING ASYNCHRONOUS CONNECTION EXECUTION 14
4.6. CLOSING CONNECTIONS FROM THE CLIENT SIDE 15
4.6.1. Configuring a client to use dynamic discovery 15
Configuring dynamic discovery using JMS 15
4.7. CONFIGURING A CLIENT TO USE STATIC DISCOVERY 16
Configuring static discovery 16

APPENDIX A. USING YOUR SUBSCRIPTION 18
Accessing your account 18
Activating a subscription 18
Downloading ZIP and TAR files 18
Registering your system for packages 18

APPENDIX B. USING RED HAT MAVEN REPOSITORIES 19
B.1. USING THE ONLINE REPOSITORY 19
Adding the repository to your Maven settings 19
Adding the repository to your POM file 20
B.2. USING A LOCAL REPOSITORY 20

CHAPTER 1. OVERVIEW

AMQ Core Protocol JMS is a Java Message Service (JMS) 2.0 client for use in messaging applications that send and receive Artemis Core Protocol messages.

AMQ Core Protocol JMS is part of AMQ Clients, a suite of messaging libraries supporting multiple languages and platforms. For an overview of the clients, see AMQ Clients Overview. For information about this release, see AMQ Clients 2.6 Release Notes .

AMQ Core Protocol JMS is based on the JMS client from Apache ActiveMQ Artemis.

1.1. KEY FEATURES

- JMS 1.1 and 2.0 compatible
- SSL/TLS for secure communication
- Automatic reconnect and failover
- Distributed transactions (XA)
- Pure-Java implementation

1.2. SUPPORTED STANDARDS AND PROTOCOLS

AMQ Core Protocol JMS supports the following industry-recognized standards and network protocols:

- Version 2.0 of the Java Message Service API
- Modern TCP with IPv6

1.3. SUPPORTED CONFIGURATIONS

AMQ Core Protocol JMS supports the following OS and language versions.

- Red Hat Enterprise Linux 6, 7, and 8 with the following JDKs:
  - OpenJDK 8
  - Oracle JDK 8
  - IBM JDK 8
- Microsoft Windows 10 Pro with Oracle JDK 8
- Microsoft Windows Server 2012 R2 and 2016 with Oracle JDK 8

For more information, see Red Hat AMQ 7 Supported Configurations .

1.4. TERMS AND CONCEPTS

This section introduces the core API entities and describes how they operate together.

Table 1.1. API terms

Entity	Description
ConnectionFactory	An entry point for creating connections.
Connection	A channel for communication between two peers on a network. It contains sessions.
Session	A context for producing and consuming messages. It contains message producers and consumers.
MessageProducer	A channel for sending messages to a destination. It has a target destination.
MessageConsumer	A channel for receiving messages from a destination. It has a source destination.
Destination	A named location for messages, either a queue or a topic.
Queue	A stored sequence of messages.
Topic	A stored sequence of messages for multicast distribution.
Message	An application-specific piece of information.

AMQ Core Protocol JMS sends and receives *messages*. Messages are transferred between connected peers using *message producers* and *consumers*. Producers and consumers are established over *sessions*. Sessions are established over *connections*. Connections are created by *connection factories*.

A sending peer creates a producer to send messages. The producer has a *destination* that identifies a target queue or topic at the remote peer. A receiving peer creates a consumer to receive messages. Like the producer, the consumer has a destination that identifies a source queue or topic at the remote peer.

A destination is either a *queue* or a *topic*. In JMS, queues and topics are client-side representations of named broker entities that hold messages.

A queue implements point-to-point semantics. Each message is seen by only one consumer, and the message is removed from the queue after it is read. A topic implements publish-subscribe semantics. Each message is seen by multiple consumers, and the message remains available to other consumers after it is read.

See the JMS tutorial for more information.

1.5. DOCUMENT CONVENTIONS

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

CHAPTER 2. INSTALLATION

This chapter guides you through the steps to install AMQ Core Protocol JMS in your environment.

2.1. USING THE RED HAT MAVEN REPOSITORY

The client uses Apache Maven as its build tool. You can configure your Maven environment to download the library from the Red Hat Maven repository.

Procedure

1. Add the Red Hat repository to your Maven settings or POM file. For example configuration files, see Section B.1, "Using the online repository" .

```
<repository>
  <id>red-hat-ga</id>
  <url>https://maven.repository.redhat.com/ga</url>
</repository>
```

2. Add the library dependency to your POM file.

```
<dependency>
  <groupId>org.apache.activemq</groupId>
  <artifactId>artemis-jms-client</artifactId>
  <version>2.10.0.redhat-00004</version>
</dependency>
```

The client is now available in your Maven project.

2.2. INSTALLING A LOCAL MAVEN REPOSITORY

As an alternative to the online repository, AMQ Core Protocol JMS can be installed to your local filesystem as a file-based Maven repository. Note that AMQ Core Protocol JMS is delivered as part of the AMQ Broker component.

Procedure

1. Use your subscription to download the **AMQ Broker Maven Repository**zip file.

2. Extract the file contents into a directory of your choosing.
   On Linux or UNIX, use the **unzip** command to extract the file contents.

```
unzip amq-broker-<version>-maven-repository.zip
```

   On Windows, right-click on the zip file and select **Extract All**.

3. Configure Maven to use the repository in the **maven-repository** directory inside the extracted install directory. For more information, see Section B.2, "Using a local repository" .

2.3. INSTALLING THE ZIP FILE

AMQ Core Protocol JMS is delivered as part of the AMQ Broker component. The AMQ Broker zip file contains the examples and a distribution of the client libraries for those not using Maven. If you are using Maven and do not require the examples, you do not need to install the zip file.

Procedure

1. Use your subscription to download the **AMQ Broker** zip file.
2. Extract the file contents into a directory of your choosing.
   On Linux or UNIX, use the **unzip** command to extract the file contents.

   ```
   unzip amq-broker-<version>-bin.zip
   ```

   On Windows, right-click on the zip file and select **Extract All**.

   When you extract the contents of the zip file, a directory named **amq-broker-<version>** is created. This is the top-level directory of the installation and is referred to as **<install-dir>** throughout this document.
3. (Optional) If you are not using Maven, add the jar files in the **<install-dir>/lib** directory to your Java classpath.

CHAPTER 3. GETTING STARTED

This chapter guides you through a simple exercise to help you get started using AMQ Core Protocol JMS.

3.1. PREREQUISITES

- The example programs are located in the AMQ Broker zip file. To get started, you must install the zip file.
- To build the example, Maven must be configured to use the Red Hat repository or a local repository.

3.2. PREPARING THE BROKER

The example programs require a running broker with a queue named **exampleQueue**. Follow these steps to define the queue and start the broker:

Procedure

1. Install the broker.
2. Create a broker instance. Enable anonymous access.
3. Start the broker instance and check the console for any critical errors logged during startup.

   ```
   $ <broker-instance-dir>/bin/artemis run
   ...
   14:43:20,158 INFO  [org.apache.activemq.artemis.integration.bootstrap] AMQ101000:
   Starting ActiveMQ Artemis Server
   ...
   15:01:39,686 INFO  [org.apache.activemq.artemis.core.server] AMQ221020: Started
   Acceptor at 0.0.0.0:5672 for protocols [AMQP]
   ...
   15:01:39,691 INFO  [org.apache.activemq.artemis.core.server] AMQ221007: Server is now
   live
   ```

4. Use the **artemis queue** command to create a queue called **exampleQueue**.

   ```
   <broker-instance-dir>/bin/artemis queue create --name exampleQueue --auto-create-
   address --anycast
   ```

   You are prompted to answer a series of questions. For yes or no questions, type **N**. Otherwise, press Enter to accept the default value.

3.3. RUNNING YOUR FIRST EXAMPLE

Procedure

1. Use Maven to build the examples by running the following command in the **<install-dir>/examples/features/standard/queue** directory.

   ```
   mvn clean package dependency:copy-dependencies -DincludeScope=runtime -DskipTests
   ```

–

The addition of **dependency:copy-dependencies** results in the dependencies being copied into the **target/dependency** directory.

2. Use the **java** command to run the example.
   On Linux or UNIX:

   ```
   java -cp "target/classes:target/dependency/*"
   org.apache.activemq.artemis.jms.example.QueueExample
   ```

   On Windows:

   ```
   java -cp "target\classes;target\dependency\*"
   org.apache.activemq.artemis.jms.example.QueueExample
   ```

The example creates a consumer and producer for a queue named **exampleQueue**. It sends a text message and then receives it back, printing the received message to the console.

Running it on Linux results in the following output.

```
$ java -cp "target/classes:target/dependency/*"
org.apache.activemq.artemis.jms.example.QueueExample
Sent message: This is a text message
Received message: This is a text message
```

The source code for the example is in the **<install-dir>/examples/features/standard/queue/src** directory. Additional examples are available in the **<install-dir>/examples/features/standard** directory.

CHAPTER 4. RECONNECT AND FAILOVER

AMQ Core Protocol JMS supports automatic reconnect after temporary network failures.

The client can also reconnect, or fail over, to alternate servers. This is often used in combination with HA server clusters. For more information, see Implementing high availability.

4.1. 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 recreated 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 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.

4.1.1. 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 Core Protocol 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> (1)
  ...
 </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.

4.1.2. 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.

4.1.3. 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.

4.1.4. 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 4.1. 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

4.2. 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.

4.3. 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 the 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, or 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, or 10 seconds using Core JMS clients.

Procedure

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

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

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

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

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

4.4. 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, or 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>30</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 **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, or every 1 second. By default, the checks are done every **2000** milliseconds, or 2 seconds.

Configuring time-to-live on the client

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

Procedure

- Set the Time-To-Live for a Client Connection.
  - If you are using JNDI to instantiate your connection factory, you can specify it in the xml config, using the parameter **connectionTtl**.

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

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

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

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

4.5. 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>
```

4.6. 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();
   }
}
```

4.6.1. Configuring a client to use dynamic discovery

You can configure a Red Hat AMQ 7.5 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(groupPort);
discoveryGroupConfiguration.setBroadcastEndpointFactory(udpBroadcastEndpointFactory);

ConnectionFactory jmsConnectionFactory = ActiveMQJMSClient.createConnectionFactoryWithHA
    (discoveryGroupConfiguration, 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()**.

4.7. 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 7.5 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.ActiveMQInitialContextFactory
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);
```

APPENDIX A. 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 B. USING RED HAT MAVEN REPOSITORIES

This section describes how to use Red Hat-provided Maven repositories in your software.

B.1. USING THE ONLINE REPOSITORY

Red Hat maintains a central Maven repository for use with your Maven-based projects. For more information, see the repository welcome page .

There are two ways to configure Maven to use the Red Hat repository:

- Add the repository to your Maven settings
- Add the repository to your POM file

Adding the repository to your Maven settings

This method of configuration applies to all Maven projects owned by your user, as long as your POM file does not override the repository configuration and the included profile is enabled.

Procedure

1. Locate the Maven **settings.xml** file. It is usually inside the **.m2** directory in the user home directory. If the file does not exist, use a text editor to create it.
   On Linux or UNIX:

   ```
   /home/<username>/.m2/settings.xml
   ```

   On Windows:

   ```
   C:\Users\<username>\.m2\settings.xml
   ```

2. Add a new profile containing the Red Hat repository to the **profiles** element of the **settings.xml** file, as in the following example:

   #### Example: A Maven settings.xml file containing the Red Hat repository

   ```
   <settings>
     <profiles>
       <profile>
         <id>red-hat</id>
         <repositories>
           <repository>
             <id>red-hat-ga</id>
             <url>https://maven.repository.redhat.com/ga</url>
           </repository>
         </repositories>
         <pluginRepositories>
           <pluginRepository>
             <id>red-hat-ga</id>
             <url>https://maven.repository.redhat.com/ga</url>
             <releases>
               <enabled>true</enabled>
             </releases>
             <snapshots>
   ```

```
          <enabled>false</enabled>
        </snapshots>
      </pluginRepository>
    </pluginRepositories>
  </profile>
</profiles>
<activeProfiles>
  <activeProfile>red-hat</activeProfile>
</activeProfiles>
</settings>
```

For more information about Maven configuration, see the Maven settings reference.

Adding the repository to your POM file

To configure a repository directly in your project, add a new entry to the **repositories** element of your POM file, as in the following example:

Example: A Maven **pom.xml** file containing the Red Hat repository

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.example</groupId>
  <artifactId>example-app</artifactId>
  <version>1.0.0</version>

  <repositories>
    <repository>
      <id>red-hat-ga</id>
      <url>https://maven.repository.redhat.com/ga</url>
    </repository>
  </repositories>
</project>
```

For more information about POM file configuration, see the Maven POM reference.

B.2. USING A LOCAL REPOSITORY

Red Hat provides file-based Maven repositories for some of its components. These are delivered as downloadable archives that you can extract to your local filesystem.

To configure Maven to use a locally extracted repository, apply the following XML in your Maven settings or POM file:

```
<repository>
  <id>red-hat-local</id>
  <url>${repository-url}</url>
</repository>
```

${repository-url} must be a file URL containing the local filesystem path of the extracted repository.

Table B.1. Example URLs for local Maven repositories

Operating system	Filesystem path	URL
Linux or UNIX	**/home/alice/maven-repository**	**file:/home/alice/maven-repository**
Windows	**C:\repos\red-hat**	**file:C:\repos\red-hat**

Revised on 2020-02-26 17:17:07 UTC