Red Hat JBoss A-MQ 6.1

Client Connectivity Guide

Creating and tuning clients connections to message brokers

Last Updated: 2017-10-13

Red Hat JBoss A-MQ 6.1 Client Connectivity Guide

Creating and tuning clients connections to message brokers

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Abstract

Red Hat JBoss A-MQ supports a number of different wire protocols and message formats. This guide provides a quick reference for understanding how to configure connections between clients and message brokers.

Table of Contents

CHAPTER 1. INTRODUCTION

Abstract

Red Hat JBoss A-MQ clients can connect to a broker using a variety of transports and APIs. The connections are highly configurable and can be tuned for the majority of use cases.

1.1. JBOSS A-MQ CLIENT APIS

Transports and protocols

Red Hat JBoss A-MQ uses OpenWire as its default on the wire message protocol. OpenWire is a JMS compliant wire protocol that is designed to be fully-featured and highly performant. It is the default protocol of JBoss A-MQ. OpenWire can use a number of transports including TCP, SSL, and HTTP.

In addition to OpenWire, JBoss A-MQ clients can also use a number of other transports including:

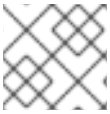
- Simple Text Orientated Messaging Protocol(STOMP)—allows developers to use a wide variety of client APIs to connect to a broker.
- Discovery—allows clients to connect to one or more brokers without knowing the connection details for a specific broker. See *Using Networks of Brokers*.
- VM—allows clients to directly communicate with other clients in the same virtual machine. See Chapter 5, *Intra-JVM Connections*.
- Peer—allows clients to communicate with each other without using an external message broker. See Chapter 6, *Peer Protocol*.

For details of using the different the transports see the *Connection Reference*.

Supported Client APIs

JBoss A-MQ provides a standard JMS client library. In addition to the standard JMS APIs the Java client library has a few implementation specific APIs.

JBoss A-MQ also has a C++ client library and .Net client library that are developed as part of the Apache ActiveMQ project. You can download them from them from the Red Hat customer portal. You will need to compile them yourselves.



NOTE

This guide only deals with the JBoss A-MQ client libraries.

The STOMP protocol allows you to use a number of other clients including:

- C clients
- C++ clients
- C# and .NET clients
- Delphi clients

- Flash clients
- Perl clients
- PHP clients
- Pike clients
- Python clients

Configuration

There are two types of properties that effects client connections:

- transport options—configured on the connection. These options are configured using the connection URI and may be set by the broker. They apply to all clients using the connection.
- destination options—configured on a per destination basis. These options are configured when the destination is created and impact all of the clients that send or receive messages using the destination. They are always set by clients.

Some properties, like prefect and redelivery, can be configured as both connection options and destination oprions.

1.2. PREPARING TO USE MAVEN

Overview

This section gives a brief overview of how to prepare Maven for building Red Hat JBoss A-MQ projects and introduces the concept of Maven coordinates, which are used to locate Maven artifacts.

Prerequisites

In order to build a project using Maven, you must have the following prerequisites:

- *Maven installation*—Maven is a free, open source build tool from Apache. You can download the latest version from the Maven download page.
- *Network connection*—whilst performing a build, Maven dynamically searches external repositories and downloads the required artifacts on the fly. By default, Maven looks for repositories that are accessed over the Internet. You can change this behavior so that Maven will prefer searching repositories that are on a local network.

NOTE

Maven can run in an offline mode. In offline mode Maven will only look for artifacts in its local repository.

Adding the Red Hat JBoss A-MQ repository

In order to access artifacts from the Red Hat JBoss A-MQ Maven repository, you need to add it to Maven's `settings.xml` file. Maven looks for your `settings.xml` file in the `.m2` directory of the user's home directory. If there is not a user specified `settings.xml` file, Maven will use the system-level `settings.xml` file at `M2_HOME/conf/settings.xml`.

To add the JBoss A-MQ repository to Maven's list of repositories, you can either create a new **.m2/settings.xml** file or modify the system-level settings. In the **settings.xml** file, add the **repository** element for the JBoss A-MQ repository as shown in bold text in Example 1.1, "Adding the Red Hat JBoss A-MQ Repositories to Maven".

Example 1.1. Adding the Red Hat JBoss A-MQ Repositories to Maven

```
<settings>
    <profiles>
        <profile>
            <id>my-profile</id>
            <activation>
                <activeByDefault>true</activeByDefault>
            </activation>
            <repositories>
                <repository>
                    <id>fusesource</id>

<url>http://repo.fusesource.com/nexus/content/groups/public/</url>
                    <snapshots>
                        <enabled>false</enabled>
                    </snapshots>
                    <releases>
                        <enabled>true</enabled>
                    </releases>
                </repository>                <repository>
                    <id>fusesource.snapshot</id>

<url>http://repo.fusesource.com/nexus/content/groups/public-
snapshots/</url>
                    <snapshots>
                        <enabled>true</enabled>
                    </snapshots>
                    <releases>
                        <enabled>false</enabled>
                    </releases>
                </repository>
                <repository>
                    <id>apache-public</id>

<url>https://repository.apache.org/content/groups/public/</url>
                    <snapshots>
                        <enabled>true</enabled>
                    </snapshots>
                    <releases>
                        <enabled>true</enabled>
                    </releases>
                </repository>
                ...
            </repositories>
        </profile>
    </profiles>
    ...
</settings>
```

The preceding example also shows repository element for the following repositories:

- **`fusesource-snapshot`** repository—if you want to experiment with building your application using an Red Hat JBoss A-MQ snapshot kit, you can include this repository.
- **`apache-public`** repository—you might not always need this repository, but it is often useful to include it, because JBoss A-MQ depends on many of the artifacts from Apache.

Artifacts

The basic building block in the Maven build system is an *artifact*. The output of an artifact, after performing a Maven build, is typically an archive, such as a JAR or a WAR.

Maven coordinates

A key aspect of Maven functionality is the ability to locate artifacts and manage the dependencies between them. Maven defines the location of an artifact using the system of *Maven coordinates*, which uniquely define the location of a particular artifact. A basic coordinate tuple has the form, **{*`groupId`*, *`artifactId`*, *`version`*}**. Sometimes Maven augments the basic set of coordinates with the additional coordinates, *packaging* and *classifier*. A tuple can be written with the basic coordinates, or with the additional *packaging* coordinate, or with the addition of both the *packaging* and *classifier* coordinates, as follows:

```
groupId:artifactId:version
groupId:artifactId:packaging:version
groupId:artifactId:packaging:classifier:version
```

Each coordinate can be explained as follows:

groupId

Defines a scope for the name of the artifact. You would typically use all or part of a package name as a group ID—for example, **`org.fusesource.example`**.

artifactId

Defines the artifact name (relative to the group ID).

version

Specifies the artifact's version. A version number can have up to four parts: **`n.n.n.n`**, where the last part of the version number can contain non-numeric characters (for example, the last part of **`1.0-SNAPSHOT`** is the alphanumeric substring, **`0-SNAPSHOT`**).

packaging

Defines the packaged entity that is produced when you build the project. For OSGi projects, the packaging is **`bundle`**. The default value is **`jar`**.

classifier

Enables you to distinguish between artifacts that were built from the same POM, but have different content.

The group ID, artifact ID, packaging, and version are defined by the corresponding elements in an artifact's POM file. For example:

```
<project ... >
  ...
  <groupId>org.fusesource.example</groupId>
  <artifactId>bundle-demo</artifactId>
  <packaging>bundle</packaging>
  <version>1.0-SNAPSHOT</version>
  ...
</project>
```

For example, to define a dependency on the preceding artifact, you could add the following **dependency** element to a POM:

```
<project ... >
  ...
  <dependencies>
    <dependency>
      <groupId>org.fusesource.example</groupId>
      <artifactId>bundle-demo</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
  ...
</project>
```

NOTE

It is *not* necessary to specify the **bundle** package type in the preceding dependency, because a bundle is just a particular kind of JAR file and **jar** is the default Maven package type. If you do need to specify the packaging type explicitly in a dependency, however, you can use the **type** element.

CHAPTER 2. NATIVE ACTIVEMQ CLIENT APIS

Abstract

The Red Hat JBoss A-MQ client APIs follow the standard JMS pattern.

Regardless of the API in use, the pattern for establishing a connection between a messaging client and a message broker is the same. You must:

1. Get an instance of the Red Hat JBoss A-MQ connection factory.

   Depending on the environment, the application can create a new instance of the connection factory or use JNDI, or another mechanism, to look up the connection factory.

2. Use the connection factory to create a connection.

3. Get an instance of the destination used for sending or receiving messages.

   Destinations are administered objects that are typically created by the broker. The JBoss A-MQ allows clients to create destinations on-demand. You can also look up destinations using JNDI or another mechanism.

4. Use the connection to create a session.

   The session is the factory for creating producers and consumers. The session also is a factory for creating messages.

5. Use the session to create the message consumer or message producer.

6. Start the connection.

NOTE

You can add configuration information when creating connections and destinations.

2.1. NATIVE JMS CLIENT API

Overview

Red Hat JBoss A-MQ clients use the standard JMS APIs to interact with the message broker. Most of the configuration properties can be set using the connection URI and the destination specification used.

Developers can also use the JBoss A-MQ specific implementations to access JBoss A-MQ configuration features. Using these APIs will make your client non-portable.

The connection factory

The connection factory is an administered object that is created by the broker and used by clients wanting to connect to the broker. Each JMS provider is responsible for providing an implementation of the connection factory and the connection factory is stored in JNDI and retrieved by clients using a JNDI lookup.

The JBoss A-MQ connection factory, **`ActiveMQConnectionFactory`**, is used to create connections

to brokers and does not need to be looked up using JNDI. Instances are created using a broker URI that specifies one of the transport connectors configured on a broker and the connection factory will do the heavy lifting.

Example 2.1, "Connection Factory Constructors" shows the syntax for the available **ActiveMQConnectionFactory** constructors.

Example 2.1. Connection Factory Constructors

```
ActiveMQConnectionFactory(String brokerURI);
ActiveMQConnectionFactory(URI brokerURI);
ActiveMQConnectionFactory(String username,
                          String password,
                          String brokerURI);
ActiveMQConnectionFactory(String username,
                          String password,
                          URI brokerURI);
```

The broker URI also specifies connection configuration information. For details on how to construct a broker URI see the *Connection Reference*.

The connection

The connection object is created from the connection factory and is the object responsible for maintaining the link between the client and the broker. The connection object is used to create session objects that manage the resources used by message producers and message consumers.

For more applications the standard JMS **Connection** object will suffice. However, JBoss A-MQ does provide an implementation, **ActiveMQConnection**, that provides a number of additional methods for working with the broker. Using **ActiveMQConnection** will make your client code less portable between JMS providers.

The session

The session object is responsible for managing the resources for the message consumers and message producers implemented by a client. It is created from the connection, and is used to create message consumers, message producers, messages, and other objects involved in sending and receiving messages from a broker.

Example

Example 2.2, "JMS Producer Connection" shows code for creating a message producer that sends messages to the queue **EXAMPLE.FOO**.

Example 2.2. JMS Producer Connection

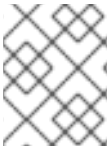
```
import org.apache.activemq.ActiveMQConnectionFactory;

import javax.jms.Connection;
import javax.jms.DeliveryMode;
import javax.jms.Destination;
import javax.jms.ExceptionListener;
import javax.jms.JMSException;
import javax.jms.Message;
```

```
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;

...

// Create a ConnectionFactory
ActiveMQConnectionFactory connectionFactory = new
ActiveMQConnectionFactory("tcp://localhost:61616");

// Create a Connection
Connection connection = connectionFactory.createConnection();

// Create a Session
Session session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);

// Create the destination
Destination destination = session.createQueue("EXAMPLE.FOO");

// Create a MessageProducer from the Session to the Queue
MessageProducer producer = session.createProducer(destination);

// Start the connection
connection.start();
```

2.2. NATIVE C++ CLIENT API

Overview

The CMS API is a C++ corollary to the JMS API. The CMS makes every attempt to maintain parity with the JMS API as possible. It only diverges when a JMS feature depended on features in the Java programming language. Even though there are some differences most are minor and for the most part CMS adheres to the JMS spec. Having a firm grasp on how JMS works should make using the C++ API easier.



NOTE

In order to use the CMS API, you will need to download the source and build it for your environment.

The connection factory

The first interface you will use in the CMS API is the **`ConnectionFactory`**. A **`ConnectionFactory`** allows you to create connections which maintain a connection to a message broker.

The simplest way to obtain an instance of a **`ConnectionFactory`** is to use the static **`createCMSConnectionFactory()`** method that all CMS provider libraries are required to implement. Example 2.3, “Creating a Connection Factory” demonstrates how to obtain a new **`ConnectionFactory`**.

Example 2.3. Creating a Connection Factory

```
std::auto_ptr<cms::ConnectionFactory> connectionFactory(
        cms::ConnectionFactory::createCMSConnectionFactory(
"tcp://127.0.0.1:61616" ) );
```

The **`createCMSConnectionFactory()`** takes a single string parameter which a URI that defines the connection that will be created by the factory. Additionally configuration information can be encoded in the URI. For details on how to construct a broker URI see the *Connection Reference*.

The connection

Once you've created a connection factory, you need to create a connection using the factory. A **`Connection`** is a object that manages the client's connection to the broker. Example 2.4, "Creating a Connection" shows the code to create a connection.

Example 2.4. Creating a Connection

```
std::auto_ptr<cms::Connection> connection( connectionFactory-
>createConnection() );
```

Upon creation the connection object attempts to connect to the broker, if the connection fails then an **`CMSException`** is thrown with a description of the error that occurred stored in its message property.

The connection interface defines an object that is the client's active connection to the CMS provider. In most cases the client will only create one connection object since it is considered a heavyweight object.

A connection serves several purposes:

- It encapsulates an open connection with a JMS provider. It typically represents an open TCP/IP socket between a client and a provider service daemon.
- Its creation is where client authentication takes place.
- It can specify a unique client identifier.
- It provides a **`ConnectionMetaData`** object.
- It supports an optional **`ExceptionListener`** object.

The session

After creating the connection the client must create a Session in order to create message producers and consumers. Example 2.5, "Creating a Session" shows how to create a session object from the connection.

Example 2.5. Creating a Session

```
std::auto_ptr<cms::Session> session( connection-
>createSession(cms::Session::CLIENT_ACKNOWLEDGE) );
```

When a client creates a session it must specify the mode in which the session will acknowledge the messages that it receives and dispatches. The modes supported are summarized in Table 2.1, “Support Acknowledgement Modes”.

Table 2.1. Support Acknowledgement Modes

Acknowledge Mode	Description
AUTO_ACKNOWLEDGE	The session automatically acknowledges a client's receipt of a message either when the session has successfully returned from a call to receive or when the message listener the session has called to process the message successfully returns.
CLIENT_ACKNOWLEDGE	The client acknowledges a consumed message by calling the message's acknowledge method. Acknowledging a consumed message acknowledges all messages that the session has consumed.
DUPS_OK_ACKNOWLEDGE	The session to lazily acknowledges the delivery of messages. This is likely to result in the delivery of some duplicate messages if the broker fails, so it should only be used by consumers that can tolerate duplicate messages. Use of this mode can reduce session overhead by minimizing the work the session does to prevent duplicates.
SESSION_TRANSACTED	The session is transacted and the acknowledge of messages is handled internally.
INDIVIDUAL_ACKNOWLEDGE	Acknowledges are applied to a single message only.

NOTE

If you do not specify an acknowledgement mode, the default is **AUTO_ACKNOWLEDGE**.

A session serves several purposes:

- It is a factory for producers and consumers.
- It supplies provider-optimized message factories.
- It is a factory for temporary topics and temporary queues.
- It provides a way to create a queue or a topic for those clients that need to dynamically manipulate provider-specific destination names.
- It supports a single series of transactions that combine work spanning its producers and consumers into atomic units.
- It defines a serial order for the messages it consumes and the messages it produces.

- It retains messages it consumes until they have been acknowledged.
- It serializes execution of message listeners registered with its message consumers.

NOTE

A session can create and service multiple producers and consumers.

Resources

The API reference documentation for the A-MQ C++ API can be found at http://activemq.apache.org/cms/api.html.

Example

Example 2.6, "CMS Producer Connection" shows code for creating a message producer that sends messages to the queue **EXAMPLE.FOO**.

Example 2.6. CMS Producer Connection

```
#include <decaf/lang/Thread.h>
#include <decaf/lang/Runnable.h>
#include <decaf/util/concurrent/CountDownLatch.h>
#include <decaf/lang/Integer.h>
#include <decaf/util/Date.h>
#include <activemq/core/ActiveMQConnectionFactory.h>
#include <activemq/util/Config.h>
#include <cms/Connection.h>
#include <cms/Session.h>
#include <cms/TextMessage.h>
#include <cms/BytesMessage.h>
#include <cms/MapMessage.h>
#include <cms/ExceptionListener.h>
#include <cms/MessageListener.h>
...

using namespace activemq::core;
using namespace decaf::util::concurrent;
using namespace decaf::util;
using namespace decaf::lang;
using namespace cms;
using namespace std;

...

// Create a ConnectionFactory
auto_ptr<ConnectionFactory> connectionFactory(
                ConnectionFactory::createCMSConnectionFactory(
"tcp://127.1.0.1:61616?wireFormat=openwire" ) );

// Create a Connection
connection = connectionFactory->createConnection();
connection->start();

// Create a Session
```

```
session = connection->createSession( Session::AUTO_ACKNOWLEDGE );
destination = session->createQueue( "EXAMPLE.FOO" );

// Create a MessageProducer from the Session to the Queue
producer = session->createProducer( destination );

...
```

2.3. NATIVE .NET CLIENT API

Overview

The Red Hat JBoss A-MQ NMS client is a .Net client that communicates with the JBoss A-MQ broker using its native Openwire protocol. This client supports advanced features such as failover, discovery, SSL, and message compression.

For complete details of using the .Net API see http://activemq.apache.org/nms/index.html.

NOTE

In order to use the NMS API, you will need to download the source and build it for your environment.

Resources

The API reference documentation for the A-MQ .Net API can be found at http://activemq.apache.org/nms/nms-api.html.

You can find examples of using the A-MQ .Net API at http://activemq.apache.org/nms/nms-examples.html.

Example

Example 2.7, “NMS Producer Connection” shows code for creating a message producer that sends messages to the queue **EXAMPLE.FOO**.

Example 2.7. NMS Producer Connection

```
using System;
using Apache.NMS;
using Apache.NMS.Util;
...

// NOTE: ensure the nmsprovider-activemq.config file exists in the
executable folder.
IConnectionFactory factory = new
ActiveMQ.ConnectionFactory("tcp://localhost:61616);

// Create a Connection
IConnection connection = factory.CreateConnection();

// Create a Session
```

```
ISession session = connection.CreateSession();

// Create the destination
IDestination destination = SessionUtil.GetDestination(session,
"queue://EXAMPLE.FOO");

// Create a message producer from the Session to the Queue
IMessageProducer producer = session.CreateProducer(destination);

// Start the connection
connection.Start();
...
```

2.4. CONFIGURING NMS.ACTIVEMQ

Abstract

All configuration settings can be accessed through URI-encoded options, which can be set either on a connection or on a destination. Using the URI syntax, you can configure virtually every facet of an NMS.ActiveMQ client.

Connection configuration using the generic NMSConnectionFactory class

Using the Generic **`NMSConnectionFactory`** class, you can configure an ActiveMQ endpoint as follows:

```
var cf = new NMSConnectionFactory(
    "activemq:tcp://localhost:61616?
wireFormat.tightEncodingEnabled=true");
```

Connection configuration using the ActiveMQ ConnectionFactory class

Using the ActiveMQ **`ConnectionFactory`** class, you can configure an ActiveMQ endpoint as follows:

```
var cf = new Apache.NMS.ActiveMQ.ConnectionFactory(
    "tcp://localhost:61616?wireFormat.tightEncodingEnabled=true");
```

Protocol variants

The following variants of the OpenWire protocol are supported:

Option Name	Description
`tcp`	Uses TCP/IP Sockets to connect to the Broker.
`ssl`	Uses TCP/IP Sockets to connect to the Broker with an added SSL layer.

`discovery`	Uses The Discovery Transport to find a Broker.
`failover`	Uses the Failover Transport to connect and reconnect to one or more Brokers.

TCP transport options

The `tcp` transport supports the following options:

Option Name	Default	Description
`transport.useLogging`	`false`	Log data that is sent across the Transport.
`transport.receiveBufferSize`	`8192`	Amount of Data to buffer from the Socket.
`transport.sendBufferSize`	`8192`	Amount of Data to buffer before writing to the Socket.
`transport.receiveTimeout`	`0`	Time to wait for more data, zero means wait infinitely.
`transport.sendTimeout`	`0`	Timeout on sends, 0 means wait forever for completion.
`transport.requestTimeout`	`0`	Time to wait before a Request Command is considered to have failed.

Failover transport options

The `failover` transport supports the following options:

Option Name	Default	Description
`transport.timeout`	`-1`	Time that a send operation blocks before failing.
`transport.initialReconnectDelay`	`10`	Time in Milliseconds that the transport waits before attempting to reconnect the first time.
`transport.maxReconnectDelay`	`30000`	The max time in Milliseconds that the transport will wait before attempting to reconnect.

transport.backOffMultiplier	**2**	The amount by which the reconnect delay will be multiplied by if useExponentialBackOff is enabled.
transport.useExponentialBackOff	**true**	Should the delay between connection attempt grow on each try up to the max reconnect delay.
transport.randomize	**true**	Should the Uri to connect to be chosen at random from the list of available Uris.
transport.maxReconnectAttempts	**0**	Maximum number of time the transport will attempt to reconnect before failing (0 means infinite retries)
transport.startupMaxReconnectAttempts	**0**	Maximum number of time the transport will attempt to reconnect before failing when there has never been a connection made. (0 means infinite retries) *(included in NMS.ActiveMQ v1.5.0+)*
transport.reconnectDelay	**10**	The delay in milliseconds that the transport waits before attempting a reconnection.
transport.backup	**false**	Should the Failover transport maintain hot backups.
transport.backupPoolSize	**1**	If enabled, how many hot backup connections are made.
transport.trackMessages	**false**	keep a cache of in-flight messages that will flushed to a broker on reconnect
transport.maxCacheSize	**256**	Number of messages that are cached if trackMessages is enabled.
transport.updateURIsSupported	**true**	Update the list of known brokers based on BrokerInfo messages sent to the client.

Connection Options

Connection options can either be set using either the **connection.** prefix or the **nms.** prefix (in a similar way to the Java client's **jms.** prefixed settings).

Option Name	Default	Description
`connection.AsyncSend`	`false`	Are message sent Asynchronously.
`connection.AsyncClose`	`true`	Should the close command be sent Asynchronously
`connection.AlwaysSyncSend`	`false`	Causes all messages a Producer sends to be sent Asynchronously.
`connection.CopyMessageOnSend`	`true`	Copies the Message objects a Producer sends so that the client can reuse Message objects without affecting an in-flight message.
`connection.ProducerWindowSize`	`0`	The ProducerWindowSize is the maximum number of bytes in memory that a producer will transmit to a broker before waiting for acknowledgement messages from the broker that it has accepted the previously sent messages. In other words, this how you configure the producer flow control window that is used for async sends where the client is responsible for managing memory usage. The default value of 0 means no flow control at the client. See also Producer Flow Control
`connection.useCompression`	`false`	Should message bodies be compressed before being sent.
`connection.sendAcksAsync`	`false`	Should message acks be sent asynchronously
`connection.messagePrioritySupported`	`true`	Should messages be delivered to the client based on the value of the Message Priority header.
`connection.dispatchAsync`	`false`	Should the broker dispatch messages asynchronously to the connection's consumers.
`connection.watchTopicAdvisories`	`true`	Should the client watch for advisory messages from the broker to track the creation and deletion of temporary destinations.

OpenWire options

The following options are used to configure the OpenWire protocol:

Option Name	Default	Description
`wireFormat.stackTraceEnabled`	`false`	Should the stack trace of exception that occur on the broker be sent to the client? Only used by openwire protocol.
`wireFormat.cacheEnabled`	`false`	Should commonly repeated values be cached so that less marshalling occurs? Only used by openwire protocol.
`wireFormat.tcpNoDelayEnabled`	`false`	Does not affect the wire format, but provides a hint to the peer that TCP nodelay should be enabled on the communications Socket. Only used by openwire protocol.
`wireFormat.sizePrefixDisabled`	`false`	Should serialized messages include a payload length prefix? Only used by openwire protocol.
`wireFormat.tightEncodingEnabled`	`false`	Should wire size be optimized over CPU usage? Only used by the openwire protocol.
`wireFormat.maxInactivityDuration`	`30000`	The maximum inactivity duration (before which the socket is considered dead) in milliseconds. On some platforms it can take a long time for a socket to appear to die, so we allow the broker to kill connections if they are inactive for a period of time. Use by some transports to enable a keep alive heart beat feature. Set to a value <= 0 to disable inactivity monitoring.
`maxInactivityDurationInitalDelay`	`10000`	The initial delay in starting the maximum inactivity checks (and, yes, the word 'Inital' is supposed to be misspelled like that)

Destination configuration

A destination URI can be configured as shown in the following example:

```
d = session.CreateTopic("com.foo?
consumer.prefetchSize=2000&consumer.noLocal=true");
```

General options

The following destination URI options are generally supported for all protocols:

Option Name	Default	Description
`consumer.prefetchSize`	`1000`	The number of message the consumer will prefetch.
`consumer.maximumPendingMessageLimit`	`0`	Use to control if messages are dropped if a slow consumer situation exists.
`consumer.noLocal`	`false`	Same as the noLocal flag on a Topic consumer. Exposed here so that it can be used with a queue.
`consumer.dispatchAsync`	`false`	Should the broker dispatch messages asynchronously to the consumer.
`consumer.retroactive`	`false`	Is this a Retroactive Consumer.
`consumer.selector`	`null`	JMS Selector used with the consumer.
`consumer.exclusive`	`false`	Is this an Exclusive Consumer.
`consumer.priority`	`0`	Allows you to configure a Consumer Priority.

OpenWire specific options

The following destination URI options are supported *only* for the OpenWire protocol:

Option Name	Default	Description
`consumer.browser`	`false`	
`consumer.networkSubscription`	`false`	

consumer.optimizedAcknowledge	**false**	Enables an optimised acknowledgement mode where messages are acknowledged in batches rather than individually. Alternatively, you could use **Session.DUPS_OK_ACKNOWLEDGE** acknowledgement mode for the consumers which can often be faster. **WARNING:** enabling this issue could cause some issues with auto-acknowledgement on reconnection
consumer.noRangeAcks	**false**	
consumer.retroactive	**false**	Sets whether or not retroactive consumers are enabled. Retroactive consumers allow non-durable topic subscribers to receive old messages that were published before the non-durable subscriber started.

CHAPTER 3. QPID JMS CLIENT API

WARNING

The Qpid JMS client API is a technical preview only, and is *not* supported in JBoss A-MQ 6.1.

3.1. GETTING STARTED WITH AMQP

3.1.1. Introduction to AMQP

What is AMQP?

The Advanced Message Queuing Protocol (AMQP) is an open standard messaging system, which has been designed to facilitate interoperability between messaging systems. The key features of AMQP are:

- Open standard (defined by the OASIS AMQP Technical Committee)
- Defines a wire protocol
- Defines APIs for multiple languages (C++, Java)
- Interoperability between different AMQP implementations

WARNING

The Qpid JMS client API is a technical preview only, and is *not* supported in JBoss A-MQ 6.1.

JMS is an API

It is interesting to contrast the Java Message Service (JMS) with AMQP. The JMS is first and foremost an API and is designed to enable Java code to be portable between different messaging products. JMS does *not* describe how to implement a messaging service (although it imposes significant constraints on the messaging behaviour), nor does JMS specify any details of the wire protocol for transmitting messages. Consequently, different JMS implementations are generally *not* interoperable.

AMQP is a wire protocol

AMQP, on the other hand, does specify complete details of a wire protocol for messaging (in an open standard). Moreover, AMQP also specifies APIs in several different programming languages (for example, Java and C++). An implementation of AMQP is therefore much more constrained than a comparable JMS implementation. One of the benefits of this is that different AMQP implementations ought to be interoperable with each other.

AMQP-to-JMS requires message conversion

If you want to bridge from an AMQP messaging system to a JMS messaging system, the messages must be converted from AMQP format to JMS format. Usually, this involves a fairly lightweight conversion, because the message body can usually be left intact while message headers are mapped to equivalent headers.

AMQP support in JBoss A-MQ

AMQP support in JBoss A-MQ is based on the following main components:

- *AMQP endpoint in the broker*—an endpoint on the broker that supports the AMQP wire protocol and implicitly converts messages between AMQP format and JMS format (which is used inside the JBoss A-MQ broker).
- *AMQP JMS client*—is based on the Apache Qpid JMS client , which is compatible with the broker's AMQP endpoint.

Getting started with AMQP

To run a simple demonstration of AMQP in JBoss A-MQ, you need to set up the following parts of the application:

- *Configure the broker to use AMQP*—to enable AMQP in the broker, add an AMQP endpoint to the broker's configuration. This implicitly activates the broker's AMQP integration, ensuring that incoming messages are converted from AMQP message format to JMS message format, as required.
- *Implement the AMQP clients*—the AMQP clients are based on the Apache Qpid JMS client libraries.

3.1.2. Configuring the Broker for AMQP

Overview

Configuring the broker to use AMQP is relatively straightforward in JBoss A-MQ, because the required AMQP packages are pre-installed in the container. There are essentially two main points you need to pay attention to:

- Make sure that you have appropriate user entries in the `etc/users.properties` file, so that the AMQP clients will be able to log on to the broker.
- Add an AMQP endpoint to the broker (by inserting a `transportConnector` element into the broker's XML configuration).

Steps to configure the broker

Perform the following steps to configure the broker with an AMQP endpoint:

1. This example assumes that you are working with a fresh install of a standalone JBoss A-MQ broker, *InstallDir*.
2. Define a JAAS user for the AMQP clients, so that the AMQP clients can authenticate themselves to the broker using JAAS security (security is enabled by default in the broker). Edit the *`InstallDir`*`/etc/users.properties` file and add a new user entry, as follows:

```
#
# This file contains the valid users who can log into JBoss A-MQ.
# Each line has to be of the format:
#
# USER=PASSWORD,ROLE1,ROLE2,...
#
# All users and roles entered in this file are available after JBoss
A-MQ startup
# and modifiable via the JAAS command group. These users reside in a
JAAS domain
# with the name "karaf"..
#
# You must have at least one users to be able to access JBoss A-MQ
resources

#admin=admin,admin
amqpuser=secret
```

At this point, you can add entries for any other secure users you want. In particular, it is advisable to have at least one user with the **admin** role, so that you can log into the secure container remotely (remembering to choose a *secure* password for the admin user).

3. Add an AMQP endpoint to the broker configuration. Edit the broker configuration file, ***InstallDir*/etc/activemq.xml**. As shown in the following XML fragment, add the highlighted **transportConnector** element as a child of the **transportConnectors** element in the broker configuration:

```
<beans ...>
    ...
    <broker ...>
        ...
        <transportConnectors>
            <transportConnector name="openwire"
uri="tcp://0.0.0.0:0?maximumConnections=1000"/>
 <transportConnector name="amqp" uri="amqp://0.0.0.0:5672"/>
        </transportConnectors>
    </broker>

</beans>
```

4. To start the broker, open a new command prompt, change directory to ***InstallDir*/bin**, and enter the following command:

```
./amq
```

Message conversion

The AMQP endpoint in the broker implicitly converts incoming AMQP format messages into JMS format messages (which is the format in which messages are stored in the broker). The endpoint configuration shown here uses the default options for this conversion.

Reference

For full details of how to configure an AMQP endpoint in the broker, see the "Advanced Message Queueing Protocol (AMQP)" chapter from the *Connection Reference*. This also includes details of how to customize the message conversion from AMQP format to JMS format.

3.1.3. AMQP Example Clients

Overview

This section explains how to implement two basic AMQP clients: an AMQP producer client, which sends messages to a queue on the broker; and an AMQP consumer client, which pulls messages off the queue on the broker. The clients themselves use generic JMS code to access the messaging system. The key details of the AMQP configuration are retrieved using JNDI properties.

Prerequisites

Before building the example clients, you must install and configure the Apache Maven build tool, as described in Section 1.2, “Preparing to use Maven”.

AMQP connection URI

The critical piece of configuration for establishing a connection with the broker is the AMQP URI (defined as a JNDI property in the **`jndi.properties`** file, in this demonstration). This example uses the following AMQP URI for the clients:

```
amqp://amqpuser:secret@localhost/test/?brokerlist='tcp://localhost:5672'
```

The first part of the URI, **`amqpuser:secret@localhost`**, has the format ***`Username`*`:`*`Password`*`@`*`ClientID`**. In order to authenticate the clients successfully with the broker, it is essential that there is a corresponding JAAS user entry on the broker side.

The **`brokerlist`** option defines the location of the AMQP port on the broker, which is **`tcp://localhost:5672`** for this example.

Steps to implement and run the AMQP clients

Perform the following steps to implement and run an AMQP producer client and an AMQP consumer client:

1. At any convenient location, create the directory, **`activemq-amqp-example`**, to hold the example code:

   ```
   mkdir activemq-amqp-example
   ```

2. Create the directory hierarchy for the example code. Change directory to **`activemq-amqp-example`** and run the following script at a command prompt:

   ```
   mkdir src
   mkdir src/main
   mkdir src/main/java
   mkdir src/main/java/org
   mkdir src/main/java/org/fusebyexample
   mkdir src/main/java/org/fusebyexample/activemq
   mkdir src/main/resources
   ```

–

After executing the preceding commands, you should have the following directory structure for the **`activemq-amqp-example`** project:

```
activemq-amqp-example/
    src/
        main/
            java/
                org/fusebyexample/activemq
            resources/
```

3. Create a POM file for the Maven project. Using a text editor, create a new file, **`activemq-amqp-example/pom.xml`**, with the following contents:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.fusebyexample.activemq</groupId>
    <artifactId>activemq-amqp-example</artifactId>
    <version>5.8.0</version>

    <name>ActiveMQ AMQP Example</name>

    <properties>
        <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>

        <activemq.version>5.9.0.redhat-610379</activemq.version>

        <qpid.version>0.22</qpid.version>

        <slf4j-version>1.6.6</slf4j-version>
        <log4j-version>1.2.17</log4j-version>
    </properties>

    <repositories>
        <repository>
            <id>fusesource.releases</id>
            <name>FuseSource Release Repository</name>

<url>http://repo.fusesource.com/nexus/content/repositories/releases<
/url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </repository>
    </repositories>
    <pluginRepositories>
        <pluginRepository>
```

```
            <id>fusesource.releases</id>
            <name>FuseSource Release Repository</name>

<url>http://repo.fusesource.com/nexus/content/repositories/releases<
/url>
            <releases>
                <enabled>true</enabled>
            </releases>
            <snapshots>
                <enabled>false</enabled>
            </snapshots>
        </pluginRepository>
    </pluginRepositories>

    <dependencies>
        <dependency>
            <groupId>org.apache.qpid</groupId>
            <artifactId>qpid-amqp-1-0-client-jms</artifactId>
            <version>${qpid.version}</version>
        </dependency>

        <dependency>
            <groupId>org.apache.geronimo.specs</groupId>
            <artifactId>geronimo-jms_1.1_spec</artifactId>
            <version>1.1.1</version>
        </dependency>

        <!-- Logging -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>${slf4j-version}</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>${slf4j-version}</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>${log4j-version}</version>
            <scope>runtime</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <source>1.6</source>
                    <target>1.6</target>
```

```
                </configuration>
            </plugin>
        </plugins>
    </build>

    <profiles>
        <profile>
            <id>consumer</id>
            <build>
                <defaultGoal>package</defaultGoal>

                <plugins>
                    <plugin>
                        <groupId>org.codehaus.mojo</groupId>
                        <artifactId>exec-maven-plugin</artifactId>
                        <version>1.2.1</version>
                        <executions>
                            <execution>
                                <phase>package</phase>
                                <goals>
                                    <goal>java</goal>
                                </goals>
                                <configuration>

<mainClass>org.fusebyexample.activemq.SimpleConsumer</mainClass>
                                </configuration>
                            </execution>
                        </executions>
                    </plugin>
                </plugins>
            </build>
        </profile>

        <profile>
            <id>producer</id>
            <build>
                <defaultGoal>package</defaultGoal>

                <plugins>
                    <plugin>
                        <groupId>org.codehaus.mojo</groupId>
                        <artifactId>exec-maven-plugin</artifactId>
                        <version>1.2.1</version>
                        <executions>
                            <execution>
                                <phase>package</phase>
                                <goals>
                                    <goal>java</goal>
                                </goals>
                                <configuration>

<mainClass>org.fusebyexample.activemq.SimpleProducer</mainClass>
                                </configuration>
                            </execution>
                        </executions>
                    </plugin>
```

```
                </plugins>
            </build>
        </profile>
    </profiles>

</project>
```

4. Define the Java implementation of an AMQP consumer class, **SimpleConsumer**. Using a text editor, create the **SimpleConsumer.java** file under the **activemq-amqp-example/src/main/java/org/fusebyexample/activemq/** directory, with the following contents:

```
package org.fusebyexample.activemq;

import javax.jms.*;
import javax.naming.Context;
import javax.naming.InitialContext;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class SimpleConsumer {
    private static final Logger LOG =
LoggerFactory.getLogger(SimpleConsumer.class);

    private static final Boolean NON_TRANSACTED = false;
    private static final String CONNECTION_FACTORY_NAME =
"myJmsFactory";
    private static final String DESTINATION_NAME = "queue/simple";
    private static final int MESSAGE_TIMEOUT_MILLISECONDS = 120000;

    public static void main(String args[]) {
        Connection connection = null;

        try {
            // JNDI lookup of JMS Connection Factory and JMS
Destination
            Context context = new InitialContext();
            ConnectionFactory factory = (ConnectionFactory)
context.lookup(CONNECTION_FACTORY_NAME);
            Destination destination = (Destination)
context.lookup(DESTINATION_NAME);

            connection = factory.createConnection();
            connection.start();

            Session session =
connection.createSession(NON_TRANSACTED, Session.AUTO_ACKNOWLEDGE);
            MessageConsumer consumer =
session.createConsumer(destination);

            LOG.info("Start consuming messages from " +
destination.toString() + " with " + MESSAGE_TIMEOUT_MILLISECONDS +
"ms timeout");

            // Synchronous message consumer
```

```
            int i = 1;
            while (true) {
                Message message =
consumer.receive(MESSAGE_TIMEOUT_MILLISECONDS);
                if (message != null) {
                    if (message instanceof TextMessage) {
                        String text = ((TextMessage)
message).getText();
                        LOG.info("Got " + (i++) + ". message: " +
text);
                    }
                } else {
                    break;
                }
            }

            consumer.close();
            session.close();
        } catch (Throwable t) {
            LOG.error("Error receiving message", t);
        } finally {
            // Cleanup code
            // In general, you should always close producers,
consumers,
            // sessions, and connections in reverse order of
creation.
            // For this simple example, a JMS connection.close will
            // clean up all other resources.
            if (connection != null) {
                try {
                    connection.close();
                } catch (JMSException e) {
                    LOG.error("Error closing connection", e);
                }
            }
        }
    }
}
```

5. Define the Java implementation of an AMQP producer class, **SimpleProducer**. Using a text editor, create the **SimpleProducer.java** file under the **activemq-amqp-example/src/main/java/org/fusebyexample/activemq/** directory, with the following contents:

```
package org.fusebyexample.activemq;

import javax.jms.*;
import javax.naming.Context;
import javax.naming.InitialContext;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class SimpleProducer {
    private static final Logger LOG =
LoggerFactory.getLogger(SimpleProducer.class);
```

```
    private static final Boolean NON_TRANSACTED = false;
    private static final long MESSAGE_TIME_TO_LIVE_MILLISECONDS = 0;
    private static final int MESSAGE_DELAY_MILLISECONDS = 100;
    private static final int NUM_MESSAGES_TO_BE_SENT = 100;
    private static final String CONNECTION_FACTORY_NAME =
"myJmsFactory";
    private static final String DESTINATION_NAME = "queue/simple";

    public static void main(String args[]) {
        Connection connection = null;

        try {
            // JNDI lookup of JMS Connection Factory and JMS
Destination
            Context context = new InitialContext();
            ConnectionFactory factory = (ConnectionFactory)
context.lookup(CONNECTION_FACTORY_NAME);
            Destination destination = (Destination)
context.lookup(DESTINATION_NAME);

            connection = factory.createConnection();
            connection.start();

            Session session =
connection.createSession(NON_TRANSACTED, Session.AUTO_ACKNOWLEDGE);
            MessageProducer producer =
session.createProducer(destination);


producer.setTimeToLive(MESSAGE_TIME_TO_LIVE_MILLISECONDS);

            for (int i = 1; i <= NUM_MESSAGES_TO_BE_SENT; i++) {
                TextMessage message = session.createTextMessage(i +
". message sent");
                LOG.info("Sending to destination: " +
destination.toString() + " this text: '" + message.getText());
                producer.send(message);
                Thread.sleep(MESSAGE_DELAY_MILLISECONDS);
            }

            // Cleanup
            producer.close();
            session.close();
        } catch (Throwable t) {
            LOG.error("Error sending message", t);
        } finally {
            // Cleanup code
            // In general, you should always close producers,
consumers,
            // sessions, and connections in reverse order of
creation.
            // For this simple example, a JMS connection.close will
            // clean up all other resources.
            if (connection != null) {
                try {
```

```
                    connection.close();
                } catch (JMSException e) {
                    LOG.error("Error closing connection", e);
                }
            }
        }
    }
}
```

6. Configure the JNDI properties for the AMQP clients. Using a text editor, create the **jndi.properties** file under the **activemq-amqp-example/src/main/resources/** directory, with the following contents:

```
#
# Copyright (C) Red Hat, Inc.
# http://www.redhat.com
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#

# JNDI properties file to setup the JNDI server within ActiveMQ

#
# Default JNDI properties settings
#
java.naming.factory.initial =
org.apache.qpid.amqp_1_0.jms.jndi.PropertiesFileInitialContextFactor
y
java.naming.provider.url = src/main/resources/jndi.properties

#
# Set the connection factory name(s) as well as the destination
names. The connection factory name(s)
# as well as the second part (after the dot) of the left hand side
of the destination definition
# must be used in the JNDI lookups.
#
connectionfactory.myJmsFactory =
amqp://amqpuser:secret@localhost/test/?
brokerlist='tcp://localhost:5672'

queue.queue/simple = test.queue.simple
```

7. Configure the client logging with log4j. Using a text editor, create the **log4j.properties** file under the **activemq-amqp-example/src/main/resources/** directory, with the following contents:

```
#
# Copyright (C) Red Hat, Inc.
# http://www.redhat.com
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#

#
# The logging properties used by the standalone ActiveMQ broker
#
log4j.rootLogger=INFO, stdout

# CONSOLE appender
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{HH:mm:ss} %-5p
%m%n

# Log File appender
log4j.appender.logfile=org.apache.log4j.FileAppender
log4j.appender.logfile.layout=org.apache.log4j.PatternLayout
log4j.appender.logfile.layout.ConversionPattern=%d [%-15.15t] %-5p
%-30.30c{1} - %m%n
log4j.appender.logfile.file=./target/log/exercises.log
log4j.appender.logfile.append=true

#
#  You can change logger levels here.
#
log4j.logger.org.apache.activemq=INFO
log4j.logger.org.apache.activemq.spring=WARN
```

8. Make sure that the broker is already configured and running with an AMQP endpoint, as described in Section 3.1.2, “Configuring the Broker for AMQP”.

9. Run the AMQP producer client as follows. Open a new command prompt, change directory to the project directory, **activemq-amqp-example/**, and enter the following Maven command:

```
mvn -P producer
```

After building the code (and downloading any packages required by Maven), this target proceeds to run the producer client, which sends 100 messages to the **test.queue.simple** queue in the broker. If the producer runs successfully, you should see output like the following in the console window:

```
13:31:43 INFO  Sending to destination:
org.apache.qpid.amqp_1_0.jms.impl.QueueImpl@fabdfd0b this text: '1.
message sent
13:31:43 INFO  Sending to destination:
org.apache.qpid.amqp_1_0.jms.impl.QueueImpl@fabdfd0b this text: '2.
message sent
13:31:43 INFO  Sending to destination:
org.apache.qpid.amqp_1_0.jms.impl.QueueImpl@fabdfd0b this text: '3.
message sent
...
13:31:53 INFO  Sending to destination:
org.apache.qpid.amqp_1_0.jms.impl.QueueImpl@fabdfd0b this text: '99.
message sent
13:31:53 INFO  Sending to destination:
org.apache.qpid.amqp_1_0.jms.impl.QueueImpl@fabdfd0b this text:
'100. message sent
```

10. Run the AMQP consumer client as follows. Open a new command prompt, change directory to the project directory, **activemq-amqp-example/**, and enter the following Maven command:

```
mvn -P consumer
```

After building the code, this target proceeds to run the consumer client, which reads messages from the **test.queue.simple** queue. You should see output like the following in the console window:

```
13:32:12 INFO  Start consuming messages from
org.apache.qpid.amqp_1_0.jms.impl.QueueImpl@fabdfd0b with 120000ms
timeout
13:32:12 INFO  Got 1. message: 1. message sent
13:32:12 INFO  Got 2. message: 2. message sent
13:32:12 INFO  Got 3. message: 3. message sent
...
13:32:12 INFO  Got 99. message: 99. message sent
13:32:12 INFO  Got 100. message: 100. message sent
```

3.2. A SIMPLE MESSAGING PROGRAM IN JAVA JMS

The following program shows how to send and receive a message using the Qpid JMS client. JMS programs typically use JNDI to obtain connection factory and destination objects which the application needs. In this way the configuration is kept separate from the application code itself.

In this example, we create a JNDI context using a properties file, use the context to lookup a connection factory, create and start a connection, create a session, and lookup a destination from the JNDI context. Then we create a producer and a consumer, send a message with the producer and receive it with the consumer. This code should be straightforward for anyone familiar with Java JMS.

Example 3.1. "Hello world!" in Java

```
package org.apache.qpid.example.jmsexample.hello;

import javax.jms.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.util.Properties;

public class Hello {

public Hello() {
}

public static void main(String[] args) {
Hello producer = new Hello();
producer.runTest();
}

private void runTest() {
try {
Properties properties = new Properties();

properties.load(this.getClass().getResourceAsStream("hello.properties"))
(1);
(2)   Context context = new InitialContext(properties);

ConnectionFactory connectionFactory
       = (ConnectionFactory)
(3)context.lookup("qpidConnectionfactory");
(4)   Connection connection = connectionFactory.createConnection();
(5)    connection.start();

Session
(6)session=connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
   Destination destination = (Destination)
(7)context.lookup("topicExchange");

MessageProducer messageProducer =
(8)session.createProducer(destination);
   MessageConsumer messageConsumer =
(9)session.createConsumer(destination);

TextMessage message = session.createTextMessage("Hello world!");
messageProducer.send(message);

(10)   message = (TextMessage)messageConsumer.receive();
   System.out.println(message.getText());

(11)   connection.close();
(12)    context.close();
       }
catch (Exception exp) {
exp.printStackTrace();
}
}
}
```

1. Loads the JNDI properties file, which specifies connection properties, queues, topics, and addressing options. See Section 3.3, “Apache Qpid JNDI Properties for AMQP Messaging” for details.
2. Creates the JNDI initial context.
3. Creates a JMS connection factory for Qpid.
4. Creates a JMS connection.
5. Activates the connection.
6. Creates a session. This session is not transactional (transactions='false'), and messages are automatically acknowledged.
7. Creates a destination for the topic exchange, so senders and receivers can use it.
8. Creates a producer that sends messages to the topic exchange.
9. Creates a consumer that reads messages from the topic exchange.
10. Reads the next available message.
11. Closes the connection, all sessions managed by the connection, and all senders and receivers managed by each session.
12. Closes the JNDI context.

The contents of the hello.properties file are shown below.

Example 3.2. JNDI Properties File for "Hello world!" example

```
java.naming.factory.initial =
org.apache.qpid.jndi.PropertiesFileInitialContextFactory

# connectionfactory.[jndiname] = [ConnectionURL]
connectionfactory.qpidConnectionfactory =
1 amqp://guest:guest@clientid/test?brokerlist='tcp://localhost:5672'

# destination.[jndiname] = [address_string]
2 destination.topicExchange = amq.topic
```

1. Defines a connection factory from which connections can be created. The syntax of a ConnectionURL is given in Section 3.3, “Apache Qpid JNDI Properties for AMQP Messaging”.
2. Defines a destination for which MessageProducers and/or MessageConsumers can be created to send and receive messages. The value for the destination in the properties file is an address string. In the JMS implementation MessageProducers are analogous to senders in the Qpid Message API, and MessageConsumers are analogous to receivers.

3.3. APACHE QPID JNDI PROPERTIES FOR AMQP MESSAGING

Apache Qpid defines JNDI properties that can be used to specify JMS Connections and Destinations. Here is a typical JNDI properties file:

Example 3.3. JNDI Properties File

```
java.naming.factory.initial =
org.apache.qpid.jndi.PropertiesFileInitialContextFactory

# connectionfactory.[jndiname] = [ConnectionURL]
connectionfactory.qpidConnectionfactory =
amqp://guest:guest@clientid/test?brokerlist='tcp://localhost:5672'

# destination.[jndiname] = [address_string]
destination.topicExchange = amq.topic
```

The following sections describe the JNDI properties that Qpid uses.

3.3.1. JNDI Properties for Apache Qpid

Apache Qpid supports the properties shown in the following table:

Table 3.1. JNDI Properties supported by Apache Qpid

Property	Purpose
connectionfactory.<jndiname>	The Connection URL that the connection factory uses to perform connections.
queue.<jndiname>	A JMS queue, which is implemented as an amq.direct exchange in Apache Qpid.
topic.<jndiname>	A JMS topic, which is implemented as an amq.topic exchange in Apache Qpid.
destination.<jndiname>	Can be used for defining all amq destinations, queues, topics and header matching, using an address string. [a]

[a] Binding URLs, which were used in earlier versions of the Qpid Java JMS client, can still be used instead of address strings.

3.3.2. Connection URLs

In JNDI properties, a Connection URL specifies properties for a connection. The format for a Connection URL is:

```
amqp://[<user>:<pass>@][<clientid>]<virtualhost>[?<option>='<value>'[&
<option>='<value>']]
```

For instance, the following Connection URL specifies a user name, a password, a client ID, a virtual host ("test"), a broker list with a single broker, and a TCP host with the host name “localhost” using port 5672:

```
amqp://username:password@clientid/test?brokerlist='tcp://localhost:5672'
```

Apache Qpid supports the following properties in Connection URLs:

Table 3.2. Connection URL Properties

Option	Type	Description
brokerlist	see below	List of one or more broker addresses.
maxprefetch	integer	The maximum number of pre-fetched messages per consumer. If not specified, default value of 500 is used. Note: You can also set the default per-consumer prefetch value on a client-wide basis by configuring the client using Java system properties.
sync_publish	{'persistent' | 'all'}	A sync command is sent after every persistent message to guarantee that it has been received; if the value is 'persistent', this is done only for persistent messages.
sync_ack	boolean	A sync command is sent after every acknowledgement to guarantee that it has been received.
use_legacy_map_msg_format	boolean	If you are using JMS Map messages and deploying a new client with any JMS client older than 0.8 release, you must set this to true to ensure the older clients can understand the map message encoding.

Option	Type	Description
failover	{'singlebroker' | 'roundrobin' | 'failover_exchange' | 'nofailover' | '<class>'}	This option controls failover behaviour. The method **singlebroker** uses only the first broker in the list, **roundrobin** will try each broker given in the broker list until a connection is established, **failover_exchange** connects to the initial broker given in the broker URL and will receive membership updates via the failover exchange. **nofailover** disables all retry and failover logic. Any other value is interpreted as a classname which must implement the **org.apache.qpid.jms.failover.FailoverMethod** interface. The broker list options **retries** and **connectdelay** (described below) determine the number of times a connection to a broker will be retried and the the length of time to wait between successive connection attempts before moving on to the next broker in the list. The failover option **cyclecount** controls the number of times to loop through the list of available brokers before finally giving up. Defaults to **roundrobin** if the brokerlist contains multiple brokers, or **singlebroker** otherwise.
ssl	boolean	If **ssl='true'**, use SSL for all broker connections. Overrides any per-broker settings in the brokerlist (see below) entries. If not specified, the brokerlist entry for each given broker is used to determine whether SSL is used. Introduced in version 0.22.

Broker lists are specified using a URL in this format:

```
brokerlist=<transport>://<host>[:<port>](?<param>='<value>')(&
<param>='<value>')*
```

For instance, this is a typical broker list:

```
brokerlist='tcp://localhost:5672'
```

A broker list can contain more than one broker address; if so, the connection is made to the first broker in the list that is available. In general, it is better to use the failover exchange when using multiple brokers, since it allows applications to fail over if a broker goes down.

Example 3.4. Broker Lists

A broker list can specify properties to be used when connecting to the broker, such as security options. This broker list specifies options for a Kerberos connection using GSSAPI:

```
amqp://guest:guest@test/test?sync_ack='true'
&brokerlist='tcp://ip1:5672?sasl_mechs='GSSAPI''
```

This broker list specifies SSL options:

```
amqp://guest:guest@test/test?sync_ack='true'
&brokerlist='tcp://ip1:5672?ssl='true'&ssl_cert_alias='cert1''
```

This broker list specifies two brokers using the connectdelay and retries broker options. It also illustrates the failover connection URL property.

```
amqp://guest:guest@/test?failover='roundrobin?cyclecount='2''
&brokerlist='tcp://ip1:5672?
retries='5'&connectdelay='2000';tcp://ip2:5672?
retries='5'&connectdelay='2000''
```

The following broker list options are supported.

Table 3.3. Broker List Options

Option	Type	Description
heartbeat	integer	frequency of heartbeat messages (in seconds)

Option	Type	Description
sasl_mechs	--	For secure applications, we suggest CRAM-MD5, DIGEST-MD5, or GSSAPI. The ANONYMOUS method is not secure. The PLAIN method is secure only when used together with SSL. For Kerberos, sasl_mechs must be set to GSSAPI, sasl_protocol must be set to the principal for the qpidd broker, e.g. qpidd/, and sasl_server must be set to the host for the SASL server, e.g. sasl.com. SASL External is supported using SSL certification, e.g. **ssl='true'&sasl_mechs='EXTERNAL'**
sasl_encryption	Boolean	If **sasl_encryption='true'**, the JMS client attempts to negotiate a security layer with the broker using GSSAPI to encrypt the connection. Note that for this to happen, GSSAPI must be selected as the sasl_mech.
sasl_protocol	--	Used only for Kerberos. **sasl_protocol** must be set to the principal for the qpidd broker, e.g. **qpidd/**
sasl_server	--	For Kerberos, sasl_mechs must be set to GSSAPI, sasl_server must be set to the host for the SASL server, e.g. **sasl.com**.
trust_store	--	path to trust store
trust_store_password	--	Trust store password
key_store	--	path to key store
key_store_password	--	key store password

Option	Type	Description
`ssl`	Boolean	If `ssl='true'`, the JMS client will encrypt the connection to this broker using SSL. This can also be set/overridden for all brokers using the Connection URL options.
`ssl_verify_hostname`	Boolean	When using SSL you can enable hostname verification by using `ssl_verify_hostname='true'` in the broker URL.
`ssl_cert_alias`	--	If multiple certificates are present in the keystore, the alias will be used to extract the correct certificate.
`retries`	integer	The number of times to retry connection to each broker in the broker list. Defaults to 1.
`connectdelay`	integer	Length of time (in milliseconds) to wait before attempting to reconnect. Defaults to 0.
`connecttimeout`	integer	Length of time (in milliseconds) to wait for the socket connection to succeed. A value of 0 represents an infinite timeout, i.e. the connection attempt will block until established or an error occurs. Defaults to 30000.
`tcp_nodelay`	Boolean	If `tcp_nodelay='true'`, TCP packet batching is disabled. Defaults to true since Qpid 0.14.

3.4. JAVA JMS MESSAGE PROPERTIES

The following table shows how Qpid Messaging API message properties are mapped to AMQP 0-10 message properties and delivery properties. In this table **msg** refers to the Message class defined in the Qpid Messaging API, **mp** refers to an AMQP 0-10 **message-properties** struct, and **dp** refers to an AMQP 0-10 **delivery-properties** struct.

Table 3.4. Java JMS Mapping to AMQP 0-10 Message Properties

Java JMS Message Property	AMQP 0-10 Property[a]
JMSMessageID	mp.message_id
qpid.subject[b]	mp.application_headers["qpid.subject"]
JMSXUserID	mp.user_id
JMSReplyTo	mp.reply_to[c]
JMSCorrelationID	mp.correlation_id
JMSDeliveryMode	dp.delivery_mode
JMSPriority	dp.priority
JMSExpiration	dp.ttl[d]
JMSRedelivered	dp.redelivered
JMS Properties	mp.application_headers
JMSType	mp.content_type

[a] In these entries, **mp** refers to an AMQP message property, and **dp** refers to an AMQP delivery property.

[b] This is a custom JMS property, set automatically by the Java JMS client implementation.

[c] The reply_to is converted from the protocol representation into an address.

[d] JMSExpiration = dp.ttl + currentTime

3.5. JMS MAPMESSAGE TYPES

Qpid supports the Java JMS **MapMessage** interface, which provides support for maps in messages. The following code shows how to send a **MapMessage** in Java JMS.

Example 3.5. Sending a Java JMS MapMessage

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.MapMessage;
import javax.jms.MessageProducer;
import javax.jms.Session;
```

```
import java.util.Arrays;

// !!! SNIP !!!

MessageProducer producer = session.createProducer(queue);

MapMessage m = session.createMapMessage();
m.setIntProperty("Id", 987654321);
m.setStringProperty("name", "Widget");
m.setDoubleProperty("price", 0.99);

List<String> colors = new ArrayList<String>();
colors.add("red");
colors.add("green");
colors.add("white");
m.setObject("colours", colors);

Map<String,Double> dimensions = new HashMap<String,Double>();
dimensions.put("length",10.2);
dimensions.put("width",5.1);
dimensions.put("depth",2.0);
m.setObject("dimensions",dimensions);

List<List<Integer>> parts = new ArrayList<List<Integer>>();
parts.add(Arrays.asList(new Integer[] {1,2,5}));
parts.add(Arrays.asList(new Integer[] {8,2,5}));
m.setObject("parts", parts);

Map<String,Object> specs = new HashMap<String,Object>();
specs.put("colours", colors);
specs.put("dimensions", dimensions);
specs.put("parts", parts);
m.setObject("specs",specs);

producer.send(m);
```

The following table shows the datatypes that can be sent in a **MapMessage**, and the corresponding datatypes that will be received by clients in Python or C++.

Table 3.5. Java Datatypes in Maps

Java Datatype	Python	C++
boolean	bool	bool
short	int | long	int16
int	int | long	int32
long	int | long	int64

Java Datatype	Python	C++
float	float	float
double	float	double
java.lang.String	unicode	std::string
java.util.UUID	uuid	qpid::types::Uuid
java.util.Map[a]	dict	Variant::Map
java.util.List	list	Variant::List

[a] In Qpid, maps can nest. This goes beyond the functionality required by the JMS specification.

3.6. JMS CLIENT LOGGING

The JMS Client logging is handled using the Simple Logging Facade for Java (SLF4J). As the name implies, slf4j is a facade that delegates to other logging systems like log4j or JDK 1.4 logging. For more information on how to configure slf4j for specific logging systems, please consult the slf4j documentation.

When using the log4j binding, please set the log level for org.apache.qpid explicitly. Otherwise log4j will default to DEBUG which will degrade performance considerably due to excessive logging. The recommended logging level for production is **WARN**.

The following example shows the logging properties used to configure client logging for slf4j using the log4j binding. These properties can be placed in a log4j.properties file and placed in the **CLASSPATH**, or they can be set explicitly using the **`-Dlog4j.configuration`** property.

Example 3.6. log4j Logging Properties

```
log4j.logger.org.apache.qpid=WARN, console
log4j.additivity.org.apache.qpid=false

log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.Threshold=all
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%t %d %p [%c{4}] %m%n
```

3.7. CONFIGURING THE JMS CLIENT

The Qpid JMS Client allows several configuration options to customize it's behaviour at different levels of granualarity.

- JVM level using JVM arguments : Configuration that affects all connections, sessions, consumers and producers created within that JVM.

Ex. `-Dmax_prefetch=1000` property specifies the message credits to use.

- Connection level using Connection/Broker properties : Affects the respective connection and sessions, consumers and produces created by that connection.

  Ex. `amqp://guest:guest@test/test?max_prefetch='1000'&brokerlist='tcp://localhost:5672'` property specifies the message credits to use. This overrides any value specified via the JVM argument `max_prefetch`.

  Please refer to the Section 3.3.2, “Connection URLs” section for a complete list of all properties and how to use them.

- Destination level using Addressing options : Affects the producer(s) and consumer(s) created using the respective destination.

  Ex. `my-queue; {create: always, link:{capacity: 10}}`, where `capacity` option specifies the message credits to use. This overrides any connection level configuration.

Some of these config options are available at all three levels (Ex. `max_prefetch`), while others are available only at JVM or connection level.

3.7.1. Qpid JVM Arguments

Table 3.6. Config Options For Connection Behaviour

Property Name	Type	Default Value	Description
qpid.amqp.version	string	0-10	Sets the AMQP version to be used - currently supports one of {0-8,0-9,0-91,0-10}. The client will begin negotiation at the specified version and only negotiate downwards if the Broker does not support the specified version.
qpid.heartbeat	int	120 (secs)	The heartbeat interval in seconds. Two consective misssed heartbeats will result in the connection timing out. This can also be set per connection using the Connection URL options.

Property Name	Type	Default Value	Description
ignore_setclientID	boolean	false	If a client ID is specified in the connection URL it's used or else an ID is generated. If an ID is specified after it's been set Qpid will throw an exception. Setting this property to 'true' will disable that check and allow you to set a client ID of your choice later on.

Table 3.7. Config Options For Session Behaviour

Property Name	Type	Default Value	Description
qpid.session.command_limit	int	65536	Limits the # of unacked commands
qpid.session.byte_limit	int	1048576	Limits the # of unacked commands in terms of bytes
qpid.use_legacy_map_message	boolean	false	If set will use the old map message encoding. By default the Map messages are encoded using the 0-10 map encoding. This can also be set per connection using the Connection URL options.
qpid.jms.daemon.dispatcher	boolean	false	Controls whether the Session dispatcher thread is a daemon thread or not. If this system property is set to true then the Session dispatcher threads will be created as daemon threads. This setting is introduced in version 0.16.

Table 3.8. Config Options For Consumer Behaviour

Property Name	Type	Default Value	Description

Property Name	Type	Default Value	Description
max_prefetch	int	500	Maximum number of pre-fetched messages per consumer. This can also be defaulted for consumers created on a particular connection using the Connection URL options, or per destination (see the `capacity` option under link properties in addressing)
qpid.session.max_ack_delay	long	1000 (ms)	Timer interval to flush message acks in buffer when using AUTO_ACK and DUPS_OK. When using the above ack modes, message acks are batched and sent if one of the following conditions are met (which ever happens first). • When the ack timer fires. • if un_acked_msg_count > max_prefetch/2. The ack timer can be disabled by setting it to 0.

Property Name	Type	Default Value	Description
sync_ack	boolean	false	If set, each message will be acknowledged synchronously. When using AUTO_ACK mode, you need to set this to "true", in order to get the correct behaviour as described by the JMS spec. This is set to false by default for performance reasons, therefore by default AUTO_ACK behaves similar to DUPS_OK. This can also be set per connection using the Connection URL options.

Table 3.9. Config Options For Producer Behaviour

Property Name	Type	Default Value	Description
sync_publish	string	"" (disabled)	If one of {persistent|all} is set then persistent messages or all messages will be sent synchronously. This can also be set per connection using the Connection URL options.

Table 3.10. Config Options For Threading

Property Name	Type	Default Value	Description
qpid.thread_factory	string	org.apache.qpid.thread.DefaultThreadFactory	Specifies the thread factory to use. If using a real time JVM, you need to set the above property to `org.apache.qpid.thread.RealtimeThreadFactory`.
qpid.rt_thread_priority	int	20	Specifies the priority (1-99) for Real time threads created by the real time thread factory.

Table 3.11. Config Options For I/O

Property Name	Type	Default Value	Description
qpid.transport	string	org.apache.qpid.transport.network.io.IoNetworkTransport	The transport implementation to be used. A user could specify an alternative transport mechanism that implements the interface **`org.apache.qpid.transport.network.OutgoingNetworkTransport`**.
qpid.sync_op_timeout	long	60000	The length of time (in milliseconds) to wait for a synchronous operation to complete. For compatibility with older clients, the synonym **`amqj.default_syncwrite_timeout`** is supported.
qpid.tcp_nodelay	boolean	true	Sets the TCP_NODELAY property of the underlying socket. The default was changed to true as of Qpid 0.14. This can also be set per connection using the Connection URL options. For compatibility with older clients, the synonym **`amqj.tcp_nodelay`** is supported.
qpid.send_buffer_size	integer	65535	Sets the SO_SNDBUF property of the underlying socket. Added in Qpid 0.16. For compatibility with older clients, the synonym **`amqj.sendBufferSize`** is supported.

Property Name	Type	Default Value	Description
qpid.receive_buffer_size	integer	65535	Sets the SO_RCVBUF property of the underlying socket. Added in Qpid 0.16. For compatibility with older clients, the synonym `amqj.receiveBufferSize` is supported.
qpid.failover_method_timeout	long	60000	During failover, this is the timeout for each attempt to try to re-establish the connection. If a reconnection attempt exceeds the timeout, the entire failover process is aborted. It is only applicable for AMQP 0-8/0-9/0-9-1 clients.

Table 3.12. Config Options For Security

Property Name	Type	Default Value	Description
qpid.sasl_mechs	string	PLAIN	The SASL mechanism to be used. More than one could be specified as a comma separated list. We currently support the following mechanisms {PLAIN | GSSAPI | EXTERNAL}. This can also be set per connection using the Connection URL options.
qpid.sasl_protocol	string	AMQP	When using GSSAPI as the SASL mechanism, `sasl_protocol` must be set to the principal for the qpidd broker, e.g. `qpidd`. This can also be set per connection using the Connection URL options.

Property Name	Type	Default Value	Description
qpid.sasl_server_name	string	localhost	When using GSSAPI as the SASL mechanism, **`sasl_server`** must be set to the host for the SASL server, e.g. **`example.com`**. This can also be set per connection using the Connection URL options.

Table 3.13. Config Options For Security - Standard JVM properties needed when using GSSAPI as the SASL mechanism.[a]

Property Name	Type	Default Value	Description
javax.security.auth.useSubjectCredsOnly	boolean	true	If set to 'false', forces the SASL GASSPI client to obtain the kerberos credentials explicitly instead of obtaining from the "subject" that owns the current thread.

java.security.auth.login.config	string		Specifies the jass configuration file.
			
			`Ex-Djava.security.auth.login.config=myjas.conf`
			
			Here is the sample myjas.conf JASS configuration file:
			
			`com.sun.security.jgss.initiate {`
			
			`com.sun.security.auth.module.Krb5LoginModule required useTicketCache=true;`
			`};`

Property Name	Type	Default Value	Description
[a] Please refer to the Java security documentation for a complete understanding of the above properties.			

Table 3.14. Config Options For Security - Using SSL for securing connections or using EXTERNAL as the SASL mechanism.

Property Name	Type	Default Value	Description
qpid.ssl_timeout	long	60000	Timeout value used by the Java SSL engine when waiting on operations.
qpid.ssl.KeyManagerFactory.algorithm	string	-	The key manager factory algorithm name. If not set, defaults to the value returned from the Java runtime call **`KeyManagerFactory.getDefaultAlgorithm()`** For compatibility with older clients, the synonym **`qpid.ssl.keyStoreCertType`** is supported.
qpid.ssl.TrustManagerFactory.algorithm	string	-	The trust manager factory algorithm name. If not set, defaults to the value returned from the Java runtime call **`TrustManagerFactory.getDefaultAlgorithm()`** For compatibility with older clients, the synonym **`qpid.ssl.trustStoreCertType`** is supported.

Table 3.15. Config Options For Security - Standard JVM properties needed when Using SSL for securing connections or using EXTERNAL as the SASL mechanism.[a]

Property Name	Type	Default Value	Description

Property Name	Type	Default Value	Description
javax.net.ssl.keyStore	string	jvm default	Specifies the key store path. This can also be set per connection using the Connection URL options.
javax.net.ssl.keyStorePassword	string	jvm default	Specifies the key store password. This can also be set per connection using the Connection URL options.
javax.net.ssl.trustStore	string	jvm default	Specifies the trust store path. This can also be set per connection using the Connection URL options.
javax.net.ssl.trustStorePassword	string	jvm default	Specifies the trust store password. This can also be set per connection using the Connection URL options.

[a] Qpid allows you to have per connection key and trust stores if required. If specified per connection, the JVM arguments are ignored.

CHAPTER 4. STOMP HEARTBEATS

Abstract

The Stomp 1.1 protocol support a heartbeat policy that allows clients to send keepalive messages to the broker.

STOMP 1.1 HEARTBEATS

Stomp 1.1 adds support for heartbeats (keepalive messages) on Stomp connections. Negotiation of a heartbeat policy is normally initiated by the client (Stomp 1.1 clients only) and the client must be configured to enable heartbeats. No broker settings are required to enable support for heartbeats, however.

At the level of the Stomp wire protocol, heartbeats are negotiated when the client establishes the Stomp connection and the following messages are exchanged between client and server:

```
CONNECT
heart-beat:CltSend,CltRecv

CONNECTED:
heart-beat:SrvSend,SrvRecv
```

The *CltSend*, *CltRecv*, *SrvSend*, and *SrvRecv* fields are interpreted as follows:

CltSend

Indicates the minimum frequency of messages *sent from the client*, expressed as the maximum time between messages in units of milliseconds. If the client does not send a regular Stomp message within this time limit, it must send a special heartbeat message, in order to keep the connection alive.

A value of zero indicates that the client does not send heartbeats.

CltRecv

Indicates how often the client expects to *receive* message from the server, expressed as the maximum time between messages in units of milliseconds. If the client does not receive any messages from the server within this time limit, it would time out the connection.

A value of zero indicates that the client does not expect heartbeats and will not time out the connection.

SrvSend

Indicates the minimum frequency of messages *sent from the server*, expressed as the maximum time between messages in units of milliseconds. If the server does not send a regular Stomp message within this time limit, it must send a special heartbeat message, in order to keep the connection alive.

A value of zero indicates that the server does not send heartbeats.

SrvRecv

Indicates how often the server expects to *receive* message from the client, expressed as the maximum time between messages in units of milliseconds. If the server does not receive any messages from the client within this time limit, it would time out the connection.

A value of zero indicates that the server does not expect heartbeats and will not time out the connection.

In order to ensure that the rates of sending and receiving required by the client and the server are mutually compatible, the client and the server negotiate the heartbeat policy, adjusting their sending and receiving rates as needed.

STOMP 1.0 HEARTBEAT COMPATIBILITY

A difficulty arises, if you want to support an inactivity timeout on your Stomp connections when legacy Stomp 1.0 clients are connected to your broker. The Stomp 1.0 protocol does *not* support heartbeats, so Stomp 1.0 clients are not capable of negotiating a heartbeat policy.

To get around this limitation, you can specify the **`transport.defaultHeartBeat`** option in the broker's **`transportConnector`** element, as follows:

```
<transportConnector name="stomp" uri="stomp://0.0.0.0:0?
transport.defaultHeartBeat=5000,0" />
```

The effect of this setting is that the broker now behaves *as if* the Stomp 1.0 client had sent the following Stomp frame when it connected:

```
CONNECT
heart-beat:5000,0
```

This means that the broker will expect the client to send a message at least once every 5000 milliseconds (5 seconds). The second integer value, **`0`**, indicates that the client does not expect to receive any heartbeats from the server (which makes sense, because Stomp 1.0 clients do not understand heartbeats).

Now, if the Stomp 1.0 client does not send a regular message after 5 seconds, the connection will time out, because the Stomp 1.0 client is not capable of sending out a heartbeat message to keep the connection alive. Hence, you should choose the value of the timeout in **`transport.defaultHeartBeat`** such that the connection will stay alive, as long as the Stomp 1.0 clients are sending messages at their normal rate.

CHAPTER 5. INTRA-JVM CONNECTIONS

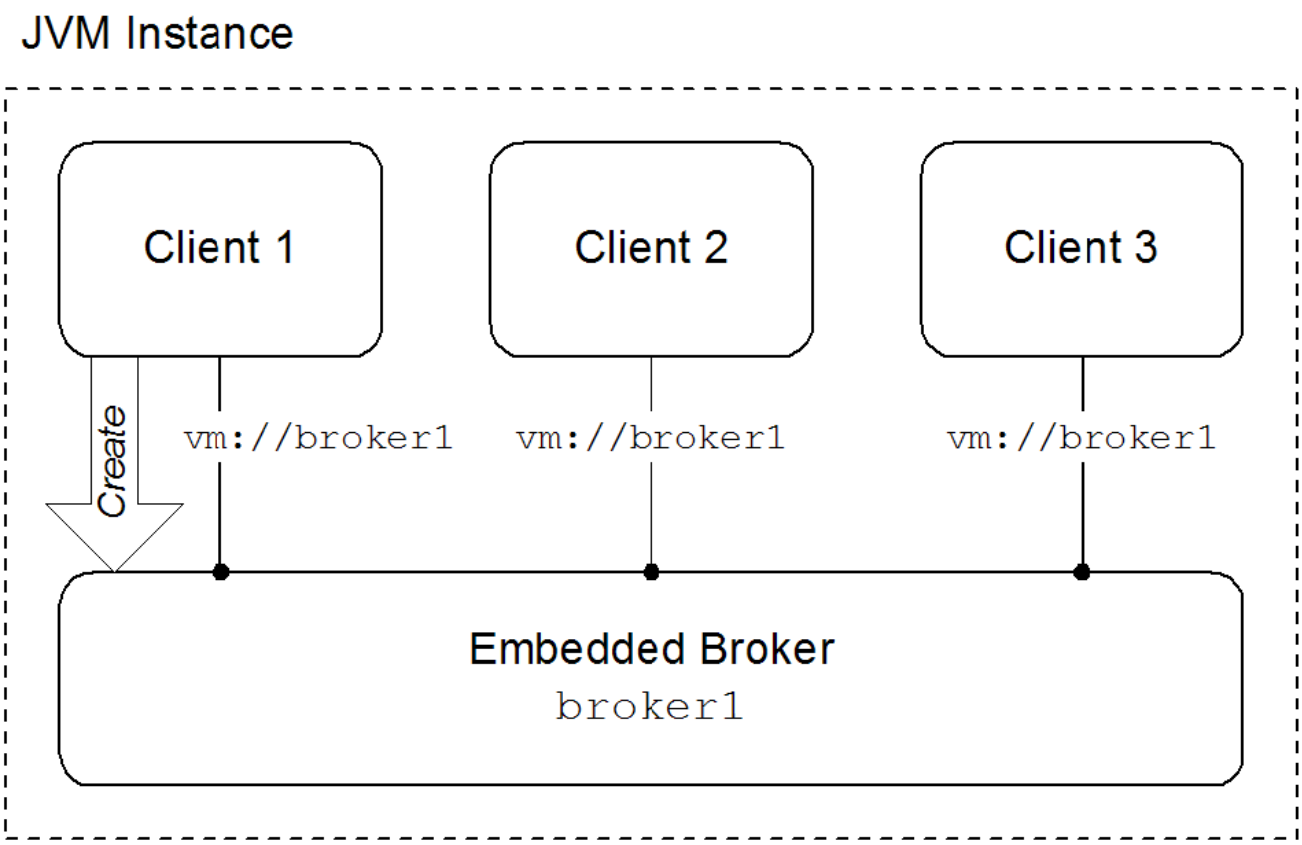
Abstract

Red Hat JBoss A-MQ uses a VM transport to allow clients to connect to each other inside the Java Virtual Machine (JVM) without the overhead of network communication.

OVERVIEW

Red Hat JBoss A-MQ's VM transport enables Java clients running inside the same JVM to communicate with each other without having to resort to a using a network connection. The VM transport does this be implicitly creating an embedded broker the first time it is accessed. Figure 5.1, “Clients Connected through the VM Transport” shows the basic architecture of the VM protocol.

Figure 5.1. Clients Connected through the VM Transport



EMBEDDED BROKERS

The VM transport uses a broker embedded in the same JVM as the clients to facilitate communication between the clients. The embedded broker can be created in several ways:

- explicitly defining the broker in the application's configuration
- explicitly creating the broker using the Java APIs
- automatically when the first client attempts to connect to it using the VM transport

The VM transport uses the broker name to determine if an embedded broker needs to be created. When a client uses the VM transport to connect to a broker, the transport checks to see if an

embedded broker by that name already exists. If it does exist, the client is connected to the broker. If it does not exist, the broker is created and then the client is connected to it.

IMPORTANT

When using explicitly created brokers there is a danger that your clients will attempt to connect to the embedded broker before it is started. If this happens, the VM transport will auto-create an instance of the broker for you. To avoid this conflict you can set the `waitForStart` option or the `create=false` option to manage how the VM transport determines when to create a new embedded broker.

USING THE VM TRANSPORT

The URI used to specify the VM transport comes in two flavors to provide maximum control over how the embedded broker is configured:

- simple

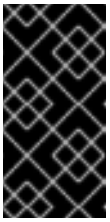
  The simple VM URI is used in most situations. It allows you to specify the name of the embedded broker to which the client will connect. It also allows for some basic broker configuration.

  Example 5.1, "Simple VM URI Syntax" shows the syntax for a simple VM URI.

  Example 5.1. Simple VM URI Syntax

  ```
  vm://BrokerName?TransportOptions
  ```

  - *BrokerName* specifies the name of the embedded broker to which the client connects.
  - *TransportOptions* specifies the configuration for the transport. They are specified in the form of a query list. For details about the available options see the *Connection Reference*.

    IMPORTANT

    The broker configuration options specified on the VM URI are only meaningful if the client is responsible for instantiating the embedded broker. If the embedded broker is already started, the transport will ignore the broker configuration properties.

- advanced

  The advanced VM URI provides you full control over how the embedded broker is configured. It uses a broker configuration URI similar to the one used by the administration tool to configure the embedded broker.

  Example 5.2, "Advanced VM URI Syntax" shows the syntax for an advanced VM URI.

  Example 5.2. Advanced VM URI Syntax

  ```
  vm://(BrokerConfigURI)?TransportOptions
  ```

- *BrokerConfigURI* is a broker configuration URI.
- *TransportOptions* specifies the configuration for the transport. They are specified in the form of a query list. For details about the available options see the *Connection Reference*.

EXAMPLES

Example 5.3, "Basic VM URI" shows a basic VM URI that connects to an embedded broker named `broker1`.

Example 5.3. Basic VM URI

```
vm://broker1
```

Example 5.4, "Simple URI with broker options" creates and connects to an embedded broker that uses a non-persistent message store.

Example 5.4. Simple URI with broker options

```
vm://broker1?broker.persistent=false
```

Example 5.5, "Advanced VM URI" creates and connects to an embedded broker configured using a broker configuration URI.

Example 5.5. Advanced VM URI

```
vm:(broker:(tcp://localhost:6000)?persistent=false)?marshal=false
```

CHAPTER 6. PEER PROTOCOL

Abstract

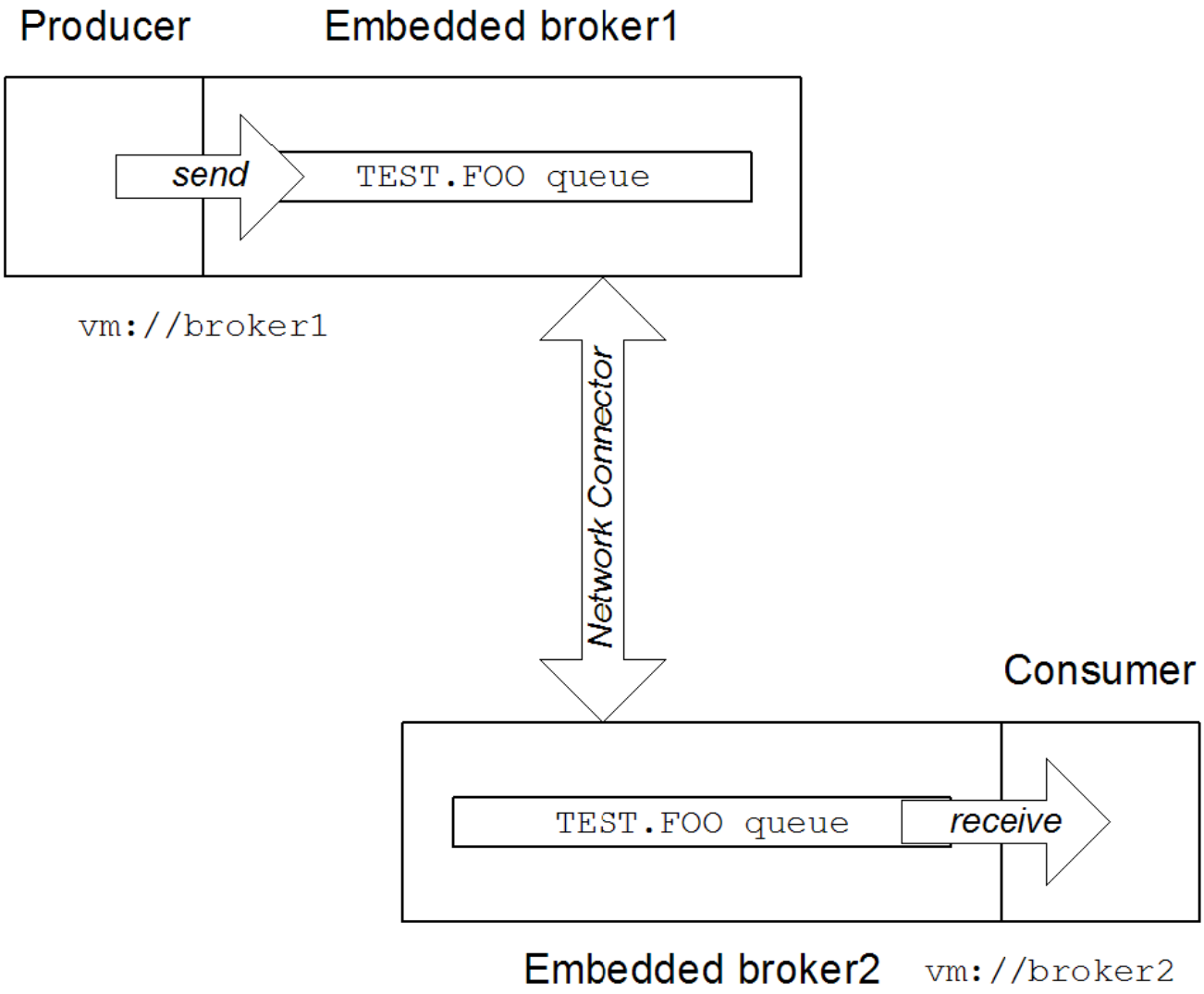
The peer protocol enables messaging clients to communicate with each other directly, eliminating the requirement to route messages through an external message broker. It does this by embedding a message broker in each client and using the embedded brokers to mediate the interactions.

OVERVIEW

The peer protocol enables messaging clients to communicate without the need for a separate message broker. It creates a peer-to-peer network by creating an embedded broker inside each peer endpoint and setting up a network connector between them. The messaging clients are formed into a network-of-brokers.

Figure 6.1, “Peer Protocol Endpoints with Embedded Brokers” illustrates the peer-to-peer network topology for a simple two-peer network.

Figure 6.1. Peer Protocol Endpoints with Embedded Brokers

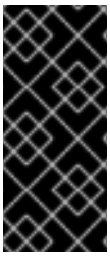


The producer sends messages to its embedded broker, **broker1**, by connecting to the local VM endpoint, **vm://broker1**. The embedded brokers, **broker1** and **broker2**, are linked together using a network connector which allows messages to flow in either direction between the brokers. When the

producer sends a message to the queue, **broker1** pushes the message across the network connector to **broker2**. The consumer receives the message from **broker2**.

PEER ENDPOINT DISCOVERY

The peer protocol uses multicast discovery to locate active peers on the network. As the embedded brokers are instantiated they use a multicast discovery agent to locate other embedded brokers in the same multicast group. The multicast group ID is provided as part of the peer URI.



IMPORTANT

To use the peer protocol, you must ensure that the IP multicast protocol is enabled on your operating system.

For more information about using multicast discovery and network connectors see *Using Networks of Brokers*.

URI SYNTAX

A **peer** URI must conform to the following syntax:

```
peer://PeerGroup/BrokerName?BrokerOptions
```

Where the group name, *PeerGroup*, identifies the set of peers that can communicate with each other. A given peer can connect only to the set of peers that specify the *same PeerGroup* name in their URLs. The *BrokerName* specifies the broker name for the embedded broker. The broker options, *BrokerOptions*, are specified in the form of a query list.

SAMPLE URI

The following is an example of a peer URL that belongs to the peer group, **groupA**, and creates an embedded broker with broker name, **broker1**:

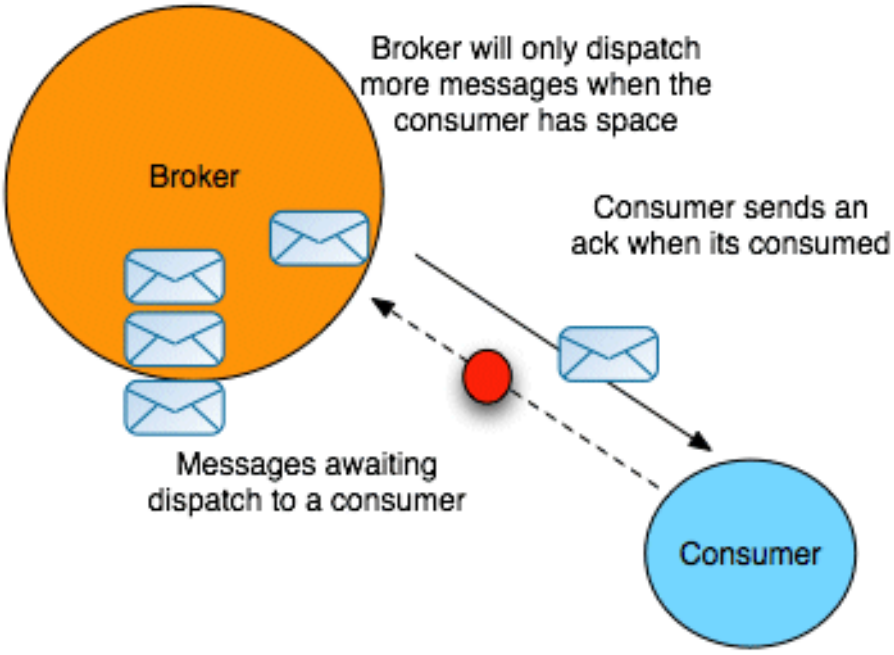
```
peer://groupA/broker1?persistent=false
```

CHAPTER 7. MESSAGE PREFETCH BEHAVIOR

OVERVIEW

Figure 7.1, “Consumer Prefetch Limit” illustrates the behavior of a broker, as it waits to receive acknowledgments for the messages it has already sent to a consumer.

Figure 7.1. Consumer Prefetch Limit



If a consumer is slow to acknowledge messages, the broker may send it another message before the previous message is acknowledged. If the consumer continues to be slow, the number of unacknowledged messages can grow continuously larger. The broker does not continue to send messages indefinitely. When the number of unacknowledged messages reaches a set limit—the *prefetch limit*—the server ceases sending new messages to the consumer. No more messages will be sent until the consumer starts sending back some acknowledgments.

NOTE

The broker relies on acknowledgement of delivery to determine if it can dispatch additional messages to a consumer's prefetch buffer. So, if a consumer's prefetch buffer is set to 1 and it is slow to acknowledge the processing of the message, it is possible that the broker will dispatch an additional message to the consumer and the pending message count will be 2.

Red Hat JBoss A-MQ has a provides a lot of options for fine tuning prefetch limits for specific circumstances. The prefetch limits can be specified for different types of consumers. You can also set the prefect limits on a per broker, per connection, or per destination basis.

CONSUMER SPECIFIC PREFETCH LIMITS

Different prefetch limits can be set for each consumer type. Table 7.1, “Prefect Limit Defaults” list the property name and default value for each consumer type's prefetch limit.

Table 7.1. Prefect Limit Defaults

Consumer Type	Property	Default
Queue consumer	**queuePrefetch**	**1000**
Queue browser	**queueBrowserPrefetch**	**500**
Topic consumer	**topicPrefetch**	**32766**
Durable topic subscriber	**durableTopicPrefetch**	**100**

SETTING PREFETCH LIMITS PER BROKER

You can define the prefetch limits for all consumers that attach to a particular broker by setting a destination policy on the broker. To set the destination policy, add a **destinationPolicy** element as a child of the **broker** element in the broker's configuration, as shown in Example 7.1, "Configuring a Destination Policy".

Example 7.1. Configuring a Destination Policy

```
<broker ... >
  ...
  <destinationPolicy>
    <policyMap>
      <policyEntries>
        <policyEntry queue="queue.>" queuePrefetch="1"/>
        <policyEntry topic="topic.>" topicPrefetch="1000"/>
      </policyEntries>
    </policyMap>
  </destinationPolicy>
  ...
</broker>
```

In Example 7.1, "Configuring a Destination Policy", the queue prefetch limit for all queues whose names start with **queue.** is set to 1 (the **>** character is a wildcard symbol that matches one or more name segments); and the topic prefetch limit for all topics whose names start with **topic.** is set to 1000.

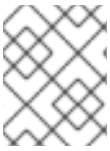
SETTING PREFETCH LIMITS PER CONNECTION

In a consumer, you can specify the prefetch limits on a connection by setting properties on the **ActiveMQConnectionFactory** instance. Example 7.2, "Setting Prefetch Limit Properties Per Connection" shows how to specify the prefetch limits for all consumer types on a connection factory.

Example 7.2. Setting Prefetch Limit Properties Per Connection

```
ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory();

Properties props = new Properties();
props.setProperty("prefetchPolicy.queuePrefetch", "1000");
props.setProperty("prefetchPolicy.queueBrowserPrefetch", "500");
props.setProperty("prefetchPolicy.durableTopicPrefetch", "100");
```

```
props.setProperty("prefetchPolicy.topicPrefetch", "32766");

factory.setProperties(props);
```



NOTE

You can also set the prefetch limits using the consumer properties as part of the broker URI used when creating the connection factory.

SETTING PREFETCH LIMITS PER DESTINATION

At the finest level of granularity, you can specify the prefetch limit on each destination instance that you create in a consumer. Example 7.3, “Setting the Prefect Limit on a Destination” shows code create the queue **TEST.QUEUE** with a prefetch limit of 10. The option is set as a destination option as part of the URI used to create the queue.

Example 7.3. Setting the Prefect Limit on a Destination

```
Queue queue = new ActiveMQQueue("TEST.QUEUE?consumer.prefetchSize=10");

MessageConsumer consumer = session.createConsumer(queue);
```

DISABLING THE PREFETCH EXTENSION LOGIC

The default behavior of a broker is to use delivery acknowledgements to determine the state of a consumer's prefect buffer. For example, if a consumer's prefect limit is configured as 1 the broker will dispatch 1 message to the consumer and when the consumer acknowledges receiving the message, the broker will dispatch a second message. If the initial message takes a long time to process, the message sitting in the prefect buffer cannot be processed by a faster consumer.

This behavior can also cause issues when using the JCA resource adapter and transacted clients.

If the behavior is causing issues, it can be changed such that the broker will wait for the consumer to acknowledge that the message is processed before refilling the prefetch buffer. This is accomplished by setting a destination policy on the broker to disable the prefect extension for specific destinations.

Example 7.4, “Disabling the Prefetch Extension” shows configuration for disabling the prefect extension on all of a broker's queues.

Example 7.4. Disabling the Prefetch Extension

```
<broker ... >
  ...
  <destinationPolicy>
    <policyMap>
      <policyEntries>
        <policyEntry queue=">" usePrefetchExtension=”false”/>
      </policyEntries>
    </policyMap>
```

```
    </destinationPolicy>
    ...
</broker>
```

CHAPTER 8. MESSAGE REDELIVERY

OVERVIEW

Messages are redelivered to a client when any of the following occurs:

- A transacted session is used and `rollback()` is called.
- A transacted session is closed before commit is called.
- A session is using `CLIENT_ACKNOWLEDGE` and `Session.recover()` is called.

The policy used to control how messages are redelivered and when they are determined dead can be configured in a number of ways:

- On the broker, using the broker's redelivery plug-in,
- On the connection factory, using the connection URI,
- On the connection, using the `RedeliveryPolicy`,
- On destinations, using the connection's `RedeliveryPolicyMap`.

REDELIVERY PROPERTIES

Table 8.1, “Redelivery Policy Options” list the properties that control message redelivery.

Table 8.1. Redelivery Policy Options

Option	Default	Description
`collisionAvoidanceFactor`	`0.15`	Specifies the percentage of range of collision avoidance.
`maximumRedeliveries`	`6`	Specifies the maximum number of times a message will be redelivered before it is considered a poisoned pill and returned to the broker so it can go to a dead letter queue. `-1` specifies an infinite number of redeliveries.
`maximumRedeliveryDelay`	`-1`	Specifies the maximum delivery delay that will be applied if the `useExponentialBackOff` option is set. `-1` specifies that no maximum be applied.
`initialRedeliveryDelay`	`1000`	Specifies the initial redelivery delay in milliseconds.

Option	Default	Description
redeliveryDelay	**1000**	Specifies the delivery delay, in milliseconds, if **initialRedeliveryDelay** is 0.
useCollisionAvoidance	**false**	Specifies if the redelivery policy uses collision avoidance.
useExponentialBackOff	**false**	Specifies if the redelivery time out should be increased exponentially.
backOffMultiplier	**5**	Specifies the back-off multiplier.

CONFIGURING THE BROKER'S REDELIVERY PLUG-IN

Configuring a broker's redelivery plug-in is a good way to tune the redelivery of messages to all of the consumer's that use the broker. When using the broker's redelivery plug-in, it is recommended that you disable redelivery on the consumer side (if necessary, by setting **maximumRedeliveries** to 0 on the destination).

The broker's redelivery policy configuration is done through the **redeliveryPlugin** element. As shown in Example 8.1, "Configuring the Redelivery Plug-In" this element is a child of the broker's **plugins** element and contains a policy map defining the desired behavior.

Example 8.1. Configuring the Redelivery Plug-In

```
<broker xmlns="http://activemq.apache.org/schema/core" ... >
  ....
  <plugins>
    <redeliveryPlugin ... >
      <redeliveryPolicyMap>
        <redeliveryPolicyMap>
(1)         <redeliveryPolicyEntries>
              <!-- a destination specific policy -->
            <redeliveryPolicy queue="SpecialQueue"
                              maximumRedeliveries="3"
                              initialRedeliveryDelay="3000" />
          </redeliveryPolicyEntries>
          <!-- the fallback policy for all other destinations -->
(2)         <defaultEntry>
              <redeliveryPolicy maximumRedeliveries="3"
                              initialRedeliveryDelay="3000" />
          </defaultEntry>
        </redeliveryPolicyMap>
      </redeliveryPolicyMap>
    </redeliveryPlugin>
  </plugins>
  ...
</broker>
```

1. The **redeliveryPolicyEntries** element contains a list of **redeliveryPolicy** elements that configures redelivery policies on a per-destination basis.
2. The **defaultEntry** element contains a single **redeliveryPolicy** element that configures the redelivery policy used by all destinations that do not match the one with a specific policy.

CONFIGURING THE REDELIVERY USING THE BROKER URI

Clients can specify their preferred redelivery by adding redelivery policy information as part of the connection URI used when getting the connection factory. Example 8.2, "Setting the Redelivery Policy using a Connection URI" shows code for setting the maximum number of redeliveries to 4.

Example 8.2. Setting the Redelivery Policy using a Connection URI

```
ActiveMQConnectionFactory connectionFactory =
  new ActiveMQConnectionFactory("tcp://localhost:61616?
jms.redeliveryPolicy.maximumRedeliveries=4");
```

For more information on connection URIs see the *Connection Reference*.

SETTING THE REDELIVERY POLICY ON A CONNECTION

The **ActiveMQConnection** class' **getRedeliveryPolicy()** method allows you to configure the redelivery policy for all consumer's using that connection.

getRedeliveryPolicy() returns a **RedeliveryPolicy** object that controls the redelivery policy for the connection. The **RedeliveryPolicy** object has setters for each of the properties listed in Table 8.1, "Redelivery Policy Options".

Example 8.3, "Setting the Redelivery Policy for a Connection" shows code for setting the maximum number of redeliveries to 4.

Example 8.3. Setting the Redelivery Policy for a Connection

```
ActiveMQConnection connection =
  connectionFactory.createConnetion();

// Get the redelivery policy
RedeliveryPolicy policy = connection.getRedeliveryPolicy();

// Set the policy
policy.setMaximumRedeliveries(4);
```

SETTING THE REDELIVERY POLICY ON A DESTINATION

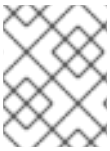
For even more fine grained control of message redelivery, you can set the redelivery policy on a per-destination basis. The **ActiveMQConnection** class' **getRedeliveryPolicyMap()** method returns a

RedeliveryPolicyMap object that is a map of **RedeliveryPolicy** objects with destination names as the key.

NOTE

You can also specify destination names using wildcards.

Each **RedeliveryPolicy** object controls the redelivery policy for all destinations whose name match the destination name specified in the map's key.



NOTE

If a destination does not match one of the entries in the map, the destination will use the redelivery policy set on the connection.

Example 8.4, "Setting the Redelivery Policy for a Destination" shows code for specifying that messages in the queue **FRED.JOE** can only be redelivered 4 times.

Example 8.4. Setting the Redelivery Policy for a Destination

```
ActiveMQConnection connection =
  connectionFactory.createConnetion();

// Get the redelivery policy
RedeliveryPolicy policy = new RedeliveryPolicy();
policy.setMaximumRedeliveries(4);

//Get the policy map
RedeliveryPolicyMap map = connection.getRedeliveryPolicyMap();
map.put(new ActiveMQQueue("FRED.JOE"), queuePolicy);
```

INDEX

A

B

C

V