Red Hat JBoss A-MQ 6.0

Client Connectivity Guide

Creating and tuning clients connections to message brokers

Last Updated: 2017-10-13

Red Hat JBoss A-MQ 6.0 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 **3**
TRANSPORTS AND PROTOCOLS 3
SUPPORTED CLIENT APIS 3
CONFIGURATION 4

CHAPTER 2. CONNECTING TO A BROKER **5**
2.1. CONNECTING WITH THE JAVA API 5
2.2. CONNECTING WITH THE C++ API 7
2.3. CONNECTING WITH THE .NET API 11

CHAPTER 3. STOMP HEARTBEATS **13**
STOMP 1.1 HEARTBEATS 13
STOMP 1.0 HEARTBEAT COMPATIBILITY 14

CHAPTER 4. INTRA-JVM CONNECTIONS **15**
OVERVIEW 15
EMBEDDED BROKERS 15
USING THE VM TRANSPORT 16
EXAMPLES 17

CHAPTER 5. PEER PROTOCOL **18**
OVERVIEW 18
PEER ENDPOINT DISCOVERY 19
URI SYNTAX 19
SAMPLE URI 19

CHAPTER 6. MESSAGE PREFETCH BEHAVIOR **20**
OVERVIEW 20
CONSUMER SPECIFIC PREFETCH LIMITS 20
SETTING PREFETCH LIMITS PER BROKER 21
SETTING PREFETCH LIMITS PER CONNECTION 21
SETTING PREFETCH LIMITS PER DESTINATION 22
DISABLING THE PREFETCH EXTENSION LOGIC 22

CHAPTER 7. MESSAGE REDELIVERY **24**
OVERVIEW 24
REDELIVERY PROPERTIES 24
CONFIGURING THE BROKER'S REDELIVERY PLUG-IN 25
CONFIGURING THE REDELIVERY USING THE BROKER URI 26
SETTING THE REDELIVERY POLICY ON A CONNECTION 26
SETTING THE REDELIVERY POLICY ON A DESTINATION 26

INDEX **27**

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.

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 4, *Intra-JVM Connections*.
- Peer—allows clients to communicate with each other without using an external message broker. See Chapter 5, *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.

CHAPTER 2. CONNECTING TO A BROKER

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.

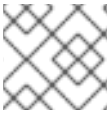
   Destinations are administered objects that are typically created by the broker. The JBoss A-MQ allows clients to create destinations on-demand. You can also look up destinations using JNDI or another mechanism.

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. CONNECTING WITH THE JAVA 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. CONNECTING WITH THE C++ 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.

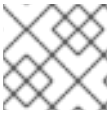
NOTE

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. CONNECTING WITH THE .NET 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 can download the Red Hat JBoss A-MQ 6.2.0 .NET Client binaries from the Red Hat Customer Portal. Binaries for version 6.0 are not available, but the later versions of the .NET client binaries are compatible with 6.0 (the OpenWire endpoint will negotiate down to the correct protocol version for the 6.0 broker). Ideally, though, it is recommended that you upgrade your brokers to version 6.2.0 as well.

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();
...
```

CHAPTER 3. 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 4. INTRA-JVM CONNECTIONS

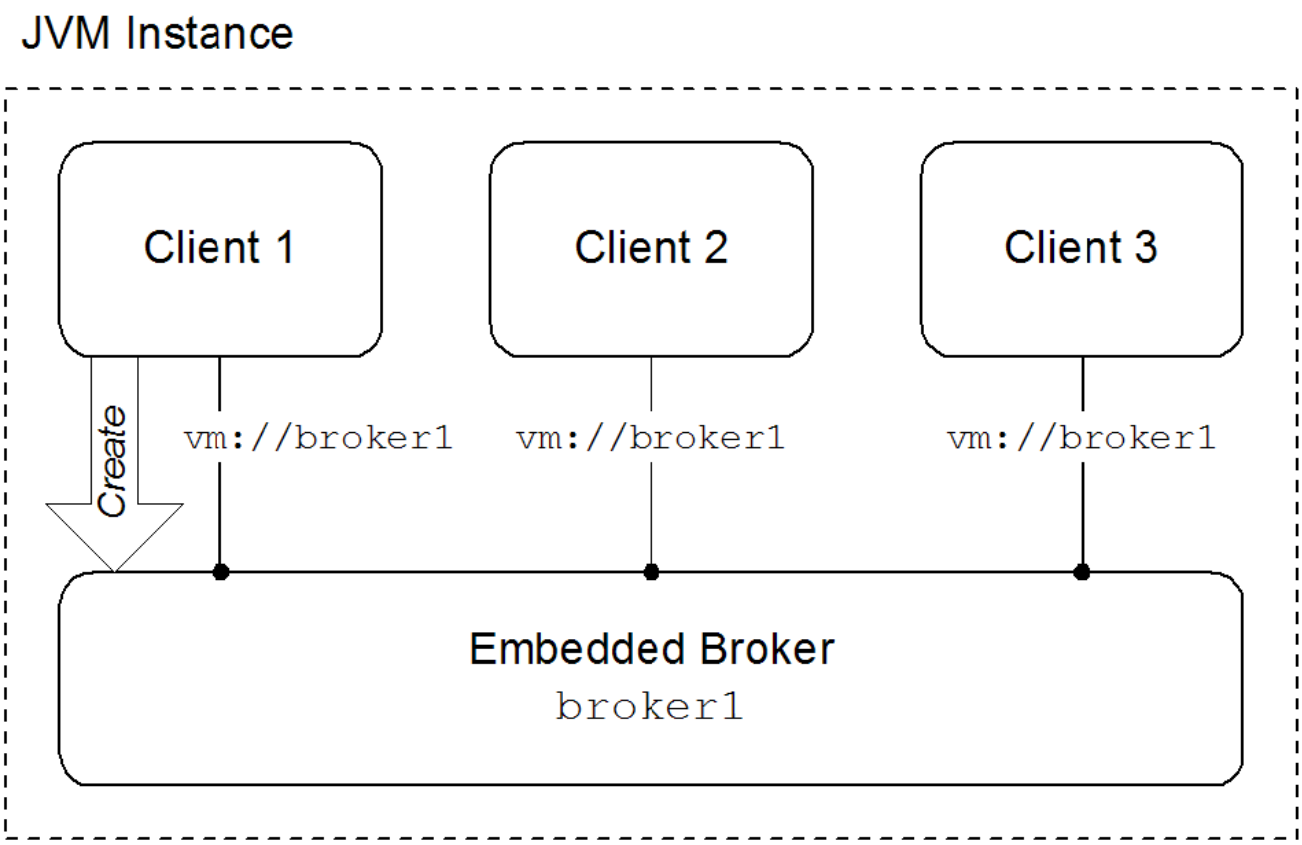
Abstract

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 4.1, “Clients Connected through the VM Transport” shows the basic architecture of the VM protocol.

Figure 4.1. Clients Connected through the VM Transport



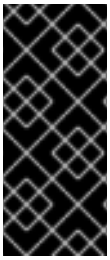
EMBEDDED BROKERS

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

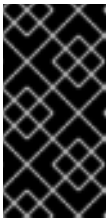
  The simple VM URI is used in most situations. It allows you to specify the name of the embedded broker to which the client will connect. It also allows for some basic broker configuration.

  Example 4.1, “Simple VM URI Syntax” shows the syntax for a simple VM URI.

  Example 4.1. 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 4.2, “Advanced VM URI Syntax” shows the syntax for an advanced VM URI.

  Example 4.2. 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 4.3, “Basic VM URI” shows a basic VM URI that connects to an embedded broker named **`broker1`**.

Example 4.3. Basic VM URI

```
vm://broker1
```

Example 4.4, “Simple URI with broker options” creates and connects to an embedded broker that uses a non-persistent message store.

Example 4.4. Simple URI with broker options

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

Example 4.5, “Advanced VM URI” creates and connects to an embedded broker configured using a broker configuration URI.

Example 4.5. Advanced VM URI

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

CHAPTER 5. PEER PROTOCOL

Abstract

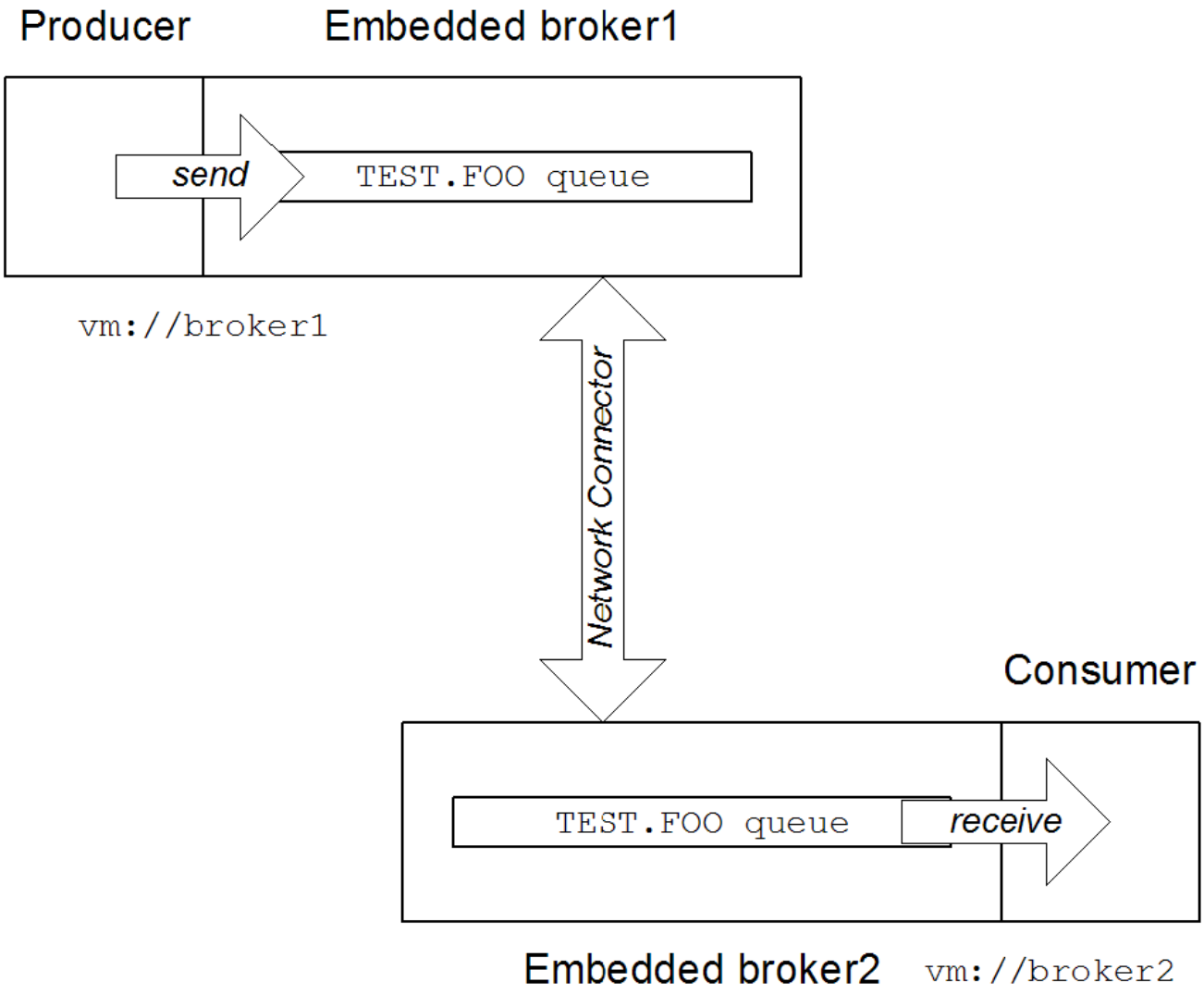
The peer protocol enables messaging clients to communicate with each other directly, eliminating the requirement to route messages through an external message broker. It does this by embedding a message broker in each client and using the embedded brokers to mediate the interactions.

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

Figure 5.1. 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**:

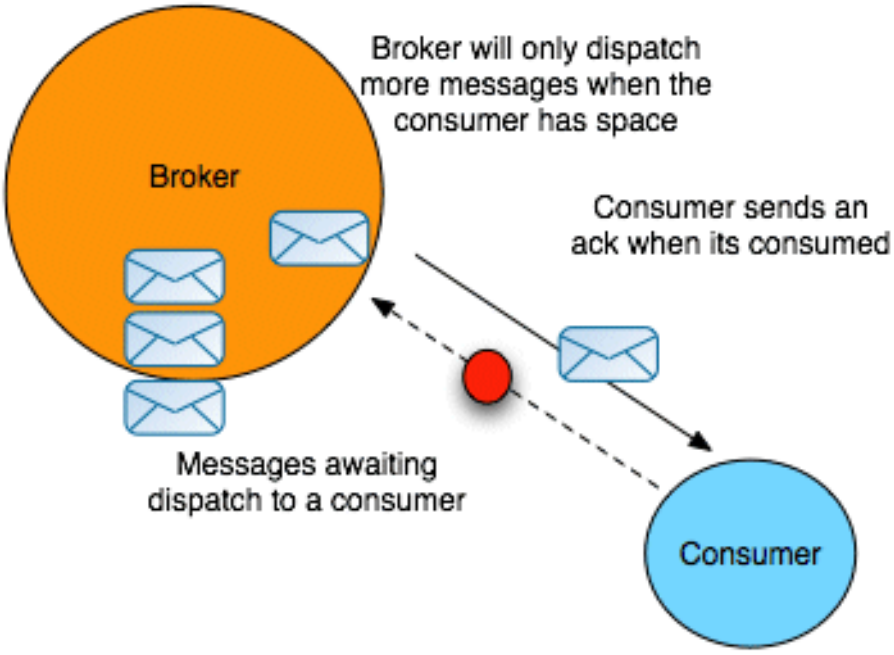
```
peer://groupA/broker1?persistent=false
```

CHAPTER 6. MESSAGE PREFETCH BEHAVIOR

OVERVIEW

Figure 6.1, “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 6.1. 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 6.1, “Prefect Limit Defaults” list the property name and default value for each consumer type's prefetch limit.

Table 6.1. 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 6.1, “Configuring a Destination Policy”.

Example 6.1. Configuring a Destination Policy

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

In Example 6.1, “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 6.2, “Setting Prefetch Limit Properties Per Connection” shows how to specify the prefetch limits for all consumer types on a connection factory.

Example 6.2. 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 6.3, “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 6.3. 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 6.4, “Disabling the Prefetch Extension” shows configuration for disabling the prefect extension on all of a broker's queues.

Example 6.4. Disabling the Prefetch Extension

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

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

CHAPTER 7. 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 7.1, “Redelivery Policy Options” list the properties that control message redelivery.

Table 7.1. 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 7.1, "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 7.1. 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 7.2, “Setting the Redelivery Policy using a Connection URI” shows code for setting the maximum number of redeliveries to 4.

Example 7.2. 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 7.1, “Redelivery Policy Options”.

Example 7.3, “Setting the Redelivery Policy for a Connection” shows code for setting the maximum number of redeliveries to 4.

Example 7.3. Setting the Redelivery Policy for a Connection

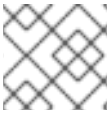
```
ActiveMQConnection connection =
  connectionFactory.createConnetion();

// 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 7.4, “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 7.4. 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

ActiveMQConnection, The connection, Setting the redelivery policy on a connection, Setting the redelivery policy on a destination

ActiveMQConnectionFactory, The connection factory

B

backOffMultiplier, Redelivery properties

C

collisionAvoidanceFactor, Redelivery properties

Connection, The connection

ConnectionFactory, The connection factory

D

durableTopicPrefetch, Consumer specific prefetch limits

E

embedded broker, Embedded brokers

G

getRedeliveryPolicy(), Setting the redelivery policy on a connection

getRedeliveryPolicyMap(), Setting the redelivery policy on a destination

I

initialRedeliveryDelay, Redelivery properties

M

maximumRedeliveries, Redelivery properties

maximumRedeliveryDelay, Redelivery properties

P

prefetch

- **per broker**, Setting prefetch limits per broker
- **per connection**, Setting prefetch limits per connection
- **per destination**, Setting prefetch limits per destination

Q

queueBrowserPrefetch, Consumer specific prefetch limits

queuePrefetch, Consumer specific prefetch limits

R

redeliveryDelay, Redelivery properties

redeliveryPlugin, Configuring the broker's redelivery plug-in

RedeliveryPolicy, Setting the redelivery policy on a connection, Setting the redelivery policy on a destination

RedeliveryPolicyMap, Setting the redelivery policy on a destination

T

topicPrefetch, Consumer specific prefetch limits

U

useCollisionAvoidance, Redelivery properties

useExponentialBackOff, Redelivery properties

usePrefetchExtension, Disabling the prefetch extension logic

V

VM

advanced URI, Using the VM transport

broker name, Using the VM transport

create, Embedded brokers

embedded broker, Embedded brokers

simple URI, Using the VM transport

waitForStart, Embedded brokers

VM URI

advanced, Using the VM transport

simple, Using the VM transport