Progress FUSE

FUSE™ Message Broker

Connectivity Guide

Version 5.3
Febuary 2009

PROGRESS SOFTWARE

Connectivity Guide

Version 5.3

Publication date 23 Jul 2009

Legal Notices

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Protocol Summary

FUSE Message Broker supports a wide variety of protocols for client-to-broker, broker-to-broker, and client-to-client connections. The intention is that the variety of protocols will make it easier to connect to a range of client types. Different network topologies can also be supported with the help of special protocols, such as discovery and peer-to-peer.

Simple Connections

Overview

The following protocols can be used either for straightforward client-to-broker connections (transport connector) or broker-to-broker connections (network connector). For each wire protocol (that is, on-the-wire message encoding), FUSE Message Broker supports one or more associated transport protocols. Hence, you can configure connections with a wide variety of wire protocol/transport protocol combinations.

Protocols for simple connections

Table 1.1 on page 12 shows the protocol combinations that messaging clients can use to connect directly to the message broker.

Table 1.1. Protocols for Simple Connections

Wire Protocol	Transport Protocol	Sample URL	Description
OpenWire	TCP	`tcp://Host:Port`	Connect to the message broker endpoint at `Host:Port` using the OpenWire over TCP protocol. This URL is also used to configure the transport connector in a broker.
OpenWire	SSL	`ssl://Host:Port`	Connect to the message broker endpoint at `Host:Port` using the OpenWire over SSL protocol. This URL is also used to configure the transport connector in a broker.
OpenWire	HTTP	`http://Host:Port`	Connect to the message broker endpoint at `Host:Port` using the OpenWire over HTTP protocol (HTTP tunneling). You can use this protocol to navigate through firewalls. This URL is also used to configure the transport connector in a broker.
OpenWire	HTTPS	`https://Host:Port`	Connect to the message broker endpoint at `Host:Port` using the OpenWire over HTTPS protocol This URL is also used to configure the transport connector in a broker.
Stomp	TCP	`stomp://Host:Port`	Connect to the message broker endpoint at `Host:Port` using the Stomp over TCP protocol.

Wire Protocol	Transport Protocol	Sample URL	Description
			This URL is also used to configure the transport connector in a broker.
Stomp	SSL	`stomp+ssl://Host:Port`	Connect to the message broker endpoint at `Host:Port` using the Stomp over SSL protocol. This URL is also used to configure the transport connector in a broker.
REST	HTTP	`http://Host:Port/demo/message/FOO/BAR?timeout=10000&type=queue`	Connect to the message broker endpoint at `Host:Port` using the REST protocol. The REST endpoint is implemented as a servlet deployed in a servlet engine. For example, the sample URL is built up from a Web context name, `demo`, followed by the servlet name, `message`, followed by a destination name, `FOO/BAR`, and some query options. This URL is *not* used to configure the REST transport connector in a broker. Use the `<jetty>` tag to configure the REST endpoint in the broker.
RESTs	HTTPS	`http://Host:Port/demo/message/FOO/BAR?timeout=10000&type=queue`	
XMPP	TCP	`xmpp://Host:Port`	Configure the transport connector in a message broker to accept XMPP connections on `Host:Port` (for example, from an Instant Messaging client).
VM	*N/A*	`vm://BrokerName`	Configure clients to connect to a broker embedded within the same Java Virtual Machine (JVM). The `BrokerName` is the broker name of the embedded broker.

Discovery Protocols

Overview

A discovery protocol builds a connection to a message broker in two steps, as follows:

1. Obtain a list of available broker endpoints (represented by URIs).
2. Connect to an endpoint randomly selected from the given list.

Discovery protocols are particularly useful for clients that connect to a cluster of message brokers.

Summary of discovery protocols

Table 1.1 on page 12 describes the discovery protocols that clients can use.

Table 1.2. Summary of Discovery Protocols

Protocol	Sample URL	Description
Failover	`failover://(uri1,...,uriN)?TransportOptions`	Configure clients to connect to one of the broker endpoints from the URI list, `uri1,...,uriN`. The transport options, `?TransportOptions`, are specified in the form of a query list. If no transport options are required, you can omit the parentheses and the question mark, `?`.
Discovery	`discovery://(DiscoveryAgentUri)?TransportOptions`	Configure clients to connect to one of the broker endpoints from a URI list that is dynamically discovered at runtime, using a discovery agent. The discovery agent URI, `DiscoveryAgentUri`, is normally a multicast discovery agent—for example, `multicast://default`.

Discovery agents

The discovery protocol supports a number of discovery agents, which are also specified in the form of a URI. For details of the supported discovery agents, see "Discovery Agents" on page 87 .

Note

Although discovery agent URIs look superficially like transport URIs, they are not the same thing. A discovery agent URI can only be used in certain contexts and *cannot* be used in place of a transport URI.

Peer-to-Peer Protocols

Overview

Peer-to-peer protocols enable messaging clients to communicate with each other directly, eliminating the requirement to route messages through an external message broker.

Summary of peer-to-peer protocols

Table 1.3 on page 16 describes the peer-to-peer protocols that clients can use.

Table 1.3. Summary of Peer-to-Peer Protocols

Protocol	Sample URL	Description
Peer	`peer://PeerGroup/BrokerName?BrokerOptions`	Configure clients to connect to their peers in the group with the group name, *`PeerGroup`*. The *`BrokerName`* specifies the broker name for the embedded broker. The broker options, *`BrokerOptions`*, are specified in the form of a query list (for example, `?persistent=true`).

Broker options

The `peer` protocol supports a variety of broker options. For details, see the broker options listed in Table 7.1 on page 93 .

Chapter 2. OpenWire Protocol

The OpenWire protocol is the default on-the-wire protocol for FUSE Message Broker. This chapter provides a brief introduction to the protocol, illustrating how to use OpenWire with a variety of transport protocols.

Introduction to the OpenWire Protocol

Overview

The OpenWire protocol is a JMS compliant wire protocol (defining message types and message encodings) that is native to the FUSE Message Broker. The protocol is designed to be fully-featured, JMS-compliant, and highly performant. It is the default protocol of the FUSE Message Broker.

Transport protocols

Table 2.1 on page 18 shows the transport protocols supported by the OpenWire wire protocol:

Table 2.1. Transport Protocols Supported by OpenWire

Transport Protocol	URL	Description
TCP	`tcp://Host:Port`	Endpoint URL for OpenWire over TCP/IP. The broker listens for TCP connections on the host machine, *`Host`*, and IP port, *`Port`*.
SSL	`ssl://Host:Port`	*(Java clients only)* Endpoint URL for OpenWire over TCP/IP, where the socket layer is secured using SSL (or TLS). For details of how to configure an SSL connection, see the *FUSE Message Broker Security Guide*.
HTTP	`http://Host:Port`	*(Java clients only)* Endpoint URL for OpenWire over HTTP.
HTTPS	`https://Host:Port`	*(Java clients only)* Endpoint URL for OpenWire over HTTP, where the socket layer is secured by SSL (or TLS). For details of how to configure a HTTPS connection, see the *FUSE Message Broker Security Guide*. [REVISIT - Insert Olink.]

Transport options

OpenWire supports a number of transport options, which can be set as query options on the transport URL. For example, to specify that error messages should omit the stack trace, use a URL like the following:

```
tcp://localhost:61616?wireformat.stackTraceEnabled=false
```

Where the `wireformat.stackTraceEnabled` property is set to `false` to disable the inclusion of stack traces in error messages. Table 2.2 on page 19 gives the complete list of transport options for OpenWire.

Table 2.2. Transport Options Supported by OpenWire Protocol

Property	Default	Description	Negotiation policy
`wireformat .stackTraceEnabled`	`true`	Should the stack trace of an exception occuring on the broker be sent to the client?	Set to `false` if either side is `false`.
`wireformat .tcpNoDelayEnabled`	`false`	Provides a hint to the peer that TCP `nodelay` should be enabled on the communications Socket.	Set to `false` if either side is `false`.
`wireformat .cacheEnabled`	`true`	Should commonly repeated values be cached so that less marshalling occurs?	Set to `false` if either side is `false`.
`wireformat .tightEncodingEnabled`	`true`	Should wire size be optimized over CPU usage?	Set to `false` if either side is `false`.
`wireformat .prefixPacketSize`	`true`	Should the size of the packet be prefixed before each packet is marshalled?	Set to `true` if both sides are `true`.
`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. Set to a value `<= 0` to disable inactivity monitoring.	Use the smaller of the two values.
`wireformat .cacheSize`	`1024`	If `cacheEnabled` is `true`, this property specifies the maximum number of values to cache.	Use the smaller of the two values.

Supported clients

FUSE Message Broker currently supports the following client types for the OpenWire protocol:

- *Java clients*—the Java API conforms fully to the JMS specification.

If you want to develop an OpenWire client using other programming languages, try one of the following client types from the Apache ActiveMQ[1] project:

[1] http://activemq.apache.org/

- *C++ clients*—for C++ clients, Apache ActiveMQ provides the CMS (C++ Messaging Service) API, which is closely modelled on the JMS specification. Only the TCP transport is supported for C++ clients.

OpenWire Example

Overview

It is relatively straightforward to try out the various OpenWire+transport combinations using the sample code provided. After configuring the broker to add the relevant transport connectors, you can use the sample producer tool and the consumer tool to transmit messages through the broker using the following protocols: OpenWire over TCP or OpenWire over HTTP.

Note

The secure socket protocols—OpenWire over SSL, and OpenWire over HTTPS—are discussed in the *FUSE Message Broker Security Guide*.

Example prerequisites

Before you can build and run the sample clients, you must have installed the Apache Ant build tool, version 1.6 or later (see http://ant.apache.org/).

The OpenWire examples depend on the sample producer and consumer clients located in the following directory:

```
FUSEInstallDir/fuse-message-broker-Version/example
```

Example steps

To try out the OpenWire protocol, perform the following steps:

1. "Configure the broker" on page 21 .
2. "Run the broker" on page 22 .
3. "Run the consumer" on page 22 .
4. "Run the producer with the TCP protocol" on page 23 .
5. "Run the producer with the HTTP protocol" on page 24 .

Configure the broker

Add the following transport connectors to the default broker configuration file (in `fuse-message-broker-Version/conf/activemq.xml`):

```
<beans>
    ...
    <transportConnectors>
       ...
       <transportConnector name="openwire" uri="tcp://local
host:61616"/>
       <transportConnector name="http"   uri="http://local
host:61620"/>    </transportConnectors>
    ...
</beans>
```

Run the broker

Run the default broker by entering the following at a command line:

```
activemq
```

The default broker automatically takes its configuration from the default configuration file.

Note

The `activemq` script automatically sets the `ACTIVEMQ_HOME` and `ACTIVEMQ_BASE` environment variables to *`FUSEInstallDir`*`/fuse-message-broker-`*`Version`* by default. If you want the `activemq` script to pick up its configuration from a non-default `conf` directory, you can set `ACTIVEMQ_BASE` explicitly in your environment. The configuration files will then be taken from `$ACTIVEMQ_BASE/conf`.

Run the consumer

To connect the consumer tool to the `tcp://localhost:61616` endpoint (OpenWire over TCP), change directory to `fuse-message-broker-`*`Version`*`/example` and enter the following command:

```
ant consumer -Durl=tcp://localhost:61616 -Dmax=100
```

You should see some output like the following:

```
Buildfile: build.xml
init:
compile:
consumer:
     [echo] Running consumer against server at $url = tcp://local
```

```
host:61616
for subject $subject = TEST.FOO
     [java] Connecting to URL: tcp://localhost:61616
     [java] Consuming queue: TEST.FOO
     [java] Using a non-durable subscription
     [java] We are about to wait until we consume: 100 message(s)
then
we will shutdown
```

Run the producer with the TCP protocol

To connect the producer tool to the `tcp://localhost:61616` endpoint (OpenWire over TCP), open a new command prompt, change directory to `fuse-message-broker-`*`Version`*`/example` and enter the following command:

```
ant producer -Durl=tcp://localhost:61616
```

In the window where the *consumer tool* is running, you should see some output like the following:

```
     [java] Received: Message: 0 sent at: Wed Sep 19 14:38:06
 BST
2007  ...
     [java] Received: Message: 1 sent at: Wed Sep 19 14:38:06
 BST
2007  ...
     [java] Received: Message: 2 sent at: Wed Sep 19 14:38:06
 BST
2007  ...
     [java] Received: Message: 3 sent at: Wed Sep 19 14:38:06
 BST
2007  ...
     [java] Received: Message: 4 sent at: Wed Sep 19 14:38:06
 BST
2007  ...
     [java] Received: Message: 5 sent at: Wed Sep 19 14:38:06
 BST
2007  ...
     [java] Received: Message: 6 sent at: Wed Sep 19 14:38:06
 BST
2007  ...
     [java] Received: Message: 7 sent at: Wed Sep 19 14:38:06
 BST
2007  ...
     [java] Received: Message: 8 sent at: Wed Sep 19 14:38:06
 BST
2007  ...
```

```
    [java] Received: Message: 9 sent at: Wed Sep 19 14:38:06
 BST
2007  ...
```

Run the producer with the HTTP protocol

To connect the producer tool to the `http://localhost:61620` endpoint (OpenWire over HTTP), enter the following command from the `example` directory:

```
ant producer -Durl=http://localhost:61620
```

This command sends ten new messages to the consumer client.

Note

The JAR files for the HTTP protocol are currently located in the `lib/optional` subdirectory. If you construct the CLASSPATH manually, you must be sure to include the JAR files from this subdirectory.

Chapter 3. Stomp Protocol

The Stomp protocol is a simplified messaging protocol that is specially designed for implementing clients using scripting languages. This chapter provides a brief introduction to the protocol, illustrating how to run Stomp clients implemented in Ruby.

Introduction to the Stomp Protocol

Overview

The Stomp protocol is a simplified messaging protocol that is being developed as an open source project (http://stomp.codehaus.org/). The advantage of the stomp protocol is that you can easily improvise a messaging client—even when a specific client API is not available—because the protocol is so simple.

Transport protocols

Table 3.1 on page 26 shows the transport protocols supported by the Stomp wire protocol:

Table 3.1. Transport Protocols Supported by Stomp

Transport Protocol	URL	Description
TCP	`stomp://Host:Port`	Endpoint URL for Stomp over TCP/IP. The broker listens for TCP connections on the host machine, *`Host`*, and IP port, *`Port`*.
SSL	`stomp+ssl://Host:Port`	Endpoint URL for secure Stomp over SSL. The broker listens for TCP connections on the host machine, *`Host`*, and IP port, *`Port`*.

Supported clients

Stomp currently supports the following client types:

- *C clients*.
- *C++ clients*.
- *C# and .NET clients*.
- *.NET clients*.
- *Delphi clients*.
- *Flash clients*.
- *Perl clients*.
- *PHP clients*.
- *Pike clients*.

- *Python clients*.

Stomp Example

Overview

FUSE Message Broker provides some sample code in `fuse-message-broker-Version/example/ruby` that enables you to experiment with the Stomp protocol in the Ruby programming language.

Example prerequisites

You must download and install the requisite packages to support the Ruby programming language before you can run the Stomp example. Install the following packages:

- *Ruby programming language*—download and install the Ruby programming language from http://www.ruby-lang.org/en/downloads. Add the Ruby `/bin` directory to your `PATH`.
- *RubyGems package manager*—RubyGems (http://www.rubygems.org) is a utility for installing and managing add-ons to the Ruby language. Download and install RubyGems as follows:
  1. Download a RubyGems archive file (`.tgz`, `.zip`, or `.gem`) from the RubyForge (http://rubyforge.org/frs/?group_id=126).
  2. Unzip the RubyGems archive.
  3. Initialize RubyGems by entering the following command:

     ```
     ruby GemsInstallDir/setup.rb
     ```

  4. Add `GemsInstallDir/bin` to your `PATH`.
- *Stomp package for Ruby*—install the Stomp package for Ruby by running the following command:

```
gem install stomp
```

RubyGems downloads and installs the requisite package to support the Ruby Stomp client API.

Example steps

To try out the Stomp protocol, perform the following steps:

1. "Configure the broker" on page 29 .
2. "Run the broker" on page 29 .
3. "Run the Ruby listener" on page 30 .
4. "Run the Ruby publisher" on page 30

Configure the broker

Check that the the Stomp connector is present in the default broker configuration file (in `fuse-message-broker-`*`Version`*`/conf/activemq.xml`), as follows:

```
<beans>
    ...
    <transportConnectors>
       ...
       <transportConnector name="stomp"   uri="stomp://local
host:61613"/>
    </transportConnectors>
    ...
</beans>
```

Run the broker

Run the default broker by entering the following at a command line:

```
activemq
```

The default broker automatically takes its configuration from the default configuration file.

Note

The `activemq` script automatically sets the `ACTIVEMQ_HOME` and `ACTIVEMQ_BASE` environment variables to *`FUSEInstallDir`*`/fuse-message-broker-`*`Version`* by default. If

you want the `activemq` script to pick up its configuration from a non-default `conf` directory, you can set `ACTIVEMQ_BASE` explicitly in your environment. The configuration files will then be taken from `$ACTIVEMQ_BASE/conf`.

Run the Ruby listener

To connect the listener tool to the `stomp://localhost:61613` endpoint (Stomp over TCP), change directory to `fuse-message-broker-`*`Version`*`/example/ruby` and enter the following command:

```
ruby listener.rb
```

They Ruby listener connects to the endpoint, `stomp://localhost:61613`, by default. You could change this endpoint address by editing the `listener.rb` script.

Run the Ruby publisher

To connect the publisher tool to the `stomp://localhost:61613` endpoint (Stomp over TCP), change directory to `fuse-message-broker-`*`Version`*`/example/ruby` and enter the following command:

```
ruby publisher.rb
```

You should see some output like the following:

```
Sent 1000 messages
Sent 2000 messages
Sent 3000 messages
Sent 4000 messages
Sent 5000 messages
Sent 6000 messages
Sent 7000 messages
Sent 8000 messages
Sent 9000 messages
Sent 10000 messages
Received report: Received 10000 in 4.567 seconds, remaining:
 9
```

Protocol Details

Overview

This section describes the format of Stomp data packets , as well as the semantics of the data packet exchanges. Stomp is a relatively simple wire protocol—it is even possible to communicate manually with a Stomp broker using a `telnet` client (see "Stomp Tutorial" on page 40).

Transport protocols

In principal, Stomp can be combined with any transport protocol, including connection-oriented and non-connection-oriented transports. In practice, though, Stomp is usually implemented over TCP and this is the only transport currently supported by FUSE Message Broker.

Licence

The Stomp specification is licensed under the Creative Commons Attribution v2.5[1]

Stomp frame format

The Stomp specification defines the term *frame* to refer to the data packets transmitted over a Stomp connection. A Stomp frame has the following general format:

```
<StompCommand>
<HeaderName_1>:<HeaderValue_1>
<HeaderName_2>:<HeaderValue_2>

<FrameBody>
^@
```

A Stomp frame always starts with a Stomp command (for example, `SEND`) on a line by itself. The Stomp command may then be followed by zero or more header lines: each header is in a `<key>:<value>` format and terminated by a newline. A blank line indicates the end of the headers and the beginning of the body, `<FrameBody>`, (which is empty for many of the commands). The frame is terminated by the null character, which is represented as `^@` above (Ctrl-@ in ASCII)).

Oneway and RPC commands

Most Stomp commands have oneway semantics (that is, after sending a frame, the sender does not expect any reply). The only exceptions are:

[1] http://creativecommons.org/licenses/by/2.5/

- `CONNECT` *command*—after a client sends a `CONNECT` frame, it expects the server to reply with a `CONNECTED` frame.
- *Commands with a* `receipt` *header*—a client can force the server to acknowledge receipt of a command by adding a `receipt` header to the outgoing frame.
- *Erroneous commands*—if a client sends a frame that is malformed, or otherwise in error, the server may reply with an `ERROR` frame. Note, however, that the `ERROR` frame is not formally correlated with the original frame that caused the error (Stomp frames are not required to include a unique identifier).

Receipt header

Any client frame, other than `CONNECT`, may specify a `receipt` header with an arbitrary value. This causes the server to acknowledge receipt of the frame with a `RECEIPT` frame, which contains the value of this header as the value of the `receipt-id` header in the `RECEIPT` frame. For example, the following frame shows a `SEND` command that includes a receipt header:

```
SEND
destination:/queue/a
receipt:message-12345

Hello a!^@
```

Client commands

Table 3.2 on page 32 lists the client commands for the Stomp protocol. The Reply column indicates whether or not the server sends a reply frame by default.

Table 3.2. Client Commands for the Stomp Protocol

Command	Reply?	Role	Description
"CONNECT" on page 33	Yes	Producer, Consumer	Open a connection to a Stomp broker (server).
"SEND" on page 33	No	Producer	Send a message to a particular queue or topic on the server.
"SUBSCRIBE" on page 34	No	Consumer	Subscribe to a particular queue or topic on the server.

Command	Reply?	Role	Description
"UNSUBSCRIBE" on page 35	No	Consumer	Cancel a subscription to a particular queue or topic.
"ACK" on page 36	No	Consumer	Acknowledge receipt of one message.
"BEGIN" on page 36	No	Producer, Consumer	Start a transaction (applies to `SEND` or `ACK` commands).
"COMMIT" on page 36	No	Producer, Consumer	Commit a transaction.
"ABORT" on page 37	No	Producer, Consumer	Roll back a transaction.
"DISCONNECT" on page 37	No	Producer, Consumer	Shut down the existing connection gracefully.

CONNECT

After opening a socket to connect to the remote server, the client sends a `CONNECT` command to initiate a Stomp session. For example, the following frame shows a typical `CONNECT` command, including a `login` header and a `passcode` header:

```
CONNECT
login: <username>
passcode:<passcode>

^@
```

After the client sends the `CONNECT` frame, the server always acknowledges the connection by sending a frame, as follows:

```
CONNECTED
session: <session-id>

^@
```

The `session-id` header is a unique identifier for this session (currently unused).

SEND

The `SEND` command sends a message to a *destination*—for example, a queue or a topic—in the messaging system. It has one required header, `destination`, which indicates where to send the message. The body of the

SEND command is the message to be sent. For example, the following frame sends a message to the /queue/a destination:

```
SEND
destination:/queue/a
hello queue a

^@
```

From the client's perspective, the destination name, /queue/a, is an arbitrary string. Despite seeming to indicate that the destination is a queue it does not, in fact, specify any such thing. Destination names are simply strings that are mapped to some form of destination on the server; how the server translates these strings is up to the implementation.

The SEND command also supports the following optional headers:

- transaction—specifies the transaction ID. Include this header, if the SEND command partakes in a transaction (see "BEGIN" on page 36).
- content-length—specifies the byte count for the length of the message body. If a content-length header is included, this number of bytes should be read, regardless of whether or not there are null characters in the body. The frame still needs to be terminated with a null byte and if a content-length is not specified, the first null byte encountered signals the end of the frame.

SUBSCRIBE

The SUBSCRIBE command registers a client's interest in listening to a specific destination. Like the SEND command, the SUBSCRIBE command requires a destination header. Henceforth, any messages received on the subscription are delivered as MESSAGE frames, from the server to the client. For example, the following frame shows a client subscribing to the destination, /queue/a:

```
SUBSCRIBE
destination: /queue/foo
ack: client
```

```
^@
```

In this case the `ack` header is set to `client`, which means that messages are considered delivered only after the client specifically acknowledges them with an `ACK` frame. The body of the `SUBSCRIBE` command is ignored.

The `SUBSCRIBE` command supports the following optional headers:

- `ack`—specify the acknowledgement mode for this subscription. The following modes are recognized:
  - `auto`—messages are considered delivered as soon as the server delivers them to the client (in the form of a `MESSAGE` command). The server does *not* expect to receive any `ACK` commands from the client for this subscription.
  - `client`—messages are considered delivered only after the client specifically acknowledges them with an `ACK` frame.
- `selector`—specifies a SQL 92 selector on the message headers, which acts as a filter for content based routing.
- `id`—specify an ID to identify this subscription. Later, you can use the ID to `UNSUBSCRIBE` from this subscription (you may end up with overlapping subscriptions, if multiple selectors match the same destination).

  When an `id` header is supplied, the server should append a `subscription` header to any `MESSAGE` commands sent to the client. When using wildcards and selectors, this enables clients to figure out which subscription triggered the message.

UNSUBSCRIBE

The `UNSUBSCRIBE` command removes an existing subscription, so that the client no longer receives messages from that destination. It requires either a `destination` header or an `id` header (if the previous `SUBSCRIBE` operation passed an `id` value). For example, the following frame cancels the subscription to the `/queue/a` destination:

```
UNSUBSCRIBE
destination: /queue/a
```

```
^@
```

ACK

The ACK command acknowledges the consumption of a message from a subscription. If the client issued a SUBSCRIBE frame with an ack header set to client, any messages received from that destination are not considered to have been consumed until the message is acknowledged by an ACK frame.

The ACK command has one required header, message-id, which must contain a value matching the message-id for the MESSAGE being acknowledged. Optionally, a transaction header may be included, if the acknowledgment participates in a transaction. For example, the following frame acknowledges a message in the context of a transaction:

```
ACK
message-id: <message-identifier>
transaction: <transaction-identifier>

^@
```

BEGIN

The BEGIN command initiates a transaction. Transactions can be applied to SEND and ACK commands. Any messages sent or acknowledged during a transaction can either be commited or rolled back at the end of the transaction.

```
BEGIN
transaction: <transaction-identifier>

^@
```

The transaction header is required and the <transaction-identifier> can be included in SEND, COMMIT, ABORT, and ACK frames to bind them to the named transaction.

COMMIT

The COMMIT command commits a specific transaction.

```
COMMIT
transaction: <transaction-identifier>
```

```
^@
```

The `transaction` header is required and specifies the transaction, `<transaction-identifier>`, to commit.

ABORT

The `ABORT` command rolls back a specific transaction.

```
ABORT
transaction: <transaction-identifier>

^@
```

The `transaction` header is required and specifies the transaction, `<transaction-identifier>`, to roll back.

DISCONNECT

The `DISCONNECT` command disconnects gracefully from the server.

```
DISCONNECT

^@
```

Server commands

Table 3.3 on page 37 lists the commands that the server can send to a Stomp client. These commands all have oneway semantics.

Table 3.3. Server Commands for the Stomp Protocol

Command	Description
"MESSAGE" on page 37	Send a message to the client, where the client has previously registered a subscription with the server.
"RECEIPT" on page 38	Acknowledges receipt of a client command, if the client requested a receipt by included a `receipt-id` header.
"ERROR" on page 38	Error message sent from the server to the client.

MESSAGE

The `MESSAGE` command conveys messages from a subscription to the client. The `MESSAGE` frame must include a `destination` header, which identifies the destination from which the message is taken. The `MESSAGE` frame also contains a `message-id` header with a unique message identifier. The frame

body contains the message contents. For example, the following frame shows a typical MESSAGE command with destination and message-id headers:

```
MESSAGE
destination:/queue/a
message-id: <message-identifier>

hello queue a^@
```

The MESSAGE command supports the following optional headers:

- content-length—specifies the byte count for the length of the message body. If a content-length header is included, this number of bytes should be read, regardless of whether or not there are null characters in the body. The frame still needs to be terminated with a null byte and if a content-length is not specified, the first null byte encountered signals the end of the frame.

RECEIPT

A RECEIPT frame is issued from the server whenever the client requests a receipt for a given command. The RECEIPT frame includes a receipt-id, containing the value of the receipt-id from the original client request. For example, the following frame shows a typical RECEIPT command with receipt-id header:

```
RECEIPT
receipt-id:message-12345

^@
```

The receipt body is always empty.

ERROR

The server may send ERROR frames if something goes wrong. The error frame should contain a message header with a short description of the error. The body may contain more detailed information (or may be empty). For example, the following frame shows an ERROR command with a non-empty body:

```
ERROR
message: malformed packet received

The message:
-----
```

```
MESSAGE
destined:/queue/a
Hello queue a!
-----
Did not contain a destination header, which is required for
message
propagation.
^@
```

The `ERROR` command supports the following optional headers:

- `content-length`—specifies the byte count for the length of the message body. If a `content-length` header is included, this number of bytes should be read, regardless of whether or not there are null characters in the body. The frame still needs to be terminated with a null byte and if a content-length is not specified, the first null byte encountered signals the end of the frame.

Stomp Tutorial

Telnet client

Because Stomp frames consist of plain text, it is possible to improvise a Stomp client by starting up a `telnet` session and entering Stomp frames directly at the keyboard. This can be a useful diagnostic tool and is also a good way to learn about the Stomp protocol.

Typing the null character

While most characters in a Stomp frame are just plain text, there is one required character, null, that you might have difficulty typing at the keyboard. On some keyboards, you can type null as Ctrl-@. Other keyboards might require you to do a bit of research, however.

For example, to type a null character on the 101-key keyboard that is commonly used with a Windows PC, proceed as follows:

1. Enable `NumLock` on the numeric keypad (this is essential).
2. While holding down the `Alt` key, type zero, `0`, four times in succession *on the numeric keypad*.

Tutorial steps

To send and receive messages over the Stomp protocol using telnet clients, perform the following steps:

1. "Start the broker" on page 41 .
2. "Start a telnet session for the producer" on page 41 .
3. "Start a Stomp session for the producer" on page 41 .
4. "Send a message to a queue" on page 42 .
5. "Check the queue status using JMX" on page 42 .
6. "Start a telnet session for the consumer" on page 44 .
7. "Start a Stomp session for the consumer" on page 44 .
8. "Subscribe to a queue" on page 45 .
9. "Acknowledge a message" on page 45 .

10. "Unsubscribe from the queue" on page 46 .

11. "Disconnect the clients" on page 46 .

Start the broker

Start the default broker by entering the following at a command prompt:

```
activemq
```

Normally, the default broker is configured to initialize a Stomp connector that listens on port, 61613. Look for a line like the following in the broker's log:

```
INFO  TransportServerThreadSupport   - Listening for connec
tions at:
stomp://localhost:61613
```

If the Stomp connector is not present in the broker, you will have to configure it—see "Configure the broker" on page 29 for details.

Start a telnet session for the producer

Open a new command prompt and start a new `telnet` session for the producer client, by entering the following command:

```
telnet
```

This command starts `telnet` in interactive mode. Now enter the following `telnet` commands (the `telnet` prompt that begins each line is implementation dependent):

```
telnet> set localecho
Local echo on
telnet> open localhost 61613
```

After entering the `open` command, `telnet` should connect to the Stomp socket on your local ActiveMQ broker (where the Stomp port is presumed to be 61613 here). You should now see a blank screen, where you can directly type the contents of the Stomp frames you want to send over TCP.

Start a Stomp session for the producer

Start a Stomp session for the producer by entering the following Stomp frame in the `telnet` window:

```
CONNECT
login:foo
passcode:bar

^@
```

The `login` and `passcode` headers are currently ignored by the ActiveMQ broker, so you can enter any values you like for these headers. *Don't forget to insert a blank line after the headers.* Finally, you must terminate the frame by typing the null character, `^@` (for notes on how to type the null character at your keyboard, see "Typing the null character" on page 40).

If all goes well, you will see a response similar to the following:

```
CONNECTED
session:ID:fboltond820-2290-1190810591249-3:0
```

Send a message to a queue

Send a message to the `FOO.BAR` queue by entering the following frame:

```
SEND
destination:/queue/FOO.BAR
receipt:

Hello, queue FOO.BAR
^@
```

As soon as you have finished typing the null character, `^@`, you should receive the following `RECEIPT` frame from the server:

```
RECEIPT
receipt-id:
```

It is a good idea to include a `receipt` header in the frames you send from a telnet client. It enables you to confirm that the connection is working normally.

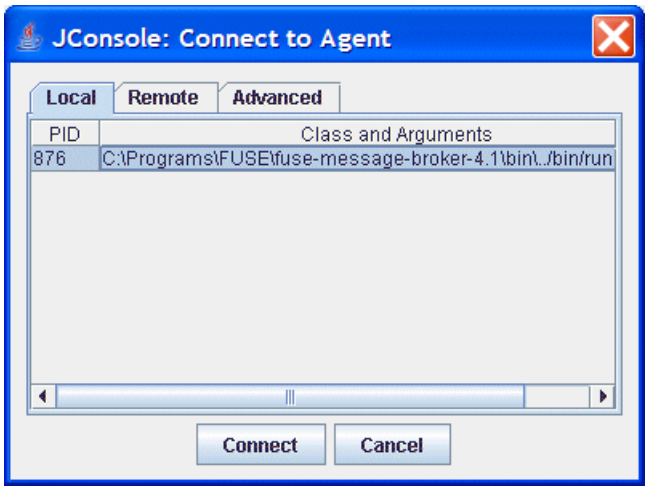
Check the queue status using JMX

The status of the ActiveMQ broker can be monitored through a JMX port. To monitor the broker, start a new command prompt and enter the following command:

```
jconsole
```

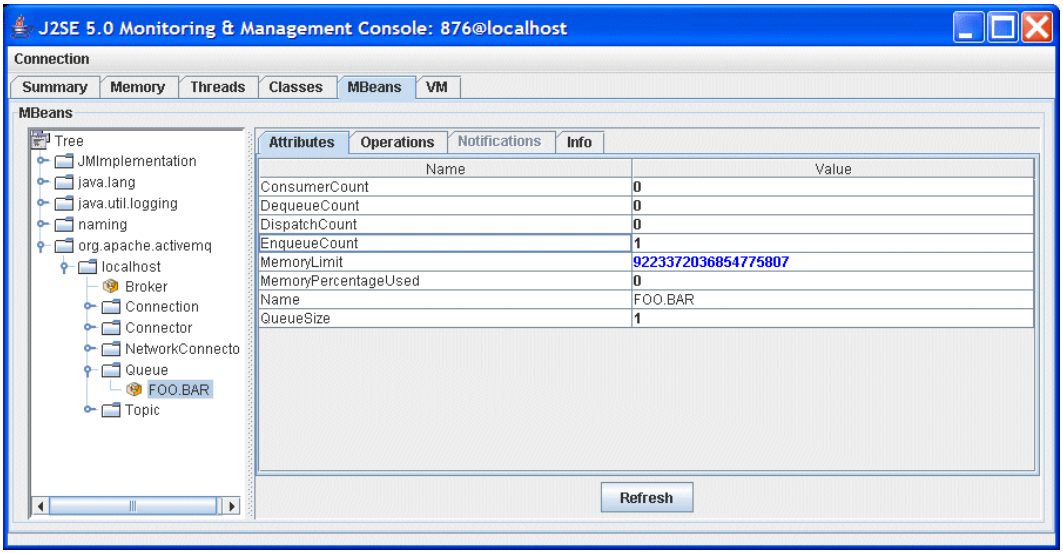
The `jconsole` utility is a standard JMX client that is included with Sun's Java Development Kit (JDK). When you start the `jconsole` utility, a dialog appears and prompts you to connect to a JMX process, as shown in Figure 3.1 on page 43 .

Figure 3.1. Connecting to the ActiveMQ JMX Port



Select the ActiveMQ broker process and click **Connect**. The main `jconsole` window opens. To view the current status of the `FOO.BAR` message queue, click on the **MBeans** tab and use the tree on the left hand side to drill down to `org.apache.activemq/localhost/Queue/FOO.BAR`. Click on the `FOO.BAR` icon to view the current status, as shown in Figure 3.2 on page 44 .

Figure 3.2. Monitoring the Status of the FOO.BAR Queue



The status shows an **EnqueueCount** of 1, which tells you that the producer has successfully enqueued one message in the `FOO.BAR` queue.

Start a telnet session for the consumer

Open a new command prompt and start a new `telnet` session for the consumer client, by entering the following command:

```
telnet
```

Enter the following `telnet` commands to connect to the Stomp socket on the broker:

```
telnet> set localecho
Local echo on
telnet> open localhost 61613
```

Start a Stomp session for the consumer

Start a Stomp session for the consumer by entering the following Stomp frame in the consumer's `telnet` window:

```
CONNECT
```

```
login:foo
passcode:bar

^@
```

If all goes well, you will see a response similar to the following:

```
CONNECTED
session:ID:fboltond820-2290-1190810591249-3:1
```

Subscribe to a queue

Subscribe to the `FOO.BAR` queue by entering the following Stomp frame in the consumer's `telnet` window:

```
SUBSCRIBE
destination:/queue/FOO.BAR
ack:client

^@
```

The `ack` header is set to the value `client`, which implies that the consumer client is expected to acknowledge each message it receives from the broker. After typing the terminating null character, `^@`, the broker dispatches the sole message on the `FOO.BAR` queue by sending a `MESSAGE` frame, as follows:

```
MESSAGE
destination:/queue/FOO.BAR
receipt:
timestamp:1190811984837
priority:0
expires:0
message-id:ID:fboltond820-2290-1190810591249-3:0:-1:1:1

Hello, queue FOO.BAR
```

To see what effect this has on the queue status, go to the `jconsole` window and click **Refresh** on the MBeans tab. The **DispatchCount** attribute is now equal to 1, indicating that the broker has dispatched the message to the consumer. The **DequeueCount** is equal to 0, however; this is because the message is not considered to be dequeued until the consumer client sends an acknowledgement.

Acknowledge a message

Acknowledge the received message by entering the following Stomp frame in the consumer's `telnet` window:

```
ACK
message-id:ID:fboltond820-2290-1190810591249-3:0:-1:1:1

^@
```

Where the message ID must match the value from the `message-id` header in the received `MESSAGE` frame. To check that the acknowledgement has been effective, go back to the `jconsole` window and click **Refresh** on the **MBeans** tab. You should now find that the **DequeueCount** has increased to 1.

Unsubscribe from the queue

Unsubscribe from the `FOO.BAR` queue by entering the following Stomp frame in the consumer's `telnet` window:

```
UNSUBSCRIBE
destination:/queue/FOO.BAR
receipt:

^@
```

Disconnect the clients

To shut down both the producer and consumer gracefully, enter the following `DISCONNECT` frame in each of their respective `telnet` windows:

```
DISCONNECT

^@
```

Chapter 4. REST Protocols

The REST protocol is a simple HTTP-based protocol that enables you to interact with the message broker using HTML forms and DHTML scripts. This chapter provides a brief introduction to the protocol, illustrating how to contact the message broker from a Web browser.

Introduction to the REST Protocol

Overview

The REST protocol is a simple HTTP-based protocol that enables you to contact the message broker through a Web browser. You can contact the message broker by navigating to appropriately formatted URLs or by posting HTML forms.

Transport protocols

The FUSE Message Broker's REST protocol is based on a subset of the HTTP protocol. Hence, HTTP is the only supported transport.

Supported clients

REST supports the following client types:

- *Web forms*—use conventional HTML forms to `POST` a message to a destination (queue or topic) or to `GET` a message from a destination—see "Example of posting a message" on page 62 .
- *Ajax clients*—an Asynchronous JavaScript And Xml (Ajax) library that enables you to communicate with a REST endpoint using JavaScript in a DHTML Web page.

REST servlets

The REST protocol is implemented by the following servlets running in a Web container:

- `message` *servlet*—supports the sending and consuming of messages.
- `queueBrowse` *servlet*—enables you to view the current status of a particular queue.

REST Example

Overview

This section describes how to run the REST example, which consists of a servlet engine integral to the message broker binary, and some demonstration servlets that run as a Web application. To connect to the Web applications, you can use your favorite Web browser.

Example prerequisites

You must ensure that the message broker is configured to instantiate an embedded servlet engine. In your broker configuration file, `conf/activemq.xml`, check that there is a `jetty` element configured as shown in Example 4.1 on page 49 .

Example 4.1. Configuration of an Embedded Servlet Engine

```
<!-- Embedded servlet engine for serving up the Admin console
-->
<jetty xmlns="http://mortbay.com/schemas/jetty/1.0">
  <connectors>
    <nioConnector port="8161" />
  </connectors>
  <handlers>
    <webAppContext contextPath="/admin"
      resourceBase="${activemq.base}/webapps/admin"
      logUrlOnStart="true" />
    <webAppContext contextPath="/demo"
      resourceBase="${activemq.base}/webapps/demo"
      logUrlOnStart="true" />
  </handlers>
</jetty>
```

With the configuration shown in Example 4.1 on page 49 , the servlet engine opens up a HTTP port on IP port, 8161. The following Web applications are loaded:

- Demonstration application (from `webapps/demo`),
- REST protocol servlets (from `webapps/demo`).
- Web console servlet (from `webapps/admin`),

Example steps

To run the REST Web example, perform the following steps:

1. "Run the servlet engine" on page 50 .
2. "Open a Web browser" on page 50 .
3. "Send a message" on page 51 .
4. "Browse the message queue" on page 52 .
5. "Receive a message from the queue" on page 56 .

Run the servlet engine

To run the embedded servlet engine, open a new command window and enter the following command to start the default message broker:

```
activemq
```

This step assumes that your broker is configured as described in "Example prerequisites" on page 49 .

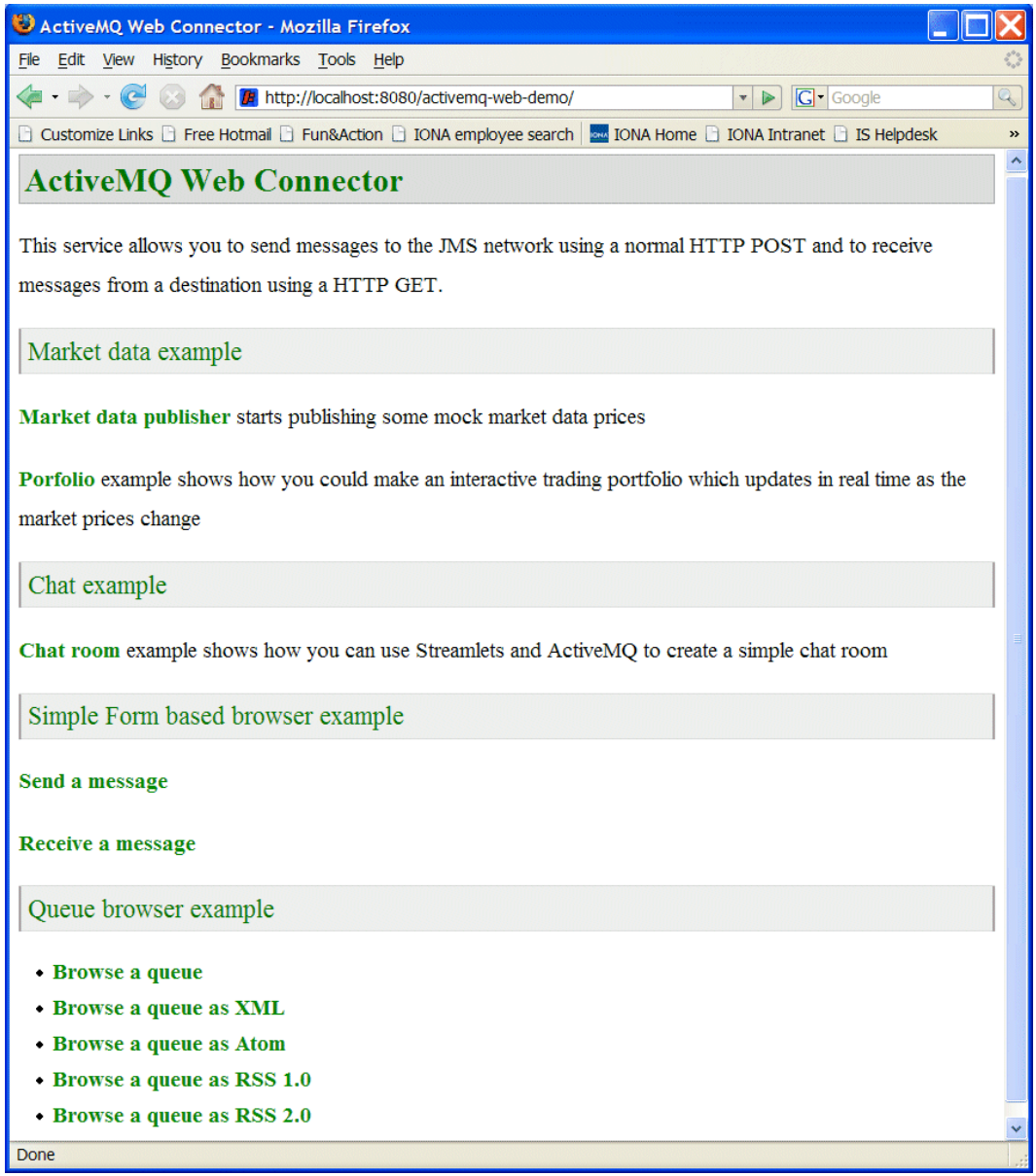
Open a Web browser

Open your favorite Web browser (for example, Firefox or Internet Explorer) and navigate to the following URL:

```
http://localhost:8161/demo
```

Your browser should now show the welcome page for the Web examples, as shown in Figure 4.1 on page 51 .

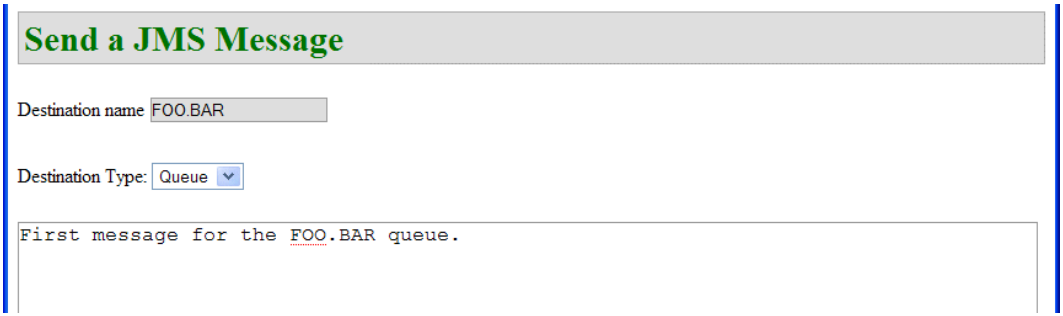
Figure 4.1. Welcome Page for Web Examples



Send a message

To view the form for publishing messages, click the link, Send a message[1]. The **Send a JMS Message** form now appears in your browser, as shown in Figure 4.2 on page 52.

Figure 4.2. The Send a JMS Message Form



In the **Destination name** text field, enter `FOO.BAR` to send a message to the `FOO.BAR` queue. Leave the **Destination Type** as `Queue`. Then enter an arbitrary text message in the large message text box. Click the **Send** button at the bottom of the form to send the message.

Browse the message queue

Using the history feature of your browser, navigate back to the example welcome page (see Figure 4.1 on page 51). The `queueBrowse` servlet supports a variety of ways to browse the contents of a queue and these are listed at the bottom of the welcome page. The following browsing options are listed:

- Browse a queue.[2]
- Browse a queue as XML[3].
- Browse a queue as Atom[4].
- Browse a queue as RSS 1.0[5].
- Browse a queue as RSS 2.0[6].

[1] http://localhost:8080/activemq-web-demo/send.html
[2] http://localhost:8080/activemq-web-demo/queueBrowse/FOO/BAR
[3] http://localhost:8080/activemq-web-demo/queueBrowse/FOO/BAR?view=xml
[4] http://localhost:8080/activemq-web-demo/queueBrowse/FOO/BAR?view=rss&feedType=atom_1.0
[5] http://localhost:8080/activemq-web-demo/queueBrowse/FOO/BAR?view=rss&feedType=rss_1.0
[6] http://localhost:8080/activemq-web-demo/queueBrowse/FOO/BAR?view=rss&feedType=rss_2.0

If you click on **Browse a queue**, you should see a page like Figure 4.3 on page 53 .

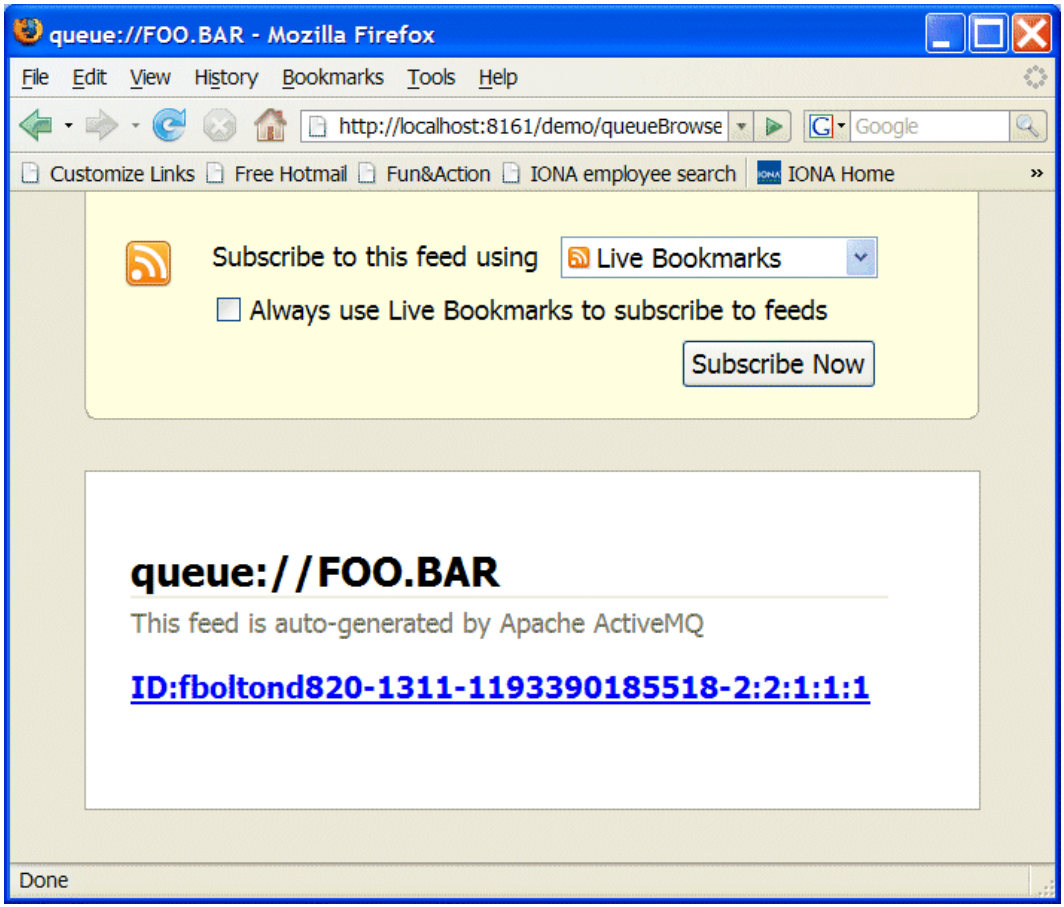
Figure 4.3. Default Option to Browse a Queue

If you click on **Browse a queue as XML**, you should see a page like Figure 4.4 on page 54 .

Figure 4.4. Option to Browse a Queue as XML

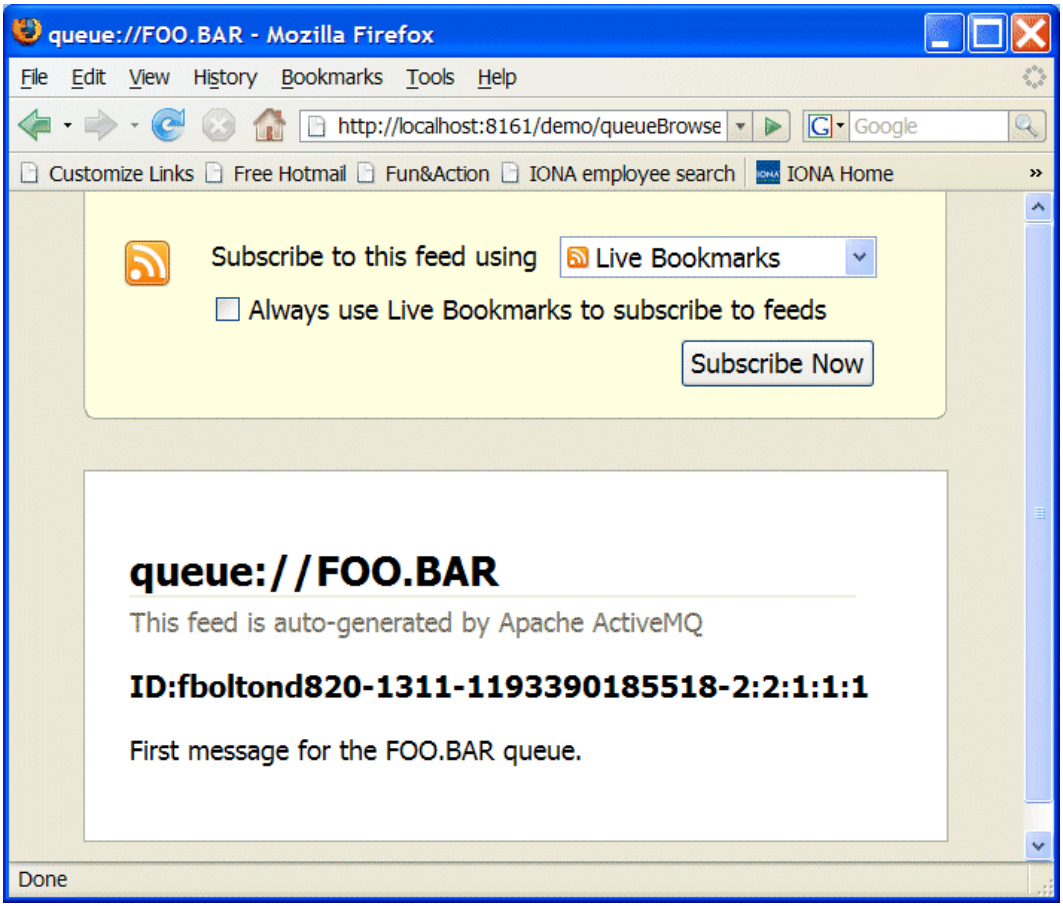
If you click on **Browse a queue as Atom**, you should see a page like Figure 4.5 on page 55 .

Figure 4.5. Option to Browse a Queue as Atom



If you click on **Browse a queue as RSS 1.0** or **Browse a queue as RSS 2.0**, you should see a page like Figure 4.6 on page 56 .

Figure 4.6. Option to Browse a Queue as RSS 1.0



Receive a message from the queue

To receive a message from the `FOO.BAR` queue, open the example welcome page in your browser, http://localhost:8161/demo[7], and click the link, Receive a message[8].

[7] http://localhost:8080/activemq-web-demo
[8] http://localhost:8080/activemq-web-demo/message/FOO/BAR?timeout=10000&type=queue

You should now see the text of the message that you sent earlier. You will probably also receive an error from your browser, if the message is not formatted as HTML or XML (which the browser expects).

Protocol Details

What is REST?

Representational State Transfer (REST) is a software architecture designed for distributed systems, like the World Wide Web. For details of the REST architecture and the philosophy underlying it, see the REST Wikipedia[9] article.

One of the key concepts of a RESTful architecture is that the interaction between different network nodes should take on a very simple form. In particular, the number of operations in a RESTful protocol must be kept small: for example, the REST protocol in FUSE requires just three operations.

Outline of a REST interaction

In general, a REST interaction consists of the following elements:

- *Operation*—belongs to a restricted, well-known set of operations—for example, in the HTTP protocol, the main operations are `GET`, `POST`, `PUT`, and `DELETE`. The advantage of this approach is that, in contrast to RPC architectures, there is no need to define interfaces for a RESTful protocol. The operations are all known in advance.
- *URI*—identifies the resource that the operation acts on. For example, a HTTP `GET` operation acts on the URI by fetching data from the resource identified by the URI.
- *Data (if required)*—needed for operations that send data to the remote resource.

HTTP as a RESTful protocol

HTTP is a good example of a protocol demonstrating RESTful design principles. In fact, proponents of REST argue that it is precisely the RESTful qualities of HTTP that enabled the rapid expansion of the World Wide Web. In keeping with REST principles, HTTP has a restricted operation set, consisting of only eight operations: `GET`, `POST`, `PUT`, `DELETE`, `OPTIONS`, `HEAD`, `TRACE`, and `CONNECT`.

For the purpose of implementing a RESTful protocol, the first four HTTP operations—`GET`, `POST`, `PUT`, and `DELETE`—are the most important. The semantics of these operations are described briefly in Table 4.1 on page 59.

[9] http://en.wikipedia.org/wiki/Representational_State_Transfer#REST.27s_Central_Principle:_Resources

Table 4.1. HTTP RESTful Operations

Operation	Description
`GET`	Fetch the remote resource identified by the URI.
`POST`	Add/append/insert data to the remote resource identified by the URI.
`PUT`	Replace the remote resource identified by the URI with the data from this operation.
`DELETE`	Delete the remote resource identified by the URI.

This simple set of operations—analogous to the classic CRUD (Create, Replace, Update, and Delete) operations for a database—turns out to be remarkably powerful and flexible.

REST protocol servlets

The following servlets—which are automatically deployed in the message broker Web console—implement RESTful access to the FUSE message queues:

- "message servlet" on page 59 .
- "queueBrowse servlet" on page 61 .

message servlet

The RESTful service implemented by the FUSE `message` servlet enables you to enqueue and dequeue messages over HTTP. You can, therefore, use the message servlet to implement message producers and message consumers as Web forms.

To interact with the FUSE `message` servlet, construct a URL of the following form:

```
http://Host:Port/WebContext/message/Destination
Path?Opt1=Val1&Opt2=Val2...
```

Where the URL is constructed from the following parts:

- *`Host`*`:`*`Port`*—the host and port of the servlet engine. For example, in the default message broker configuration, a HTTP port is opened on `localhost:8161`.

- *WebContext*—in a Web application, it is usual to group related components (servlets and so on) under a particular Web context, *WebContext*. For example, for the REST demonstration servlets, the Web context is demo by default.
- message—routes this URL to the message servlet.
- *DestinationPath*—specifies the compound name of a queue or topic in the message broker. For example, the FOO.BAR queue has the destination path, FOO/BAR.
- *?Opt1=Val1&Opt2=Val2*—you can add some options in order to qualify how the URL is processed.

For example, the following URL can be used to fetch a message from the FOO.BAR queue, where the Web console has the default configuration:

```
http://localhost:8161/demo/mes
sage/FOO/BAR?type=queue&timeout=5000
```

Table 4.2 on page 60 shows the URL options recognized by the message servlet:

Table 4.2. URL Options Recognized by the Message Servlet

URL Option	Description
type	Can be either queue or topic.
timeout	When consuming a message from a queue, specifies the length of time (in units of milliseconds) the client is prepared to wait.

Three HTTP operations—GET, POST, and DELETE—are recognized by the message servlet. The semantics of these operations are described briefly in Table 4.3 on page 60 .

Table 4.3. Message Servlet RESTful HTTP Operations

Operation	Description
GET	Consume a single message from the destination (queue or topic) specified by the URL.

Operation	Description
POST	Send a single message to the destination (queue or topic) specified by the URL.
DELETE	Consume a single message from the destination (queue or topic) specified by the URL. This operation has the same effect as GET.

For details of the form properties recognized by the message servlet (for POSTing a message), see "Example of posting a message" on page 62 .

queueBrowse servlet

The RESTful service implemented by the queueBrowse servlet enables you to monitor the contents and status of any queue or topic in the Web console. Effectively, the queueBrowse servlet is a simple management tool.

To interact with the FUSE queueBrowse servlet, construct a URL of the following form:

```
http://Host:Port/WebContext/queueBrowse/Destination
Path?Opt1=Val1&Opt2=Val2...
```

The queueBrowse URL has a similar structure to the message URL (see "message servlet" on page 59), except that the queueBrowse URL is built from *WebContext*/queueBrowse instead of *WebContext*/message.

For example, the following URL can be used to browse the FOO.BAR queue, where the Web console has the default configuration:

```
http://localhost:8161/demo/queueBrowse/FOO/BAR
```

Table 4.4 on page 61 shows the URL options recognized by the queueBrowse servlet:

Table 4.4. URL Options Recognized by the QueueBrowse Servlet

URL Option	Description
view	Specifies the format for viewing the queue/topic. The following views are supported: • simple—(*default*) displays a compact summary of the queue in XML format, where each message is shown as a message element with ID.

URL Option	Description
	• `xml`—displays a detailed summary of the queue in XML format, where each message is shown in full. • `rss`—displays a compact summary of the queue in the form of an RSS 1.0, 2.0 or Atom 0.3 feed. You can configure the type of feed using `feedType`.
`feedType`	In combination with the setting, `view=rss`, you can use this option to specify one of the following feeds: • `rss_1.0` • `rss_2.0` • `atom_0.3`
`contentType`	Override the MIME content type of the view.
`maxMessages`	The maximum number of messages to render.

Example of posting a message

Example 4.2 on page 62 shows an example of the Web form used to send a message to the `FOO.BAR` queue in the Web console, as demonstrated in "Send a message" on page 51 .

Example 4.2. Web Form for Sending a Message to a Queue or Topic

```
<html>
<head>
  <title>Send a JMS Message</title>
  <link rel="stylesheet" href="style.css" type="text/css">
</head>
<body>
<h1>Send a JMS Message</h1>
<form action="message/FOO/BAR" method="post">
  <p>
    <label for="destination">Destination name</label>
    <input type="text" name="destination"/>
  </p>
  <p>
```

```
    <label for="type">Destination Type: </label>
    <select name="type">
      <option selected value="queue">Queue</option>
      <option type" value="topic">Topic</option>
   </select>
  </p>
  <p>
    <textarea name="body" rows="30" cols="80">
Enter some text here for the message body...
    </textarea>
  </p>
  <p>
    <input type="submit" value="Send"/>
    <input type="reset"/>
  </p>
</form>
</body>
</html>
```

Table 4.5 on page 63 describes the form properties that are recognized by the message servlet.

Table 4.5. Form Properties Recognized by Message Servlet

Form Property	Description
Form action	The `action` attribute of the `<form>` tag has the format, `message/`*`DestinationPath`*, where *`DestinationPath`* is the compound name of the queue or topic, using forward slash, `/`, as the delimiter (for example, `FOO/BAR`).
`destination`	The compound name of the destination queue or topic, using a period, `.`, as the delimiter (for example, `FOO.BAR`). If this property is specified in the form, it overrides the value of the *`DestinationPath`* in the form action.
`type`	Destination type, equals `queue` or `topic`.
`body`	Message body.

Example of getting a message

To consume a message from a topic or queue, send a HTTP GET operation (for example, by following a hypertext link) using the URL format described in "message servlet" on page 59 . For example, to consume a message from the `FOO.BAR` queue, navigate to the following URL:

```
http://localhost:8161/demo/mes
sage/FOO/BAR?timeout=10000&type=queue
```

Examples of browsing a queue

To browse a queue using the `queueBrowse` servlet, simply navigate to an URL of the appropriate form, as described in "queueBrowse servlet" on page 61 .

For example, to browse the `FOO.BAR` queue in XML format:

```
http://localhost:8161/demo/queueBrowse/FOO/BAR?view=xml
```

To browse the `FOO.BAR` queue as an Atom 1.0 feed:

```
http://localhost:8161/demo/queueBrowse/FOO/BAR?view=rss&feed
Type=atom_1.0
```

To browse the `FOO.BAR` queue as an RSS 1.0 feed:

```
http://localhost:8161/demo/queueBrowse/FOO/BAR?view=rss&feed
Type=rss_1.0
```

Chapter 5. VM Protocol

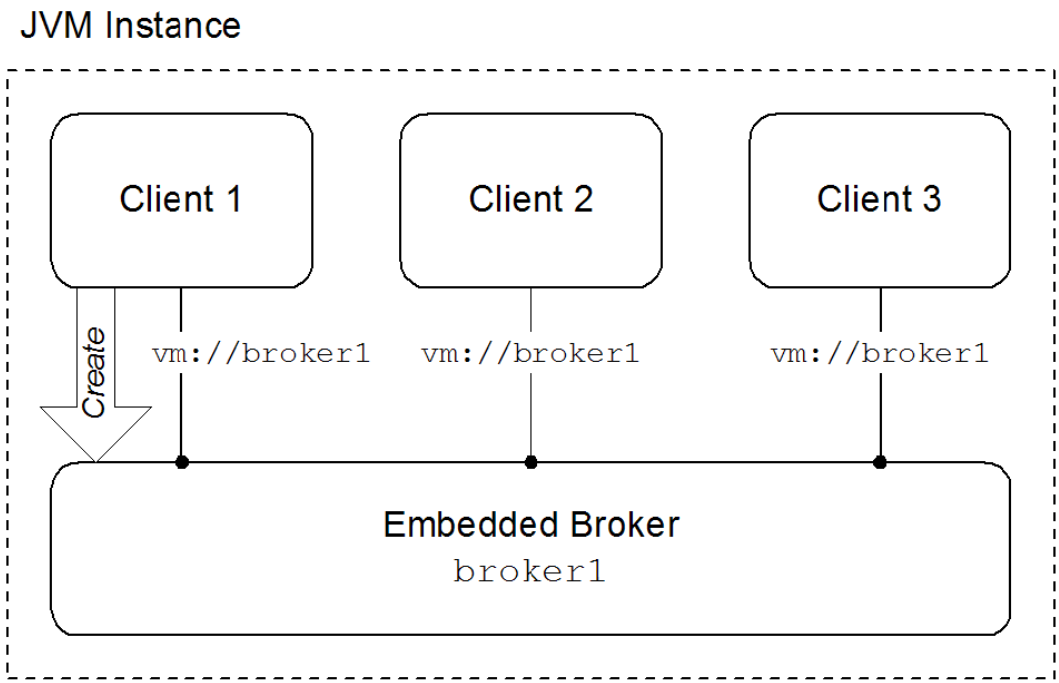
The VM transport allows clients to connect to each other inside the Java Virtual Machine (JVM) without the overhead of network communication.

Introduction to the VM Protocol

Overview

The VM protocol enables Java clients running inside the same JVM to communicate with each other inside the JVM, without having to resort to a using a network protocol. The clients still require a broker to mediate the exchange of messages, however. The VM protocol implicitly creates an embedded broker the first time it is accessed. Figure 5.1 on page 66 shows the basic architecture of the VM protocol.

Figure 5.1. Clients Connected through the VM Protocol



Embedded broker lifecycle

The embedded broker has the following lifecycle:

1. The first client that attempts to access a specific broker (for example, `broker1`) causes the broker to be instantiated. In some cases—for example,

if the VM URI contains special broker configuration details—it might be important to control which client instantiates the broker.

2. Subsequent clients connect to the pre-existing embedded broker.
3. After all of the connections are closed by the clients, the embedded broker is automatically shut down.

Simple URI syntax

A VM URI can be constructed with the following simple URI syntax:

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

Where *BrokerName* specifies the name of the embedded broker to which the client connects. The transport options, *?TransportOptions*, are specified in the form of a query list, where you can use any of the options shown in Table 5.1 on page 67 and Table 5.2 on page 68 .

Advanced URI syntax

Alternatively, you can construct a VM URI using the following advanced URI syntax:

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

Where *BrokerConfigURI* is a broker configuration URI (see "Broker configuration URI" on page 69). With this syntax, you can use only the options shown in Table 5.1 on page 67.

Transport options

Table 5.1 on page 67 shows the transport options you can use with either the simple or advanced VM URI syntax.

Table 5.1. VM Transport Options (for All URI Syntaxes)

Option	Description
`marshal`	If `true`, forces each command sent over the transport to be marshalled and unmarshalled using the specified wire format. Default is `false`.
`wireFormat`	The name of the wire format to use.
`wireFormat.*`	Properties prefixed by `wireFormat.` configure the specified wire format.

Table 5.2 on page 68 shows transport options that are valid *only* for the simple URI syntax.

Table 5.2. VM Transport Options (for Simple URI Syntax Only)

Option	Description
`broker.*`	Properties prefixed by `broker.` configure the embedded broker. You can specify any of the standard broker options (see TABLE) in this way.
`brokerConfig`	Specifies an external broker configuration file. For example, to pick up the broker configuration file, `activemq.xml`, you would set `brokerConfig` as follows: `brokerConfig=xbean:activemq.xml`.

Broker options

Table 5.3 on page 68 shows the broker options.

Table 5.3. Broker Options

Option	Description
`useJmx`	If `true`, enables JMX. Default is `true`.
`persistent`	If `true`, the broker uses persistent storage. Default is `true`.
`populateJMSXUserID`	If `true`, the broker populates the `JMSXUserID` message property with the sender's authenticated username. Default is `false`.
`useShutdownHook`	If `true`, the broker installs a shutdown hook, so that it can shut down properly when it receives a JVM kill. Default is `true`.
`brokerName`	Specifies the broker name. Default is `localhost`.
`deleteAllMessagesOnStartup`	If `true`, deletes all the messages in the persistent store as the broker starts up. Default is `false`.
`enableStatistics`	If `true`, enables statistics gathering in the broker. Default is `true`.

Example URIs

You can construct the following example URIs using the simple syntax:

- "Basic VM URI" on page 69.
- "Simple URI with broker options" on page 69.

- "Simple URI with external configuration file" on page 69 .

Basic VM URI

To connect to the embedded broker with broker name, `broker1`, you can use the following URI:

```
vm://broker1
```

Simple URI with broker options

To create and connect to the embedded broker, `broker1`, where you want the broker to have a non-persistent message store, use the following URI:

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

Evidently, the broker options (such as `broker.persistent`) can only be taken into account, if the VM URI makes the first connection to the embedded broker (thus causing the broker to be instantiated). If the VM URI connects to an existing embedded broker, it is too late to change the broker configuration.

Simple URI with external configuration file

To create and connect to the embedded broker, `broker1`, where the broker is to be configured by the file, `activemq.xml`, use the following URI:

```
vm://broker1?brokerConfig=xbean:activemq.xml
```

Where the `brokerConfig` option enables you to specify the location of the external configuration file, `activemq.xml`.

Broker configuration URI

The broker configuration URI is the very same URI that you use on the command line to configure the standalone broker, `activemq`. There are three different URI schemes supported for broker configuration: `broker:`, `properties:`, and `xbean:`. Of these URI schemes, the `broker:` URI is the most useful one for constructing VM URIs.

The broker configuration URI has the following syntax:

```
broker://(TransportURI, ..., network:NetworkURI,
...)/BrokerName?BrokerOptions
```

Where the `broker://` prefix is immediately followed by a list of URIs in parentheses. Inside the URI list you can put URIs, *`TransportURI`*, for the broker's transport endpoints and URIs, `network:`*`NetworkURI`*, for the broker's network connectors. This is followed by the broker's name, *`BrokerName`*, and any broker options from Table 5.3 on page 68 .

Advanced example URI

The following VM URI uses the advanced URI syntax to create and connect to an embedded broker, where the broker is configured using a broker configuration URI.

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

Chapter 6. Discovery Protocols

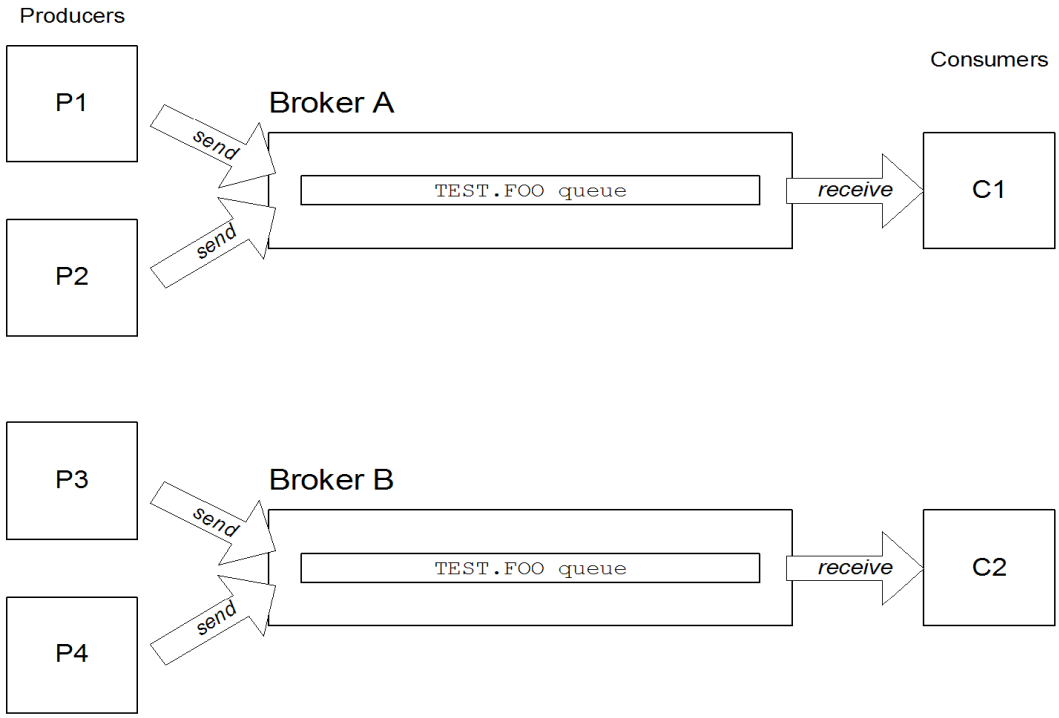
This chapter introduces the simplest kind of broker cluster: a collection of isolated broker instances (no network connectors). When used in combination with the discovery protocols, such a cluster can be used as a simple load balancing system.

Configuring a Simple Broker Cluster

Simple cluster architecture

Figure 6.1 on page 72 shows an example of the kind of simple broker cluster that is the subject of discussion in this chapter, where the cluster shown in Figure 6.1 on page 72 consists of just two brokers: broker A and broker B.

Figure 6.1. Simple Cluster Architecture



The preceding figure illustrates the simplest type of cluster, where no network connectors are enabled on the brokers and the brokers remain unaware of each other (isolated brokers). Assuming that the producers do not care which consumer processes the messages, this cluster architecture can be useful for

balancing load across multiple hosts. There are, however, some serious limitations with this type of cluster.

Failover protocols

The cluster architecture shown in Figure 6.1 on page 72 can support some simple modes of failover protection. In particular, if a producer or consumer loses its connection to a broker, it can use one of the failover URIs to manage reconnection to an alternative broker in the cluster. Currently, the following failover protocols support reconnection logic:

- "Failover Protocol" on page 78 .
- "Dynamic Discovery Protocol" on page 82 .
- "Discovery Agents" on page 87 .

Limitations of simple broker clusters

The cluster architecture shown in Figure 6.1 on page 72 suffers from the following limitations:

- Each broker must have a consumer for each type of queue.
- If a consumer for a particular queue on a particular broker becomes unavailable, messages on that queue will accumulate in the broker without being processed.
- Producers have no way of finding out whether a particular queue instance has an associated consumer.

In principle, these limitations can be overcome by linking brokers together using *network connectors*.

Steps to create multiple broker instance

Perform the following steps to create multiple message broker instances:

1. "Create a directory for the new broker configuration" on page 74 .
2. "Copy configuration files" on page 74 .
3. "Customize port numbers" on page 74 .
4. "Customize the broker name" on page 75 .
5. "(Optionally) Disable network connectors" on page 75 .
6. "Create a script to run the broker" on page 76

7. "Repeat as necessary" on page 77 .

Create a directory for the new broker configuration

Create a new directory to hold the configuration files for a new broker instance. For example, in your working directory, *FuseWorking*, create a directory, `broker_a`, for broker A by entering the following command:

```
mkdir FuseWorking/broker_a
```

Copy configuration files

From the FUSE Message Broker install directory, *InstallDir*, copy the `conf` and `webapps` directories into the `broker_a` directory. For example:

Windows

```
mkdir FuseWorking\broker_a\conf
copy InstallDir\conf FuseWorking\broker_a\conf
mkdir FuseWorking\broker_a\webapps
copy InstallDir\webapps FuseWorking\broker_a\webapps
```

UNIX

```
cp InstallDir/conf FuseWorking/broker_a/conf
cp InstallDir/webapps FuseWorking/broker_a/webapps
```

Where the `conf` directory contains the basic configuration files for the broker and the `webapps` directory contains the files required to run the Web console management tool.

Customize port numbers

You must customize the port numbers used by broker A, in order to avoid clashes with other brokers. To customize the port numbers, edit the `broker_a/conf/activemq.xml` configuration file. For example, you might customize the transport connector ports as follows:

```
<transportConnectors>
        <transportConnector name="openwire"
                            uri="tcp://localhost:61716"
                       discoveryUri="multicast://default"/>

        ...
</transportConnectors>
```

Where the TCP connector port has been changed from `61616` to `61716`. You also need to customize the port for the servlet engine (which hosts the Web console application), as follows:

```
<jetty xmlns="http://mortbay.com/schemas/jetty/1.0">
    <connectors>
        <nioConnector port="8171" />
    </connectors>
    ...
</jetty>
```

Where the servlet engine port has been changed from `8161` to `8171`.

Customize the broker name

You must also customize the broker name, by setting the `brokerName` attribute of the `broker` element. For example, to set the broker name for broker A, edit the `broker_a/conf/activemq.xml` configuration file as follows:

```
<broker xmlns="http://activemq.org/config/1.0"
    brokerName="brokera"
    dataDirectory="${activemq.base}/data">
    ...
</broker>
```

Where the broker name is set to `broker_a`.

Note

It is important for each broker in a cluster to have a distinct name. For example, discovery agents and network connectors require that each broker in the cluster has a distinct name.

(Optionally) Disable network connectors

By default, the standard broker configuration comes with a network connector enabled. For the basic cluster discussed in this chapter, however, each broker is isolated and unaware of its peers. To configure this type of cluster, you must edit the `broker_a/conf/activemq.xml` configuration file, commenting out any network connector elements, `networkConnector`, as shown in the following example:

```
<networkConnectors>
      <!-- Comment out the network connectors ... -->
      <!--
      <networkConnector name="default-nc"
                       uri="multicast://default"/>
```

```
      -->
    </networkConnectors>
```

Note

Of course, network connectors are useful and a network of brokers is much more responsive and flexible than a simple collection of isolated broker instances. If you prefer, you can leave the network connectors enabled, but you should be aware that the examples will behave somewhat differently from the descriptions given in the text.

Create a script to run the broker

The standalone message broker, `activemq`, reads the environment variable, `ACTIVEMQ_BASE`, to determine the location of its configuration directory. Hence, in order to run a standalone broker using the broker A configuration, you must set `ACTIVEMQ_BASE` to the directory, *`FuseWorking`*`/broker_a` before running `activemq`.

The easiest approach to take is to create a dedicated script to run broker A. For example, you might create a script similar to the following:

Windows

Create a Windows `.bat` file, `broker_a.bat`, according to the following outline:

```
@echo off
REM Set generic FUSE Message Broker environment
REM ... (Not shown - for details, see install guide)
set ACTIVEMQ_BASE=FuseWorking\broker_a
echo Running Broker A...
activemq
```

UNIX

Assuming you are using a Bourne shell, create a shell script, `broker_a`, according to the following outline:

```
#! /bin/sh
# Set generic FUSE Message Broker environment
# ... (Not shown - for details, see install guide)
ACTIVEMQ_BASE=FuseWorking/broker_a; export ACTIVEMQ_BASE
```

```
echo Running Broker A...
activemq
```

Repeat as necessary

Repeat the preceding steps to create as many broker instances as required—for example, broker B, C, D, and so on.

Example broker cluster

To run the examples described in this chapter, two brokers are required: broker A and broker B. Broker A is configured with the following transport connectors:

```
<transportConnectors>
        <transportConnector
            name="openwire" uri="tcp://localhost:61716" dis
coveryUri="multicast://default"/>
</transportConnectors>
```

And broker B is configured with the following transport connectors:

```
    <transportConnectors>
        <transportConnector
            name="openwire" uri="tcp://localhost:61816" dis
coveryUri="multicast://default"/>
    </transportConnectors>
```

Where the TCP ports are configured to have distinct values (`61716` and `61816`, respectively). You must also ensure that any other port numbers are configured to be distinct (for example, the servlet engine port inside the `<jetty>` tag).

Failover Protocol

Overview

The *failover protocol* overlays reconnect logic on top of any of the other transports. The failover URI is composed of multiple URIs that represent different broker endpoints. By default, the protocol randomly chooses a URI from the list and attempts to establish a connection to it. If it does not succeed, or if it subsequently fails, a new connection is established to one of the other URIs in the list.

Configuration syntax

A `failover` URI must conform to the following syntax:

```
failover://(uri1,...,uriN)?TransportOptions
```

Where the URI list, *uri1,...,uriN*, is a comma-separated list containing one or more broker endpoint URIs. The transport options, *?TransportOptions*, are specified in the form of a query list (where the supported options are described in Table 6.1 on page 78). If no transport options are required, you can use the following alternative syntax:

```
failover://uri1,...,uriN
```

Transport options

The failover protocol supports the transport options described in Table 6.1 on page 78.

Table 6.1. Failover Transport Options

Option Name	Default	Description
`initialReconnectDelay`	`10`	How long to wait before the first reconnect attempt (in ms).
`maxReconnectDelay`	`30000`	The maximum amount of time to wait between reconnect attempts (in ms).
`useExponentialBackOff`	`true`	If `true`, use an exponential back-off between reconnect attempts.
`backOffMultiplier`	`2`	The exponent used in the exponential back-off algorithm.
`maxReconnectAttempts`	`0`	If not `0`, this is the maximum number of reconnect attempts before an error is sent back to the client.

Option Name	Default	Description
randomize	true	If true, choose a URI at random from the list provided.

Sample URI

The following is an example of a failover URI that randomly connects to one of two message brokers:

```
failover://(tcp://localhost:61616,tcp://remotehost:61616)?initialReconnectDelay=100
```

Example of using the failover protocol

To try out the failover protocol, perform the following steps:

1. "Start up broker A" on page 79 .
2. "Start up consumer A" on page 79 .
3. "Start up broker B" on page 80 .
4. "Start up consumer B" on page 80 .
5. "Start up a producer with failover URL" on page 80 .
6. "Kill the active broker" on page 81 .

Start up broker A

Assuming that you have already configured a simple broker cluster as described in "Configuring a Simple Broker Cluster" on page 72 , start broker A by running the relevant script. For example:

```
broker_a
```

Start up consumer A

Start a consumer that consumes messages from the TEST.FOO queue on broker A. The consumer tool is located in the example directory of the FUSE Message Broker install directory, *InstallDir*. Assuming that broker A's TCP connector is listening on port 61716 on the local host, open a new command window and start consumer A as follows:

```
cd InstallDir/example
ant consumer -Durl=tcp://localhost:61716 -Dmax=100
```

The consumer tool should log output similar to the following as it starts up:

```
Buildfile: build.xmlinit:compile:consumer:     [echo] Running
 consumer
against server at $url = tcp://localhost:61716 for subject
$subject = TEST.FOO
   [java] Connecting to URL: tcp://localhost:61716     [java]
 Consuming
queue: TEST.FOO     [java] Using a non-durable subscription
    [java] We
are about to wait until we consume: 100 message(s) then we
will shutdown
```

Start up broker B

Start broker B by running the relevant script. For example, open a new command window and enter:

```
broker_b
```

Start up consumer B

Start a consumer that consumes messages from the `TEST.FOO` queue on broker B. Assuming that broker B's TCP connector is listening on port `61816` on the local host, open a new command window and start consumer B as follows:

```
cd InstallDir/example
ant consumer -Durl=tcp://localhost:61816 -Dmax=100
```

Start up a producer with failover URL

Start a producer with a failover URL such that it can connect *either* to broker A or to broker B. Open a new command window and start the producer as follows:

```
cd InstallDir/example
ant producer -Durl="failover://(tcp://localhost:61716,tcp://loc
alhost:61816)?initialReconnectDelay=100"
-DsleepTime=5000
```

It is important to include the double quotes around the failover URL, otherwise the comma-separated list would be parsed as two arguments. The sleep time

is set to 5 seconds between messages, in order to give you enough time to perform the next step.

Kill the active broker

After starting the producer, observe which of the consumers is receiving messages. If consumer A is receiving messages, the producer must be connected to broker A, in which case broker A is the active broker. If consumer B is receiving messages, broker B is the active broker.

To test the failover functionality on the producer side, kill the *active broker* (not consumer!) by switching focus to the relevant broker window and typing `Ctrl-C`. If the producer is still producing messages (it produces ten in total), it will attempt to reconnect to the other broker. You should see output similar to the following in the producer log:

```
     [java] 13:50:31 WARN  Transport failed, attempting to
automatically reconnect due to: java.io.EOFException
     [java] java.io.EOFException
     [java]     at java.io.DataInputStream.readInt(DataInput
Stream.java:358) ...
     [java] Sending message: Message: 7 sent at: Wed Oct 10
13:50:31 BST 2007  ...
     [java] 13:50:32 INFO  Successfully reconnected to
tcp://localhost:61716
```

Dynamic Discovery Protocol

Overview

The *dynamic discovery protocol* combines reconnect logic with the capability to auto-discover broker endpoints in the local network. The discovery protocol invokes a discovery agent in order to build up a list of broker URIs. The protocol then randomly chooses a URI from the list and attempts to establish a connection to it. If it does not succeed, or if it subsequently fails, a new connection is established to one of the other URIs in the list.

Discovery agents

A *discovery agent* is a bootstrap mechanism that enables a message broker, consumer, or producer to obtain a list of broker URIs, where the URIs represent connector endpoints. The broker, consumer, or producer can subsequently connect to one of the URIs in the list.

The following kinds of discovery agent are currently supported in FUSE Message Broker:

- Simple (static) discovery agent.
- Multicast discovery agent.
- Rendezvous discovery agent.

For more details, see "Discovery Agents" on page 87.

Configuring a transport connector with a discovery agent

Before you can use the discovery protocol, you must make your broker's endpoints discoverable by adding a discovery agent to each transport connector. For example, to make a TCP transport connector discoverable, set the `discoveryUri` attribute on the `transportConnector` element as follows:

```
    <transportConnectors>
        <transportConnector
            name="openwire" uri="tcp://localhost:61716" dis
coveryUri="multicast://default"/>
    </transportConnectors>
```

Where the TCP transport connector is configured to use the multicast discovery agent, `multicast://default`.

Configuration syntax

A `discovery` URI must conform to the following syntax:

```
discovery://(DiscoveryAgentUri)?TransportOptions
```

Where the discovery agent URI, *DiscoveryAgentUri*, identifies a discovery agent, as described in "Discovery agents" on page 82 above. The transport options, *?TransportOptions*, are specified in the form of a query list (where the supported options are described in Table 6.2 on page 83). If no transport options are required, you can use the following alternative syntax:

```
discovery://DiscoveryAgentUri
```

Transport options

The discovery protocol supports the transport options described in Table 6.2 on page 83

Table 6.2. Discovery Transport Options

Option Name	Default	Description
`initialReconnectDelay`	`10`	How long to wait before the first reconnect attempt (in ms).
`maxReconnectDelay`	`30000`	The maximum amount of time to wait between reconnect attempts (in ms).
`useExponentialBackOff`	`true`	If `true`, use an exponential back-off between reconnect attempts.
`backOffMultiplier`	`2`	The exponent used in the exponential back-off algorithm.
`maxReconnectAttempts`	`0`	If not `0`, this is the maximum number of reconnect attempts before an error is sent back to the client.

Sample URI

The following is an example of a discovery URI that uses a multicast discovery agent:

```
discovery://(multicast://default)?initialReconnectDelay=100
```

Example of using the dynamic discovery protocol

To try out the dynamic discovery protocol, perform the following steps:

1. "Configure the brokers' transport connectors" on page 84 .
2. "Start up broker A" on page 84 .

3. "Start up consumer A" on page 84 .

4. "Start up broker B" on page 85 .

5. "Start up consumer B" on page 85 .

6. "Start up a producer with discovery URL" on page 85 .

7. "Kill the active broker" on page 86

Configure the brokers' transport connectors

To run the current example you need a cluster of two brokers: broker A and broker B. For details of how to set up the broker cluster, see "Configuring a Simple Broker Cluster" on page 72 . In particular, you must ensure that each broker configures its TCP transport connector with a discovery URI, as follows:

```
    <transportConnectors>
        <transportConnector
            name="openwire" uri="tcp://localhost:61716" dis
coveryUri="multicast://default"/>
    </transportConnectors>
```

Where the `discoveryUri` attribute on the `transportConnector` element is initialized to `multicast://default`.

Start up broker A

Assuming that you have already configured a simple broker cluster as described in "Configuring a Simple Broker Cluster" on page 72 , start broker A by running the relevant script. For example:

```
broker_a
```

Start up consumer A

Start a consumer that consumes messages from the `TEST.FOO` queue on broker A. Assuming that broker A's TCP connector is listening on port `61716` on the local host, open a new command window and start consumer A as follows:

```
cd InstallDir/example
ant consumer -Durl=tcp://localhost:61716 -Dmax=100
```

Start up broker B

Start broker B by running the relevant script. For example, open a new command window and enter:

```
broker_b
```

Start up consumer B

Start a consumer that consumes messages from the `TEST.FOO` queue on broker B. Assuming that broker B's TCP connector is listening on port `61816` on the local host, open a new command window and start consumer B as follows:

```
cd InstallDir/example
ant consumer -Durl=tcp://localhost:61816 -Dmax=100
```

Start up a producer with discovery URL

Start a producer with a discovery URL such that it can connect *either* to broker A or to broker B. Open a new command window and start the producer as follows:

```
cd InstallDir/example
ant producer -Durl=discovery://(multicast://default) -DsleepTime=2000
```

The sleep time is set to 2 seconds between messages, in order to give you enough time to perform the next step.

As the producer starts up, it should log the following lines to the screen:

```
Buildfile: build.xml
init:
compile:
producer:
     [echo] Running producer against server at $url = discovery://(multicast://default)
for subject $subject = TEST.FOO
     [java] Connecting to URL: discovery://(multicast://default)
```

```
     [java] Publishing a Message with size 1000 to queue:
TEST.FOO
     [java] Using non-persistent messages
     [java] Sleeping between publish 2000 ms
     [java] 16:27:55 WARN  brokerName not set
     [java] 16:27:56 INFO  Adding new broker connection URL:
 tcp://fboltond820:61816
     [java] 16:27:56 INFO  Adding new broker connection URL:
 tcp://fboltond820:61716
     [java] 16:27:56 INFO  Successfully reconnected to
tcp://fboltond820:61816
```

You can ignore the warning message, `WARN brokerName not set`. The next two `INFO` messages show that the discovery mechanism is working: each of the discovered URLs is logged here. Finally, the producer connects to one of the discovered URLs and starts sending message to that broker.

Kill the active broker

After starting the producer, observe which of the consumers is receiving messages. If consumer A is receiving messages, the producer must be connected to broker A, in which case broker A is the active broker. If consumer B is receiving messages, broker B is the active broker.

To test the failover functionality on the producer side, kill the *active broker* (not consumer!) by switching focus to the relevant broker window and typing `Ctrl-C`. If the producer is still producing messages, it will attempt to reconnect to the other broker. You should see output similar to the following in the producer log:

```
     [java] 16:28:10 WARN  Transport failed, attempting to
automaticallyreconnect due to: java.io.EOFException
     [java] java.io.EOFException
     [java]     at java.io.DataInputStream.readInt(DataInput
Stream.java:358) ...
     [java] Sending message: Message: 7 sent at: Wed Oct 10
13:50:31 BST 2007  ...
     [java] 16:28:10 INFO  Successfully reconnected to
tcp://fboltond820:61716
```

Discovery Agents

Overview

A discovery agent is a bootstrap mechanism that enables a client or message broker to discover other broker instances on a network. On the client side, the purpose of the discovery agent is simply to obtain a list of broker URIs. The list of URIs is then processed by the dynamic discovery protocol, `discovery://(...)`, which opens a connection to one of the URIs in the list.

Discovery agents typically use some form of ping mechanism to discover the broker URIs. Hence, it is usually necessary to enable the discovery mechanism on the server side as well (an exception to this requirement is the `simple` discovery agent).

Configuring discovery agents on the message broker

For certain kinds of discovery agent (for example, multicast or rendezvous), it is necessary to enable the discovery agent in the message broker configuration. For example, to enable the multicast discovery agent on an Openwire endpoint, you should edit the relevant `transportConnector` element as follows:

```
    <transportConnectors>
        <transportConnector
            name="openwire" uri="tcp://localhost:61716" dis
coveryUri="multicast://default"/>
    </transportConnectors>
```

Where the `discoveryUri` attribute on the `transportConnector` element is initialized to `multicast://default`. You can associate multiple endpoints with the same discovery agent. For example, to configure both an Openwire endpoint and a Stomp endpoint to use the `multicast://default` discovery agent:

```
    <transportConnectors>
        <transportConnector
            name="openwire" uri="tcp://localhost:61716" dis
coveryUri="multicast://default"/>
        <transportConnector
            name="stomp" uri="stomp://localhost:61613"  dis
```

```
coveryUri="multicast://default"/>
    </transportConnectors>
```

Using a discovery agent on the client side

You *cannot* use a discovery agent URI directly, on the client side. A discovery agent is not a transport protocol and it is not recognized as such by messaging clients. To use a discovery agent on the client side, the agent URI,*DiscoveryAgentUri*, is embedded inside a discovery URL, as follows:

```
discovery://(DiscoveryAgentUri)?TransportOptions
```

The client recognizes the discovery URL as a transport. It first obtains a list of available endpoint URLs using the specified discovery agent and then connects to one of the discovered URLs. For more details about the discovery protocol, see "Dynamic Discovery Protocol" on page 82 .

Configuring broker networks

Discovery agents

FUSE Message Broker currently supports the following discovery agents:

- "Simple (static) discovery agent" on page 88 .
- "Multicast discovery agent" on page 89 .
- "Rendezvous discovery agent" on page 89 .

Simple (static) discovery agent

The simple (static) discovery agent provides an explicit list of broker URLs for a client to connect to. For example:

```
simple://(tcp://localhost:61716,tcp://localhost:61816)
```

In general, the URI for a simple discovery agent must conform to the following syntax:

```
simple://(URI1,URI2,URI3,...)
```

Or equivalently:

```
static://(URI1,URI2,URI3,...)
```

The two prefixes, `simple:` and `static:`, are exactly equivalent. In order to use the agent URI, it *must* be embedded inside a discovery URL—for example:

```
discovery://(static://(tcp://localhost:61716,tcp://local
host:61816))
```

This discovery agent is only used on the client side. No extra configuration is required in the broker.

Multicast discovery agent

The *multicast discovery agent* uses the IP multicast protocol to find any message brokers currently active on the local network. In order for the protocol to work, a multicast discovery agent must be enabled on *each* broker you want to advertise and messaging clients must be configured to use a `discovery` URI.

The URI for a multicast discovery agent must conform to the following syntax:

```
multicast://GroupID
```

Where the *`GroupID`* is an alphanumeric identifier. All participants in the same discovery network must use the same *`GroupID`*. For example, the FUSE Message Broker is usually configured to use the URI, `multicast://default`.

Rendezvous discovery agent

The *rendezvous discovery agent* is derived from Apple's Bonjour Networking[1] technology, which defines the rendezvous protocol as a mechanism for discovering services on a network. To enable the protocol, a multicast discovery agent must be configured on *each* broker you want to advertise and messaging clients must be configured to use a `discovery` URI.

The URI for a rendezvous discovery agent must conform to the following syntax:

```
rendezvous://GroupID
```

Where the *`GroupID`* is an alphanumeric identifier. All participants in the same discovery network must use the same *`GroupID`*.

[1] http://developer.apple.com/networking/bonjour/

For example, to use a rendezvous discovery agent on the client side, where the client needs to connect to the `groupA` group, you would construct a discovery URL like the following:

```
discovery://(rendezvous://groupA)
```

Chapter 7. Peer-to-Peer Protocols

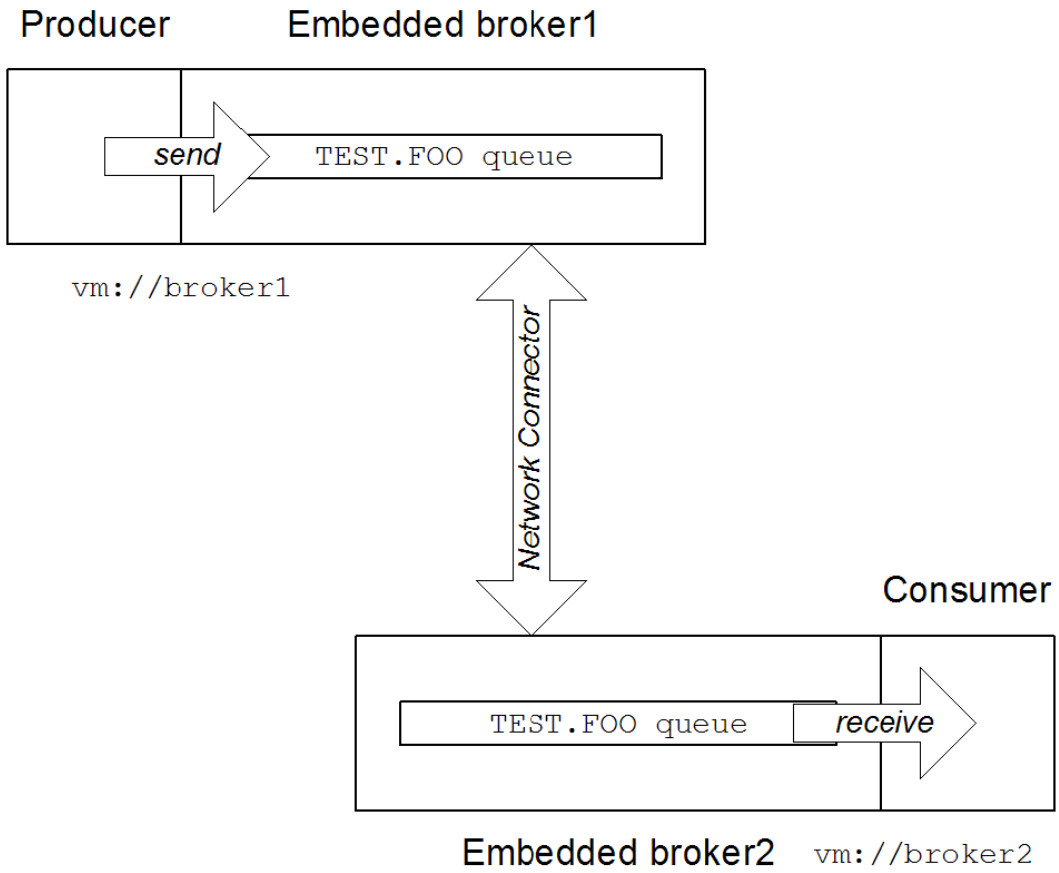
Peer-to-peer protocols enable messaging clients to communicate with each other directly, eliminating the requirement to route messages through an external message broker.

Peer Protocol

Overview

The peer protocol enables you to set up a peer-to-peer network by creating an embedded broker inside each peer endpoint. Figure 7.1 on page 92 illustrates the peer-to-peer network topology for a simple two-peer network.

Figure 7.1. Peer Protocol Endpoints with Embedded Brokers



In this topology, a standalone broker is *not* required, because each peer instantiates its own embedded broker. As shown in Figure 7.1 on page 92 , the producer sends messages to its embedded broker, `broker1`, by

connecting to the local VM endpoint, `vm://broker1`—see "VM Protocol" on page 65 . 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, `TEST.FOO`, the first embedded broker, `broker1`, automatically pushes the message across the network connector and on to the remote embedded broker, `broker2`. The consumer can then receive the message from its embedded broker, `broker2`.

Discovering peer endpoints

Implicitly, the peer protocol uses multicast discovery to locate active peers on the network. In order for this to work, you must ensure that the IP multicast protocol is enabled on your operating system. See "Dynamic Discovery Protocol" on page 82 for details.

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. That is, 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 (for example, `?persistent=true`).

Broker options

The peer URL supports the broker options described in Table 7.1 on page 93 .

Table 7.1. Broker Options

Option	Description
`useJmx`	If `true`, enables JMX. Default is `true`.
`persistent`	If `true`, the broker uses persistent storage. Default is `true`.
`populateJMSXUserID`	If `true`, the broker populates the `JMSXUserID` message property with the sender's authenticated username. Default is `false`.
`useShutdownHook`	If `true`, the broker installs a shutdown hook, so that it can shut down properly when it receives a JVM kill. Default is `true`.

Option	Description
brokerName	Specifies the broker name. Default is localhost.
deleteAllMessagesOnStartup	If true, deletes all the messages in the persistent store as the broker starts up. Default is false.
enableStatistics	If true, enables statistics gathering in the broker. Default is true.

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
```

Example of using the peer protocol

To try out the peer protocol, perform the following steps:

1. "Start up consumer with embedded broker" on page 94 .

2. "Start up producer with embedded broker" on page 95 .

Start up consumer with embedded broker

Start a consumer that consumes messages from the TEST.FOO queue belonging to the group peer group. To start the consumer, run the consumer tool with a peer group URL as follows:

```
cd InstallDir/example
ant consumer -Durl="peer://group/broker1?persistent=false" -
Dmax=100
```

Where the first component of the URL path, group, specifies that this peer belongs to the group peer group. The second component, broker1, specifies the name of the embedded broker and the setting, persistent=false, sets a broker option. When the consumer starts up, you should see output like the following in the command window:

```
consumer:
     [echo] Running consumer against server at $url =
peer://group/broker1?persistent=false for subject $subject =
 TEST.FOO
     [java] Connecting to URL: peer://group/broker1?persist
```

```
ent=false
     [java] Consuming queue: TEST.FOO
     [java] Using a non-durable subscription
     [java] 15:43:10 INFO  ActiveMQ null JMS Message Broker
(broker1) is starting
    [java] 15:43:10 INFO  For help or more information please
 see:
http://activemq.apache.org/
     [java] 15:43:10 INFO  Using Persistence Adapter: Memory
PersistenceAdapter
     [java] 15:43:10 INFO  Listening for connections at:
tcp://fboltond820:2399
     [java] 15:43:10 INFO  Connector tcp://fboltond820:2399
Started
     [java] 15:43:10 INFO  Network Connector org.apache.act
ivemq.transport.discovery.multicast.MulticastDiscoveryA
gent@da4b71 Started
     [java] 15:43:10 INFO  ActiveMQ JMS Message Broker
(broker1, ID:fboltond820-2398-1192200190327-2:0) started
     [java] 15:43:10 INFO  Connector vm://broker1 Started
     [java] 15:43:10 INFO  JMX consoles can connect to ser
vice:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi
     [java] We are about to wait until we consume: 100 mes
sage(s) then we will shutdown
```

While the consumer is starting up, it activates an embedded broker with broker name, `broker1`, and attempts to connect to its peers using a multicast discovery agent.

Start up producer with embedded broker

Start a producer that sends messages to the `TEST.FOO` queue on the `group` peer group. To start the producer, run the producer tool with a peer group URL as follows:

```
cd InstallDir/example
ant producer -Durl="peer://group/broker2?persistent=false" -
DsleepTime=1000
```

Where the name of the embedded broker is set to `broker2` and the sleep time (time between successive messages) is set to 1000 ms. When the producer starts up, the output log should include some lines like the following:

```
     [java] 15:43:27 INFO  Establishing network connection
between from vm://broker2 to tcp://fboltond820:2399
     [java] 15:43:28 INFO  Network connection between
```

```
vm://broker2#2 and tcp://localhost/127.0.0.1:2399(broker1)
has been established.
```

These lines indicate that a peer-to-peer connection was successfully established between the embedded brokers, `broker1` and `broker2`. The consumer should now be able to receive the messages sent by the producer.