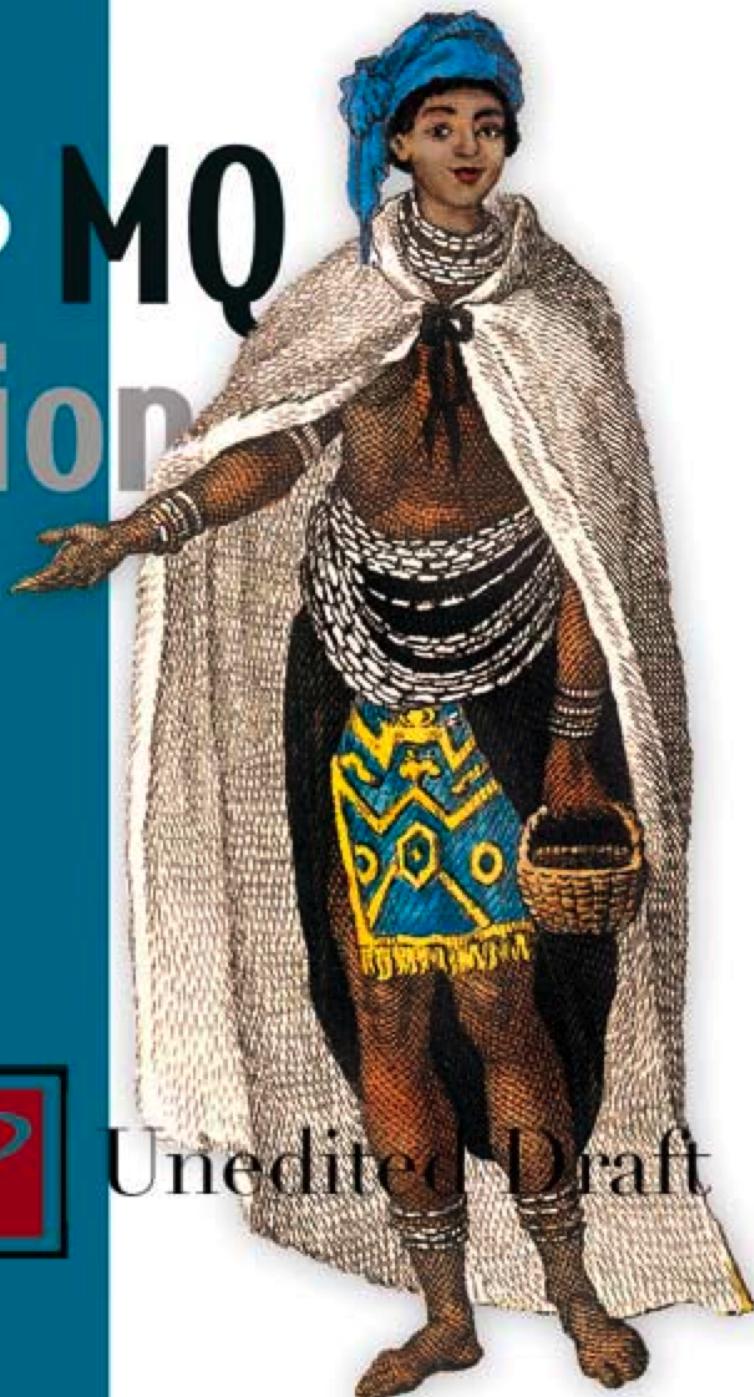


Active MQ in Action

Bruce Snyder
Rob Davies
Dejan Bosanac



 MANNING



Unedited Draft



**MEAP Edition
Manning Early Access Program**

Copyright 2009 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

ActiveMQ In Action

1.0-Alpha

Tue Oct 13 11:35:10 MDT 2009

Bruce Snyder, Rob Davies, Dejan Bosanac

Please post comments or corrections to the [Author Online Forum](#)

I. An Introduction to Messaging and ActiveMQ	1
1. Introduction to Apache ActiveMQ	2
1.1. What is ActiveMQ?	2
1.1.1. ActiveMQ Features	3
1.1.2. Why Use ActiveMQ?	5
1.1.3. When and Where to Use ActiveMQ	8
1.2. Getting Started With ActiveMQ	10
1.2.1. Download and Install the Java SE	10
1.2.2. Download ActiveMQ	11
1.2.3. Examining the ActiveMQ Directory	11
1.2.4. Starting Up ActiveMQ	13
1.2.5. Verifying Your ActiveMQ Setup With the Examples	15
1.3. Summary	20
2. Understanding Message-Oriented Middleware and JMS	21
2.1. Introduction to Enterprise Messaging	22
2.2. What is Message Oriented Middleware?	25
2.3. What is the Java Message Service?	27
2.3.1. Messaging Clients	30
2.3.2. The JMS Provider	32
2.3.3. Anatomy of a JMS Message	33
2.3.4. Message Selectors	40
2.3.5. JMS Domains	44
2.3.6. Administered Objects	49
2.3.7. Using the JMS APIs to Create JMS Applications	50
2.3.8. Message-Driven Beans	55
2.4. Summary	57
3. The ActiveMQ In Action Examples	59
3.1. Understanding the Example Use Cases	59
3.1.1. Prerequisites	60
3.1.2. ActiveMQ In Action Examples	63
3.2. Summary	77
II. How to Configure ActiveMQ	78
4. Connecting to ActiveMQ	79

Please post comments or corrections to the [Author Online Forum](#)

4.1. Understanding Connector URIs	79
4.2. Configuring Transport Connectors	82
4.2.1. Using Transport Connectors	84
4.2.2. Using Network Protocols	87
4.2.3. Using the Virtual Machine Protocol	105
4.3. Configuring Network Connectors	108
4.3.1. Defining Static Networks	111
4.3.2. Defining Dynamic networks	119
4.4. Summary	128
5. Message Persistence	129
5.1. How Are Messages Stored by ActiveMQ?	129
5.2. Available Message Stores in ActiveMQ	131
5.2.1. The AMQ Message Store	132
5.2.2. The KahaDB Message Store	140
5.2.3. The JDBC Message Store	144
5.2.4. The Memory Message Store	150
5.3. Caching Messages in the Broker for Consumers	152
5.3.1. How Message Caching for Consumers Works	152
5.3.2. The ActiveMQ Subscription Recovery Policies	153
5.3.3. Configuring The Subscription Recovery Policy	156
5.4. Summary	157
6. Securing ActiveMQ	158
6.1. Introducing Basic Security Concepts	158
6.2. Authentication	159
6.2.1. Configuring the Simple Authentication Plugin	159
6.2.2. Configuring the JAAS Plugin	161
6.3. Authorization	164
6.3.1. Operation Level Authorization	164
6.3.2. Message Level Authorization	167
6.4. Broker Level Operations	169
6.4.1. Building A Custom Security Plugin	170
6.5. Summary	174
III. Using ActiveMQ to Build Messaging Applications	175
7. Creating Java Applications With ActiveMQ	176

7.1. Integrating Broker	176
7.1.1. Embedding The Broker	176
7.1.2. Integrating With Spring Framework	180
7.2. Summary	193
8. Embedding ActiveMQ In Other Java Containers	195
8.1. Introduction	195
8.2.	195
8.3. Summary	196
9. Connecting to ActiveMQ With Other Languages	197
9.1. Preparing Examples	197
9.2. Communicating with the STOMP protocol	200
9.2.1. Writing Ruby client	204
9.2.2. Creating Python client	207
9.2.3. Building PHP client	212
9.2.4. Implementing Perl client	214
9.2.5. Understanding Stomp transactions	216
9.2.6. Working with Durable Topic Subscribers	220
9.3. Learning NMS (.Net Message Service) API	223
9.4. Introducing CMS (C++ Messaging Service) API	225
9.5. Messaging on the Web	230
9.5.1. Using REST API	230
9.5.2. Understanding Ajax API	234
9.6. Summary	239
IV. Advanced Features in ActiveMQ	241
10. Broker Topologies	242
10.1.	242
10.2. Broker High Availability	242
10.2.1. Shared Nothing Master/Slave	242
10.2.2. Shared Database Master/Slave	246
10.2.3. Shared File system Master/Slave	248
10.3. Networks of Brokers	250
10.3.1. Store and Forward	250
10.3.2. Network Discovery	255
10.3.3. Network Configuration	258

Please post comments or corrections to the [Author Online Forum](#)

10.4. Scaling Applications	263
10.4.1. Vertical Scaling	263
10.4.2. Horizontal Scaling	267
10.4.3. Traffic Partitioning	268
10.5. Summary	269
11. Advanced ActiveMQ Broker Features	270
11.1. Introduction	270
11.2. Wildcards and Composite Destinations	270
11.2.1. Subscribing to Wildcard Destinations	270
11.2.2. Sending a Message to Multiple Destinations	272
11.3. Advisory Messages	273
11.4. Virtual Topics	276
11.5. Retroactive Consumers	278
11.6. Message Redelivery and Dead-letter Queues	280
11.7. Summary	282
12. Advanced Client Options	283
12.1. Exclusive Consumer	283
12.1.1. Exclusive Consumer Example	285
12.2. Message Groups	287
12.3. ActiveMQ Streams	290
12.4. Blob Messages	292
12.5. Summary	294
13. Tuning ActiveMQ For Performance	295
13.1. General Techniques	295
13.1.1. Persistent vs Non-Persistent Messages	296
13.1.2. Transactions	298
13.1.3. Embedding Brokers	299
13.1.4. Tuning the OpenWire protocol	302
13.1.5. Tuning the TCP Transport	305
13.2. Optimizing Message Producers	305
13.2.1. Asynchronous send	306
13.2.2. Producer Flow Control	307
13.3. Optimizing Message Consumers	309
13.3.1. Prefetch Limit	310

Please post comments or corrections to the [Author Online Forum](#)

13.3.2. Delivery and Acknowledgement of messages	313
13.3.3. Asynchronous dispatch	315
13.4. Putting it all Together	318
13.5. Summary	322
14. Administering and Monitoring ActiveMQ	323
14.1. APIs	323
14.1.1. JMX	324
14.1.2. Advisory Messages	342
14.2. Tools	350
14.2.1. Command-Line Tools	350
14.2.2. Command Agent	357
14.2.3. JConsole	361
14.2.4. Web Console	365
14.3. Logging	367
14.3.1. Client Logging	370
14.3.2. Logging Interceptor	372
14.4. Summary	373

Part I. An Introduction to Messaging and ActiveMQ

Apache ActiveMQ is a message broker for remote communication between systems using the Java Message Service specification. Although ActiveMQ is written in Java, APIs for many languages than other Java are provided including C/C++, .NET, Perl, PHP, Python, Ruby and many more. This book provides the information needed to understand, configure and use ActiveMQ successfully to meet the requirements of many business applications.

In Part I, you will be introduced to ActiveMQ, the concepts surrounding enterprise messaging and the examples that will be used throughout the book. These chapters provide a good base set of knowledge for the rest of the book.

Chapter 1. Introduction to Apache ActiveMQ

Enterprise messaging software has been in existence since the late 1980s. Not only is messaging a style of communication between applications, it is also a style of integration. Therefore, messaging fulfills the need for both notification as well as interoperation amongst applications. However, it's only within the last 10 year that open source solutions have emerged. Apache ActiveMQ is one such solution, providing the ability for applications to communicate in an asynchronous, loosely-coupled manner. This chapter will introduce you to ActiveMQ.

Your first steps with ActiveMQ are important to your success in using it for your own work. To the novice user, ActiveMQ may appear to be daunting and yet to the seasoned hacker, it might be easier to understand. This chapter will walk you through the task of becoming familiar with ActiveMQ in a simple manner.

In this chapter, readers will achieve the following:

- Gain an understanding of ActiveMQ and its features
- Make sense of why you might use ActiveMQ
- Acquire ActiveMQ and getting started using it
- Install and verify that ActiveMQ is running properly by using the ActiveMQ examples
- Identify and understand the overall use cases to be covered by the examples for the book

1.1. What is ActiveMQ?

ActiveMQ is an open source, JMS 1.1 compliant, message-oriented middleware (MOM) from the Apache Software Foundation that provides high-availability,

performance, scalability, reliability and security for enterprise messaging. ActiveMQ is licensed using the Apache License, one of the most liberal and business friendly OSI-approved licenses available. Because of the Apache License, anyone can use or modify ActiveMQ without any repercussions for the redistribution of changes. This is a critical point for many businesses who use ActiveMQ in a strategic manner. As described later in Chapter 2, the job of a MOM is to mediate events and messages amongst distributed applications, guaranteeing that they reach their intended recipients. So it's vital that a MOM must be highly available, performant and scalable.

The goal of ActiveMQ is to provide standards-based, message-oriented application integration across as many languages and platforms as possible. ActiveMQ implements the JMS spec and offers many additional features and value on top of this spec. These additional features including items such as JMX management of the broker, master/slave capability, message grouping, total ordering of messages, consumer priority for location affinity, retroactive consumer feature for receiving old messages upon subscription activation, handling for slow consumer situations, virtual destinations to lower the number of required connections to the broker, sophisticated message persistence, support for cursoring to handle large messages, support for message transformation, support for the EIP patterns via Apache Camel, mirrored queues for easier monitoring and much, much more.

1.1.1. ActiveMQ Features

JMS Compliance - A good starting point for understanding the features in ActiveMQ is that ActiveMQ is an implementation of the JMS 1.1 spec. ActiveMQ is standards-based in that it is a JMS 1.1 compliant MOM. As discussed later in this chapter , the JMS spec provides many benefits and guarantees including synchronous or asynchronous message delivery, once-and-only-once message delivery, message durability for subscribers and much more. By adhering to the JMS spec for such features means that no matter what JMS provider is used, the same base set of features will be made available.

Connectivity - ActiveMQ provides a wide range of connectivity options including support for protocols such as HTTP/S, JGroups, JXTA, multicast, SSL, TCP, UDP,

XMPP and more. Support for such a wide range of protocols equates to more flexibility. Many existing systems utilize a particular protocol and don't have the option to change so a messaging platform that supports many protocols lowers the barrier to adoption. Though connectivity is very important, the ability to closely integrate with other containers is also very important. Chapter 4 addresses both the transport connectors and the network connectors in ActiveMQ.

Pluggable Persistence and Security - ActiveMQ provides multiple flavors of persistence and you can choose between them. Also, security in ActiveMQ can be completely customized for the type of authentication and authorization that's best for your needs. These two topics are discussed in Chapter 5 and Chapter 6.

Building Messaging Applications With Java - The most common route with ActiveMQ is with Java applications for sending and receiving messages. This task entails use of the JMS spec APIs with ActiveMQ and is covered in Chapter 7, *Creating Java Applications With ActiveMQ*.

Integration With Other Java Containers - ActiveMQ provides the ability to easily integrate with many popular Java containers including Apache Geronimo, Apache Tomcat, JBoss, Jetty, etc. Integrating ActiveMQ with such Java containers is covered in Chapter 8.

Client APIs - ActiveMQ also provides a client API for many languages besides just Java including C/C++, .NET, Perl, PHP, Python, Ruby and more. This opens the door to many more opportunities where ActiveMQ can be utilized outside of just the Java world. Many other languages also have access to all of the features and benefits provided by ActiveMQ through these various client APIs. Of course, the ActiveMQ broker still runs in a Java VM but the clients can be written using any of the supported languages. Connectivity to ActiveMQ is covered in Chapter 9.

Broker Clustering - Many ActiveMQ brokers can work together as a federated network of brokers for scalability purposes. This is known as a network of brokers and can support many different topologies. This topic is covered in Chapter 10.

Many Advanced Broker Features and Client Options - ActiveMQ provides many sophisticated features for both the broker and the clients connecting to the broker as well as a minimal introduction to Apache Camel. These features are

discussed in Chapter 11 and Chapter 12.

Dramatically Simplified Administration - ActiveMQ is designed with developers in mind and as such it doesn't require a dedicated administrator because it provides many easy to use yet very powerful administration features. This is all covered in Chapter 13.

This is just a taste of the features offered by ActiveMQ. As you can see, these topics will be addressed through the rest of the chapters of the book. For demonstration purposes, a couple of simple examples will be carried throughout and these examples will be introduced in this chapter. But before we take a look at the examples and given the fact that you've been presented with many different features, I'm sure you have some questions about why you might use ActiveMQ.

1.1.2. Why Use ActiveMQ?

When considering distributed application design, application coupling is important. Coupling refers to the interdependence of two or more applications or systems. An easy way to think about this are changes to any application and the implications across the other applications in the architecture. Do changes to one application force changes to other applications involved? If the answer is yes, then those applications are tightly coupled. However, if one application can be changed without affecting other applications, then those applications are loosely coupled. The overall lesson here is that tightly coupled applications are more difficult to maintain compared to loosely coupled applications. Said another way, loosely coupled applications can easily deal with unforeseen changes.

Technologies such as those discussed in Chapter 2 (COM, CORBA, DCE and EJB) using techniques called Remote Procedural Calls (RPC) are considered to be tightly coupled. Using RPC, when one application calls another application, the caller is blocked until the callee returns control to the caller. The diagram in Figure 1.1 below depicts this concept.

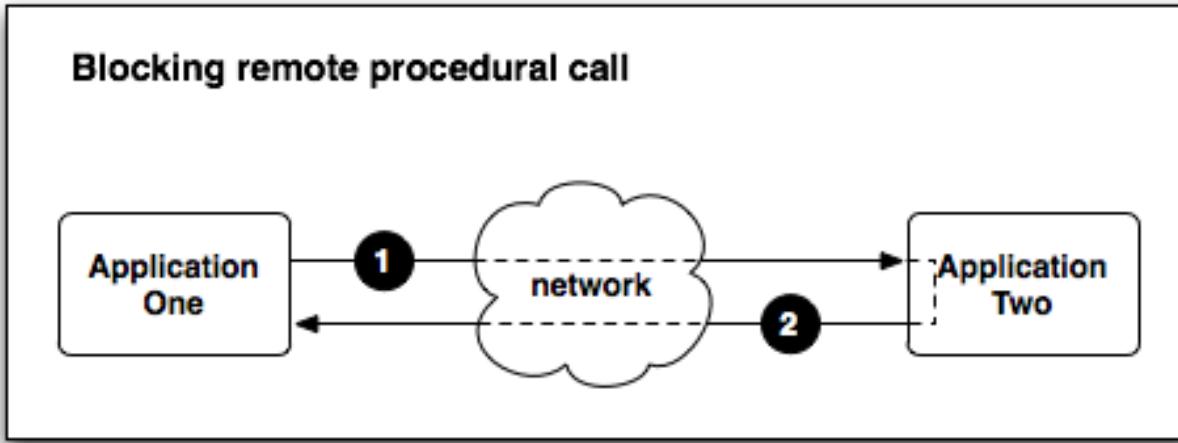


Figure 1.1. Two tightly-coupled RPC applications

The caller (application one) in Figure 1.1, “Two tightly-coupled RPC applications” is blocked until the callee (application two) returns control. Many system architectures have used RPC and been very successful. However, there are many disadvantages to such tightly coupled technologies, most commonly resulting in higher maintenance costs since even small changes ripple throughout the system architecture. Timing is very important between the two applications whereby application two must be available to receive the call from application one, otherwise the call fails and the whole architecture fails. Compare and contrast this with an architecture where two applications are completely unaware of one another such as that depicted in Figure 1.2.

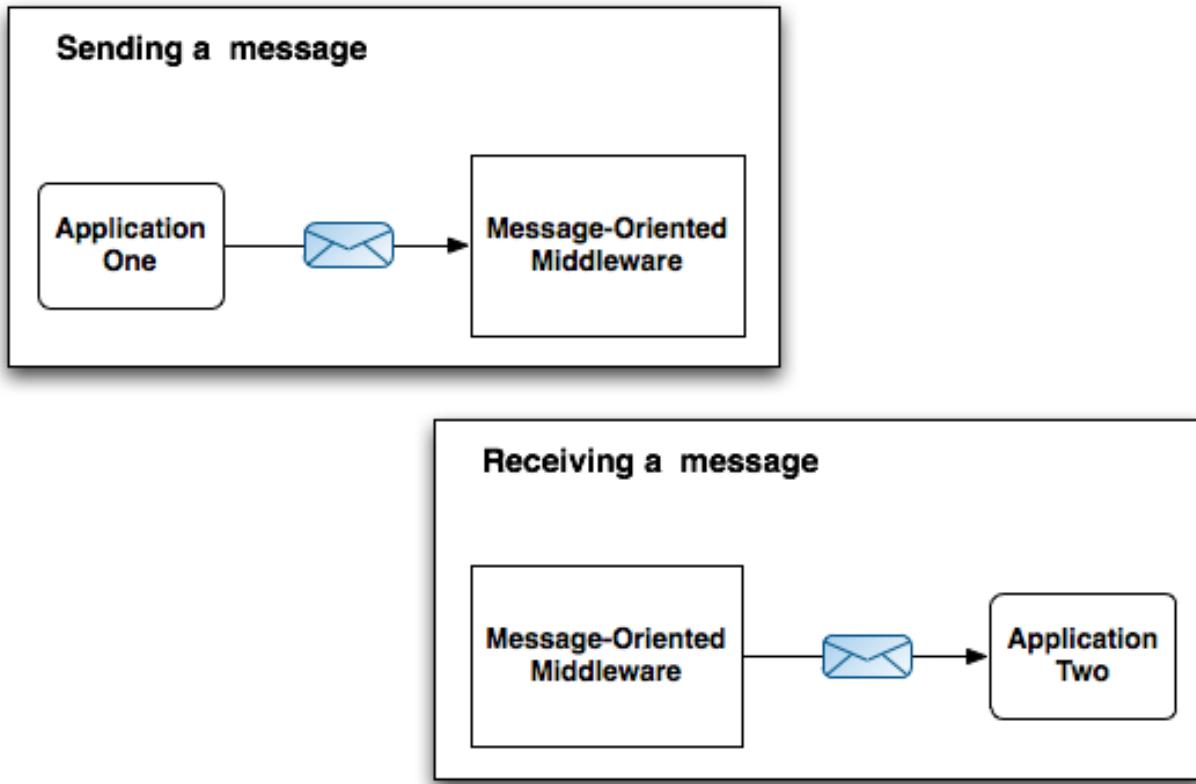


Figure 1.2. Two loosely-coupled JMS applications

Application one in Figure 1.2 makes a call to send a message to the MOM in a one-way fashion. Then, possibly sometime later, application two makes a call to receive a message from the MOM, again, in a one-way fashion. Neither application has any knowledge that the other even exists and there is no timing between the two applications. This one-way style of interaction results in much lower maintenance because changes in one application have little to no effect on the other application. For these reasons, loosely coupled applications offer some big advantages over tightly coupled architectures when considering distributed application design. This is where ActiveMQ enters the picture.

ActiveMQ provides the benefits of loose coupling as noted above and is commonly introduced into an architecture to mitigate the classic tight coupling of RPC style applications. Such design is considered to be asynchronous where the calls from either application have no bearing on one another; there is no interdependence. The

applications can rely upon ActiveMQ's implementation of the JMS spec and it's guaranteed delivery of messages. Because of this, it is often said that applications sending messages just fire and forget. They send the message to ActiveMQ and are not concerned with how or when the message is delivered. In the same rite, the consuming applications have no concerns with where the messages originated or how they were sent to ActiveMQ. This is an especially powerful benefit in heterogeneous environments allowing clients to be written using different languages and even possibly different wire protocols. ActiveMQ acts as the middleman allowing heterogeneous integration and interaction in an asynchronous manner.

So ActiveMQ is a good solution to introduce loose coupling into an architecture and to provide heterogeneous integration. So when and where should it be used to introduce these benefits?

1.1.3. When and Where to Use ActiveMQ

As already discussed, tightly coupled applications and systems can be problematic for many reasons. Switching from such a tightly coupled design to a loosely coupled one can be a very viable option, but making such a switch requires some planning. Performing a large-scale conversion from one style of design to another is always a difficult move, both in terms of the technical work involved as well as in the impact on users of the applications. Most commonly a large-scale conversion means that the users are affected quite dramatically and need to be educated about such a change. Introducing loose coupling into a design offers many user benefits, most prominent of which is the switch from synchrony to asynchrony.

Applications using RPC style synchronous calls are widespread and even though many have been very successful, conversion to the use of asynchronous calls can bring many more benefits without giving up the guarantee of a response. In refactoring such a design toward asynchronous calls, applications are freed up from waiting for control to return to the caller so that other tasks may be performed instead. Consider the two use cases that will be used throughout the rest of the book - the stock portfolio example and the job queues example. These two examples are discussed in detail in Chapter 3.

Any situation where two applications need to communicate is a potential spot to use JMS messaging, no matter whether that communication is local to a single machine or distributed across machines. Communication between two disparate applications that are distributed is low-hanging fruit for applying ActiveMQ. But even in situations where two applications reside on the same machine is a very viable option for using ActiveMQ. The benefits of guaranteed messaging and asynchronous communication provided by ActiveMQ can be used in both of the these environments as ActiveMQ is versatile enough to handle them both.

In an environment where two applications reside on the same machine and they need to communicate, depending on how those applications are deployed, you might consider running ActiveMQ stand alone on that machine or embedding ActiveMQ in a Java application server. Using either deployment style, each application can send and receive messages using destinations in ActiveMQ. This provides the option of using either the pub/sub or the point-to-point messaging domain without waiting for a synchronous call to complete. Each application can simply send a message and continue along with other work immediately; there is no requirement to make a blocking call that must be completed before performing other tasks. By working in this manner, the guarantee of message delivery is no longer in the hands of each application as this responsibility has been offloaded to ActiveMQ.

Many solutions are provided for using messaging between distributed applications, two of which include the use of a single ActiveMQ instance or the use of multiple ActiveMQ instances in a federated manner. The first scenario is the simplest solution whereby a single ActiveMQ instance is used. Then each application sends and receives messages to ActiveMQ in much the same manner as mentioned above with two applications on the same machine. A single instance of ActiveMQ sits between the distributed applications for mediating messages. This instance of ActiveMQ could be installed on the same machine as one of the applications or on a completely separate machine. What's most important is that each application needs to be able to communicate directly with ActiveMQ so you must consider the implications of these choices with regard to your network design.

The second scenario is more complex but relies upon ActiveMQ to handle all remote communications instead of the applications themselves. In this scenario, an

ActiveMQ instance is set up locally with each application (either embedded or stand alone) and the application sends and receives messages from this local ActiveMQ instance. Then the ActiveMQ instances are networked together in a federated manner so that messages are delivered remotely between the brokers based on demand from each application. In ActiveMQ parlance, this is known as a *network of brokers*. This concept is most commonly used to increase the amount of messages ActiveMQ can handle but it is also used to mitigate various network topologies where direct remote connection to an ActiveMQ instance is not feasible. In the latter case, sometimes different protocols can be used to allow ActiveMQ to traverse a network in an easier manner.

Bear in mind that these are just a couple of scenarios where ActiveMQ can be used. Both pub/sub and point-to-point messaging are flexible enough to be applied to many different business scenarios and along with the advanced features in ActiveMQ just about any messaging need can be met. Now it's time to get started using ActiveMQ.

1.2. Getting Started With ActiveMQ

Getting started with ActiveMQ is not very difficult. You simply need to start up the broker and make sure that it's running and capable of accepting connections and send messages. ActiveMQ comes with some simple examples that will help you with this task, but first we need to install Java and download ActiveMQ.

1.2.1. Download and Install the Java SE

ActiveMQ requires a minimum of the Sun Java SE 1.5. This must be installed prior to attempting this section. If you do not have the Sun J2SE 1.5 installed and you're using Linux, Solaris or Windows, download and install it from the following URL:

http://java.sun.com/javase/downloads/index_jdk5.jsp

Make sure that you *do not* download JDK 5.0 with Netbeans or the Java Runtime Environment! You need to download JDK 5.0 Update 16. If you're using MacOS X, you should already have Java installed. But just in case you don't, you can grab

Please post comments or corrections to the [Author Online Forum](#)

it from the following URL:

<http://developer.apple.com/java/download/>

Once you have the Java SE installed, you'll need to test that it is set up correctly. To do this, open a terminal or command line and enter the following command:

```
[~]$ java -version  
java version "1.5.0_13"  
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_13-b05-237)  
Java HotSpot(TM) Client VM (build 1.5.0_13-119, mixed mode, sharing)
```

Your output may be slightly different depending on the operating system you're using, but the important part is that there is output from the Java SE. The command above tells us two things; that the J2SE is installed correctly and that you're using version 1.5. If you did not see similar output, then you'll need to rectify this situation before moving on to the next section.

1.2.2. Download ActiveMQ

ActiveMQ is available from the Apache ActiveMQ website at the following URL:

<http://activemq.apache.org/download.html>

Click on the link to the 5.3.0 release and you will find both tarball and zip formats available (the tarball is for Linux and Unix while the zip is for Windows). Once you have downloaded one of the archives, expand it and you're ready to move along. Once you get to this point, you should have the Java SE all set up and working correctly and you've ready to take a peek at the ActiveMQ directory.

1.2.3. Examining the ActiveMQ Directory

From the command line, move into the apache-activemq-5.3.0 directory and list its contents:

```
[apache-activemq-5.3.0]$ ls -1  
LICENSE  
NOTICE  
README.txt  
WebConsole-README.txt  
activemq-all-5.3.0.jar  
bin
```

Please post comments or corrections to the [Author Online Forum](#)

```
conf  
data  
docs  
example  
lib  
user-guide.html  
webapps
```

The contents of the directory are fairly straightforward:

- **LICENSE** - A file required by the ASF for legal purposes; contains the licenses of all libraries used by ActiveMQ
- **NOTICE** - Another ASF-required file for legal purposes; it contains copyright information of all libraries used by ActiveMQ
- **README.txt** - A file containing some URLs to documentation to get new users started with ActiveMQ
- **WebConsole-README.txt** - Contains information about using the ActiveMQ web console
- **activemq-all-5.3.0.jar** - A jar file that contains all of ActiveMQ; it's placed here for convenience if you need to grab it and use it
- **bin** - The bin directory contains binary/executable files for ActiveMQ; the startup scripts live in this directory
- **conf** - The conf directory holds all the configuration information for ActiveMQ
- **data** - The data directory is where the log files and message persistence data is stored
- **docs** - Contains a simple index.html file referring to the ActiveMQ website
- **example** - The ActiveMQ examples; these are what we will use momentarily to test out ActiveMQ quickly

- **lib** - The lib directory holds all the libraries needed by ActiveMQ
- **user-guide.html** - A very brief guide to starting up ActiveMQ and running the examples
- **webapps** - The webapps directory holds the ActiveMQ web console and some other web-related demos

The next task is to start up ActiveMQ and verify it using the examples.

1.2.4. Starting Up ActiveMQ

After downloading and expanding the archive, ActiveMQ is ready for use. The binary distribution provides a basic configuration to get you started easily and that's what we'll use with the examples. So start up ActiveMQ now by running the following command in a LInux/Unix environment:

```
[apache-activemq-5.3.0]$ ./bin/activemq
Java Runtime: Apple Inc. 1.5.0_16 /System/Library/Frameworks/JavaVM.framework/Versions/1.5.0
Heap sizes: current=1984k free=1449k max=520256k
JVM args: -Xmx512M -Dorg.apache.activemq.UseDedicatedTaskRunner=true
-Djava.util.logging.config.file=logging.properties -Dcom.sun.management.jmxremote
-Dactivemq.classpath=/tmp/apache-activemq-5.3.0/conf; -Dactivemq.home=/tmp/apache-activemq-
-Dactivemq.base=/tmp/apache-activemq-5.3.0
ACTIVEMQ_HOME: /tmp/apache-activemq-5.3.0
ACTIVEMQ_BASE: /tmp/apache-activemq-5.3.0
Loading message broker from: xbean:activemq.xml
INFO | Using Persistence Adapter: org.apache.activemq.store.kahadb.KahaDBPersistenceAdapter
INFO | ActiveMQ 5.3.0 JMS Message Broker (localhost) is starting
INFO | For help or more information please see: http://activemq.apache.org/
INFO | Listening for connections at: tcp://mongoose.local:61616
INFO | Connector openwire Started
INFO | ActiveMQ JMS Message Broker (localhost, ID:mongoose.local-56371-1255406832102-0:0)
INFO | Logging to org.slf4j.impl.JCLLoggerAdapter(org.mortbay.log) via org.mortbay.log.Slf4jLog
INFO | jetty-6.1.9
INFO | ActiveMQ WebConsole initialized.
INFO | Initializing Spring FrameworkServlet 'dispatcher'
INFO | ActiveMQ Console at http://0.0.0.0:8161/admin
INFO | Initializing Spring root WebApplicationContext
INFO | Connector vm://localhost Started
INFO | Camel Console at http://0.0.0.0:8161/camel
INFO | ActiveMQ Web Demos at http://0.0.0.0:8161/demo
INFO | RESTful file access application at http://0.0.0.0:8161/fileserver
INFO | Started SelectChannelConnector@0.0.0.0:8161
```

This command starts up the ActiveMQ broker and some of its connectors to expose it to clients via a few protocols, namely TCP, SSL, STOMP and XMPP. Just be aware that ActiveMQ is started up and available to clients over those four protocols and the port numbers used for each. This is all configurable and will be discussed later in Chapter 3. For now, the output above tells you that ActiveMQ is up and running and ready for use. Now it's ready to begin handling some messages.

Uniquely Naming An ActiveMQ Broker

When developing in a team environment and using the default configuration that is part of the ActiveMQ distribution, it's highly possible (and quite probable) that two or more ActiveMQ instances will connect to one another and begin consuming one another's messages. Here are some recommendations for preventing this situation from occurring:

1. Remove the discoveryUri portion of the openwire transport connector

The transport connector whose name is openwire is configured by default to advertise the broker's TCP transport using multicast. This allows other brokers to automatically discover it and connect to it if necessary.

Below is the openwire transport connector definition from the conf/activemq.xml configuration file:

```
<transportConnector name="openwire" uri="tcp://localhost:61616"  
discoveryUri="multicast://default"/>
```

To stop the broker from advertising the TCP transport over multicast for discovery by other brokers, just change the definition to remove the discoveryUri attribute so it looks like this:

```
<transportConnector name="openwire" uri="tcp://localhost:61616" />
```

2. Comment out/remove the default-nc network connector

The network connector named default-nc utilizes the multicast transport to automatically and dynamically discover other brokers. To stop this

behavior, comment out/remove the default-nc network connector so that it won't discover other brokers without your knowledge.

Below is the default-nc network connector definition from the conf/activemq.xml configuration file:

```
<networkConnector name="default-nc" uri="multicast://default"/>
```

To disable this network connector, comment it out so it looks like this:

```
<!--networkConnector name="default-nc" uri="multicast://default"-->
```

3. **Give the broker a unique name** - The default configuration for ActiveMQ in the conf/activemq.xml file provides a broker name of localhost as shown below:

```
<broker xmlns="http://activemq.apache.org/schema/core" brokerName="localhost" dataDirectory="${activemq.base}/data">
```

In order to uniquely identify your broker instance, change the brokerName attribute from localhost to something unique. This is especially handy when searching through log files to see which brokers are taking certain actions.

1.2.5. Verifying Your ActiveMQ Setup With the Examples

Now ActiveMQ is running in one terminal, now you need to open a couple more terminals to verify that ActiveMQ is working correctly. In the second terminal, move into the example directory and look at the contents:

```
[apache-activemq-5.3.0]$ cd ./example/
bsnyder@mongoose [example]$ ls -l
build.xml
conf
perfHarness
ruby
src
transactions
```

The example directory contains a few different items including:

- **build.xml** - An Ant build configuration for use with the Java examples
- **conf** - The conf directory holds configuration information for use with the Java examples
- **perfarness** - The perfarness directory contains a script for running the IBM JMS performance harness against ActiveMQ
- **ruby** - The ruby directory contains some examples of using ActiveMQ with Ruby and the STOMP connector
- **src** - The src directory is where the Java examples live; this directory is used by the build.xml
- **transactions** - The transactions directory holds an ActiveMQ implementation of the TransactedExample from Sun's JMS Tutorial

Using the second terminal, run the following command to start up a JMS consumer:

```
[example]$ ant consumer
Buildfile: build.xml

init:
compile:
consumer:
[echo] Running consumer against server at $url = tcp://localhost:61616 for subject ${subject}
[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: 2000 message(s) then we will shutdown
```

The command above compiles the Java examples and starts up a simple JMS consumer. As you can see from the output above, this consumer is:

- Connecting to the broker using the TCP protocol (tcp://localhost:61616)

- Watching a queue named TEST.FOO
- Using non-durable subscription
- Waiting to receive 2000 messages before shutting down

Basically, the JMS consumer is connected to ActiveMQ and waiting for messages. Now you can send some messages to the TEST.FOO destination.

In the third terminal, move into the example directory and start up a JMS producer. This will immediately begin to send messages:

```
[example]$ ant producer
Buildfile: build.xml

init:

compile:

producer:
    [echo] Running producer against server at $url = tcp://localhost:61616 for subject ${subject}
    [java] Connecting to URL: tcp://localhost:61616
    [java] Publishing a Message with size 1000 to queue: TEST.FOO
    [java] Using non-persistent messages
    [java] Sleeping between publish 0 ms
    [java] Sending message: Message: 0 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
    [java] Sending message: Message: 1 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
    [java] Sending message: Message: 2 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
    [java] Sending message: Message: 3 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
    [java] Sending message: Message: 4 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
    [java] Sending message: Message: 5 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
    [java] Sending message: Message: 6 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
    [java] Sending message: Message: 7 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
    [java] Sending message: Message: 8 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
    [java] Sending message: Message: 9 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
    [java] Sending message: Message: 10 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
    [java] Sending message: Message: 11 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
    ...
    [java] Sending message: Message: 1998 sent at: Fri Jun 20 13:48:21 MDT 200...
    [java] Sending message: Message: 1999 sent at: Fri Jun 20 13:48:21 MDT 200...
    [java] Done.
    [java] connection {
    [java]     session {
    [java]         messageCount{ count: 0 unit: count startTime: 1213991298653 lastSampleTime: ...
description: Number of messages exchanged }
        [java]         messageRateTime{ count: 0 maxTime: 0 minTime: 0 totalTime: 0 averageTime: 0 ...
averageTimeExMinMax: 0.0 averagePerSecond: 0.0 averagePerSecondExMinMax: 0.0 unit: millis ...
1213991298654 lastSampleTime: 1213991298654 description: Time taken to process a message (t...
    }
    [java]         pendingMessageCount{ count: 0 unit: count startTime: 1213991298654 lastSamp...
```

Please post comments or corrections to the [Author Online Forum](#)

```

1213991298654 description: Number of pending messages }
    [java]      expiredMessageCount{ count: 0 unit: count startTime: 1213991298654 lastSamp...
1213991298654 description: Number of expired messages }
    [java]      messageWaitTime{ count: 0 maxTime: 0 minTime: 0 totalTime: 0 averageTime: 0 ...
averageTimeExMinMax: 0.0 averagePerSecond: 0.0 averagePerSecondExMinMax: 0.0 unit: millis st...
1213991298654 lastSampleTime: 1213991298654 description: Time spent by a message before bei...
    [java]      durableSubscriptionCount{ count: 0 unit: count startTime: 1213991298654 last...
1213991298654 description: The number of durable subscriptions }

    [java]      producers {
        [java]      producer queue://TEST.FOO {
            [java]      messageCount{ count: 0 unit: count startTime: 1213991298662 lastSampleT...
1213991298662 description: Number of messages processed }
            [java]      messageRateTime{ count: 0 maxTime: 0 minTime: 0 totalTime: 0 averageTime...
averageTimeExMinMax: 0.0 averagePerSecond: 0.0 averagePerSecondExMinMax: 0.0 unit: millis st...
1213991298662 lastSampleTime: 1213991298662 description: Time taken to process a message (t...
}
            [java]      pendingMessageCount{ count: 0 unit: count startTime: 1213991298662 last...
1213991298662 description: Number of pending messages }
            [java]      messageRateTime{ count: 0 maxTime: 0 minTime: 0 totalTime: 0 averageTime...
averageTimeExMinMax: 0.0 averagePerSecond: 0.0 averagePerSecondExMinMax: 0.0 unit: millis st...
1213991298662 lastSampleTime: 1213991298662 description: Time taken to process a message (t...
}
            [java]      expiredMessageCount{ count: 0 unit: count startTime: 1213991298662 last...
1213991298662 description: Number of expired messages }
            [java]      messageWaitTime{ count: 0 maxTime: 0 minTime: 0 totalTime: 0 averageTime...
averageTimeExMinMax: 0.0 averagePerSecond: 0.0 averagePerSecondExMinMax: 0.0 unit: millis st...
1213991298662 lastSampleTime: 1213991298662 description: Time spent by a message before bei...
                [java]      }
                [java]      }
                [java]      consumers {
                [java]      }
                [java]      }
                [java]      }

```

Although the output above has been truncated for readability, the command above starts up a simple JMS producer and you can see from the output that it:

- Connects to the broker using the TCP protocol (tcp://localhost:61616)
- Publishes messages to a queue named TEST.FOO
- Uses non-persistent messages
 - Does not sleep between receiving messages

Once the JMS producer is connected, it then sends 2000 messages and shuts down. This is the number of message on which the consumer is waiting to consume

before it shuts down. So as the messages are being sent by the producer in terminal number three, flip back to terminal number two and watch the JMS consumer as it consumes those messages. Below is the output you will see:

```
[java] Received: Message: 0 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
[java] Received: Message: 1 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
[java] Received: Message: 2 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
[java] Received: Message: 3 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
[java] Received: Message: 4 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
[java] Received: Message: 5 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
[java] Received: Message: 6 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
[java] Received: Message: 7 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
[java] Received: Message: 8 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
[java] Received: Message: 9 sent at: Fri Jun 20 13:48:18 MDT 2008 ...
...
[java] Received: Message: 1997 sent at: Fri Jun 20 13:48:21 MDT 200...
[java] Received: Message: 1998 sent at: Fri Jun 20 13:48:21 MDT 200...
[java] Received: Message: 1999 sent at: Fri Jun 20 13:48:21 MDT 200...
[java] Closing connection
```

Again, the output has been truncated a bit for brevity but this doesn't change the fact that the consumer received 2000 messages and shut itself down. At this time, both the consumer and the producer should be shut down but the ActiveMQ broker is still running in the first terminal. Take a look at the first terminal again and you will see that ActiveMQ appears to have not budged at all. This is because the default logging configuration doesn't output anything beyond what is absolutely necessary. (If you'd like to tweak the logging configuration to output more information as messages are sent and received you can do so, but logging will be covered later in Chapter 14.)

So what did you learn here? Through the use of the Java examples that come with ActiveMQ, it has been proven that the broker is up and running and is able to mediate messages. This doesn't seem like much but it's an important first step. If you were able to successfully run the Java examples then you know that you have no networking problems on the machine you're using and you know that ActiveMQ is behaving properly. If you were unable to successfully run the Java examples, then you'll need to troubleshoot the situation. If you need some help, heading over to the Manning forums or the ActiveMQ mailing lists are both viable options. These examples are just to get you started but can be used to test many scenarios. Throughout the rest of the book, some different examples surrounding a couple of common use cases will be used to demonstrate ActiveMQ and its

features.

1.3. Summary

ActiveMQ is clearly a very versatile message-oriented middleware, capable of adapting to many different situations. In this chapter, you have learned about some of the features in ActiveMQ and read about some scenarios where ActiveMQ might be applied. The next step is to begin to understand the ActiveMQ configuration so that you can configure connectors for clients and additional ActiveMQ brokers.

Chapter 2. Understanding Message-Oriented Middleware and JMS

At one time or another, every software developer has the need to communicate between applications or transfer data from one system to another. Not only are there many solutions to this sort of problem, but depending on your constraints and requirements, deciding how to go about such a task can be a big decision. Business requirements oftentimes place restrictions on items that directly impact such a decision including performance, scalability, reliability and more. There are many applications that we use every day that impose just such requirements including ATMs, airline reservation systems, credit card systems, point-of-sale systems and telecommunications just to name a few. Where would we be without most of these applications in our daily lives today?

For just a moment, think about how these types of services have made your life easier. These applications and others like them are made possible because of their reliable and secure nature. Behind the scenes of these applications, just about all of them are composed of many applications, usually distributed, communicating by passing events or messages back and forth. Even the most sophisticated financial trading systems are integrated in this manner, operating completely through the sending and receipt of business information amongst all the necessary systems using messaging.

In this chapter, readers will learn the following items:

- Some history behind enterprise messaging
- A definition of Message-Oriented Middleware (MOM)
- An introduction to the Java Message Service (JMS)
- Some examples of using the JMS API

2.1. Introduction to Enterprise Messaging

Most systems like those mentioned above were built using mainframe computers and many still make use of them today. So how can these applications work in such a reliable manner? To answer this and other questions, let's briefly explore some of the history behind such solutions.

Starting in the 1960s, large organizations invested in mainframes for critical applications for functions such as data processing, financial processing, statistical analysis and much more. Mainframes provided many benefits including high availability, redundancy, extreme reliability and scalability, upgradability without service interruption and many other critical features required by business.

Although these systems were extremely powerful, access to such systems was restricted as input options were few. Also, interconnectivity amongst systems had not yet been invented meaning that parallel processing was not yet possible.

Figure 2.1 shows a diagram demonstrating how terminals connect to a mainframe.

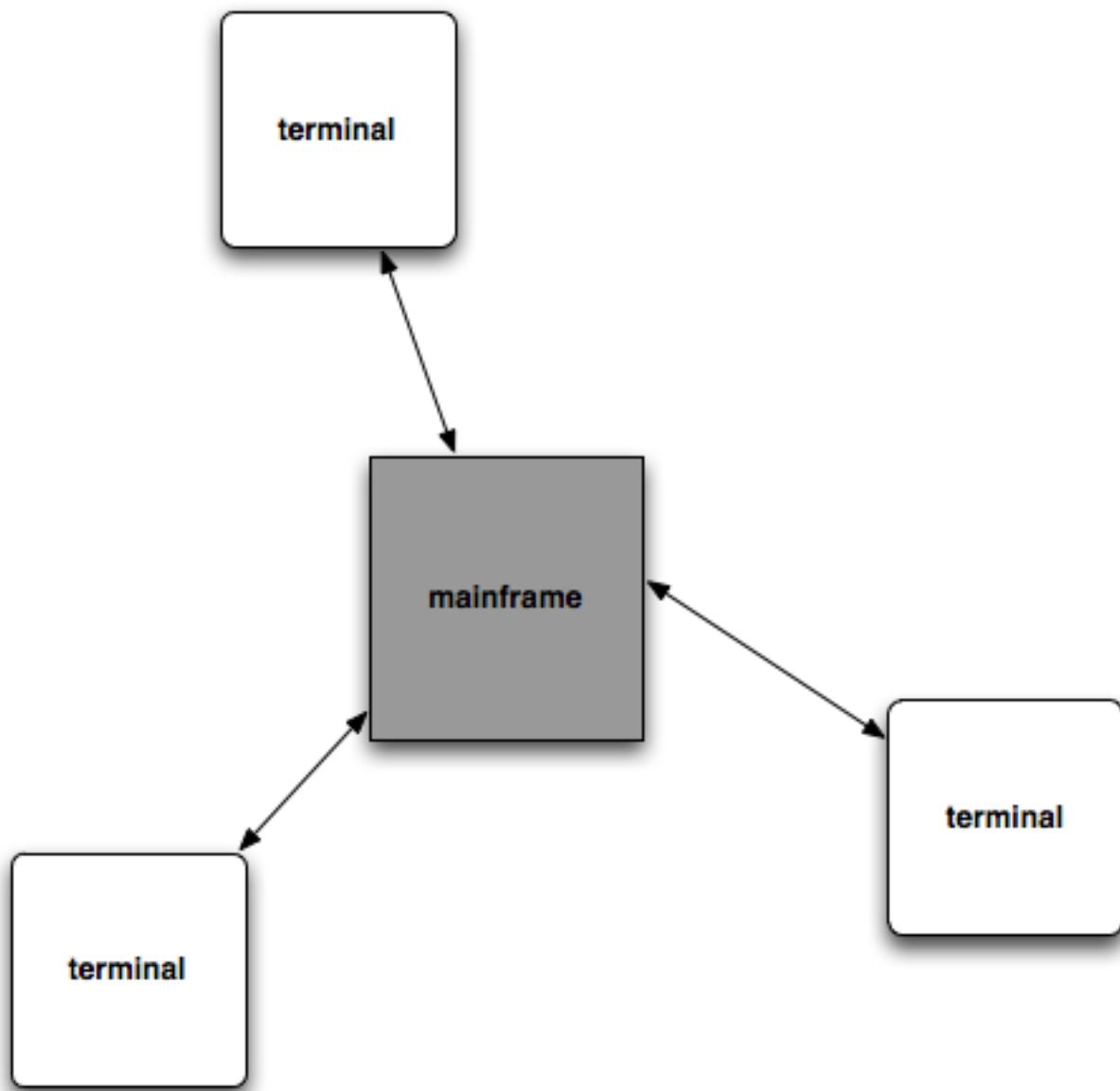


Figure 2.1. Terminals connecting to a mainframe

In the 1970s, users began to access mainframes through terminals which dramatically expanded the use of these systems by allowing thousands of concurrent users. It was during this period that computer networks were invented and connectivity amongst mainframes themselves now became possible. By the 1980s, not only were graphical terminals available, but PCs were also invented and terminal emulation software quickly became common. Interconnectivity became

even more important because applications needing access to the mainframe were being developed to run on PCs and workstations. Figure 2.2 shows these various types of connectivity to the mainframe. Notice how this expanded connectivity introduced additional platforms and protocols, posing a new set of problems to be addressed.

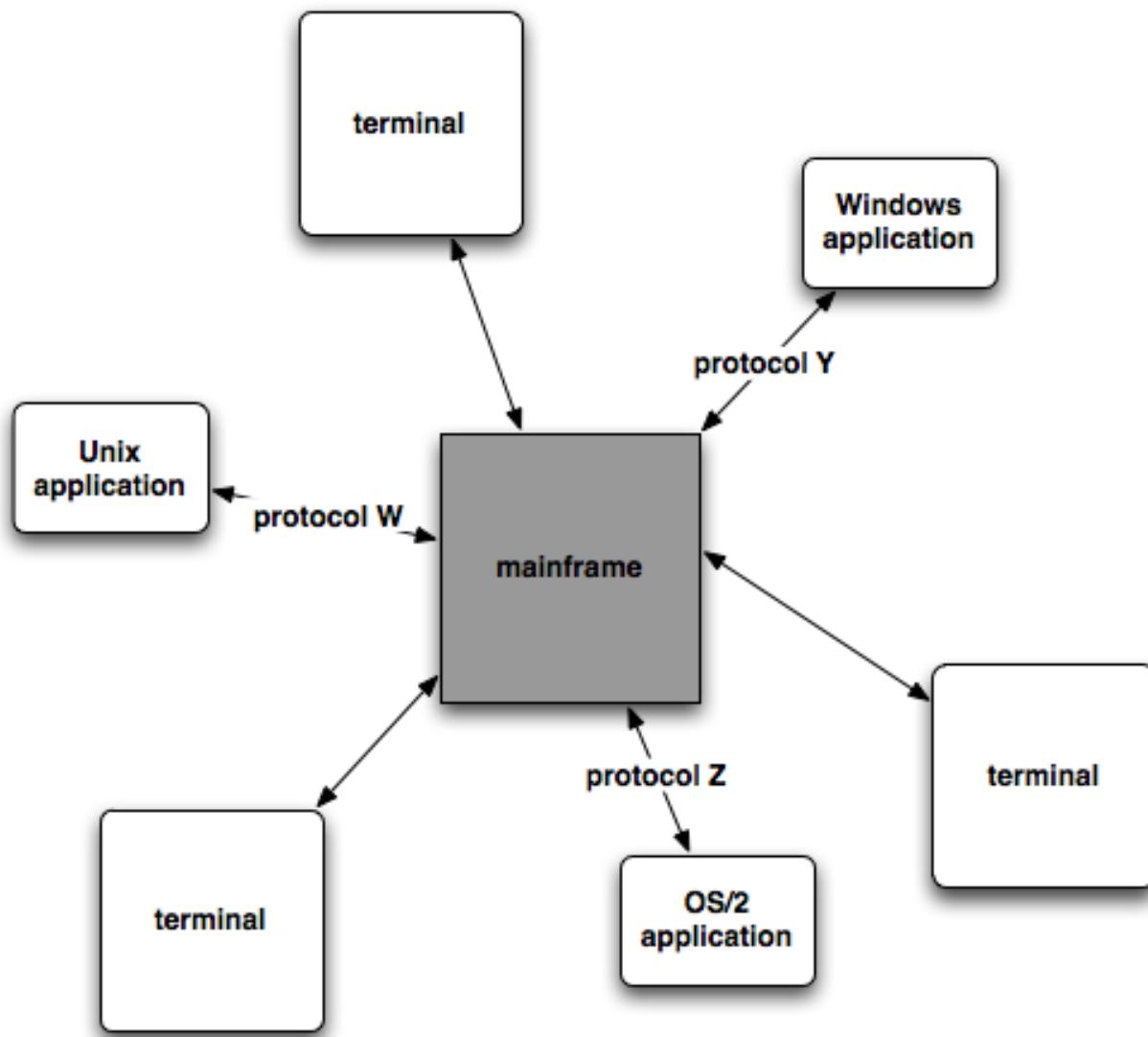


Figure 2.2. Terminals and applications connecting to a mainframe

Connecting a source system and a target system was not easy as each data format, hardware and protocol required a different type of adapter. As the list of adapters

grew so did the versions of each causing them to become difficult to maintain. Soon the effort required to maintain the adapters outweighed that of the systems themselves. This is where enterprise messaging entered the picture.

The purpose of enterprise messaging was to transfer data amongst disparate systems by sending messages from one system to another. There have been many technologies for various forms of messaging through the years, including:

- Solutions for remote procedural calls such as COM, CORBA, DCE and EJB
- Solutions for event notification, inter-process communication and message queuing that are baked into operating systems such as FIFO buffers, message queues, pipes, signals, sockets and others
- Solutions for a category of middleware that provides asynchronous, reliable message queuing such as MQSeries, SonicMQ, TIBCO and Apache ActiveMQ commonly used for Enterprise Application Integration (EAI) purposes

So there are many products that can provide messaging for various purposes, but the last category of solutions above focusing on messaging middleware is what we'll discuss here. So let's explore messaging middleware further. Necessity is the mother of invention, and this is how messaging middleware was born. A form of software became necessary for communication and data transfer capabilities that could more easily manage the disparity amongst data formats, operating systems, protocols and even programming languages. Additionally, capabilities such as sophisticated message routing and transformation began to emerge as part of or in conjunction with these solutions. Such systems came to be known as message-oriented middleware.

2.2. What is Message Oriented Middleware?

Message-oriented middleware (MOM) is best described as a category of software for communication in a loosely-coupled, reliable, scalable and secure manner amongst distributed applications or systems. MOMs were a very important concept

to the distributed computing world. They allowed application-to-application communication using APIs provided by each vendor and began to deal with many issues in the enterprise messaging space.

The overall idea with a MOM is that it acts as a message mediator between message senders and message receivers. This mediation provides a whole new level of loose coupling for enterprise messaging. Figure 2.3 demonstrates how a MOM is used to mediate connectivity and messaging not only between each application and the mainframe but also from application-to-application.

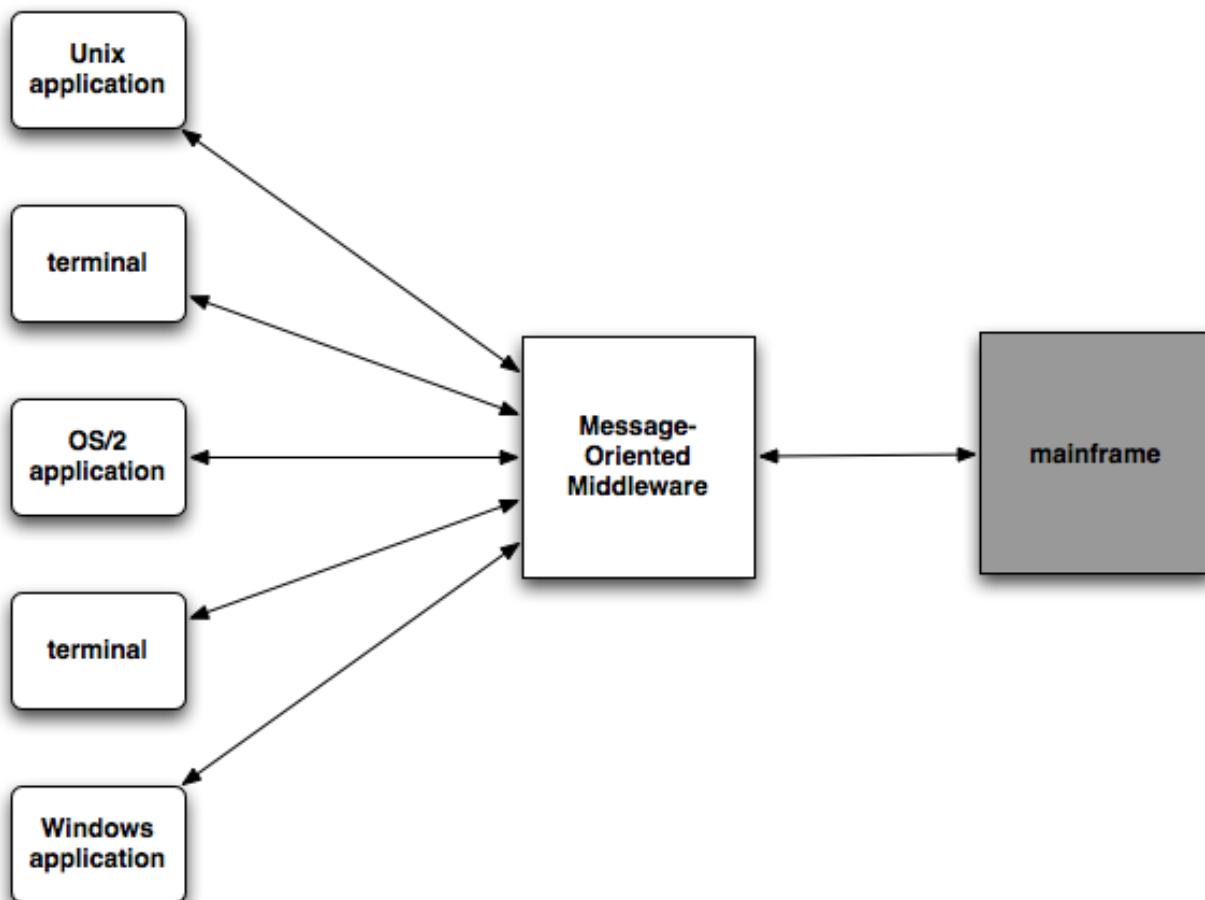


Figure 2.3. Introducing message-oriented middleware

At a high level, messages are a unit of business information that is sent from one application to another via the MOM. Applications send and receive messages via a MOM using what are known as destinations. Messages are addressed to and

delivered to receivers that connect or subscribe to the messages. This is the mechanism that allows for loose coupling between senders and receivers as there is no requirement for each to be connected to the MOM for sending and receiving messages. Senders know nothing about receivers and receives know nothing about senders. This is what is known as asynchronous messaging.

MOMs added many additional features to enterprise messaging that weren't previously possible when systems were tightly coupled. Features such as message persistence, robust communication over slow or unreliable connections, complex message routing, message transformation and much more. Message persistence helps to mitigate slow or unreliable connections made by senders and receivers or in a situation where a receiver just simply fails it won't affect the state of the sender. Complex message routing opens up a huge amount of possibilities including delivering a single message to many receivers, message routing based on properties or the content of a message, etc. Message transformation allows two applications that don't handle the same message format to now communicate.

Additionally, many MOMs on the market today provide support for a diverse set of protocols for connectivity. Some commonly supported protocols include HTTP/S, multicast, SSL, TCP/IP, UDP. Some vendors even provide support for multiple languages, further lowering the barrier to using MOMs in a wide variety of environments.

Furthermore, it's typical for a MOM to provide an API for sending and receiving messages and otherwise interacting with the MOM. For years, all MOM vendors provided their own proprietary APIs for whatever languages they chose. That is, until the Java Message Service came along.

2.3. What is the Java Message Service?

The Java Message Service (JMS) moved beyond vendor-centric MOM APIs to provide an API for enterprise messaging. JMS aims to provide a standardized API to send and receive messages using the Java programming language in a vendor-neutral manner. The JMS API minimizes the amount of enterprise messaging knowledge a Java programmer is required to possess in order to develop

complex messaging applications, while still maintaining a certain amount of portability across JMS provider implementations.

JMS is not itself a MOM. It is an API that abstracts the interaction between messaging clients and MOMs in the same manner that JDBC abstracts communication with relational databases. Figure 2.4 shows at a high level that JMS provides an API used by messaging clients to interact with MOM-specific JMS providers that handle interaction with the vendor-specific MOM. The JMS API lowers the barrier to the creation of enterprise messaging applications. It also eases the portability to other JMS providers.

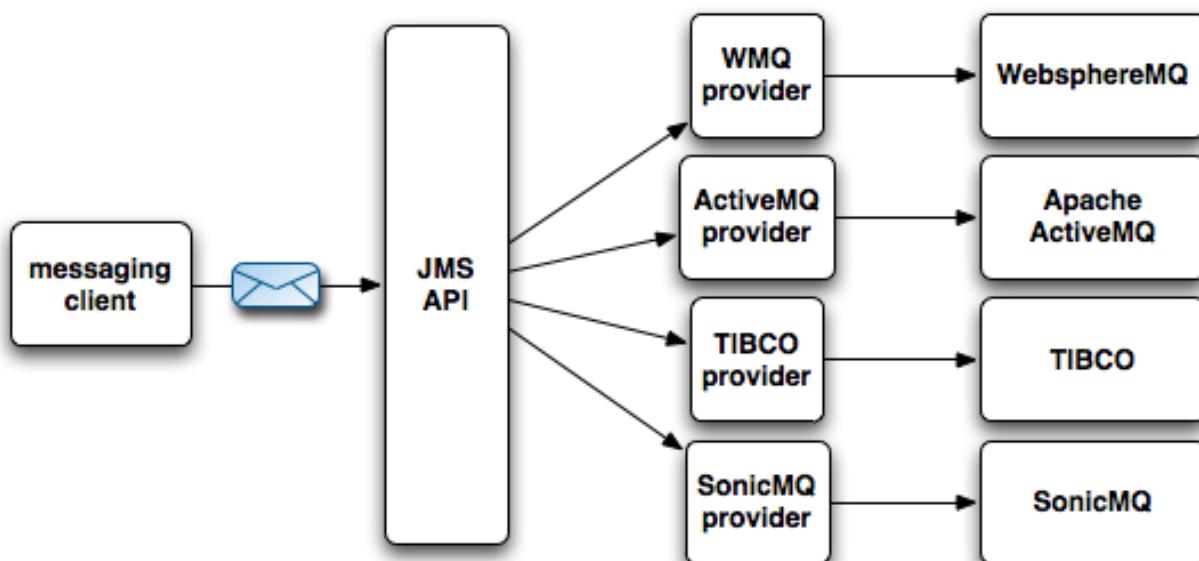


Figure 2.4. JMS allows a single client to easily connect to many JMS providers

Originally created by Sun in conjunction with a group of companies from the enterprise messaging industry, the first version of the JMS spec was released in 1998. The latest release was in 2002 and offering some necessary improvements. The JMS 1.1 release unified the two sets of APIs for working with the two messaging domains. So working with both messaging domains now only requires a single common API. This was a dramatic change that improved the APIs. But backwards compatibility with the old APIs is still supported.

In standardizing the API, JMS formally defined many concepts and artifacts from

the world of messaging:

- **JMS Client** - An application written using 100% pure Java to send and receive messages.
- **Non-JMS Client** - An application is written using the JMS provider's native client API to send and receive messages instead of JMS.
- **JMS Producer** - A client application that creates and sends JMS messages.
- **JMS Consumer** - A client application that receives and processes JMS messages.
- **JMS Provider** - The implementation of the JMS interfaces which is ideally written in 100% pure Java.
- **JMS Message** - The most fundamental concept of JMS; sent and received by JMS clients.
- **JMS Domains** - The two styles of messaging that include point-to-point and publish/subscribe.
- **Administered Objects** - Preconfigured JMS objects that contain provider-specific configuration data for use by clients. These objects are typically accessible by clients via JNDI.
 - **Connection Factory** - Clients use a connection factory to create connections to the JMS provider.
 - **Destination** - An object to which messages are addressed and sent and from which messages are received.

Besides these concepts there are others that are also important. So in the next few sections we'll dive deeper into these concepts and focus on describing these building blocks of JMS.

2.3.1. Messaging Clients

As mentioned in the previous section, the JMS spec defines two types of clients - JMS clients and non-JMS clients. The differences are worthy of a brief discussion, so let's address them briefly.

2.3.1.1. JMS Clients

JMS clients utilize the JMS API for interacting with the JMS provider. Similar in concept to the using the JDBC API to access data in relational databases, JMS clients use the JMS API for standardized access to the messaging service. Many JMS providers (including ActiveMQ) provider many features beyond those required by JMS. It's worth noting that a 100% pure JMS client would only make use of the JMS APIs and would avoid using such additional features. However, the choice to use a particular JMS provider is oftentimes driven by the additional features offered. If a JMS client makes use of such additional features, this client may not be portable to another JMS provider without a refactoring effort.

JMS clients utilize the `MessageProducer` and `MessageConsumer` interfaces in some way. It is the responsibility of the JMS provider to furnish an implementation of each of these interfaces. A JMS client that sends messages is known as a producer and a JMS client that receives messages is known as a consumer. It is possible for a JMS client to handle both the sending and receiving of messages.

2.3.1.1.1. JMS Producer

JMS clients make use of the JMS `MessageProducer` class for sending messages to a destination. The default destination for a given producer is set when the producer is created using the `Session.createProducer()` method. But this can be overridden for individual messages by using the `MessageProducer.send()` method. The `MessageProducer` interface is shown below:

```
public interface MessageProducer {  
    void setDisableMessageID(boolean value) throws JMSEException;  
  
    boolean getDisableMessageID() throws JMSEException;  
  
    void setDisableMessageTimestamp(boolean value) throws JMSEException;
```

Please post comments or corrections to the [Author Online Forum](#)

```
boolean getDisableMessageTimestamp() throws JMSEException;  
  
void setDeliveryMode(int deliveryMode) throws JMSEException;  
  
int getDeliveryMode() throws JMSEException;  
  
void setPriority(int defaultPriority) throws JMSEException;  
  
int getPriority() throws JMSEException;  
  
void setTimeToLive(long timeToLive) throws JMSEException;  
  
long getTimeToLive() throws JMSEException;  
  
Destination getDestination() throws JMSEException;  
  
void close() throws JMSEException;  
  
void send(Message message) throws JMSEException;  
  
void send(Message message, int deliveryMode, int priority,  
          long timeToLive)  
    throws JMSEException;  
  
void send(Destination destination, Message message)  
    throws JMSEException;  
  
void send(  
    Destination destination,  
    Message message,  
    int deliveryMode,  
    int priority,  
    long timeToLive)  
    throws JMSEException;  
}
```

The `MessageProducer` provides methods for not only sending messages but also methods for setting various message headers including the `JMSDeliveryMode`, the `JMSPriority`, the `JMSExpiration` (via the `get/setTimeToLive()` method) as well as a utility `send()` method for setting all three of these at once. These message headers are discussed in Section 2.3.3.

2.3.1.1.2. JMS Consumer

JMS clients make use of the `JMS MessageConsumer` class for consuming messages from a destination. The `MessageConsumer` can consume messages either synchronously by using one of the `receive()` methods or asynchronously by

Please post comments or corrections to the [Author Online Forum](#)

providing a `MessageListener` implementation to the consumer the `MessageListener.onMessage()` method is invoked as messages arrive on the destination. Below is the `MessageConsumer` interface:

```
public interface MessageConsumer {  
    String getMessageSelector() throws JMSEException;  
  
    MessageListener getMessageListener() throws JMSEException;  
  
    void setMessageListener(MessageListener listener) throws JMSEException;  
  
    Message receive() throws JMSEException;  
  
    Message receive(long timeout) throws JMSEException;  
  
    Message receiveNoWait() throws JMSEException;  
  
    void close() throws JMSEException;  
}
```

There is no method for setting the destination on the `MessageConsumer`. Instead the destination is set when the consumer is created using the `Session.createConsumer()` method.

2.3.1.2. Non-JMS Clients

As noted above, a non-JMS client uses a JMS provider's native client API instead of the JMS API. This is an important distinction because native client APIs might offer some different features than the JMS API. Such non-JMS APIs could consist of utilizing the CORBA IIOP protocol or some other native protocol beyond Java RMI. Messaging providers that pre-date the JMS spec commonly have a native client API, but many JMS providers also provide a non-JMS client API.

2.3.2. The JMS Provider

The term JMS provider refers to the vendor-specific MOM that implements the JMS API. Such an implementation provides access to the MOM via the standardized JMS API (remember the analogy to JDBC above).

2.3.3. Anatomy of a JMS Message

The JMS message is the most important concept in the JMS specification. Every concept in the JMS spec is built around handling a JMS message because it is how business data and events are transmitted through any JMS provider. A JMS message allows anything to be sent as part of the message including text and binary data as well as information in the headers as well as additional properties. As depicted in Figure 2.5, JMS messages contain three parts including headers, properties and a payload. The headers provide metadata about the message used by both clients and JMS providers. Properties are optional fields on a message to add additional custom information to the message. The payload is the actual body of the message and can hold both textual and binary data via the various message types.

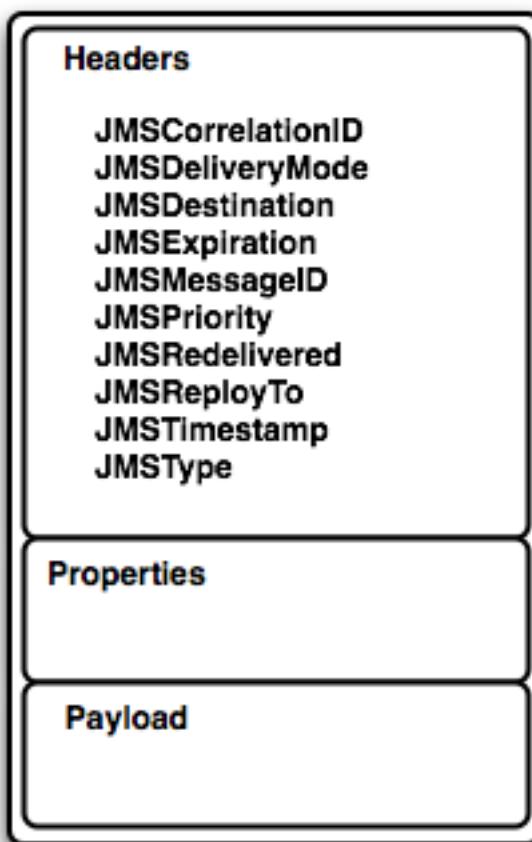


Figure 2.5. A graphical representation of a JMS message

2.3.3.1. JMS Message Headers

As shown in Figure 2.5, JMS messages support a standard list of headers and the JMS API provides methods for working with them. Many of the headers are automatically assigned. The following lists describes each of these headers and are broken down into two lists based on how the headers are assigned to the message.

Headers set automatically by the client's `send()` method:

- **JMSDestination** - The destination to which the message is being sent. This is valuable for clients who consume messages from more than one destination.
- **JMSDeliveryMode** - JMS supports two types of delivery modes for messages: persistent and non-persistent. The default delivery mode is persistent. Each delivery mode incurs its own overhead and implies a particular level of reliability.
- **Persistent** - Advises the JMS provider to persist the message so it's not lost if the provider fails. A JMS provider must deliver a persistent message *once-and-only-once*. In other words, if the JMS provider fails, the message will not be lost and will not be delivered more than once. Persistent messages incur more overhead due to the need to store the message and values reliability over performance.
- **Non-Persistent** - Instructs the JMS provider not to persist the message. A JMS provider must deliver a non-persistent message *at-most-once*. In other words, if the JMS provider fails, the message may be lost, but it will not be delivered twice. Non-persistent messages incur less overhead and values performance over reliability.
The delivery mode is set on the producer and is applied to all messages sent from that producer. But the delivery mode can be overridden for individual messages.
- **JMSExpiration** - The time that a message will expire. This header is used to

prevent delivery of a message after it has expired. The expiration value for messages can be set using either the `MessageProducer.setTimeToLive()` method to set the time-to-live globally for all messages sent from that producer or using one of the `MessageProducer.send()` methods to set the time-to-live locally for each message that is sent. Calling any of these methods sets the default length of time in milliseconds that a message should be considered usable, although the `MessageProducer.send()` methods takes precedence.

The JMSExpiration message header is calculated by adding the timeToLive to the current time in GMT. By default the time-to-live is zero meaning that the message will not expire. If a time-to-live is not specified the default value is used and the message will not expire. If the time-to-live is explicitly specified as zero, then the same is true and the message will not expire.

This header can be valuable for time-sensitive messages. But be aware that JMS providers should not deliver messages that have expired and JMS clients should be written so as to not process messages that have expired.

- **JMSMessageID** - A string that uniquely identifies a message that is assigned by the JMS provider and must begin with 'ID:'. The MessageID can be used for message processing or for historical purposes in a message storage mechanism. Because message IDs can cause the JMS provider to incur some overhead, the producer can advise the JMS provider that the JMS application does not depend on the value of this header via the `MessageProducer.setDisableMessageID()` method. If the JMS provider accepts this advice, the message ID must be set to null. But be aware that a JMS provider may ignore this call and assign a message ID anyway.
- **JMSPriority** - Used to assign a level of importance to a message. This header is also set on the message producer. Once the priority is set on a producer, it applies to all messages sent from that producer. The priority can be overridden for individual messages. JMS defines 10 levels of message priority, zero is the lowest and nine is the highest. These levels are explained below:

- Priorities 0-4 - These priorities are finer granularities of the *normal* priority.
- Priorities 5-9 - These priorities are finer granularities of *expedited* priority. JMS providers are not required to implement message ordering, although most do. They should simply attempt to deliver higher priority messages before lower priority messages.
- **JMSTimestamp** - This header denotes the time the message was sent by the producer to the JMS provider. The value of this header uses the standard Java millis time value. Similar to the JMSMessageID header above, the producer may advise the JMS provider that the JMSTimestamp header is not needed via the `MessageProducer.setDisableMessageTimestamp()` method. If the JMS provider accepts this advice, it must set the JMSTimestamp to zero.

Headers set optionally by the client:

- **JMSCorrelationID** - Used to associate the current message with a previous message. This header is commonly used to associate a response message with a request message. The value of the JMSCorrelationID can be one of the following:
 - A provider-specific message ID
 - An application-specific String
 - A provider-native byte[] value

The provider-specific message ID will begin with the 'ID:' prefix whereas the application-specific String must not start with the 'ID:' prefix. If a JMS provider supports the concept of a native correlation ID, a JMS client may need to assign a specific JMSCorrelationID value to match that expected by non-JMS clients, but this is not a requirement.

- **JMSReplyTo** - Used to specify a destination where a reply should be sent. This header is commonly used for request/reply style of messaging. Messages sent with this header populated are typically expecting a response but it is actually optional. The client must make the decision to respond or not.
- **JMSType** - Used to semantically identify the message type. This header is used by very few vendors and has nothing to do with the payload Java type of the message.

Headers set optionally by the JMS provider:

- **JMSRedelivered** - Used to indicate the likeliness that a message was previously delivered but not acknowledged. This can happen if a consumer fails to acknowledge delivery, if the JMS provider has not been notified of delivery such as an exception being thrown that prevents the acknowledgement from reaching the provider.

2.3.3.2. JMS Message Properties

Properties are more or less just additional headers that can be specified on a message. JMS provides the ability to set custom properties using the generic methods that are provided. Methods are provided for working with many primitive Java types for property values including boolean, byte, short, int, long, float, double, and also the String object type. Examples of these methods can be seen in the excerpt below taken from the `Message` interface:

```
public interface Message {  
    ...  
    boolean getBooleanProperty(String name) throws JMSEException;  
    byte getByteProperty(String name) throws JMSEException;  
    short getShortProperty(String name) throws JMSEException;  
    int getIntProperty(String name) throws JMSEException;  
    long getLongProperty(String name) throws JMSEException;
```

```
float getFloatProperty(String name) throws JMSEException;  
  
double getDoubleProperty(String name) throws JMSEException;  
  
String getStringProperty(String name) throws JMSEException;  
  
Object getObjectProperty(String name) throws JMSEException;  
  
...  
  
Enumeration getPropertyNames() throws JMSEException;  
  
boolean propertyExists(String name) throws JMSEException;  
  
...  
  
void setBooleanProperty(String name, boolean value)  
    throws JMSEException;  
  
void setByteProperty(String name, byte value) throws JMSEException;  
  
void setShortProperty(String name, short value) throws JMSEException;  
  
void setIntProperty(String name, int value) throws JMSEException;  
  
void setLongProperty(String name, long value) throws JMSEException;  
  
void setFloatProperty(String name, float value) throws JMSEException;  
  
void setDoubleProperty(String name, double value) throws JMSEException;  
  
void setStringProperty(String name, String value) throws JMSEException;  
  
void setObjectProperty(String name, Object value) throws JMSEException;  
  
...  
}
```

Also notice the two convenience methods for working with generic properties on a message, namely the `getPropertyNames()` method and the `propertyExists()` method. The `getPropertyNames()` method returns an `Enumeration` of all the properties on a given message to easily iterate through all them. The `propertyExists()` method for testing if a given property exists on a message. Note that the JMS-specific headers are not considered generic properties and not returned in the `Enumeration` return by the `getPropertyNames()` method.

There are three types of properties: arbitrary or custom properties, JMS defined

properties and provider-specific properties.

2.3.3.2.1. Custom Properties

These properties are arbitrary and are defined by a JMS application. Developers of JMS applications can freely define any properties using any Java types necessary using the generic methods shown in the previous section (i.e.,
`getBooleanProperty()`/`setBooleanProperty()`,
`getStringProperty()`/`setStringProperty()`, etc.)

2.3.3.2.2. JMS-Defined Properties

The JMS spec reserves the 'JMSX' property name prefix for JMS-defined properties and support for these properties is optional:

- **JMSXAppID** - Identifies the application sending the message.
- **JMSXConsumerTXID** - The transaction identifier for the transaction within which this message was consumed.
- **JMSXDeliveryCount** - The number of message delivery attempts.
- **JMSXGroupID** - The message group of which this message is a part.
- **JMSXGroupSeq** - The sequence number of this message within the group.
- **JMSXProducerTXID** - The transaction identifier for the transaction within which this message was produced.
- **JMSXRcvTimestamp** - The time the JMS provider delivered the message to the consumer.
- **JMSXState** - Used to define a provider-specific state.
- **JMSXUserID** - Identifies the user sending the message.

The only recommendation provided by the spec for use of these properties is for the *JMSXGroupID* and *JMSXGroupSeq* properties and that these properties should

be used by clients when grouping messages and/or grouping messages in a particular order.

2.3.3.2.3. Provider-Specific Properties

The JMS spec reserves the 'JMS_<vendor-name>' property name prefix for provider-specific properties. Each provider defines its own value for the <vendor-name> placeholder. These are most typically used for provider-specific non-JMS clients and should not be used for JMS-to-JMS messaging.

Now that JMS headers and properties on messages have been discussed, for what exactly are they used? Headers and properties are important when it comes to filtering the messages received by a client subscribed to a destination.

2.3.4. Message Selectors

Consider the fact that there are times when a JMS client is subscribed to a given destination, but it may want to filter the types of messages it receives. This is exactly where headers and properties can be used. For example, if a consumer registered to receive messages from a queue is only interested in messages about a particular stock symbol, this is an easy task as long as each message contains a property that identifies the stock symbol of interest. The JMS client can utilize JMS message selectors to tell the JMS provider that it only wants to receive messages containing a particular value in a particular property.

Message selectors allow a JMS client to specify which messages it wants to receive from a destination based on values in message headers. Selectors are conditional expressions defined using a subset of SQL92. Using boolean logic, message selectors use message headers and properties as criteria for simple boolean evaluation. Messages not matching these expressions are not delivered to the client. Message selectors cannot reference a message payload, only the message headers and properties.

Selectors use conditional expressions for selectors are passed as String arguments using some of the creation methods in the `javax.jms.Session` object. The syntax of these expressions uses various identifiers, literals and operators taken directly

from the SQL92 syntax and are defined in the Table 2.1 table below:

Table 2.1. JMS Selector Syntax

Item	Values
Literals	Booleans TRUE/FALSE; Numbers, e.g., 5, -10, +34; Numbers with decimal or scientific notation, e.g., 43.3E7, +10.5239
Identifiers	A header or property field
Operators	AND, OR, LIKE, BETWEEN, =, <>, <, >, <=, =>, +, -, *, /, IS NULL, IS NOT NULL

The items shown in Table 2.1 are used to create queries against message headers and properties. Consider the message in defined in Example 2.1 below. This message defines a couple properties that will be used for filtering messages in our example below.

Example 2.1. A JMS message with custom properties

```
public void sendStockMessage(Session session,
                             MessageProducer producer,
                             Destination destination,
                             String payload,
                             String symbol,
                             double price)
throws JMSEException {

    TextMessage textMessage = session.createTextMessage();
    textMessage.setText(payload);
    textMessage.setStringProperty("SYMBOL", symbol);❶
    textMessage.setDoubleProperty("PRICE", price);❷
    producer.send(destination, textMessage);
}
```

- ❶ A custom String property is added to the message
- ❷ A custom Double property is added to the message

Now let's look at some examples of filtering messages via message selectors using the message above.

Example 2.2. Filter messages using the SYMBOL header

```
...
String selector = "SYMBOL == 'AAPL'" ;❶
MessageConsumer consumer = session.createConsumer(destination, selector);
...
```

- ❶ Select messages containing a property named SYMBOL whose value is AAPL

Example 2.2 defines a selector to filter only messages for Apple, Inc. This consumer receive only messages matching the query defined in the selector.

Example 2.3. Filter messages using both the SYMBOL and PRICE headers

```
...
String selector = "SYMBOL == 'AAPL' AND PRICE > " + getPreviousPrice() ;❶
MessageConsumer consumer = session.createConsumer(destination, selector);
...
```

- ❶ Select messages containing a property named PRICE whose value is great than the previous price

The example above specifies a selector that filters only messages for Apple, Inc. whose price is greater than the previous price. This selector will show stock messages whose price is increasing. But what if you want to take into account the timeliness of stock messages in addition to the price and symbol? Consider the next example:

Example 2.4. Filter messages using the SYMBOL header, the PRICE header

Please post comments or corrections to the [Author Online Forum](#)

and the JMSExpiration header

```
...
String selector = "SYMBOL IN ('AAPL', 'CSCO') AND PRICE > "
    + previousPrice + " AND JMSExpiration > "
    + new Date().getTime();❶
MessageConsumer consumer = session.createConsumer(destination, selector);
...
```

- ❶ Select messages containing a property named SYMBOL whose value is AAPL or CSCO, a property named PRICE whose value is great than the previous price and whose JMSExpiration is greater than the current time

The last example of message selectors in Example 2.4 defines a more complex selector that filters only messages for Apple, Inc. and Cisco Systems, Inc. whose price is increasing and which is not expired.

These examples should be enough for you to begin using message selectors. But if you want more in-depth information, see the Javadoc for the JMS `Message` type.

2.3.4.1. Message Body

JMS defines six Java types for the message body, also known as the payload. Through the use of these objects, data and information can be sent via the message payload.

- **Message** - The base message type. Used to send a message with no payload, only headers and properties. Typically used for simple event notification.
- **TextMessage** - A message whose payload is a String. Commonly used to send simple textual and XML data.
- **MapMessage** - Uses a set of name/value pairs as it's payload. The names are of type String and the values are a Java primitive type.
- **BytesMessage** - Used to contain an array of uninterpreted bytes as the

payload.

- **StreamMessage** - A message with a payload containing a stream of primitive Java types that is filled and read sequentially.
- **ObjectMessage** - Used to hold a serializable Java object as its payload. Usually used for complex Java objects. Also supports Java Collections.

2.3.5. JMS Domains

As noted earlier, the creation of JMS was a group effort and the group contained vendors of messaging implementations. It was the influence of existing messaging implementations that resulted in JMS identifying two styles of messaging; point-to-point and publish/subscribe. Most MOMs already supported both of these messaging styles so it only made sense that the JMS API support both. So let's examine each of these messaging styles to better understand them.

2.3.5.1. The Point-to-Point Domain

The point-to-point (PTP) messaging domain uses destinations known as queues. Through the use of queues, messages are sent and received either synchronously or asynchronously. Each message received on the queue is delivered to once-and-only-once consumer. This is similar to a person-to-person email sent through a mail server. Consumers receive messages from the queue either synchronously using the `MessageConsumer.receive()` method or asynchronously by registering a `MessageListener` implementation using the `MessageConsumer.setMessageListener()` method. The queue stores all messages until they are delivered or until they expire.

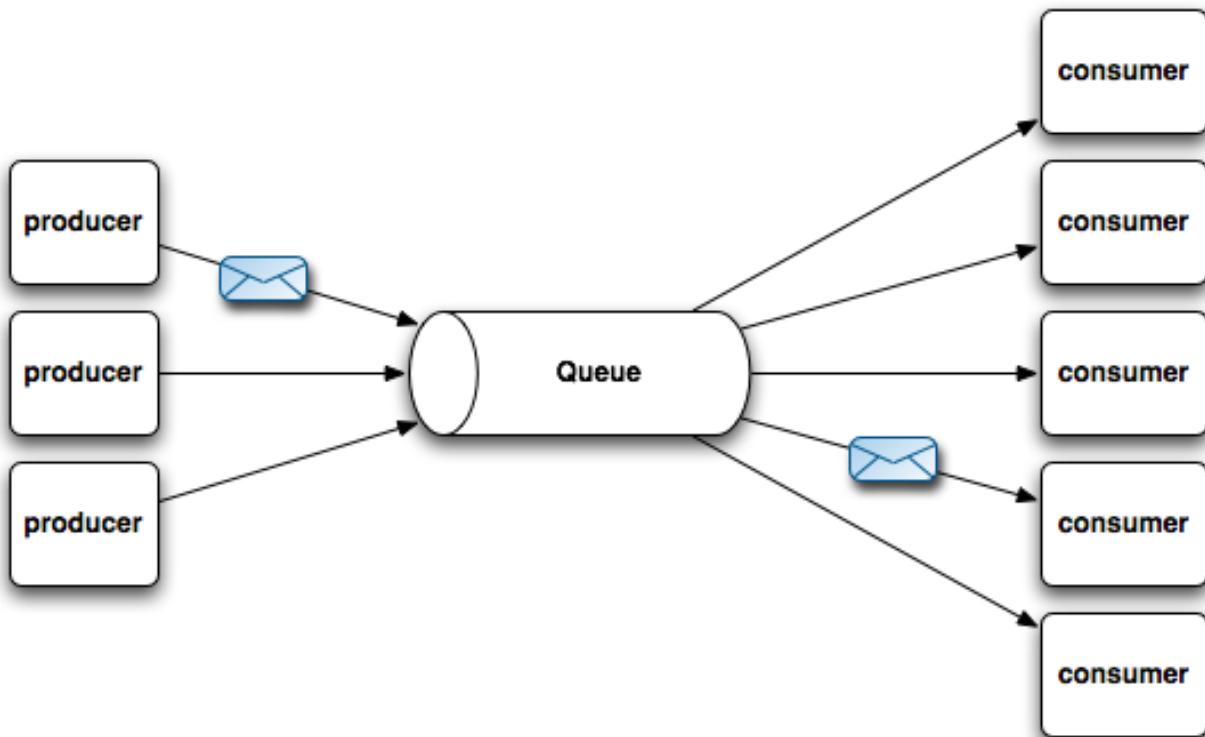


Figure 2.6. Point-to-point messaging uses a one-to-one messaging paradigm

Very much the same as PTP messaging described in the previous section, subscribers receive messages from the topic either synchronously using the `MessageConsumer.receive()` method or asynchronously by registering a `MessageListener` implementation using the `MessageConsumer.setMessageListener()` method. Multiple consumers can be registered on a single queue as shown in Figure 2.6, but only one consumer will receive a given message and then it's up to that consumer to acknowledge the message. Notice that the message in Figure 2.6 is sent from a single producer and is delivered to a single consumer, not all consumers. As mentioned above, the JMS provider guarantees the delivery of a message once-and-only-once to the next available registered consumer. In this regard, the JMS provider is doing a sort of round robin style of load-balancing of messages across all the registered consumers.

2.3.5.2. The Publish/Subscribe Domain

The publish/subscribe (pub/sub) messaging domain uses destinations known as topics. Publishers send messages to the topic and subscribers register to receive messages from the topic. Any messages sent to the topic are delivered to all subscribers via a push model, as messages are automatically delivered to the subscriber. This is similar to subscribing to a mailing list where all subscribers will receive all messages sent to the mailing list in a one-to-many paradigm. The pub/sub domain depicts this in Figure 2.7 below.

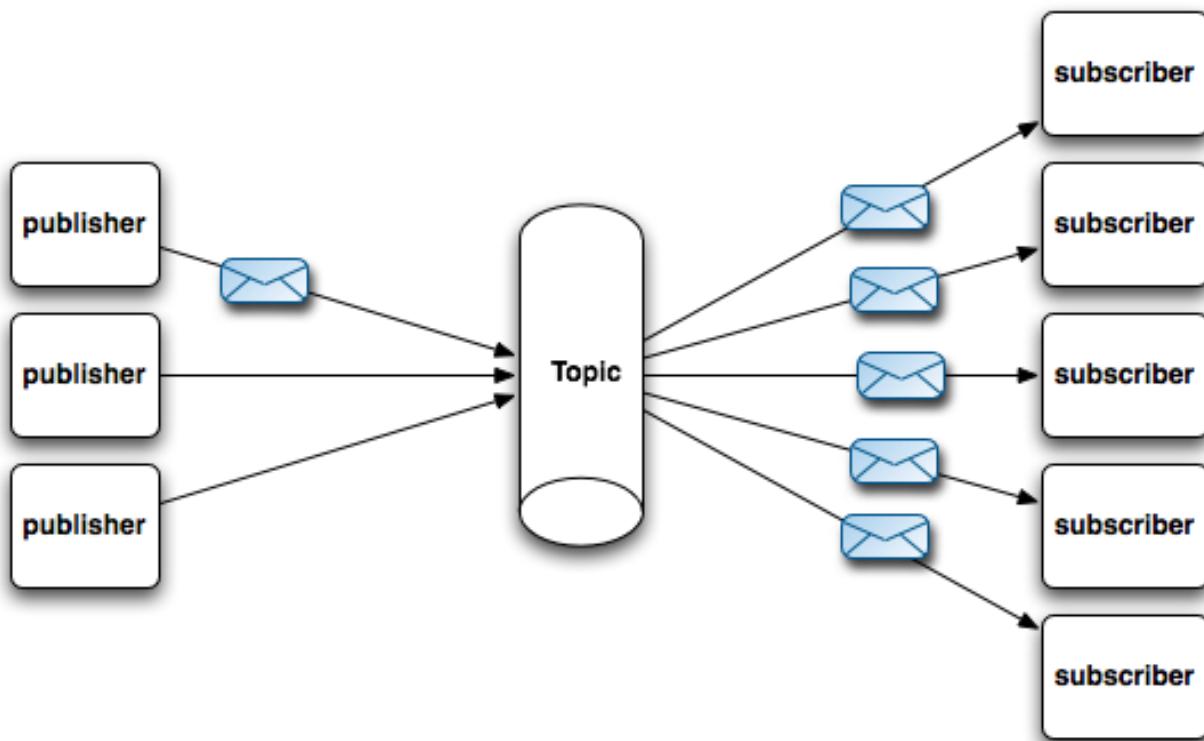


Figure 2.7. Publish/Subscribe uses a one-to-many messaging paradigm

Very much the same as PTP messaging in the previous section, subscribers register to receive messages from the topic either synchronously using the `MessageConsumer.receive()` method or asynchronously by registering a `MessageListener` implementation using the `MessageConsumer.setMessageListener()` method. Topics don't hold messages

unless it is explicitly instructed to do so. This can be achieved via the use of a durable subscription. Using a durable subscription, when a subscriber disconnects from the JMS provider, it is the responsibility of the JMS provider to store messages for the subscriber. Upon reconnecting, the durable subscriber will receive all unexpired messages from the JMS provider. Durable subscriptions allow for subscriber disconnection.

2.3.5.2.1. Distinguishing Message Durability From Message Persistence

Two points within JMS that are often confused are message durability and message persistence. Though they are similar, there are some semantic similarities though each has its specific purpose. Message durability can only be achieved with the pub/sub domain. When clients connect to a topic, they can do so using a durable or a non-durable subscription. Consider the differences of each:

- **Durable Subscription** - A durable subscription is infinite. It is registered with the topic to tell the JMS provider to preserve the subscription state in the event that the subscriber disconnects. If a subscriber disconnects, the JMS provider will hold all messages for that subscriber until it connects again. Upon reconnection, the JMS provider makes all of these messages available to the subscriber. When a durable subscriber disconnects, it is said to be inactive.
- **Non-Durable Subscription** - A non-durable subscription is finite. Its subscription state is not preserved by the JMS provider in the event that the subscriber disconnects. If a subscriber disconnects, it misses all messages during the disconnection period and the JMS provider will not hold them.

Message persistence is independent of the message domain. It is used to indicate the JMS application's ability to handle missing messages in the event of a JMS provider failure. As discussed previously, this quality of service is specified on the producer using the `JMSDeliveryMode` property using either the persistent or non-persistent property.

Creating a JMS application is not very difficult and is pretty easy to understand.

This is the next item to understand.

Request/Reply Messaging in JMS Applications

Although the JMS spec doesn't define request/reply messaging as a formal messaging domain, it does provide some message headers and a couple convenience classes for handling basic request-reply messaging.

Request-reply messaging is an asynchronous back and forth pattern utilizing either the PTP domain or the pub/sub domain through a combination of the `JMSReplyTo` and `JMSCorrelationID` message headers and temporary destinations. The `JMSReplyTo` specifies the destination where a reply should be sent and the `JMSCorrelationID` in the reply message specifies the `JMSMessageID` of the request message. These headers are used for the purposes of reference from the reply message(s) to the original request message. Temporary destinations are those that are created only for the duration of a connection and can only be consumed by the connection that created it. This style of destination can be very useful because of these restrictions are therefore very applicable to request/reply. Destinations are discussed in the next section.

The convenience classes for handling basic request/reply are the `QueueRequestor` and the `TopicRequestor`. These classes provide a `request()` method that sends a request message and waits for a reply message through the creation of a temporary destination where only one reply per request is expected. These classes are useful only for this most basic form of request/reply as shown in Figure 2.8 that is, one reply per request.

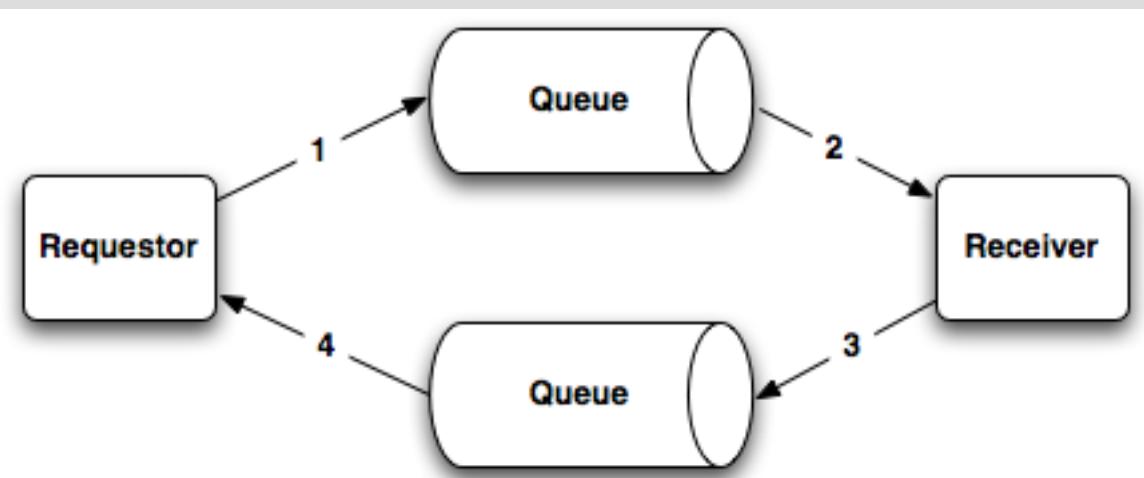


Figure 2.8. The steps involved in basic request/reply messaging

Figure 2.8, “The steps involved in basic request/reply messaging” above depicts the basic request/reply style of messaging between two endpoints. This is commonly achieved using the `JMSReplyTo` message header and a temporary queue where the reply message is sent by the receiver and consumed by the requestor. As stated previously, the `QueueRequestor` and the `TopicRequestor` can handle basic request/reply but are not designed to handle more complex cases of request/reply such as a single request and multiple replies from many receivers. Such a sophisticated use case requires a custom JMS application to be developed.

2.3.6. Administered Objects

Administered objects contain provider-specific JMS configuration information and are supposed to be created by a JMS administrator, hence the name. Administered objects are used by JMS clients. They are used to hide provider-specific details from the clients and to abstract the tasks of administration of the JMS provider. It's common to look up administered objects via JNDI, but not required. This is most common when the JMS provider is hosted in a Java EE container. The JMS spec defines two types of administered objects: `ConnectionFactory` and `Destination`.

2.3.6.1. ConnectionFactory

JMS clients use the `ConnectionFactory` object to create connections to a JMS provider. Connections represent an open TCP socket between a client and the JMS provider so the overhead for a connection is rather large. So it's a good idea to use an implementation that pools connections if possible. A connection to a JMS provider is similar to a JDBC connection to a relational database in that they are used by clients to interact with the database. JMS connections are used by JMS clients to create `javax.jms.Session` objects that represent an interaction with the JMS provider.

2.3.6.2. Destination

The `Destination` object encapsulates the provider-specific address to which messages are sent and from which messages are consumed. Although destinations are created using the `Session` object, their lifetime matches the connection from which the session was created.

Temporary destinations are unique to the connection that was used to create them. They will only live as long as the connection that created them and only the connection that created them can create consumers for them. As mentioned previously, temporary destinations are commonly used for request/reply messaging.

2.3.7. Using the JMS APIs to Create JMS Applications

JMS applications can be as simple or as complex as is necessary to situate the business requirements. Just as with other APIs such as JDBC, JNDI, EJBs,etc., it is very common to abstract the use of JMS APIs so as to not intermingle the JMS code with the business logic. This is not a concept that will be demonstrated here as this is a much lengthier exercise involving patterns and full application infrastructure. Here the simplest example will be demonstrated to show a minimalist use of the JMS APIs. A JMS application is written in the Java programming language and composed of many parts for handling different aspects of working with JMS. These parts were identified earlier in the chapter via the list

of JMS artifacts in Section 2.3 following steps:

1. Acquire a JMS connection factory
2. Create a JMS connection using the connection factory
3. Start the JMS connection
4. Create a JMS session from the connection
5. Acquire a JMS destination
6. Create a JMS producer, OR
 - a. Create a JMS producer
 - b. Create a JMS message and address it to a destination
7. Create a JMS consumer
 - a. Create a JMS consumer
 - b. Optionally register a JMS message listener
8. Send or receive JMS message(s)
9. Close all JMS resources (i.e., connection, session, producer, consumer, etc.)

The steps above are meant to be a somewhat abstract in order to demonstrate the overall simplicity of working with JMS. Using a minimal amount of code, Example 2.5 demonstrates the steps above for creating a JMS producer. Comments have been added to explain what is happening with each call.

Example 2.5. Using the JMS API to send a message

```
...
ConnectionFactory connectionFactory;
Connection connection;
Session session;
```

Please post comments or corrections to the [Author Online Forum](#)

```
Destination destination;
MessageProducer producer;
Message message;
boolean useTransaction = false;
try {
    Context ctx = new InitialContext();❶

    connectionFactory =
        (ConnectionFactory) ctx.lookup("ConnectionFactoryName");❷

    connection = connectionFactory.createConnection(); ❸

    connection.start();❹

    session = connection.createSession(useTransaction,
        Session.AUTO_ACKNOWLEDGE); ❺

    destination = session.createQueue("TEST.QUEUE");❻

    producer = session.createProducer(destination);❼

    producer.setDeliveryMode(DeliveryMode.PERSISTENT);➋

    message = session.createTextMessage("this is a test");⩿

    producer.send(message);⩾

} catch (JMSEException jmsEx) {
...
} finally {❾
    producer.close();
    session.close();
    connection.close();
}
...
```

- ❶ An initial context is usually created with a JNDI path. This one is for demonstration purposes only
- ❷ From the initial context, lookup a JMS connection factory using the unique name for the connection factory
- ❸ Using the connection factory, create a JMS connection
- ❹ Make sure to call the `start()` method on the connection factory to enable messages to start flowing
- ❺ Using the connection, create a JMS session. In this case we're just using auto-acknowledgement of messages
- ❻ Create a JMS queue using the session

- ⑦ Create a JMS producer using the session and the destination
 - ⑧ The JMS delivery mode is persistent by default, so it's being set here simply to be explicit
 - ⑨ Create a simple text message containing only a payload
 - ⑩ Using the producer, send the message to the destination
- Close all the objects that were used above

The example above in Example 2.5 demonstrates steps to create a JMS producer and send a message to a destination. Notice that there's not a concern that a JMS consumer is on the other end waiting for the message. This mediation of messages between producers and consumers is what MOMs provide and is a very big benefit when creating JMS applications. There was no special consideration to achieve this result either. The JMS APIs make this task quite simple. Now that the message has been sent to the destination, a consumer can receive the message. Example 2.6 demonstrates the steps for creating a JMS consumer and receiving the message.

Example 2.6. Using the JMS API to receive a message

```
...
ConnectionFactory connectionFactory;
Connection connection;
Session session;
Destination destination;
MessageProducer producer;
Message message;
boolean useTransaction = false;
try {
    Context ctx = new InitialContext();❶
    connectionFactory =
        (ConnectionFactory) ctx.lookup("ConnectionFactoryName");❷
    connection = connectionFactory.createConnection();❸
    connection.start(); ❹
    session = connection.createSession(useTransaction,
        Session.AUTO_ACKNOWLEDGE);❺
    destination = session.createQueue("TEST.QUEUE");❻
    consumer = session.createConsumer(destination);❼
```

Please post comments or corrections to the [Author Online Forum](#)

```
message = (TextMessage) consumer.receive(1000);❸  
  
System.out.println("Received message: " + message);❹  
  
} catch (JMSEException jmsEx) {  
...  
} finally {❺  
producer.close();  
session.close();  
connection.close();  
}  
...
```

- ❶ An initial context is usually created with a JNDI path. This one is for demonstration purposes only
- ❷ From the initial context, lookup a JMS connection factory using the unique name for the connection factory
- ❸ Using the connection factory, create a JMS connection
- ❹ Make sure to call the `start()` method on the connection factory to enable messages to start flowing
- ❺ Using the connection, create a JMS session. In this case we're just using auto-acknowledgement of messages
- ❻ Create a JMS queue using the session
- ❼ Create a JMS consumer using the session and destination
- ❽ Using the consumer, receive the message that was sent to the destination in the previous example
- ❾ Print out the message that is received
- ❿ Close all the objects that were used above

The example above in Example 2.6 is very similar to Example 2.5 because both need much of the same setup with regard to steps 1-5. Again, notice that there was no timing consideration needed to make sure that the producer is there sending a message. All mediation and temporary storage of the messagejob of the JMS provider implementation and the MOM. Consumers simply connect and receive messages.

A Note on Multithreading in JMS Applications

The JMS spec specifically defines concurrency for various objects in the JMS API and requires that only a few objects support concurrent access. The `ConnectionFactory`, `Connection` and `Destination` objects are required to support concurrent access while the `Session`, the `MessageProducer` and the `MessageConsumer` objects do not support concurrent access. The point is that the `Session`, the `MessageProducer` and the `MessageConsumer` objects should not be shared across threads in a Java application.

But consuming messages using this polling of a destination is not the only flavor of message consumption in JMS. There is another aspect to the JMS APIs for consuming messages that involves the EJB API known as message driven beans.

2.3.8. Message-Driven Beans

Message-driven beans (MDBs) were born out of the EJB 2.0 spec. The motivation was to allow very simple JMS integration into EJBs, making asynchronous message consumption by EJBs almost as easy as using the standard JMS APIs. Through the use of a JMS `MessageListener`, the EJB automatically receives messages from the JMS provider in a push style. Below is an example of a very simple MDB:

Example 2.7. A simple message-driven bean example

```
import javax.ejb.EJBException;
import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.jms.Message;
import javax.jms.MessageListener;

public class MyMessageProcessor
    implements MessageDrivenBean, MessageListener {

    public void onMessage(Message message) {
        TextMessage textMessage = null;

        try {
            if (message instanceof TextMessage) {
```

Please post comments or corrections to the [Author Online Forum](#)

```
        textMessage = (TextMessage) message;
        System.out.println("Received message: " + msg.getText());
        processMessage(textMessage);
    } else {
        System.out.println("Incorrect message type: " +
            message.getClass().getName());
    }
} catch (JMSEException jmsEx) {
    jmsEx.printStackTrace();
}
}

public void ejbRemove() throws EJBException {
    // This method is called by the EJB container
}

public void setMessageDrivenContext(MessageDrivenContext ctx)
    throws EJBException {
    // This method is called by the EJB container
}

private void processMessage(TextMessage textMessage) {
    // Do some important processing of the message here
}
}
```

Notice that the `MyMessageProcessor` class above implements both the `MessageDrivenBean` interface and the `MessageListener` interface. The `MessageDrivenBean` interface requires an implementation of the `setMessageDrivenContext()` method and the `ejbRemove()` method. Each of these methods is invoked by the EJB container for the purposes of creation and destruction of the MDB. The `MessageListener` interface contains only a single method named `onMessage()`. The `onMessage()` method is invoked automatically by the JMS provider when a message arrives in a destination on which the MDB is registered.

In addition to allowing the EJB container to manage all necessary resources including Java EE resources (such as JDBC, JMS and JCA connections), security, transactions and even JMS message acknowledgement, one of the biggest advantages of MDBs is that they can process messages concurrently. Not only do typical JMS clients need to manually manage their own resources and environment, but they are usually built for processing messages serially, that is,

one at a time (unless they're specifically built with concurrency in mind). Instead of processing messages one at a time, MDBs can process many, many more messages at the same time because the EJB container can create as many instances of the MDB as are allowed by the EJB's deployment descriptor. Such configuration is typically specific to the Java EE container. If you're using a Java EE container for this, consult the documentation for the container on how this is configured in the EJB deployment descriptor.

A disadvantage of MDBs is their requirement of a full Java EE container. Just about every EJB container available today can support MDBs only if the entire Java EE container is used. MDBs are extremely useful when using a full Java EE container, but there is an alternative that doesn't require the full Java EE container. Using the Spring Framework's JMS APIs makes developing Message Driven POJOs (MDPs) very easy. That is, Plain Old Java Objects (POJOs) that act as if they're message driven. In fact, this style of development has become quite popular in the Java development community because it avoids the overhead of using a full Java EE container. Such development with the Spring Framework will be discussed in further detail in chapter six.

Not Every EJB Container Requires a Full Java EE Container - Try OpenEJB

At the time of this writing, nearly all EJB containers on the market require a full Java EE container to support MDBs. The exception to this rule is Apache OpenEJB (<http://openejb.apache.org/>). OpenEJB supports MDBs from the EJB 1.1 spec, the EJB 2 spec and the EJB 3 spec in OpenEJB's embedded mode as well as in its stand alone mode. OpenEJB can be embedded inside of Apache Geronimo (<http://geronimo.apache.org/>), Jetty (<http://jetty.codehaus.org/>), Apache Tomcat (<http://tomcat.apache.org/>) or your own Java application and it will still provide support for MDBs.

2.4. Summary

Please post comments or corrections to the [Author Online Forum](#)

The JMS spec has had a tremendous affect on the Java world, making messaging a first-class citizen and making it available to all Java developers. This was a very important step in allowing Java to join the business of mission critical applications because it allowed a standardized manner in which to handle messaging. The examples provided in this chapter are admittedly short and simple in order to get your feet wet with JMS. As you move though the rest of the book, full examples will be discussed and made available for download.

Now that you have a basic understanding of JMS and what it provides, the next step is take a look at a JMS provider implementation. Chapter 2 provides an introduction to Apache ActiveMQ, a mature and enterprise-ready messaging implementation from the Apache Software Foundation.

Chapter 3. The ActiveMQ In Action Examples

The examples that ship as part of ActiveMQ are very good for testing the broker and provide many options. But they are more generically functional than they are specific at demonstrating particular business use cases. As mentioned in Chapter 1, *Introduction to Apache ActiveMQ*, the stock portfolio example and the job queue example will be detailed in this chapter.

In this chapter, readers will learn the following about the book examples:

- Installation and use of Maven to run the examples
- A demonstration of the publish/subscribe messaging domain with the stock portfolio example
- A demonstration of the point-to-point messaging domain with the job queue example

3.1. Understanding the Example Use Cases

The stock portfolio is a simple example of the publish/subscribe messaging domain whereby message producers called publishers broadcast stock price messages to many interested parties called subscribers. Messages are published to a JMS destination called a topic and clients with active subscriptions receive messages. Using this model, the broker pushes each message to each subscriber without them needing to poll for messages. That is, every active subscriber receives its own copy of each message published to the topic. Publishers are decoupled from subscribers via the topic. Unless durable subscriptions are used, subscribers must be active in order to receive messages sent by publishers to the topic.

The job queue is a simple example of the point-to-point messaging domain. In this example, message producers send job messages to a JMS queue from which message consumers receive the job messages for processing. There is no timing

requirement for the producers and consumers to be online at the same time with the point-to-point domain as the queue holds messages until consumers are available to receive messages. As consumers are available, messages are delivered across the consumers but no two consumers receive the same message.

Not only is each example focused around a different messaging domain, but each is also focused on a separate use case. Also, although the diagrams below for each example look nearly the same at first glance, the important difference is the use of topics for pub/sub messaging in the stock portfolio example vs. the use of queues for PTP messaging in the job queue example. The source for these examples is readily available and can be downloaded from the Manning website.

But before you get started running these examples, you will need to install either Apache Ant or Apache Maven.

3.1.1. Prerequisites

Before moving on to work with the examples, you will need to download and **install either Apache Ant or Apache Maven**, not both. This section provides some brief instructions for downloading and installing each.

3.1.1.1. Download and Install Apache Ant

If you have decided to use Apache Ant for working with the book examples, below are the steps to install Ant:

1. Download Ant from the following URL:
<http://ant.apache.org/bindownload.cgi>
Ant is provided in both tarball and zip format, depending on your operating system.
2. Expand the downloaded archive to a permanent location on your computer
3. Create an environment variable named ANT_HOME that points to the Ant directory

Please post comments or corrections to the [Author Online Forum](#)

4. On Unix, add the \$ANT_HOME/bin directory to the PATH environment variable. On Windows, add the %ANT_HOME%\bin directory to the %PATH% environment variable.
5. Verify the Ant installation. On Unix, run the following command from the command line:

```
$ ant -version  
Apache Ant version 1.7.1 compiled on June 27 2008
```

On Windows, run the following command from the command line:

```
> ant -version  
Apache Ant version 1.7.1 compiled on June 27 2008
```

You should see similar output which indicates that Ant is properly installed. If you do not see similar output, you'll need to rectify this before proceeding. See the Ant installation instructions for more information:
<http://ant.apache.org/manual/install.html>

If you have successfully installed Ant, there is no requirement to install Maven. So you can skip the next section and proceed to the begin working with the book examples.

3.1.1.2. Download and Install Apache Maven

If you have decided to use Apache Maven to work with the book examples, below are the steps to install Maven.

1. Download Maven the following URL:
<http://maven.apache.org/download.html>
Maven is provided in both tarball and zip format, depending on your operating system.
2. Expand the downloaded archive to a permanent location on your computer.
3. Create an environment variable named M2_HOME that points to the Maven

directory

4. On Unix, add the \$M2_HOME/bin directory to the PATH environment variable. On Windows, add the %M2_HOME%\bin directory to the %PATH% environment variable.
5. Verify the Maven installation. On Unix, run the following command from the command line:

```
$ mvn -version
Apache Maven 2.1.0 (r755702; 2009-03-18 13:10:27-0600)
Java version: 1.5.0_16
Java home: /System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Home
Default locale: en_US, platform encoding: MacRoman
OS name: "mac os x" version: "10.5.8" arch: "i386" Family: "unix"
```

On Windows, run the following command from the command line:

```
> mvn -version
Apache Maven 2.1.0 (r755702; 2009-03-18 13:10:27-0600)
Java version: 1.5.0_16
Java home: /System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Home
Default locale: en_US, platform encoding: MacRoman
OS name: "mac os x" version: "10.5.8" arch: "i386" Family: "unix"
```

You should see similar output which indicates that Maven is properly installed. If you do not see similar output, you'll need to rectify this before proceeding. See the Maven installation instructions for more information:
<http://maven.apache.org/download.html#Installation>

Note

Maven Needs An Internet Connection

To make use of Maven for the examples in this book, you will need a broadband connection to the Internet. This is so that Maven can connect to the necessary remote Maven repositories and download the requisite dependencies for the examples.

If you have successfully installed Maven, you can now proceed to the next section to begin working with the book examples.

3.1.2. ActiveMQ In Action Examples

The examples for this book are extremely simple. Each one only contains three classes and requires Maven for compilation. One example focuses on demonstrating JMS publish-subscribe messaging and the other example demonstrates JMS point-to-point messaging. These examples will be carried throughout the rest of the book and expanded to demonstrate many features and functionality in ActiveMQ, so it's important to understand the very basics of each one now.

3.1.2.1. Downloading the Examples

The example source code can be downloaded from the book website at the following URL:

<http://manning.com/snyder/AMQinA-src.zip>

After downloading this zip file, expand it and proceed to the next section.

3.1.2.2. Compiling the Examples

Before using the examples from the book, they must first be compiled. To compile the examples, move into the

3.1.2.3. Use Case One: The Stock Portfolio Example

As mentioned earlier in the chapter, the first use case revolves around a stock portfolio use case for demonstrating publish-subscribe messaging. This example is very simple and utilizes a Publisher class for sending stock price messages to a topic as well as a Consumer class for registering a Listener class to consume messages from topics in an asynchronous manner. These three classes embody the functionality of a generating ever-changing stock prices which are published to topics on which the consumer is subscribed.

In this example, stock prices are published to an arbitrary number of topics. The number of topics is based on the number of arguments sent to the Publisher and the Consumer on the command line. Each class will dynamically send and receive

from the number of queues on the command line (an example is provided below). First, take a look at ??? to see at a high level what the example seeks to achieve.

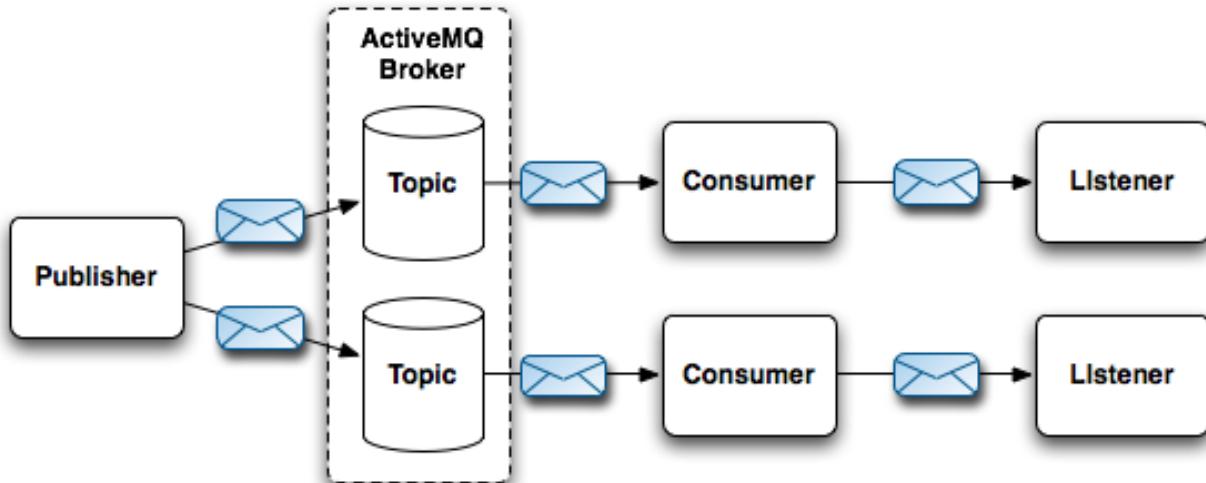


Figure 3.1. The stock portfolio example

For the sake of this demonstration, two topics will be used. The Publisher class uses a single JMS `MessageProducer` to send 1000 fictitious stock price messages in blocks of 10 randomly across the topics named in the command line argument. After it sends 1000 messages it shuts down. The Consumer class creates one JMS `MessageConsumer` per topic and registers a JMS `MessageListener` for each topic. Because this example demonstrates publish-subscribe, the Consumers must be online to consume messages being sent by the Publisher because durable messages are not used in the basic stock portfolio example. The next step is to actually run the example so that you can see them in action.

3.1.2.4. Running the Stock Portfolio Example

There are three basic steps to running this example including:

1. Start up ActiveMQ
2. Run the Consumer class

3. Run the Publisher class

These steps appear to be very simple and they are. The only item of note is that the Consumer should be started before the Publisher in order to receive all messages that are published. This is because this example demonstrates pub/sub messaging and topics will not hold messages unless the consumer makes a durable subscription and we're not using durable subscriptions here. So let's get started with the stock portfolio example.

The first task is to open a terminal or command line and execute ActiveMQ. This only requires a single command as demonstrated below:

```
[apache-activemq-5.1.0]$ ./bin/activemq
ACTIVEMQ_HOME: /Users/bsnyder/amq/apache-activemq-5.1.0
ACTIVEMQ_BASE: /Users/bsnyder/amq/apache-activemq-5.1.0
Loading message broker from: xbean:activemq.xml
INFO BrokerService - Using Persistence Adapter: AMQPersistenceAdapter
(/Users/bsnyder/amq/apache-activemq-5.1.0/data)
INFO BrokerService - ActiveMQ 5.1.0 JMS Message Broker (localhost) is started
INFO BrokerService - For help or more information please see:
http://activemq.apache.org/
INFO AMQPersistenceAdapter - AMQStore starting using directory:
/Users/bsnyder/amq/apache-activemq-5.1.0/data
INFO KahaStore - Kaha Store using data directory
/Users/bsnyder/amq/apache-activemq-5.1.0/data/kr-store/state
INFO AMQPersistenceAdapter - Active data files: []
INFO KahaStore - Kaha Store using data directory
/Users/bsnyder/amq/apache-activemq-5.1.0/data/kr-store/data
INFO TransportServerThreadSupport - Listening for connections at: tcp://mongoose.local:6161
INFO TransportConnector - Connector openwire Started
INFO TransportServerThreadSupport - Listening for connections at: ssl://mongoose.local:6161
INFO TransportConnector - Connector ssl Started
INFO TransportServerThreadSupport - Listening for connections at: stomp://mongoose.local:6161
INFO TransportConnector - Connector stomp Started
INFO TransportServerThreadSupport - Listening for connections at: xmpp://mongoose.local:6161
INFO TransportConnector - Connector xmpp Started
INFO NetworkConnector - Network Connector default-nc Started
INFO BrokerService - ActiveMQ JMS Message Broker (localhost,
ID:mongoose.local-61751-1223357347308-0:0) started
INFO log - Logging to org.slf4j.impl.JCLLoggerAdapter(org.mortbay.log.Slf4jLog)
INFO log - jetty-6.1.9
INFO WebConsoleStarter - ActiveMQ WebConsole initialized.
INFO /admin - Initializing Spring FrameworkServlet 'dispatcher'
INFO log - ActiveMQ Console at http://0.0.0.0:8161/admin
INFO log - ActiveMQ Web Demos at http://0.0.0.0:8161/demo
INFO log - RESTful file access application at http://0.0.0.0:8161
INFO log - Started SelectChannelConnector@0.0.0.0:8161
INFO FailoverTransport - Successfully connected to tcp://localhost:6161
```

Please post comments or corrections to the [Author Online Forum](#)

The next task is to open a second terminal or command line to execute the Consumer class. The Consumer is executed using the [maven-exec-plugin](#) by passing it some system properties as arguments to the maven-exec-plugin using the exec.args property. Below is the command to execute the Consumer:

```
[trunk]$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch2.portfolio.Consumer \
-Dexec.args="CSCO ORCL"
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'exec'.
[INFO] org.apache.maven.plugins: checking for updates from central
[INFO] org.codehaus.mojo: checking for updates from central
[INFO] artifact org.codehaus.mojo:exec-maven-plugin: checking for updates from central
Downloading: http://localhost:8081/nexus/content/groups/public/org/codehaus/mojo/exec-maven-
exec-maven-plugin-1.1.pom
3K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/codehaus/mojo/mojo/17/mo-
16K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/codehaus/mojo/exec-maven-
exec-maven-plugin-1.1.jar
26K downloaded
[INFO] -----
[INFO] Building amqinaction
[INFO]   task-segment: [exec:java]
[INFO] -----
[INFO] Preparing exec:java
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/plugins/
maven-compiler-plugin/2.0.2/maven-compiler-plugin-2.0.2.pom
2K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/plugins/mave-
maven-plugins-8.pom
5K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven-parent-
maven-parent-5.pom
14K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/apache/3/apache-3
3K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/plugins/
maven-compiler-plugin/2.0.2/maven-compiler-plugin-2.0.2.jar
17K downloaded
[INFO] No goals needed for project - skipping
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/activemq/activemq-
activemq-all-5.1.0.pom
2K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/activemq/activemq-
activemq-parent-5.1.0.pom
43K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/apache/4/apache-4
4K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/javax/transaction/jta/1.0.1B-
515b downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/springframework/spring/2
```

Please post comments or corrections to the [Author Online Forum](#)

```
spring-2.5.1.pom  
12K downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/xbean/xbean-spring/xbean-spring-3.3.pom  
4K downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/xbean/xbean/3.3/xbean-3.3.pom  
19K downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/commons-logging/commons-logging/commons-logging-1.0.3.pom  
866b downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/junit/junit/3.8.1/junit-3.8.1.jar  
998b downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/log4j/log4j/1.2.14/log4j-1.2.14.jar  
2K downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/javax/xml/stream/stax-api/1.0.2/stax-api-1.0-2.pom  
167b downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/activemq/activemq-activemq-all/5.1.0/activemq-all-5.1.0.jar  
3705K downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/javax/transaction/jta/1.0.1B/jta-1.0.1B.jar  
8K downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/org/springframework/spring/2.5.1/spring-2.5.1.jar  
2766K downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/xbean/xbean-spring/xbean-spring-3.3.jar  
125K downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/log4j/log4j/1.2.14/log4j-1.2.14.jar  
358K downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/javax/xml/stream/stax-api/1.0.2/stax-api-1.0-2.jar  
22K downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven-project/2.0.4/maven-project-2.0.4.pom  
1K downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven/2.0.4/maven-2.0.4.pom  
11K downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven-settings/2.0.4/maven-settings-2.0.4.pom  
1K downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven-model/2.0.4/maven-model-2.0.4.pom  
2K downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/org/codehaus/plexus/plexus-utils/1.1/plexus-utils-1.1.pom  
767b downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/org/codehaus/plexus/plexus/plexus-1.0.4/plexus-1.0.4.pom  
5K downloaded  
Downloading: http://localhost:8081/nexus/content/groups/public/org/codehaus/plexus/plexus-container-default/1.0-alpha-9/plexus-container-default-1.0-alpha-9.pom
```

Please post comments or corrections to the [Author Online Forum](#)

```
1K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/codehaus/plexus/plexus-co
1.0.3/plexus-containers-1.0.3.pom
492b downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/junit/junit/3.8.2/junit-3.8.2
747b downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven-profile
maven-profile-2.0.4.pom
1K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven-artifa
2.0.4/maven-artifact-manager-2.0.4.pom
1K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven-repos
2.0.4/maven-repository-metadata-2.0.4.pom
1K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven-artifa
maven-artifact-2.0.4.pom
765b downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/wagon/wagon-
1.0-alpha-6/wagon-provider-api-1.0-alpha-6.pom
588b downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/codehaus/plexus/plexus-u
plexus-utils-1.5.1.pom
2K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/codehaus/plexus/plexus-u
plexus-utils-1.5.1.jar
205K downloaded
[INFO] [exec:java]
```

You can see in the output above that Maven downloads the necessary artifacts it needs to run the examples. Once this has completed, the Publisher starts up and begins publishing stock prices to the two topics named on the command line, *CSCO* and *ORCL*. These two topic names were picked at random and can be replaced with any Strings you desire. The important part is that the same arguments be used for both the Consumer and the Publisher (the Publisher is shown next) via the system property named `exec.args`.

Note

BUILD ERRORS When Running the Consumer

If you receive a BUILD ERROR while attempting to run the consumer class above, you will need to compile the source code before running it.

To compile all the source, run the command below:

```
$ mvn clean install
```

This command will compile and package the source so that it's ready to be run. After this command completes, you can go back and run the command consumer using the command above

Notice that the output above just seems to stop as the Consumer hangs there. This behavior is correct because it's waiting for messages to arrive in the topics to be consumed. When the Publisher begins sending messages, the Consumer will begin to consume them.

Why Are All the Artifacts Being Downloaded From the localhost in the Output Shown Above?

As long as Maven was set up correctly in Section 3.1.1.2, “Download and Install Apache Maven”, then Maven will download all the necessary artifacts it needs to run the examples. You can see it downloading artifacts in the first portion of the output above. Notice that all the artifacts are being downloaded from the localhost using Nexus. This is because I'm running a Maven repository manager named Nexus on my machine. More information about Nexus is available here:

<http://nexus.sonatype.org/>

The next task is to open a third terminal or command line to execute the Publisher class. Notice that the same arguments are used in `exec.args` that were used for executing the Consumer class above because the maven-exec-plugin is used to execute the Publisher class as well. Below is the command to execute the Publisher:

```
[trunk]$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch2.portfolio.Publisher \
-Dexec.args="CSCO ORCL"
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'exec'.
```

Please post comments or corrections to the [Author Online Forum](#)

```
[INFO] -----  
[INFO] Building amqinaction  
[INFO]   task-segment: [exec:java]  
[INFO] -----  
[INFO] Preparing exec:java  
[INFO] No goals needed for project - skipping  
[INFO] [exec:java]  
Sending: {offer=58.31986582023755, price=58.26160421602154, up=true, stock=ORCL} on destination topic://STOCKS.ORCL  
Sending: {offer=58.46614894941039, price=58.4077412082022, up=true, stock=ORCL} on destination topic://STOCKS.ORCL  
Sending: {offer=58.718323292827435, price=58.65966362919824, up=true, stock=ORCL} on destination topic://STOCKS.ORCL  
Sending: {offer=58.345627177563735, price=58.287339837726016, up=false, stock=ORCL} on destination topic://STOCKS.ORCL  
Sending: {offer=36.37572048021212, price=36.339381099113005, up=true, stock=CSCO} on destination topic://STOCKS.CSCO  
Sending: {offer=36.57877346104115, price=36.54223122981134, up=true, stock=CSCO} on destination topic://STOCKS.CSCO  
Sending: {offer=58.2126392468028, price=58.15448476204077, up=false, stock=ORCL} on destination topic://STOCKS.ORCL  
Sending: {offer=36.695153690818174, price=36.65849519562256, up=true, stock=CSCO} on destination topic://STOCKS.CSCO  
Sending: {offer=36.68585049103857, price=36.649201289748824, up=false, stock=CSCO} on destination topic://STOCKS.CSCO  
Sending: {offer=58.153551810338584, price=58.09545635398461, up=false, stock=ORCL} on destination topic://STOCKS.ORCL  
Published '10' of '10' price messages  
Sending: {offer=36.4966203873684, price=36.460160227141266, up=false, stock=CSCO} on destination topic://STOCKS.CSCO  
Sending: {offer=58.045779233365835, price=57.98779144192392, up=false, stock=ORCL} on destination topic://STOCKS.ORCL  
Sending: {offer=36.843532042734964, price=36.80672531741755, up=true, stock=CSCO} on destination topic://STOCKS.CSCO  
Sending: {offer=36.954536453437285, price=36.917618834602685, up=true, stock=CSCO} on destination topic://STOCKS.CSCO  
Sending: {offer=58.49426529786688, price=58.43582946839849, up=true, stock=ORCL} on destination topic://STOCKS.ORCL  
Sending: {offer=58.594484920787735, price=58.53594897181593, up=true, stock=ORCL} on destination topic://STOCKS.ORCL  
Sending: {offer=59.01185791171931, price=58.952905006712605, up=true, stock=ORCL} on destination topic://STOCKS.ORCL  
Sending: {offer=37.10996253306843, price=37.072889643425015, up=true, stock=CSCO} on destination topic://STOCKS.CSCO  
Sending: {offer=37.10871702959351, price=37.071645384209305, up=false, stock=CSCO} on destination topic://STOCKS.CSCO  
Sending: {offer=59.27734586883228, price=59.218127741091195, up=true, stock=ORCL} on destination topic://STOCKS.ORCL  
Published '10' of '20' price messages  
Sending: {offer=58.79092604485173, price=58.73219385100074, up=false, stock=ORCL} on destination topic://STOCKS.ORCL  
Sending: {offer=59.25190817806627, price=59.19271546260367, up=true, stock=ORCL} on destination topic://STOCKS.ORCL
```

Please post comments or corrections to the [Author Online Forum](#)

```

Sending: {offer=37.26677291595445, price=37.22954337258187, up=true, stock=CSCO} on destination topic://STOCKS.CSCO
Sending: {offer=58.939385104126835, price=58.88050459952731, up=false, stock=ORCL} on destination topic://STOCKS.ORCL
Sending: {offer=37.39307124450011, price=37.355715528971146, up=true, stock=CSCO} on destination topic://STOCKS.CSCO
Sending: {offer=37.11888897612781, price=37.08180716895886, up=false, stock=CSCO} on destination topic://STOCKS.CSCO
Sending: {offer=37.34651626026739, price=37.30920705321418, up=true, stock=CSCO} on destination topic://STOCKS.CSCO
Sending: {offer=59.3224657152465, price=59.26320251273378, up=true, stock=ORCL} on destination topic://STOCKS.ORCL
Sending: {offer=37.416557967252466, price=37.379178788464, up=true, stock=CSCO} on destination topic://STOCKS.CSCO
Sending: {offer=59.0915505887881, price=59.032518070717394, up=false, stock=ORCL} on destination topic://STOCKS.ORCL
Published '10' of '30' price messages
...

```

When executing the Publisher class, Maven already has all the necessary dependencies from the execution of the Consumer class previously. The lower portion of the output above shows the stock price messages being sent to the two topics in blocks of 10. The example output is truncated for space, so just know that the Publisher will run until it sends a total of 1000 messages.

After running the Publisher, if you switch back to the second terminal where the Consumer was started, you should see that it's now consuming messages from the topics.

```

[INFO] [exec:java]
ORCL      58.26    58.32    up
ORCL      58.41    58.47    up
ORCL      58.66    58.72    up
ORCL      58.29    58.35    down
CSCO      36.34    36.38    up
CSCO      36.54    36.58    up
ORCL      58.15    58.21    down
CSCO      36.66    36.70    up
CSCO      36.65    36.69    down
ORCL      58.10    58.15    down
CSCO      36.46    36.50    down
ORCL      57.99    58.05    down
CSCO      36.81    36.84    up
CSCO      36.92    36.95    up
ORCL      58.44    58.49    up
ORCL      58.54    58.59    up
ORCL      58.95    59.01    up
CSCO      37.07    37.11    up
CSCO      37.07    37.11    down

```

Please post comments or corrections to the [Author Online Forum](#)

```
ORCL      59.22   59.28   up
ORCL      58.73   58.79   down
ORCL      59.19   59.25   up
CSCO      37.23   37.27   up
ORCL      58.88   58.94   down
CSCO      37.36   37.39   up
CSCO      37.08   37.12   down
CSCO      37.31   37.35   up
ORCL      59.26   59.32   up
CSCO      37.38   37.42   up
ORCL      59.03   59.09   down
ORCL      59.12   59.18   up
CSCO      37.51   37.55   up
ORCL      59.20   59.26   up
ORCL      59.40   59.46   up
ORCL      59.76   59.82   up
ORCL      59.85   59.91   up
ORCL      59.26   59.32   down
CSCO      37.54   37.58   up
ORCL      58.90   58.96   down
...
...
```

The output above comes from the Listener class that is registered by the Consumer on the two topics named ORCL and CSCO. This output shows the consumption of the stock price messages from the same two topics to which the Publisher is sending messages. Once the Publisher reaches 1000 messages sent, it will shut down. But the Consumer will continue to run and just hang there waiting for more messages to arrive on those two topics. You can simply type a CTRL-C in the second terminal to shut down the Consumer at this point.

That completes the demonstration of the stock portfolio example. The next example centers on point-to-point messaging.

3.1.2.5. Use Case Two: The Job Queue Example

The second use case focuses on job queues to illustrate point-to-point messaging. This example makes use of a Producer class to send job messages to a job queue and a Consumer class for registering a Listener class to consume messages from queues in an asynchronous manner. These three classes provide the functionality necessary to appropriately show how JMS point-to-point messaging should work. The classes in this example are extremely similar to those used in the stock portfolio example. The difference between the two examples is the JMS messaging

domain that each one uses.

The Producer class in this example sends messages to an arbitrary number of queues (again, based on the command line arguments) and the Consumer class consumes. ??? contains a high level diagram of the job queue example's functionality.

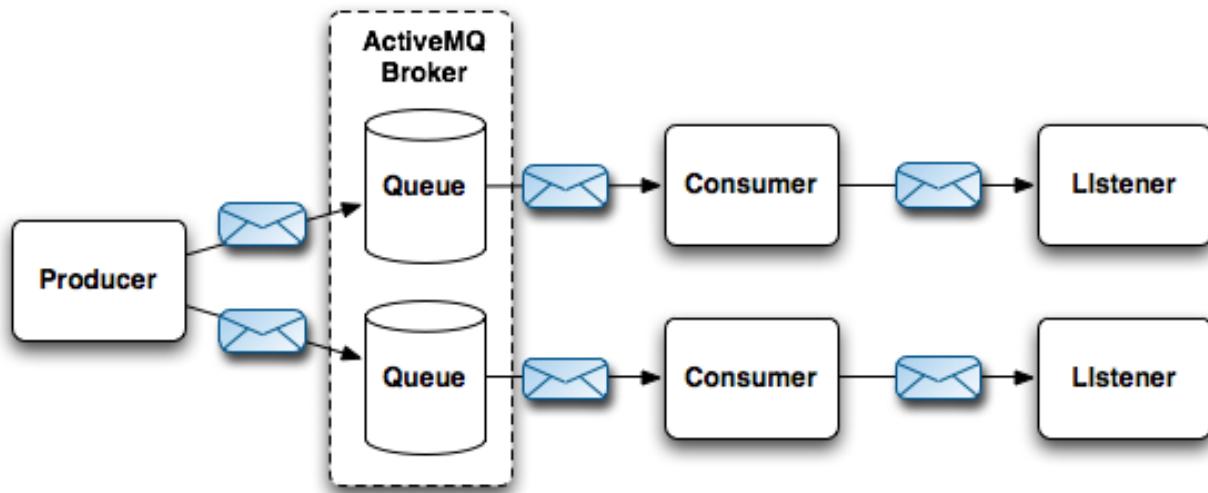


Figure 3.2. The job queue example

The same two queues will be used for this example that were used in the stock portfolio example. The Producer class uses a single JMS `MessageProducer` to send 1000 job messages in blocks of 10 randomly across the two queues. After sending 1000 messages total, it will shut down. The Consumer class uses one JMS `MessageConsumer` per queue and registers a JMS `MessageListener` on each queue to actually utilize the message and output its contents.

3.1.2.6. Running the Job Queues Example

The steps for executing the job queues example is nearly identical to the previous example including:

1. Start up ActiveMQ
2. Run the Producer class

3. Run the Consumer class

Again, these steps are very simple, but there is one exception to note. When using PTP messaging, queues will hold messages until they're consumed or the messages expire. So the Producer can be started before the Consumer and the Consumer will not miss any messages.

Just as in the stock portfolio example, the first task is to open a terminal or command line and start up ActiveMQ using the command shown below:

```
[apache-activemq-5.1.0]$ ./bin/activemq
ACTIVEMQ_HOME: /Users/bsnyder/amq/apache-activemq-5.1.0
ACTIVEMQ_BASE: /Users/bsnyder/amq/apache-activemq-5.1.0
Loading message broker from: xbean:activemq.xml
INFO BrokerService - Using Persistence Adapter: AMQPersistenceAdapter
(/Users/bsnyder/amq/apache-activemq-5.1.0/data)
INFO BrokerService - ActiveMQ 5.1.0 JMS Message Broker (localhost) is starting
INFO BrokerService - For help or more information please see:
http://activemq.apache.org/
INFO AMQPersistenceAdapter - AMQStore starting using directory:
/Users/bsnyder/amq/apache-activemq-5.1.0/data
INFO KahaStore - Kaha Store using data directory
/Users/bsnyder/amq/apache-activemq-5.1.0/data/kr-store/state
INFO AMQPersistenceAdapter - Active data files: []
INFO KahaStore - Kaha Store using data directory
/Users/bsnyder/amq/apache-activemq-5.1.0/data/kr-store/data
INFO TransportServerThreadSupport - Listening for connections at: tcp://mongoose.local:61616
INFO TransportConnector - Connector openwire Started
INFO TransportServerThreadSupport - Listening for connections at: ssl://mongoose.local:61617
INFO TransportConnector - Connector ssl Started
INFO TransportServerThreadSupport - Listening for connections at: stomp://mongoose.local:61613
INFO TransportConnector - Connector stomp Started
INFO TransportServerThreadSupport - Listening for connections at: xmpp://mongoose.local:61615
INFO TransportConnector - Connector xmpp Started
INFO NetworkConnector - Network Connector default-nc Started
INFO BrokerService - ActiveMQ JMS Message Broker (localhost,
ID:mongoose.local-61751-1223357347308-0:0) started
INFO log - Logging to org.slf4j.impl.JCLLoggerAdapter(org.mortbay.log.Slf4jLog)
via org.mortbay.log.Slf4jLog
INFO log - jetty-6.1.9
INFO WebConsoleStarter - ActiveMQ WebConsole initialized.
INFO /admin - Initializing Spring FrameworkServlet 'dispatcher'
INFO log - ActiveMQ Console at http://0.0.0.0:8161/admin
INFO log - ActiveMQ Web Demos at http://0.0.0.0:8161/demo
INFO log - RESTful file access application at http://0.0.0.0:8161/file
INFO log - Started SelectChannelConnector@0.0.0.0:8161
INFO FailoverTransport - Successfully connected to tcp://localhost:61616
```

Again, this is the same output as shown in Section 1.2.2, “Download ActiveMQ”

above and none of the default configuration has been changed.

Next, open a second terminal or command line to execute the Producer using the maven-exec-plugin as shown below:

```
[trunk]$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch2.jobs.Publisher
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'exec'.
[INFO] -----
[INFO] Building amqinaction
[INFO]   task-segment: [exec:java]
[INFO] -----
[INFO] Preparing exec:java
[INFO] No goals needed for project - skipping
[INFO] [exec:java]
Sending: id: 1000000 on queue: queue://JOBS.delete
Sending: id: 1000001 on queue: queue://JOBS.delete
Sending: id: 1000002 on queue: queue://JOBS.delete
Sending: id: 1000003 on queue: queue://JOBS.suspend
Sending: id: 1000004 on queue: queue://JOBS.suspend
Sending: id: 1000005 on queue: queue://JOBS.delete
Sending: id: 1000006 on queue: queue://JOBS.delete
Sending: id: 1000007 on queue: queue://JOBS.suspend
Sending: id: 1000008 on queue: queue://JOBS.delete
Sending: id: 1000009 on queue: queue://JOBS.suspend
Sent '10' of '10' job messages
Sending: id: 1000010 on queue: queue://JOBS.delete
Sending: id: 1000011 on queue: queue://JOBS.delete
Sending: id: 1000012 on queue: queue://JOBS.suspend
Sending: id: 1000013 on queue: queue://JOBS.delete
Sending: id: 1000014 on queue: queue://JOBS.delete
Sending: id: 1000015 on queue: queue://JOBS.delete
Sending: id: 1000016 on queue: queue://JOBS.delete
Sending: id: 1000017 on queue: queue://JOBS.suspend
Sending: id: 1000018 on queue: queue://JOBS.delete
Sending: id: 1000019 on queue: queue://JOBS.delete
Sent '10' of '20' job messages
Sending: id: 1000020 on queue: queue://JOBS.suspend
Sending: id: 1000021 on queue: queue://JOBS.suspend
Sending: id: 1000022 on queue: queue://JOBS.delete
Sending: id: 1000023 on queue: queue://JOBS.delete
Sending: id: 1000024 on queue: queue://JOBS.suspend
Sending: id: 1000025 on queue: queue://JOBS.delete
Sending: id: 1000026 on queue: queue://JOBS.delete
Sending: id: 1000027 on queue: queue://JOBS.delete
Sending: id: 1000028 on queue: queue://JOBS.suspend
Sending: id: 1000029 on queue: queue://JOBS.suspend
Sent '10' of '30' job messages
...
```

Notice that no arguments are necessary to execute the Producer. The Publisher

class contains two queues to which it publishes named *delete* and *suspend*, hence the use of those words in the output. The Producer will continue until it sends a total of 1000 messages to the two queues and then it will shut down.

The third task is to open another terminal or command line and execute the Consumer to consume the messages from the two queues. This command is shown below:

```
[trunk]$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch2.jobs.Consumer -Dexec.args=-DmqbeanName=amqinaction
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'exec'.
[INFO] -----
[INFO] Building amqinaction
[INFO]   task-segment: [exec:java]
[INFO] -----
[INFO] Preparing exec:java
[INFO] No goals needed for project - skipping
[INFO] [exec:java]
delete id:1000000
delete id:1000001
delete id:1000002
delete id:1000005
delete id:1000006
delete id:1000008
suspend id:1000003
suspend id:1000004
suspend id:1000007
suspend id:1000009
delete id:1000010
delete id:1000011
suspend id:1000012
delete id:1000013
delete id:1000014
delete id:1000015
delete id:1000016
suspend id:1000017
delete id:1000018
delete id:1000019
suspend id:1000020
suspend id:1000021
delete id:1000022
delete id:1000023
suspend id:1000024
delete id:1000025
delete id:1000026
delete id:1000027
suspend id:1000028
suspend id:1000029
delete id:1000030
delete id:1000031
delete id:1000032
```

Please post comments or corrections to the [Author Online Forum](#)

```
delete id:1000033
delete id:1000034
delete id:1000035
suspend id:1000036
delete id:1000037
delete id:1000038
delete id:1000039
delete id:1000040
delete id:1000041
...
```

The Consumer will run very fast at first, consuming all the messages already on the queues. When it catches up to where the Producer is in sending the 1000 messages, the Consumer slows down and keeps up with the Publisher until it completes. When all the messages have been sent and the Producer shuts itself down, you'll need to type a CTRL-C in the third terminal where the Consumer is running to shut it down.

3.2. Summary

[Summary paragraph goes here]

Part II. How to Configure ActiveMQ

Out of the box, so to speak, using ActiveMQ is quite straightforward; start it up, send some messages, receive some messages. But more complex situations require more advanced configuration options. Although ActiveMQ provides many configuration options, there is a core set of these options that is necessary to understand for the most basic of broker configurations.

Part II dives into the critical configuration options in ActiveMQ and helps you to quickly understand some of the possibilities with these options.

Chapter 4. Connecting to ActiveMQ

The main role of a JMS broker such as ActiveMQ is to provide a communication infrastructure for client applications. For that reason, ActiveMQ provides *connectors*, a connectivity mechanism that provides client-to-broker communications as well as broker-to-broker communications. ActiveMQ allows client applications to connect using a variety of protocols, but also others brokers to create communication channels and to build complex networks of ActiveMQ brokers.

In this chapter, we will explain such connectivity concepts as:

- *Connector URIs*, that make it possible to address brokers
- *Transport connectors*, which are used to expose brokers to clients
- *Network connectors*, which are used to create networks of brokers
- *Discovery Agents*, that allow the discovery of brokers in a cluster
- This chapter will also explain in details some of the most commonly used connectors

The details provided in this chapter will help you understand the transports used to connect to ActiveMQ brokers and choose the right setup for your needs.

4.1. Understanding Connector URIs

Before the details of connectors and their role in the overall ActiveMQ architecture are discussed, it is important to understand *connector URIs*. These connector URIs are a standardized way of addressing connectors in ActiveMQ that follow a quiet but widely used standard.

Uniform Resource Identifiers (URIs), as a concept, are not new and you have probably used it over and over again without realizing it. URIs were first introduced for addressing resources on the World Wide Web. The specification

<http://www.ietf.org/rfc/rfc2396.txt>) defines the URI as “*a compact string of characters for identifying an abstract or physical resource*”. Because of the simplicity and flexibility of the URI concept, they found their place in numerous Internet services. For example, the web URLs and email addresses we use every day are just some of the examples of URIs in practice today.

Without going too deep into discussing URIs, let's just briefly summarize the URI structure. It will be an ideal introduction to URI usage in ActiveMQ and how they are used with connectors.

Basically, every URI has the following string format:

```
<scheme>:<scheme-specific-part>
```

Consider the following URI:

```
mailto:users@activemq.apache.org
```

Notice that the `mailto` scheme is used, followed by an email address to uniquely identify both the service we are going to use and the particular resource within that service.

The most common form of URIs are hierarchical URIs, which takes the following form:

```
<scheme>://<authority><path><?query>
```

These kind of URIs are usually found on the Web, so for example the following web URL:

```
http://www.nabble.com/forum/NewTopic.jtp?forum=2356
```

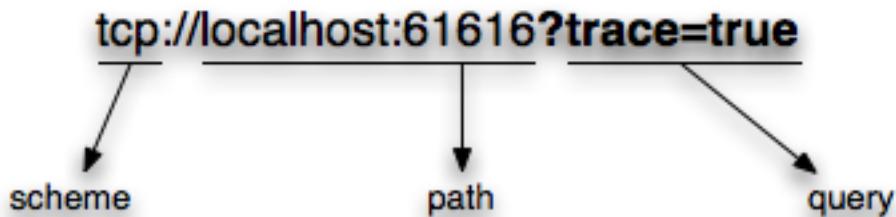
The URL above uses the `http` scheme and contains both `path` and `query` elements (query elements are used to specify additional parameters).

Because of their flexibility and simplicity, URIs are used in ActiveMQ to address specific brokers through different type of connectors. If we go back to the examples discussed in Chapter 2 you can see that the following URI was used to create a connection to the broker:

```
tcp://localhost:61616
```

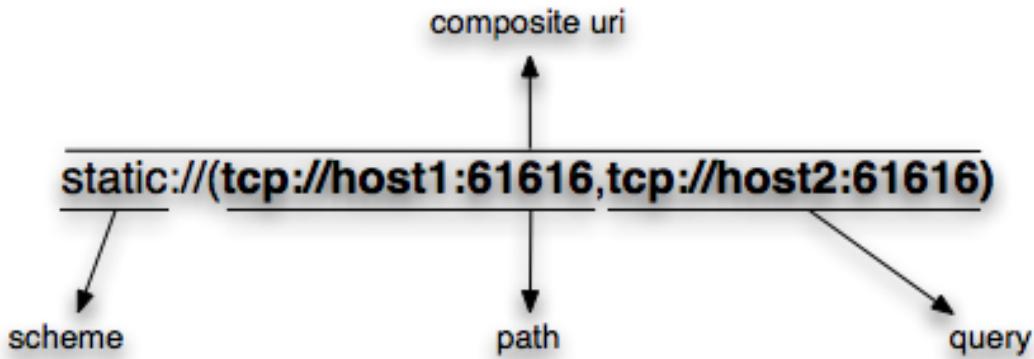
This is a typical hierarchical URI used in ActiveMQ which translates to "*create a TCP connection to localhost on port 61616*".

ActiveMQ connectors using this kind of simple hierarchical URI pattern are called *low-level connectors* and they are used to implement basic network communication protocols. Low-level protocol URIs use the scheme part to identify the underlying network protocol, the path element to identify a network resource (usually host and port) and the query element to specify additional configuration parameters for the connector. Here is a breakdown of the anatomy of a URI:



This URI extends the previous example by also telling the broker to log all commands sent over this connector (the `trace=true` part). This is just one example of an option that is available on the TCP transport.

The TCP transport in ActiveMQ supports automatic reconnection as well as the ability to connect to another broker just in case the broker to which a client is currently connected becomes unavailable. As will be discussed in Chapter 9, ActiveMQ makes this easy to use and configure through the use of *composite URIs*. These composite URIs are used to configure such automatic reconnection. In the following snippet you can see an example of a typical composite URI:



Notice that the scheme part or the URI now identifies the protocol being used (the static protocol will be described later in this chapter) and the scheme specific part contains one or more low-level URIs that will be used to create a connection. Of course, every low-level URI and the larger composite URI can contain the query part providing specific configuration options for the particular connector.

Note

Since composite URIs tend to be complex, users are often tempted to insert white spaces to make them more readable. Such white space is not allowed since the URI specification (and its standard Java implementation) does not allow it. This is a common ActiveMQ configuration mistakes, so be careful not to put white space in your URIs.

Now that you have some familiarity with ActiveMQ URI basics, let's move on to discuss the connector types supported by ActiveMQ. In the rest of this chapter, we will discuss transport connectors and network connectors and how to configure them.

4.2. Configuring Transport Connectors

As we will see in this and chapters that follows, you have a wide range of protocols available for client connectivity. This client-to-broker communication is performed through, so-called, *transport connectors*.

From the broker's perspective, the transport connector is a mechanism used to accept and listen to connections from clients. If you take a look at the default ActiveMQ configuration file (`conf/activemq.xml`), you will see the configuration snippet for transport connectors similar to the following example:

```
<!-- The transport connectors ActiveMQ will listen to -->
<transportConnectors>
    <transportConnector name="openwire" uri="tcp://localhost:61616"
        discoveryUri="multicast://default"/>
    <transportConnector name="ssl"      uri="ssl://localhost:61617"/>
    <transportConnector name="stomp"   uri="stomp://localhost:61613"/>
    <transportConnector name="xmpp"    uri="xmpp://localhost:61222"/>
</transportConnectors>
```

As you can see, transport connectors are defined within the `<transportConnectors>` element. You define particular connectors with appropriate nested `<transportConnector>` element. ActiveMQ simultaneously supports many protocols listening on different ports. The configuration for a connector must uniquely define the name and the URI attributes. In this case, the URI defines the network protocol and optional parameters through which ActiveMQ will be exposed for connectivity. The `discoveryUri` attribute as shown on the openwire connector above is optional and will be discussed further in Section 4.3.2.

The connection snippet above defines four transport connectors. Upon starting up ActiveMQ, you will see the following log out in the console as these connectors start up:

```
INFO TransportServerThreadSupport      - Listening for connections at:
  tcp://localhost:61616
INFO TransportConnector                 - Connector openwire Started
INFO TransportServerThreadSupport      - Listening for connections at:
  ssl://localhost:61617
INFO TransportConnector                 - Connector ssl Started
INFO TransportServerThreadSupport      - Listening for connections at:
  stomp://localhost:61613
INFO TransportConnector                 - Connector stomp Started
INFO TransportServerThreadSupport      - Listening for connections at:
  xmpp://localhost:61222
INFO TransportConnector                 - Connector xmpp Started
```

From the client's perspective, the transport connector URI is used to create a connection to the broker in order to send and receive messages. Sending and

receiving messages will be discussed in detail in Chapter 6, but the following code snippet should be enough to demonstrate the usage of the transport connector URIs in Java applications:

```
ActiveMQConnectionFactory factory =
    new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection connection = factory.createConnection();
connection.start();
Session session =
    connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

Notice in the small example above that the transport connector URIs defined in ActiveMQ configuration are used by the client application to create a connection to the broker. In this case, the URI for the TCP transport is used.

Note

The important thing to know is that we can use the query part of the URI to configure connection parameters both on the server and client sides.

Usually most of the parameters applies both for client and server sides of the connection, but some of them are specific to one or the other, so be sure you check the protocol reference before using the particular query parameter. F

With this basic understanding of configuring transport connectors, it's important to become aware of and understand the available transport connectors in ActiveMQ. The next section takes a look at these connectors.

4.2.1. Using Transport Connectors

In Chapter 2 a stock portfolio example was introduced which uses ActiveMQ to publish and consume stock exchange data. There, we used the fixed standard connector URI since we wanted to make those introductory examples as simple as possible. In this chapter, however, we'll explain all protocols and demonstrate them by running the stock portfolio example over each of them. For that reason, we need to modify our stock portfolio example so it can work over any of the provided protocols.

For starters, let's take a look at a modified `main()` method of our stock portfolio publisher:

Example 4.1. Listing 3.1: Modifying stock portfolio publisher to support various connector URIs

```
public static void main(String[] args) throws JMSEException {
    if (args.length == 0) {
        System.err.println("Please define a connection URI!");
        return;
    }

    Publisher publisher = new Publisher(args[0]);❶

    String[] topics = new String[args.length - 1];❷
    System.arraycopy(args, 1, topics, 0, args.length - 1);❸
    while (total < 1000) {
        for (int i = 0; i < count; i++) {
            publisher.sendMessage(topics);
        }
        total += count;
        System.out.println(
            "Published '" + count + "' of '" +
            + total + "' price messages"
        );
        try {
            Thread.sleep(1000);
        } catch (InterruptedException x) {
        }
    }
    publisher.close();
}
```

- ❶ Define connection URI
- ❷ Extract topics from rest of arguments
- ❸ Extract topics from rest of arguments

The code above ensures that the connector URI is passed as the first argument and extracts topic names from the rest of the arguments passed to the application. Now the stock portfolio publisher can be run with the following command:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Publisher \
-Dexec.args="tcp://localhost:61616 CSCO ORCL"
```

```
...
```

Please post comments or corrections to the [Author Online Forum](#)

```
Sending: {price=65.713356601409, stock=JAVA, offer=65.779069958011, up=true}
on destination: topic://STOCKS.JAVA
Sending: {price=66.071605671946, stock=JAVA, offer=66.137677277617, up=true}
on destination: topic://STOCKS.JAVA
Sending: {price=65.929035001620, stock=JAVA, offer=65.994964036622, up=false}
on destination: topic://STOCKS.JAVA

...
```

Note that one more argument has been added to the publisher; the URL to be used to connect to the appropriate broker.

The same principle can be used to modify the appropriate stock portfolio consumer. In the following code snippet, you will find stock portfolio consumer's `main()` method modified to accept the connection URI as a first parameter:

Example 4.2. Listing 3.2: Modifying stock portfolio consumer to support various connector URIs

```
public static void main(String[] args) throws JMSException {
    if (args.length == 0) {
        System.err.println("Please define connection URI!");
        return;
    }

    Consumer consumer = new Consumer(args[0]);❶

    String[] topics = new String[args.length - 1];❷
    System.arraycopy(args, 1, topics, 0, args.length - 1);❸
    for (String stock : topics) {
        Destination destination =
            consumer.getSession().createTopic("STOCKS." + stock);
        MessageConsumer messageConsumer =
            consumer.getSession().createConsumer(destination);
        messageConsumer.setMessageListener(new Listener());
    }
}
```

- ❶ Define connection URI
- ❷ Extract topics from rest of arguments
- ❸ Extract topics from rest of arguments

In order to achieve the same functionality as in the chapter 2 example, you should

Please post comments or corrections to the [Author Online Forum](#)

run the consumer with an extra URI argument. The following example shows how to do this:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Consumer \
-Dexec.args="tcp://localhost:61616 CSCO ORCL"

...
ORCL 65.71 65.78 up
ORCL 66.07 66.14 up
ORCL 65.93 65.99 down
CSCO 23.30 23.33 up
...
```

Notice that the message flow between the producer and the consumer is the same as in the original example. With these changes, the examples are now ready to be run using a variety of supported protocols.

4.2.2. Using Network Protocols

The most common usage scenario is to run ActiveMQ as a standalone Java application. This implies that clients (producer and consumer applications) will use some of the network protocols to access broker's destinations. In this section, we will describe available network protocols you can use to achieve client-to-broker communication.

4.2.2.1. Transmission Control Protocol (TCP)

Transmission Control Protocol (TCP) is today probably as important to humans as electricity. As one of the fundamental Internet protocols, we use it for almost all of our online communication. It is used as an underlying network protocol for wide range of Internet services such as email and the Web, for example.

You are probably already familiar with the basics of TCP and you've probably used it in your projects on various occasions (at least indirectly). So let's start our discussion of TCP by quoting from the specification, RFC 793 ([\[http://tools.ietf.org/html/rfc793\]](http://tools.ietf.org/html/rfc793)):

The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched

computer communication networks, and in interconnected systems of such networks.

As our broker and client applications are network hosts trying to communicate in a reliable manner (reliability is one of the main JMS characteristics), it is easy to see why TCP is an ideal network protocol for a JMS implementation. So it should not come as a surprise that the *TCP transport connector* is the most frequently used ActiveMQ connector.

Before exchanging messages over the network we need to serialize them to the suitable form. How messages are serialized from and to a byte-sequence is defined by the *wire protocol*. A default wire protocol used in ActiveMQ is called *OpenWire* (protocol specification can be found at <http://activemq.apache.org/openwire.html>). Its main purpose is to be network efficient and allow fast exchange of messages over the network. Besides that, the standardized and open protocol like this allows "native" ActiveMQ clients to be developed for various programming environments. This topic (and description of other wire protocols available for ActiveMQ) is in details covered in Chapter 14, *Administering and Monitoring ActiveMQ*. And to wrap all this up, the TCP transport connector is used to exchange messages serialized to OpenWire wire format over the TCP network.

As we have seen in previous sections, a default broker configuration starts the TCP transport listening for client connections on port 61616. The TCP connector URI has the following syntax:

```
tcp://hostname:port?key=value&key=value
```

The bold portion of the URI above denotes the required part. Any key/value pairs to the right of the question mark are optional and separated by an ampersand.

We will not discuss all transport options for appropriate protocols in this and sections that follows. This kind of material is best presented in the form of on-line reference pages. So in this chapter, we will explain basics of each protocol, how to configure and use it and point you to the appropriate protocol reference. For TCP connector, you can find an up to date reference (with all configuration options) at

the following URL: <http://activemq.apache.org/tcp-transport-reference.html>

To configure the TCP transport connector, the following configuration snippet should be added to the ActiveMQ configuration file (if it is not already there):

```
<transportConnectors>
  <transportConnector
    name="tcp"
    uri="tcp://localhost:61616?trace=true"
  />
</transportConnectors>
```

Note that the `trace` option has been added to the transport connector URI. This option instructs the broker to log all commands sent over this connector and can be helpful for debugging purposes. We have it here just as an example of tuning transport features using transport options. For full description of how to use trace option to debug ActiveMQ, see Chapter 14, *Administering and Monitoring ActiveMQ*.

Important

After changing the a configuration file, ActiveMQ must be restarted for the changes to take effect.

We have already seen in the previous section how to use this protocol in your client applications to connect to the broker. Just for the reference, the following example shows how to run the consumer over the TCP transport connector:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Consumer \
-Dexec.args="tcp://localhost:61616 CSCO ORCL"

...
ORCL 65.71 65.78 up
ORCL 66.07 66.14 up
ORCL 65.93 65.99 down
CSCO 23.30 23.33 up
...
```

Some of the benefits of the TCP transport connector include:

- Efficiency - Since this protocol uses the OpenWire marshaling protocol to

convert messages to stream of bytes (and back), it is very efficient in terms of network usage and performance

- Availability - TCP is one of the most widespread network protocols and it has been supported in Java from the early days, so it is almost certainly supported on your platform of choice
- Reliability - The TCP protocol ensures that messages won't be lost on the network (due to glitches, for example).

Now let's explore some alternatives to TCP transport connector.

4.2.2.2. New I/O API Protocol (NIO)

The *New I/O (NIO) API* was introduced in Java SE 1.4 to supplement the existing (standard) I/O API used in Java until then. Despite the prefix *new* in its name, NIO was never meant to be a replacement for the traditional Java I/O API. Its purpose was to provide an alternative approach to network programming and access to some low-level I/O operations of modern operating systems. The most prominent features of NIO are selectors and non-blocking I/O programming, allowing developers to use the same resources to handle more network clients and generally heavier loads on their servers.

From a client perspective, the *NIO transport connector* is practically the same as the standard TCP connector, in terms of its use of TCP as an underlying network protocol and OpenWire as a message serialization protocol. The only difference is under the covers with the implementation of the transport whereby the NIO transport connector is implemented using the NIO API. This makes the NIO transport connector more suitable in situations where:

- *You have a large number of clients you want to connect to the broker.*
Generally, the number of clients that can connect to the broker is limited by the number of threads supported by the operating system. Since the NIO connector implementation starts less threads per client than the TCP connector, you should consider using NIO in case TCP does not meet your needs.

- *You have a heavy network traffic to the broker.* Again, the NIO connector generally offers better performance than the TCP connector (in terms of using less resources on the broker side), so you can consider using it when you find that the TCP connector does not meet your needs.

At this point it is important to note that performance tuning of ActiveMQ is not just related to choosing the right connector. Many other aspects of ActiveMQ can be tuned including the right choice of network of brokers topology (Chapter 9) and setting various options for brokers, producers and consumers (Chapter 12).

The URI syntax for the NIO connector is practically the same as that of the TCP connector URI syntax. The only difference is the use of the `nio` scheme instead of `tcp`, as shown below:

```
nio://hostname:port?key=value
```

Now take a look at the following configuration snippet:

```
<transportConnectors>
    <transportConnector
        name="tcp"
        uri="tcp://localhost:61616?trace=true" />①
    <transportConnector
        name="nio"
        uri="nio://localhost:61618?trace=true" />②
</transportConnectors>
```

- ❶ A `tcp` connector that listens on port 61616
- ❷ A `nio` connector that listens on port 61618

Now run the stock portfolio example, but this time you will connect the publisher and consumer over different transport connectors. As Figure 4.1, “Producer sends messages using `nio` transport, consumer receives them using `tcp` transport” shows, the publisher will send messages over the NIO transport connector, while the consumer will receive those messages using the TCP transport connector.

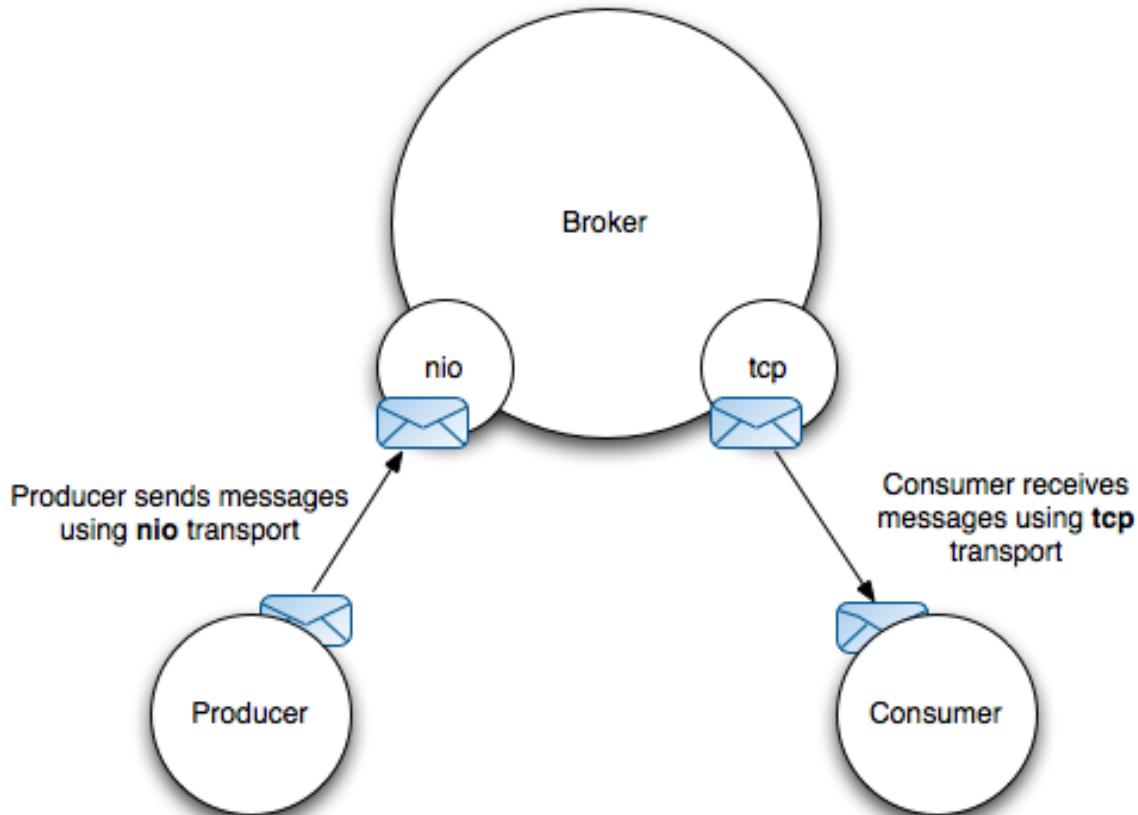


Figure 4.1. Producer sends messages using nio transport, consumer receives them using tcp transport

To achieve this, the stock portfolio publisher should be run using the following command:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Publisher \
-Dexec.args="nio://localhost:61618 CSCO ORCL"

...
Sending: {price=65.713356601409, stock=JAVA, offer=65.779069958011, up=true}
on destination: topic://STOCKS.JAVA
Sending: {price=66.071605671946, stock=JAVA, offer=66.137677277617, up=true}
on destination: topic://STOCKS.JAVA
Sending: {price=65.929035001620, stock=JAVA, offer=65.994964036622, up=false}
on destination: topic://STOCKS.JAVA
...
```

Please post comments or corrections to the [Author Online Forum](#)

Notice that the `nio` scheme is used in the connection URI to specify the NIO connector.

The consumer should use the TCP connector using the command shown earlier:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Consumer \
-Dexec.args="tcp://localhost:61616 CSCO ORCL"

...
ORCL 65.71 65.78 up
ORCL 66.07 66.14 up
ORCL 65.93 65.99 down
CSCO 23.30 23.33 up

...
```

After both the consumer and producer are started you will notice that messages are exchanged between applications as expected. The fact that they are using different connectors and protocols to communicate with the broker does not play any role in this exchange.

4.2.2.3. User Datagram Protocol (UDP)

User Datagram Protocol (UDP) along with TCP makes up the core of Internet protocols. The purpose of these two protocols is identical - to send and receive data packets (datagrams) over the network. However, there are two main differences between them, notably:

- *TCP is a stream oriented protocol*, which means that the order of data packets is guaranteed. In other words, there is no chance for data packets to be duplicated or arrive out of order. UDP, on the other hand, does not guarantee packet ordering so a receiver can expect data packets to be duplicated or arrive out of order.
- *TCP also guarantees reliability of packet delivery*, meaning that packets will not be lost during the transport. This is ensured by maintaining an active connection between the sender and receiver. On the contrary, UDP is a connectionless protocol, so it cannot make such guarantees.

As you can conclude from the discussion above, TCP is used in applications that

requires reliability (such as email), while UDP usually finds its place in applications that require fast data transfers and can handle occasional packet loss (such as VoIP or online gaming).

You can use the UDP protocol to connect to ActiveMQ by using the UDP transport connector. The URI syntax of this protocol is pretty much the same as it is for the TCP connector. The only difference is the use of `udp` scheme, as shown in the following snippet:

```
udp://hostname:port?key=value
```

The complete reference of the UDP protocol can be found at the following URL:
<http://activemq.apache.org/udp-transport-reference.html>

Comparing the TCP and UDP Transports

When considering the TCP and the UDP transports the question arises comparing the two. When should you use the UDP transport instead of default TCP transport? There are basically two such situations where the UDP transport offers an advantage, namely:

- The broker is located behind a firewall that you don't control and you can access it only over UDP ports.
- The use of very time sensitive messages and you want to eliminate network transport delay as much as possible.

But there are also a couple of pitfalls regarding the UDP connector:

- Since UDP is unreliable, you can end up with losing some of the messages, so your application should know how to deal with this situation.
- Network packets transmitted between clients and brokers are not just messages, but can also contain so called *control commands*. If some of these control commands are lost due to UDP unreliability, the JMS connection could be endangered. Thus you should always add a reliability layer (as described in later sections) over this transport just to ensure the

JMS communication between client and broker.

Now let's configure ActiveMQ to use both TCP and UDP transports on different ports. Below is an example of such a configuration:

```
<transportConnectors>
    <transportConnector
        name="tcp"
        uri="tcp://localhost:61616?trace=true"
    />
    <transportConnector
        name="udp"
        uri="udp://localhost:61618?trace=true"
    />
</transportConnectors>
```

Notice that there are two separate transport connectors on different ports.

To run a stock portfolio publisher over the UDP protocol, run the following command:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Publisher \
-Dexec.args="udp://localhost:61618 CSCO ORCL"

...
Sending: {price=65.713356601409, stock=JAVA, offer=65.779069958011, up=true}
on destination: topic://STOCKS.JAVA
Sending: {price=66.071605671946, stock=JAVA, offer=66.137677277617, up=true}
on destination: topic://STOCKS.JAVA
Sending: {price=65.929035001620, stock=JAVA, offer=65.994964036622, up=false}
on destination: topic://STOCKS.JAVA

...
```

The appropriate consumer can be run over the TCP protocol with the following command:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Consumer \
-Dexec.args="tcp://localhost:61616 CSCO ORCL"

...
ORCL 65.71 65.78 up
ORCL 66.07 66.14 up
ORCL 65.93 65.99 down
```

Please post comments or corrections to the [Author Online Forum](#)

```
CSCO 23.30 23.33 up
```

```
...
```

As expected, the behavior of the overall system is the same as it was in the original example when only the TCP transport connector was used.

4.2.2.4. Secure Sockets Layer Protocol (SSL)

Imagine yourself in a situation where you need to expose the broker over an unsecured network and you need data privacy. The same requirement emerged when the Web outgrew its academic roots and was considered for corporate usage. Sending plain data over TCP became unacceptable and a solution had to be found. The solution for secure data transfers was the *Secure Sockets Layer (SSL)*, a protocol designed to transmit encrypted data over the TCP network protocol. It uses a pair of keys (a private and a public one) to ensure a secure communication channel. ActiveMQ provides the *SSL transport connector*, which adds an SSL layer over the TCP communication channel, providing encrypted communication between brokers and clients.

The URI syntax for this protocol is:

```
ssl://hostname:port?key=value
```

Since the SSL transport is based on the TCP transport, configuration options are the same. Nevertheless, if you need a reference for this protocol, you can find it at the following URL: <http://activemq.apache.org/ssl-transport-reference.html>

ActiveMQ uses *Java Secure Socket Extension (JSSE)* to implement its SSL functionality. Since its detailed description is out of the scope of this book, please refer to the online information for JSSE

(<http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html>) before proceeding to the rest of this section.

As can see in the default configuration file, the SSL connector is configured to listen for connections on port 61617. We can create a setup that will configure only TCP and SSL transport protocols by changing the `<transportConnectors>` element in the `${ACTIVEMQ_HOME} /conf/activemq.xml` file to the following:

Please post comments or corrections to the [Author Online Forum](#)

```
<transportConnectors>
    <transportConnector
        name="tcp"
        uri="tcp://localhost:61616?trace=true"
    />
    <transportConnector
        name="ssl"
        uri="ssl://localhost:61617?trace=true"
    />
</transportConnectors>
```

Unlike other transport connectors we have described thus far, SSL needs a few more items in order to work properly. Such required items include SSL certificates for successful SSL communication. Basically, JSSE defines two types of files for storing keys and certificates. The first are so called *keystores*, which stores your own private data, certificates with their corresponding private keys. Trusted certificates of other entities (applications) are stored in *truststores*.

The default keystores and truststores used by ActiveMQ are located in the `${ACTIVEMQ_HOME}/conf/` folder. For starters, there you will find a keystore containing a default broker certificate (`broker.ks`). Of course, there is also a truststore used by broker to hold trusted client certificates (`broker.ts`). If no configuration details are provided, ActiveMQ will use `broker.ks` and `broker.ts` for SSL transport connector.

Important

The default keystore and truststore are for demonstration purposes only and should not be used for the production deployment of ActiveMQ. For production use, it is highly recommended that you create your own keystore and truststore.

Now, that we understand all the necessary elements needed for successful SSL communication, let's see how we can connect to the secured transport. In the rest of this section, we will first see how we can connect to the broker secured with its default certificate. Next, we will go step by step through the process of creating our own certificates and running a broker and clients with them.

But first of all, let's see what happens if you try to connect to a broker using SSL

and without providing any other SSL-related parameters. Connecting the stock portfolio consumer by simply changing the transport to use SSL without creating the proper stores will cause errors. Below is an example of simply changing the transport:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Consumer \
-Dexec.args="ssl://localhost:61617 CSCO ORCL"
```

Without creating and denoting the proper keystore and truststore, you can expect to see the following exceptions:

```
WARNING: Async exception with no exception listener:
javax.net.ssl.SSLHandshakeException:
sun.security.validator.ValidatorException: PKIX path building failed:
    sun.security.provider.certpath.SunCertPathBuilderException:
unable to find valid certification path to requested target
javax.net.ssl.SSLHandshakeException:
    sun.security.validator.ValidatorException:
PKIX path building failed:
    sun.security.provider.certpath.SunCertPathBuilderException:
unable to find valid certification path to requested target
```

Also, in the broker's log you will see the following error:

```
ERROR TransportConnector
- Could not accept connection : Received fatal alert: certificate_unknown
```

These errors mean that the SSL connection could not be established. This is a generic error all clients will receive when trying to connect to the untrusted broker (i.e., without the proper keystore and truststore).

When using JSSE, you must provide some SSL parameters using the appropriate system properties. In order to successfully connect to the broker via SSL, we must provide the keystore, the keystore password and the truststore to be used. This is accomplished using the following system properties:

- `javax.net.ssl.keyStore` - Defines a keystore the client should use
- `javax.net.ssl.keyStorePassword` - Defines an appropriate password for the keystore

- `javax.net.ssl.trustStore` - Defines an appropriate truststore the client should use

Now take a look at the following example of starting our stock portfolio publisher using the default client certificate stores distributed with ActiveMQ:

```
$ mvn \
-Djavax.net.ssl.keyStore=${ACTIVEMQ_HOME}/conf/client.ks \
-Djavax.net.ssl.keyStorePassword=password \
-Djavax.net.ssl.trustStore=${ACTIVEMQ_HOME}/conf/client.ts \
exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Publisher \
-Dexec.args="ssl://localhost:61617 CSCO ORCL"

...
Sending: {price=65.713356601409, stock=JAVA, offer=65.779069958011, up=true}
on destination: topic://STOCKS.JAVA
Sending: {price=66.071605671946, stock=JAVA, offer=66.137677277617, up=true}
on destination: topic://STOCKS.JAVA
Sending: {price=65.929035001620, stock=JAVA, offer=65.994964036622, up=false}
on destination: topic://STOCKS.JAVA
...
...
```

Notice the use of the JSSE system properties that provide the necessary keystore, keystore password and truststore. After providing these necessary SSL-related parameters, the publisher will connect successfully to the broker as intended without error. Of course, if the client is not located on the same computer as your broker, you'll need to copy these files and adapt the paths appropriately.

Similarly, the consumer can be run using the following command:

```
$ mvn \
-Djavax.net.ssl.keyStore=${ACTIVEMQ_HOME}/conf/client.ks \
-Djavax.net.ssl.keyStorePassword=password \
-Djavax.net.ssl.trustStore=${ACTIVEMQ_HOME}/conf/client.ts \
exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Consumer \
-Dexec.args="ssl://localhost:61617 CSCO ORCL"

...
ORCL 65.71 65.78 up
ORCL 66.07 66.14 up
ORCL 65.93 65.99 down
CSCO 23.30 23.33 up
...
...
```

Again, note the use of the JSSE system properties. Now both clients can communicate with the broker using the encrypted network channels provided by the SSL transport connector.

Working with the default certificate, keystore and truststore is OK for development purposes, but for a production system it is highly recommended that you create and use your own certificates. In most cases you will need to purchase an appropriate SSL certificate from the trusted certificate authority.

For development purposes, you'll want to create your own self-signed certificates. In the rest of this section we will go through the process of creating and sharing self-signed certificates. For that purpose we will use `keytool`, the command-line tool for managing keystores that is distributed with Java.

First of all, you must create a keystore and a certificate for the broker. Below is an example of using the `keytool`:

```
$ keytool -genkey -alias broker -keyalg RSA -keystore mybroker.ks
Enter keystore password: test123
What is your first and last name?
[Unknown]: Dejan Bosanac
What is the name of your organizational unit?
[Unknown]: Chapter 3
What is the name of your organization?
[Unknown]: ActiveMQ in Action
What is the name of your City or Locality?
[Unknown]: Belgrade
What is the name of your State or Province?
[Unknown]:
What is the two-letter country code for this unit?
[Unknown]: RS
Is CN=Dejan Bosanac, OU=Chapter 3, O=ActiveMQ in Action,
L=Belgrade, ST=Unknown, C=RS correct?
[no]: yes

Enter key password for <broker>
(RTURN if same as keystore password):
```

The `keytool` application prompts you to enter certificate data and create a keystore with the certificate in it. In this case we have created a keystore named `mybroker.ks` with the password "test123".

The next step is to export this certificate from the keystore, so it can be shared with broker's clients. This is done using the following command:

```
$ keytool -export -alias broker -keystore mybroker.ks -file mybroker_cert  
Enter keystore password: test123  
Certificate stored in file <mybroker_cert>
```

This step creates a file named `mybroker_cert`, containing a broker certificate.

Now you must create a client keystore with the appropriate certificate using a command similar to the one that was used previously to create the broker's keystore:

```
$ keytool -genkey -alias client -keyalg RSA -keystore myclient.ks  
What is your first and last name?  
[Unknown]: Dejan Bosanac  
What is the name of your organizational unit?  
[Unknown]: Chapter 3  
What is the name of your organization?  
[Unknown]: ActiveMQ in Action  
What is the name of your City or Locality?  
[Unknown]: Belgrade  
What is the name of your State or Province?  
[Unknown]:  
What is the two-letter country code for this unit?  
[Unknown]: RS  
Is CN=Dejan Bosanac, OU=Chapter 3, O=ActiveMQ in Action,  
L=Belgrade, ST=Unknown, C=RS correct?  
[no]: yes  
  
Enter key password for <client>  
(RETURN if same as keystore password):
```

The result of the command above is the `myclient.ks` file with the appropriate certificate for the client-side. Finally, the client truststore must be created and the broker's certificate must be imported into it. Again, `keytool` is used to achieve this with the following command:

```
$ keytool -import -alias broker -keystore myclient.ts -file mybroker_cert  
Enter keystore password: test123  
Owner: CN=Dejan Bosanac, OU=Chapter 3, O=ActiveMQ in Action,  
L=Belgrade, ST=Unknown, C=RS  
Issuer: CN=Dejan Bosanac, OU=Chapter 3, O=ActiveMQ in Action,  
L=Belgrade, ST=Unknown, C=RS  
Serial number: 484fdc8a  
Valid from: Wed Jun 11 16:09:14 CEST 2008 until: Tue Sep 09 16:09:14 CEST 2008  
Certificate fingerprints:  
MD5: 04:66:F2:AA:71:3A:9E:0A:3C:1B:83:C0:23:DC:EC:6F  
SHA1: FB:FA:BB:45:DC:05:9D:AE:C3:BE:5D:86:86:0F:76:84:43:C7:36:D3  
Trust this certificate? [no]: yes  
Certificate was added to keystore
```

With this step, all the necessary stores were created and the broker certificate was imported into the keystore. Now the stock portfolio example can make use of them.

Remember to start the broker using the newly created certificate. One way to do this is to replace the default keystore files and the broker cert in the `conf` directory with the ones that were just created. Another way is to include passing the SSL-related system properties to the command used to start our broker. The system properties approach will be used in the example below:

```
`${ACTIVEMQ_HOME}/bin/activemq \
-Djavax.net.ssl.keyStorePassword=test123 \
-Djavax.net.ssl.keyStore=${ACTIVEMQ_HOME}/conf/mybroker.ks
```

Now let's see how to reflect these same changes to the clients. If you try to run the client applications with the old certificate file, you will get the `unknown_certificate` exception just as it was when the client attempted to access the broker without using any certificate. So you'll have to update the command like the following:

```
$ mvn \
-Djavax.net.ssl.keyStore=${ACTIVEMQ_HOME}/conf/myclient.ks \
-Djavax.net.ssl.keyStorePassword=test123 \
-Djavax.net.ssl.trustStore=${ACTIVEMQ_HOME}/conf/myclient.ts \
exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Publisher \
-Dexec.args="ssl://localhost:61617 CSCO ORCL"

...
Sending: {price=65.713356601409, stock=JAVA, offer=65.779069958011, up=true}
on destination: topic://STOCKS.JAVA
Sending: {price=66.071605671946, stock=JAVA, offer=66.137677277617, up=true}
on destination: topic://STOCKS.JAVA
Sending: {price=65.929035001620, stock=JAVA, offer=65.994964036622, up=false}
on destination: topic://STOCKS.JAVA
...
```

The command above instructs the publisher to use the newly created client stores. And, of course, after these changes, the stock portfolio application works again.

4.2.2.5. Hypertext Transfer Protocol (HTTP/HTTPS)

In many environments, firewalls are configured to allow only basic services such as web access and email. So how can ActiveMQ be used in such an environment?

This is where the HTTP transport comes into play.

Hypertext Transfer Protocol (HTTP) was originally designed to transmit hypertext (HTML) pages over the Web. It uses TCP as an underlying network protocol and adds some additional logic for communication between browsers and web servers. After the first boom of the Internet, web infrastructure and the HTTP protocol in particular found a new role in supporting *web services*, commonly used these days to exchange information between applications. The main difference is that in case of web services, XML formatted data is transmitted using the HTTP protocol rather than HTML data.

ActiveMQ implements the *HTTP transport connector* which provides for the exchange of XML-formatted messages with the broker over the HTTP protocol. This is what allows ActiveMQ to bypass strict firewall rules. By simply using the HTTP protocol that runs on the standard web port number (80), ActiveMQ is able to use an existing hole in the firewall, so to say.

The URI syntax of this transport connector is follows:

```
http://hostname:port?key=value
```

Secure HTTP (HTTP over SSL or HTTPS) is also supported by this transport:

```
https://hostname:port?key=value
```

Notice the slight difference in the scheme used by the two examples of above based on whether security is needed or not. Let's walk through an example configuration to see how to run the examples using the HTTP transport. The transport connectors section of the XML configuration in this case looks very similar to those used in previous sections, but with the HTTP scheme:

```
<transportConnectors>
    <transportConnector
        name="tcp"
        uri="tcp://localhost:61616?trace=true"
    />
    <transportConnector
        name="http"
        uri="http://localhost:8080?trace=true"
    />
</transportConnectors>
```

Please post comments or corrections to the [Author Online Forum](#)

Notice that there are two transports configured above, one for the TCP transport and one for the HTTP transport which listens to port 8080.

In order to run the clients using the HTTP transport protocol, there is one dependency that must be added to the classpath. The HTTP transport is located in the ActiveMQ optional module, so you'll have to add it to the application's classpath (along with appropriate dependencies). Using Maven, you'll need to add the following dependency to the `pom.xml` file:

```
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-optional</artifactId>
    <version>5.2.0</version>
</dependency>
```

This will include `activemq-optional` module and all its dependencies to the classpath. In case you don't use Maven to manage your classpath, be sure to include all of these JARs into your classpath:

```
$ACTIVEMQ_HOME/lib/optional/activemq-optional-<version>.jar
$ACTIVEMQ_HOME/lib/optional/commons-httpclient-<version>.jar
$ACTIVEMQ_HOME/lib/optional/xstream-<version>.jar
$ACTIVEMQ_HOME/lib/optional/xmlpull-<version>.jar
```

Finally, the stock portfolio publisher is ready to be run using the HTTP transport:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Publisher \
-Dexec.args="http://localhost:8080 CSCO ORCL"

...
Sending: {price=65.713356601409, stock=JAVA, offer=65.779069958011, up=true}
on destination: topic://STOCKS.JAVA
Sending: {price=66.071605671946, stock=JAVA, offer=66.137677277617, up=true}
on destination: topic://STOCKS.JAVA
Sending: {price=65.929035001620, stock=JAVA, offer=65.994964036622, up=false}
on destination: topic://STOCKS.JAVA
...
```

As stated previously, when using the HTTP transport all broker-to-client communication is performed by sending XML messages. This type of communication can have an impact on the overall system performance compared to the use of the TCP transport with the OpenWire protocol (which is highly tuned

specifically for messaging purposes). So if performance is a concern, you're best to stick to the TCP transport and find some other workaround for the firewall issues.

4.2.3. Using the Virtual Machine Protocol

So far this chapter has covered protocols used to connect brokers and clients using the network stack in the operating system. As an alternative, ActiveMQ was designed to be embedded in a Java application. This allows client-to-broker communication to take place locally in the JVM, instead of via the network. In order to support this kind of intra-VM communication, ActiveMQ provides a special protocol named the VM protocol.

4.2.3.1. VM Protocol

The *VM transport connector* is used by Java applications to launch an embedded broker and connect to it. Use of the VM transport means that no network connections are created between clients and the embedded broker. Communication is performed through direct method invocations of the broker object. Because the network stack is not employed, performance improves significantly. The broker is started when the first connection is created using the VM protocol. All subsequent VM transport connections from the same virtual machine will connect to the same broker.

A broker created using the VM protocol doesn't lack any of the standard ActiveMQ features. So, for example, the broker can be configured with other transport connectors as well. When all clients that use the VM transport to the broker close their connections (and there are no other active connections), the broker will automatically shut down.

The URI syntax for VM transport is as follows:

```
vm://brokerName?key=value
```

The broker name plays an important role in the VM transport connector URI by uniquely identifying the broker. For example, you can create two different embedded brokers by specifying different broker names. This is the only required

Please post comments or corrections to the [Author Online Forum](#)

difference.

Transport options are set using the query part of the URI, the same as the previously discussed transports. The complete reference for this connector could be found at the following URL:

<http://activemq.apache.org/vm-transport-reference.html>

The important thing about options for the VM transport protocol is that you can use them to configure the broker to some extent. Options whose name begins with the prefix "broker." are used to tune the broker. For example, the following URI starts up a broker with persistence disabled (message persistence is explained in Chapter 3):

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

There is also an alternative URI syntax that can be used to configure an embedded broker:

```
vm:broker:(transportURI, network:networkURI)/brokerName?key=value
```

The complete reference of the broker URI can be found at the following URI:
<http://activemq.apache.org/broker-uri.html>

As you can see, this kind of URI can be used to configure additional transport connectors. Take a look at the following URI for example:

```
vm:broker:(tcp://localhost:6000)?brokerName=embeddedbroker&persistent=false
```

Here, we have defined an embedded broker named `embeddedBroker` and also configured a TCP transport connector that listens for connections on port 6000. Finally, persistence is also disabled in this broker. The Figure 4.2 can help to better visualize this example configuration.

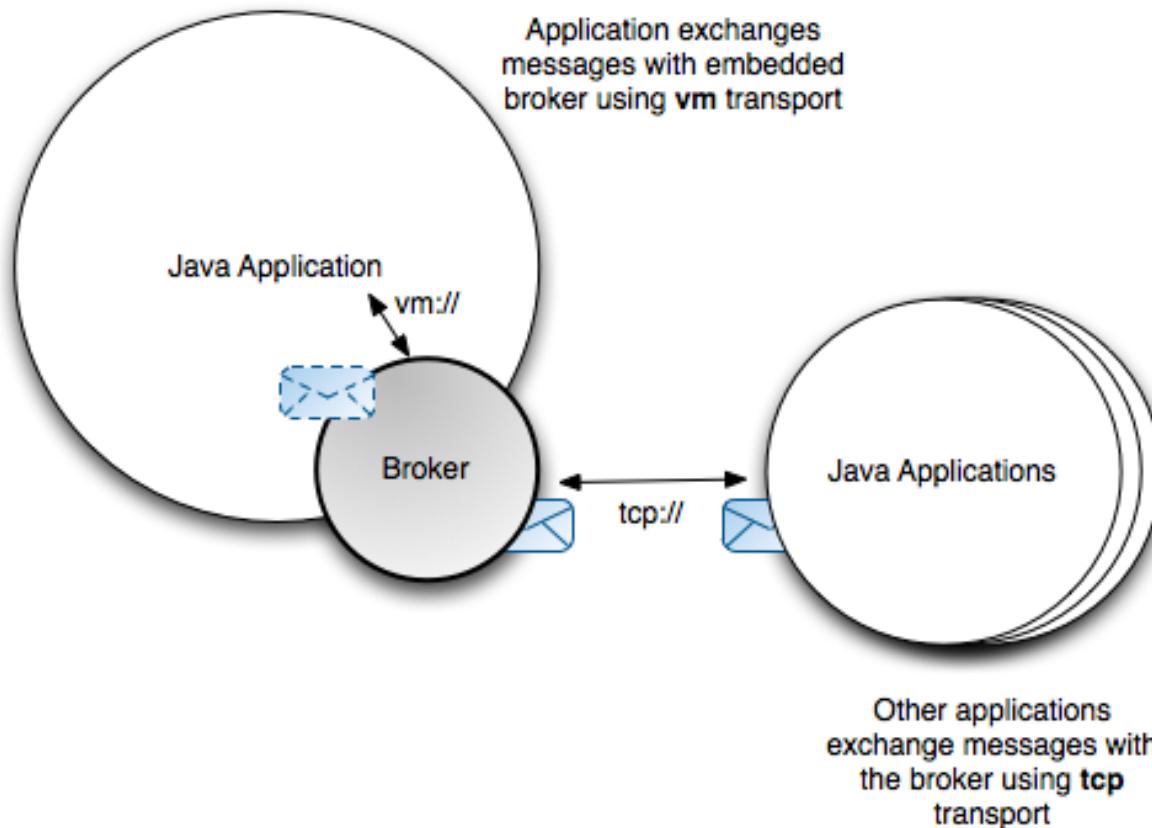


Figure 4.2. Application exchanges messages with embedded broker using vm transport

Figure Figure 4.2 demonstrates clients connecting to the broker from within the application that embeds the broker will use the VM transport. While external applications connect to that embedded broker using the TCP connector, just as they would do in case of any standalone broker.

An embedded broker using an external configuration file, this can be achieved using the `brokerConfig` transport option and by specifying the URI for the `activemq.xml` file. Below is an example:

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

The example above will locate the `activemq.xml` file in the classpath using the

xbean: protocol. Using this approach, an embedded broker can be configured just like a standalone broker using the XML configuration.

Now the stock portfolio publisher can be started with an embedded broker using the following command:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Publisher \
-Dexec.args="vm://localhost CSCO ORCL"

...
Sending: {price=65.713356601409, stock=JAVA, offer=65.779069958011, up=true}
on destination: topic://STOCKS.JAVA
Sending: {price=66.071605671946, stock=JAVA, offer=66.137677277617, up=true}
on destination: topic://STOCKS.JAVA
Sending: {price=65.929035001620, stock=JAVA, offer=65.994964036622, up=false}
on destination: topic://STOCKS.JAVA
...
```

Notice that the publisher works just fine without having started an external broker. We'll see how to configure this broker with the additional TCP transport and connect the portfolio consumer as an external Java application in Chapter 6.

One obvious advantage of the VM transport is improved performance client-to-broker communication. Also, you'll have only one Java application to run (one JVM) instead of two, which can ease your deployment process. This also means that there's one less Java process to manage. So, if you plan to use the broker mainly from one application, maybe you should consider using the embedded broker.

On the other hand, if too many Java applications that use embedded brokers exist, maintenance problems may arise when trying to consistently configure each broker as well as back up the data. In such situations, it's always easier to create a small cluster of standalone brokers instead of using embedded brokers.

In Chapter 6 various embedding techniques will be explored further.

4.3. Configuring Network Connectors

Having one ActiveMQ broker to serve all your application needs works very well

for most situations. But some environments need advanced features, such as high-availability and larger scalability. This is typically achieved using what is known as a *network of brokers*. A network of brokers creates a cluster composed of multiple ActiveMQ instances that are interconnected to meet more advanced messaging scenarios. Various topologies for broker networks, their purpose and configuration details are explained in detail in Chapter 9. This section will briefly explain *network connectors* in ActiveMQ. The previous section discussed transport connectors which provide client-to-broker communications whereas this section will discuss network connectors which provide broker-to-broker communications.

Network connectors are channels that are configured between brokers so that those brokers can communicate with one another. A network connector is a uni-directional channel by default. That is, a given broker communicates in one direction by only forwarding messages it receives to the brokers on the other side of the connection. This setup is commonly referred to as a *forwarding bridge*. In some situations, however, you may want to create a bi-directional communication channel between brokers. That is, a channel that communicates not only outward to the brokers on the other side of the connection, but it also receives messages from other brokers on that same channel. ActiveMQ supports this kind of bi-directional connector, which is usually referred to as a *duplex connector*. The Figure 3.3 shows one example of a network of brokers that contains both a forwarding bridge and duplex connectors.

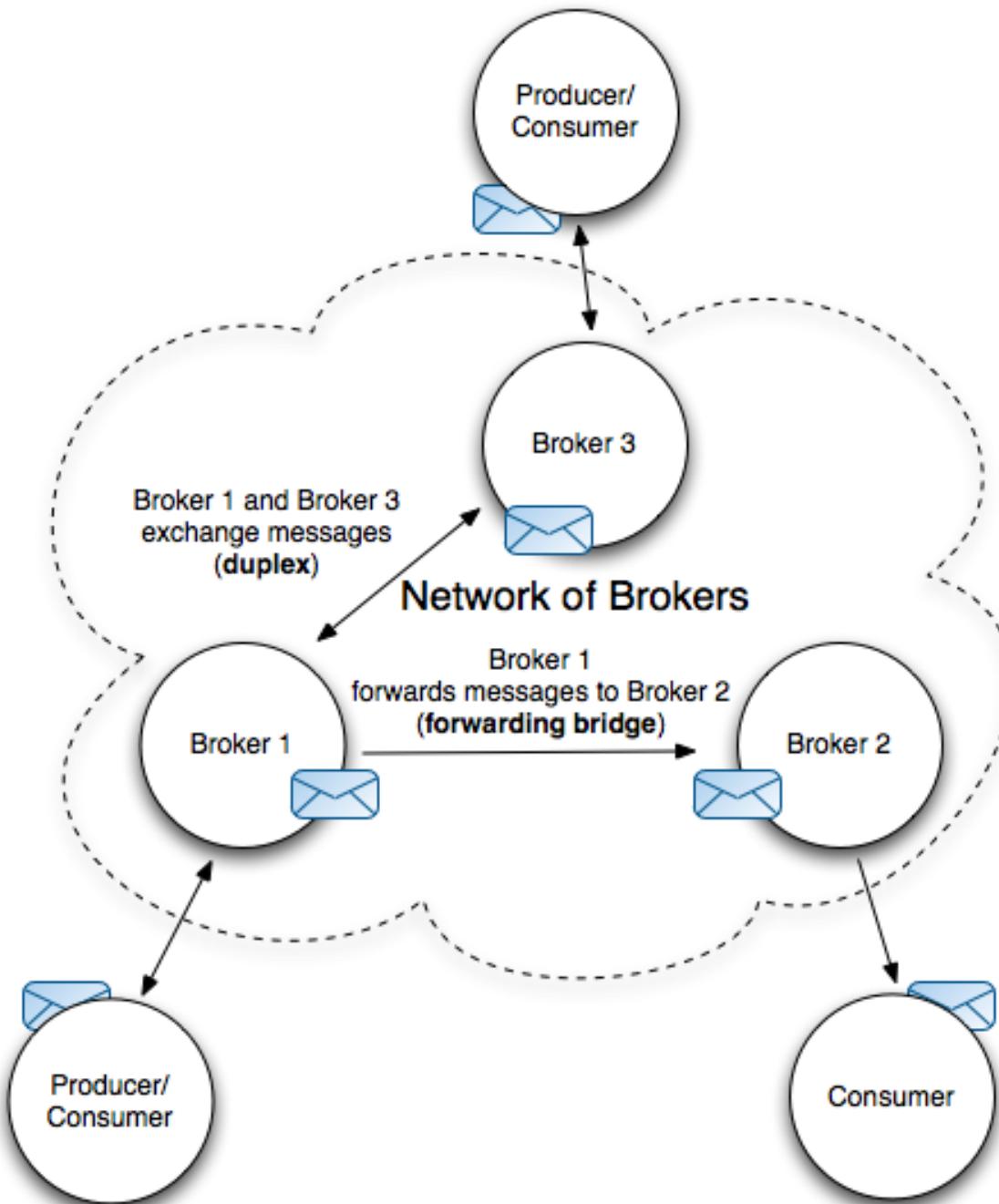


Figure 4.3. An example of complex network of brokers topology

Network connectors are configured through the ActiveMQ XML configuration file

in a similar fashion to the configuration of transport connectors. Let's take a look at the default configuration:

```
<!-- The store and forward broker networks ActiveMQ will listen to -->
<networkConnectors>
    <!-- by default just auto discover the other brokers -->
    <networkConnector name="default-nc" uri="multicast://default"/>
    <!--
    <networkConnector name="host1 and host2"
        uri="static://(tcp://host1:61616,tcp://host2:61616)"/>
    -->
</networkConnectors>
```

As you can see, networks of brokers are configured using the `<networkConnectors>` element. This element contains the configuration for one or more connectors using the `<networkConnector>` element. As was the case with transport connectors, the mandatory attributes for the `<networkConnector>` element are the `name` and the `uri`. All other attributes are optional and are used to configure additional features on the connector, as you will see in the moment.

In the rest of this chapter, various ActiveMQ protocols and techniques that are used to configure and connect to a network of brokers will be presented and discussed. But before we dive in, there is one more important ActiveMQ concept we should explain known as *discovery*. In general, discovery is a process of detecting remote broker services. Clients usually want to discover all available brokers. Brokers, on the other hand, usually want to find other available brokers so they can establish a network of brokers.

There are two approaches to configuring and connecting clients to a network of brokers. In the first approach, clients can be statically configured to access specific brokers. Protocols used for this kind of configuration are explained in the Section 4.3.1. In the second approach, brokers and clients can use what are known as *discovery agents* that dynamically detect brokers. Discovery agents, and the protocols they use, are explained in Section 4.3.2.

4.3.1. Defining Static Networks

The first approach to configuring and connecting to a network of brokers is through the use of statically configured URIs. Simply configuring a list of broker

URIs available for connection. The only prerequisite is that you know the addresses of all the brokers you want to use. Once you have these URIs, you need to know how to use them in a configuration. So let's take a look at the protocols available to create a static networks of brokers.

4.3.1.1. Static Protocol

The *static network connector* is used for to create a static configuration of multiple brokers in a network. This protocol uses a composite URI - that is, a URI that contains other URIs. A composite URI consists of multiple broker addresses or URIs that are on the other end of the network connection.

The URI syntax for the static protocol noted below:

```
static:(uri1,uri2,uri3,...)?key=value
```

and you can find the complete reference of this transport at the following address:
<http://activemq.apache.org/static-transport-reference.html>

Now take a look at the following configuration example:

```
<networkConnectors>
  <networkConnector name="local network"
    uri="static://(tcp://remotehost1:61616,tcp://remotehost2:61616)"/>
</networkConnectors>
```

Assuming that this configuration is for the broker on the `localhost` and that brokers on hosts `remotehost1` and `remotehost2` are up and running, you will notice the following messages when you start the local broker:

```
...
INFO DiscoveryNetworkConnector - Establishing network connection between
from vm://localhost to tcp://remotehost1:61616
INFO TransportConnector - Connector vm://localhost Started
INFO DiscoveryNetworkConnector - Establishing network connection between
from vm://localhost to tcp://host2:61616
INFO DemandForwardingBridge - Network connection between vm://localhost#0
and tcp://remotehost1:61616 has been established.
INFO DemandForwardingBridge - Network connection between vm://localhost#2
and tcp://remotehost2:61616 has been established.
...
```

The output shown above indicates that the broker on the `localhost` has

successfully configured a *forwarding bridge* with two other brokers running on two remote hosts. In other words, messages sent to the local broker will be forwarded to brokers running on `remotehost1` and `remotehost2`, but only if there's demand for those messages (more on this in Chapter 9).

Let's demonstrate static networks using the stock portfolio example over the network of brokers. The following diagram provides a perspective of the broker topology used in this example:

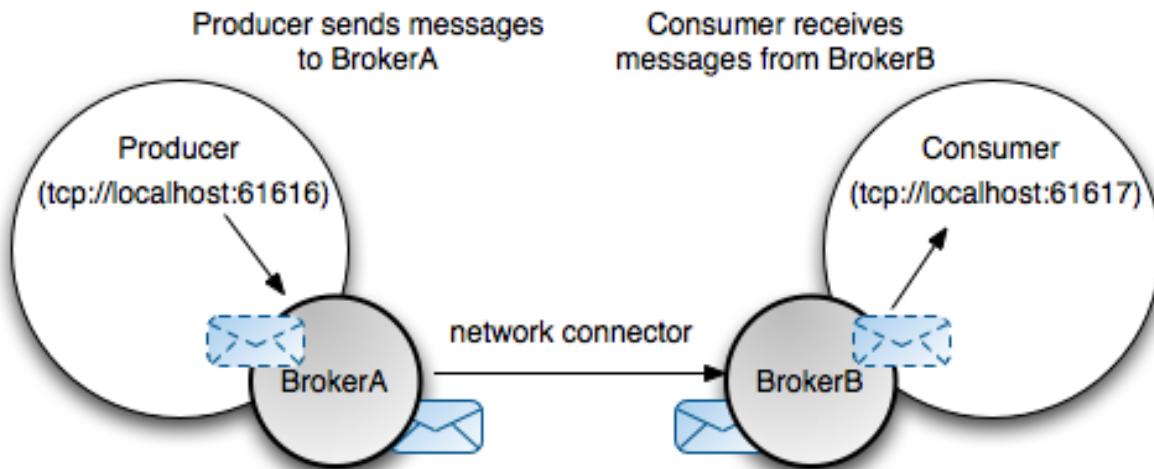


Figure 4.4. Two application exchange messages using two brokers in a static network

In the diagram above, the two brokers are networked. That is, the brokers utilize a network connector with a URI using the static protocol. A consumer is attached to a destination on BrokerB which creates demand for messages across the network connector. When the producer sends messages to the same destination on BrokerA they will be forwarded to the broker where there is demand. In this case, BrokerA forwards messages to BrokerB. The best way to understand this better is to experience it yourself using an example. The following example will walk through this basic use case.

To make this example work, first we need to start these two networked brokers. Let's start with BrokerB:

```
<broker xmlns="http://activemq.apache.org/schema/core" brokerName="BrokerB"
    dataDirectory="${activemq.base}/data">

    <!-- The transport connectors ActiveMQ will listen to -->
    <transportConnectors>
        <transportConnector name="openwire" uri="tcp://localhost:61617" />
    </transportConnectors>

</broker>
```

This simple configuration starts a broker that listens on port 61617. We can start this broker with the following command:

```
 ${ACTIVEMQ_HOME}/bin/activemq \
xbean:src/main/resources/org/apache/activemq/book/ch3/brokerA.xml
```

Now it's time to configure BrokerA:

```
<broker xmlns="http://activemq.apache.org/schema/core" brokerName="BrokerA"
    dataDirectory="${activemq.base}/data">

    <!-- The transport connectors ActiveMQ will listen to -->
    <transportConnectors>
        <transportConnector name="openwire" uri="tcp://localhost:61616" />
    </transportConnectors>

    <networkConnectors>
        <networkConnector uri="static:(tcp://localhost:61617)" />
    </networkConnectors>

</broker>
```

Besides the transport connector listening on port 61617, it defines a network connector that connects to BrokerB. In a separate console window, you can start this broker like this:

```
 ${ACTIVEMQ_HOME}/bin/activemq \
xbean:src/main/resources/org/apache/activemq/book/ch3/brokerB.xml
```

Now that we have both brokers up and running, let's run the stock portfolio example. First we will start our publisher and connect it to BrokerA

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Publisher \
-Dexec.args="tcp://localhost:61616 CSCO ORCL"
...

```

Please post comments or corrections to the [Author Online Forum](#)

```
Sending: {price=65.713356601409, stock=JAVA, offer=65.779069958011, up=true}
  on destination: topic://STOCKS.JAVA
Sending: {price=66.071605671946, stock=JAVA, offer=66.137677277617, up=true}
  on destination: topic://STOCKS.JAVA
Sending: {price=65.929035001620, stock=JAVA, offer=65.994964036622, up=false}
  on destination: topic://STOCKS.JAVA

...
```

This is practically the same command was used with the earlier TCP connector example. Now start the consumer and connect it to the BrokerB:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Consumer \
-Dexec.args="tcp://localhost:61617 CSCO ORCL"

...
ORCL 65.71 65.78 up
ORCL 66.07 66.14 up
ORCL 65.93 65.99 down
CSCO 23.30 23.33 up

...
```

Using this setup, messages are published to the BrokerA. These messages are then forwarded to the BrokerB, where they are consumed by the consumer. The overall functionality of this example has not been changed and both the publisher and the consumer behave the same as the previous single broker example. The only difference is that the publisher and the consumer are now connecting to different brokers that are networked using the static protocol.

Again, the use cases for the different broker topologies will be discussed more thoroughly in Chapter 9. But from this simple example you can conclude that this particular configuration can help you in situations when you need your distributed clients to benefit all performance advantages of communicating with the local broker.

4.3.1.1. Example Use of the Static Protocol

Configuring broker networks can be difficult depending on the situation. Use of the static protocol allows for an explicit notation that a network should exist. Consider a situation where clients in remote offices are connecting to a broker in the home office. Depending on the number of clients in each remote office, you may wind up

with far too many wide area network connections into the home office. This can cause an unnecessary burden on the network. To minimize connections, you may want to place a broker in each remote office and allow a static network connection between the remote office broker and the home office broker. Not only will this minimize the number of network connections between the remote offices and the home office, but it will allow the client applications in the remote offices to operate more efficiently. The removal of the long haul connection over the wide area network means less latency and therefore less waiting for the client application.

4.3.1.2. Failover Protocol

In all the examples so far, the clients have been configured to connect to only one specific broker. But what should you do in case you cannot connect to the desired broker or your connection fails at the later stage? Your clients have two options, either they will die gracefully or try to connect to the same or some other broker and resume their work. As you can probably guess, the stock portfolio example runs using the protocols described thus far are not immune to network problems and unavailable brokers. That's where protocols like *failover* comes in to implement automatic reconnection logic on top of low-level transport connectors described earlier. Similar to the case with the network connectors, there are two ways to provide a list of suitable brokers to which the client can connect. In the first case, you provide a static list of available brokers. This is approach used by the *failover transport connector*. In the second case, dynamic discovery of the available brokers is used. This will be explained later in the chapter. In this section, the failover transport connector will be examined.

The URI syntax for the failover connector is very similar to the previous static network connector URI. There are actually two available forms to the failover URI:

```
failover:(uri1,...,uriN)?key=value
```

or

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

The complete reference of this protocol could be found at the following URL:
<http://activemq.apache.org/failover-transport-reference.html>

By default, this protocol uses a random algorithm to choose one of the underlying connectors. If the connection fails (both on startup or at a later stage), the transport will pick another URI and try to make a connection. A default configuration also implements a *reconnection delay logic*, meaning that the transport will start with a 10ms delay for the first reconnection attempt and double this time for any subsequent attempt up to 30000ms. Also, the reconnection logic will try to reconnect indefinitely. Of course, all reconnection parameters can be reconfigured according to your needs using the appropriate transport options.

Recall the static network of brokers that was defined in the previous section. In that example, all messages sent to the local broker were forwarded to the brokers located on `remotehost1` and `remotehost2`. Because all messages were sent to both of these brokers, those messages can be consumed from either broker. The same is true here. The only difference is that the failover transport will automatically attempt a reconnect in the event of a broker failover. To experience the use of this transport, run the stock portfolio consumer and configure it to connect to the brokers using the failover connector:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Consumer \
-Dexec.args="failover:(tcp://host1:61616,tcp://host2:61616) CSCO ORCL"
```

The beauty of this solution is that it requires no changes to the application in order to add support for automatic reconnection in the event of a broker failure.

Now let's see the failover connector at work. Imagine that the random algorithm in the failover transport has chosen to connect the consumer to the broker on `host1`. You can expect that the consumer will print the following log message during the startup:

```
org.apache.activemq.transport.failover.FailoverTransport$1 iterate INFO: \
Successfully reconnected to tcp://host1:61616
```

As we already said, all messages sent by the publisher to the local broker will be forwarded to the broker on `host1` and received by the consumer. Now try to simulate a broker failure by shutting down the broker on `host1`. The consumer will

print the following log message:

```
org.apache.activemq.transport.failover.FailoverTransport handleTransportFailure
WARNING: Transport failed,
    attempting to automatically reconnect due to: java.io.EOFException
java.io.EOFException
at java.io.DataInputStream.readInt(DataInputStream.java:375)
at org.apache.activemq.openwire.OpenWireFormat.unmarshal(
    OpenWireFormat.java:268
)
at org.apache.activemq.transport.tcp.TcpTransport.readCommand(
    TcpTransport.java:192
)
at org.apache.activemq.transport.tcp.TcpTransport.doRun(
    TcpTransport.java:184
)
at org.apache.activemq.transport.tcp.TcpTransport.run(
    TcpTransport.java:172
)
at java.lang.Thread.run(Thread.java:619)
org.apache.activemq.transport.failover.FailoverTransport$1 iterate
INFO: Successfully reconnected to tcp://host2:61616
```

Notice the initial exception noting the failure, followed by the log message about reconnecting to another broker. This means that the consumer has successfully connected to the other broker and you can see that it resumed its normal operation without any assistance. You can

4.3.1.2.1. Example Use of the Failover Protocol

Due to its reconnection capabilities, it is highly advisable to use the failover protocol for all clients. Even if a client will only be connecting to a single broker. For example, the following URI will try to reestablish a connection to the same broker in the event that the broker shuts down for any reason:

```
failover:(tcp://localhost:61616)
```

The advantage of this is that clients don't need to be manually restarted in the case of a broker failure (e.g., failure, maintenance, etc.). As soon as the broker becomes available again the client will automatically reconnect. This means far more robustness for your applications by simply utilizing the a feature of ActiveMQ.

The failover transport connector plays an important role in achieving advanced functionalities such as high availability and load balancing as will be explained in

Chapter 9.

4.3.2. Defining Dynamic networks

Thus far we have seen how to setup broker networks and connect to them by explicitly specifying broker URIs (both transport and network connectors). As you will see in this section, ActiveMQ implements several mechanisms that can be used by brokers and clients to find each other and establish necessary connections.

4.3.2.1. Multicast Protocol

IP multicast is a network technique used for easy transmission of data from one source to a group of interested receivers, one-to-many communication, over an IP network. One of the fundamental concepts of IP multicast is so called *group address*. Group address is an IP address in the range of 224.0.0.0 to 239.255.255.255, used by both sources and receivers. Sources use this address as a destination for their data, while receivers use it to express their interest in data of that group.

ActiveMQ brokers use multicast protocol to advertise their services and locate services of other brokers for the purpose of creating networks of brokers. Clients, on the other hand, use multicast to locate brokers and establish a connection with them. This section will discuss how brokers use multicast, while the use of multicast by client will be discussed later.

The URI syntax for the multicast protocol is as follows:

```
multicast://ipaddress:port?key=value
```

This is really no different than the previous URIs with the exception of the scheme portion.

Below is a snippet from the default ActiveMQ configuration that makes use of multicast:

```
<broker xmlns="http://activemq.apache.org/schema/core" brokerName="multicast"
       dataDirectory="${activemq.base}/data">

    <networkConnectors>
```

Please post comments or corrections to the [Author Online Forum](#)

```
<networkConnector name="default-nc" uri="multicast://default"/>
</networkConnectors>

<transportConnectors>
    <transportConnector name="openwire" uri="tcp://localhost:61616"
        discoveryUri="multicast://default"/>
</transportConnectors>

</broker>
```

In the example above, a *default* group name is used instead of a specific IP address. This means that the broker will use 239.255.2.3 address for multicast packets. You can find a complete reference of this protocol at the following URL:
<http://activemq.apache.org/multicast-transport-reference.html>

There are two important things achieved with this configuration snippet. First, the transport connector's `discoveryUri` attribute is used to advertise this transport's URI on the default group. All clients interested in finding an available broker would use this connector. This will be demonstrated in the following section.

Next, the `uri` attribute of the network connector is used to search for available brokers and to create a network with them. In this case, the broker acts like a client and uses multicast for lookup purposes.

In the end, it is important to say that multicast works only if all brokers (and clients) are located in the same local area network (LAN). It's maybe not obvious from the start, but that is a limitation of using the IP multicast protocol.

Preventing Automatic Broker Discovery

When developing in a team environment and using the default configuration that comes with the ActiveMQ distribution, it's highly possible (and quite probable) that two or more ActiveMQ instances will automatically connect to one another and begin consuming one another's messages. Here are some recommendations for preventing this situation from occurring:

- 1. Remove the `discoveryUri` portion of the `openwire` transport connector** - The transport connector whose name is `openwire` is configured by default to advertise the broker's TCP transport using

multicast. This allows other brokers to automatically discover it and connect to it if necessary.

Below is the openwire transport connector definition from the `conf/activemq.xml` configuration file:

```
<transportConnector name="openwire" uri="tcp://localhost:61616"  
discoveryUri="multicast://default"/>
```

To stop the broker from advertising the TCP transport URI via multicast, change the definition to remove the `discoveryUri` attribute so it looks like this:

```
<transportConnector name="openwire" uri="tcp://localhost:61616" />
```

2. **Comment out/remove the default-nc network connector** - The network connector named `default-nc` utilizes the multicast transport to automatically and dynamically discover other brokers. To stop this behavior, comment out/remove the `default-nc` network connector so that it won't automatically discover other brokers.

Below is the `default-nc` network connector definition from the `conf/activemq.xml` configuration file:

```
<networkConnector name="default-nc" uri="multicast://default"/>
```

To disable this network connector, comment it out so it looks like this:

```
<!--networkConnector name="default-nc" uri="multicast://default"-->
```

3. **Give the broker a unique name** - The default configuration for ActiveMQ in the `conf/activemq.xml` file provides a broker name of `localhost` as shown below:

```
<broker xmlns="http://activemq.apache.org/schema/core" brokerName="localhost"  
dataDirectory="${activemq.base}/data">
```

In order to uniquely identify your broker instance, change the

brokerName attribute from localhost to something unique such as the example below:

```
<broker xmlns="http://activemq.apache.org/schema/core" brokerName="broker1234"  
dataDirectory="${activemq.base}/data">
```

This is especially handy when searching through log files to see which brokers are taking certain actions.

Now that you know how to configure discovery on the broker-side, let's see how to tell the client-side to use it to automatically locate brokers.

4.3.2.1.1. Example Use of the Multicast Protocol

The multicast protocol is not very different from the TCP protocol. The only difference is the automatic discovery of other brokers instead of using a static list of brokers. Use of the multicast protocol is common where brokers added and removed frequently and cases where brokers may have their IP addresses changed frequently. In these cases, instead of reconfiguring each broker manually for every change, it's oftentimes easier to utilize a discovery protocol.

The disadvantage to using the multicast protocol is that discovery is automatic. If there are brokers that you do not want to be automatically added to a given group, you must be careful in setting up the initial configuration of the broker network. Careful segmentation of broker networks is important as you don't want messages to wind up in a broker network where they don't belong.

One additional disadvantage of the multicast protocol is that it can be excessively chatty on the network. For this reason, many network administrators will not allow its use. Please check with your network administrator before taking the time to configure a network using the multicast protocol.

4.3.2.2. Discovery Protocol

The *discovery transport connector* is on the client-side of the ActiveMQ multicast functionality. This protocol is basically the same as the failover protocol in its

behavior. The only difference is that it will use multicast to discover available brokers and randomly choose one to connect to.

The syntax of this protocol is:

```
discovery:(discoveryAgentURI)?key=value
```

and its complete reference could be found at the following URL:
<http://activemq.apache.org/discovery-transport-reference.html>

Using the multicast broker configuration explained earlier, you can run the broker with the following command:

```
 ${ACTIVEMQ_HOME}/bin/activemq \
xbean:src/main/resources/org/apache/activemq/book/ch3/activemq-multicast.xml
```

Once the broker is started, run the stock portfolio publisher with the following command:

```
$ mvn -e exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Publisher \
-Dexec.args="discovery:(multicast://default) CSCO ORCL"
```

You will notice the following log messages at the application startup:

```
Jun 18, 2008 2:13:18 PM
  org.apache.activemq.transport.discovery.DiscoveryTransport onServiceAdd
INFO: Adding new broker connection URL: tcp://localhost:61616
Jun 18, 2008 2:13:19 PM
  org.apache.activemq.transport.failover.FailoverTransport doReconnect
INFO: Successfully connected to tcp://localhost:61616

...
Sending: {price=65.713356601409, stock=JAVA, offer=65.779069958011, up=true}
  on destination: topic://STOCKS.JAVA
Sending: {price=66.071605671946, stock=JAVA, offer=66.137677277617, up=true}
  on destination: topic://STOCKS.JAVA
Sending: {price=65.929035001620, stock=JAVA, offer=65.994964036622, up=false}
  on destination: topic://STOCKS.JAVA
...
```

These messages tell you that the publisher client has successfully used multicast to discover and connect to the local broker.

4.3.2.3. Peer Protocol

As we have seen before, networks of brokers and embedded brokers are very useful concepts that allow you to fit brokers to your infrastructure needs. Of course, it is theoretically possible to create networks of embedded brokers, but this would be quite cumbersome to configure manually. This is why ActiveMQ provides the *peer transport connector* as it allows you to more easily network embedded brokers. It is a superset of a VM connector that creates a *peer-to-peer* network of embedded brokers.

The URI syntax of this protocol is as follows:

```
peer://peergroup/brokerName?key=value
```

You can find its complete reference at the following URL:

<http://activemq.apache.org/peer-transport-reference.html>

When started with an appropriate peer protocol URI, the application will automatically start an embedded broker (just as was the case with the VM protocol), but it will also configure the broker to establish network connections to other brokers in the local network with the same group name.

Let's walk through a demonstration of this using the stock portfolio example with the peer protocol. In this case, both the publisher and the consumer will use their own embedded brokers, that will be appropriately networked automatically. The figure below provides a better perspective of this solution.

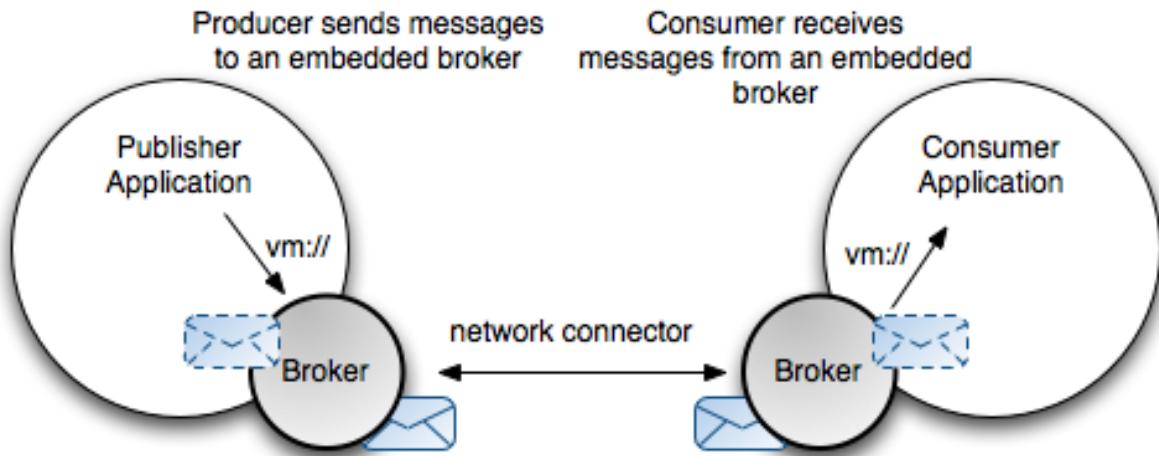


Figure 4.5. Two applications communicating using embedded brokers over peer protocol

Advise the stock portfolio publisher to create its own embedded broker using group1 like this:

```
$ mvn -e exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Publisher \
-Dexec.args="peer://group1 CSCO ORCL"

...
Sending: {price=65.713356601409, stock=JAVA, offer=65.779069958011, up=true}
on destination: topic://STOCKS.JAVA
Sending: {price=66.071605671946, stock=JAVA, offer=66.137677277617, up=true}
on destination: topic://STOCKS.JAVA
Sending: {price=65.929035001620, stock=JAVA, offer=65.994964036622, up=false}
on destination: topic://STOCKS.JAVA
...
```

Also advise the stock portfolio consumer to create its own embedded broker using group1 like this:

```
$ mvn -e exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Consumer \
-Dexec.args="peer://group1 CSCO ORCL"

...
ORCL 65.71 65.78 up
ORCL 66.07 66.14 up
```

```
ORCL 65.93 65.99 down  
CSCO 23.30 23.33 up
```

```
...
```

The two commands above start two embedded brokers (one for each application) and created a peer-to-peer broker network named `group1` between these two brokers. All messages sent to one broker will be available in the other broker as well as any other brokers that might join `group1`. Notice that the overall system operates as if these two applications are using the same centralized broker.

4.3.2.3.1. Example Use of the Peer Protocol

Consider an application that resides on the laptop of a field sales representative who often disconnects from the company network but still needs the application to run successfully in a disconnected mode. This is a common scenario where the client application just needs to continue working no matter whether the network is available. This is a case where the peer protocol can be utilized for an embedded broker to allow the application on the laptop to keep running successfully. In reality, while in disconnected mode, the application is simply sending messages to the local broker where they are queued up to be sent at a later time when the network is available again. The sales rep can still log client calls, visits, etc. while the laptop is disconnected from the network. When the laptop is again connected to the network, all of the queued messages will be sent.

4.3.2.4. Fanout Protocol

Fanout is another utility connector used by clients to simultaneously connect to multiple brokers and replicate operations to those brokers. The URI syntax of this protocol is as follows:

```
fanout:(fanoutURI)?key=value
```

You can find its complete reference at the following URL:
<http://activemq.apache.org/fanout-transport-reference.html>

The `fanoutURI`, can utilize either a static URI or a multicast URI. Consider the

following example:

```
fanout:(static:(tcp://host1:61616,tcp://host2:61616,tcp://host3:61616))
```

In the example above, the client will try to connect to three brokers statically defined using the static protocol:

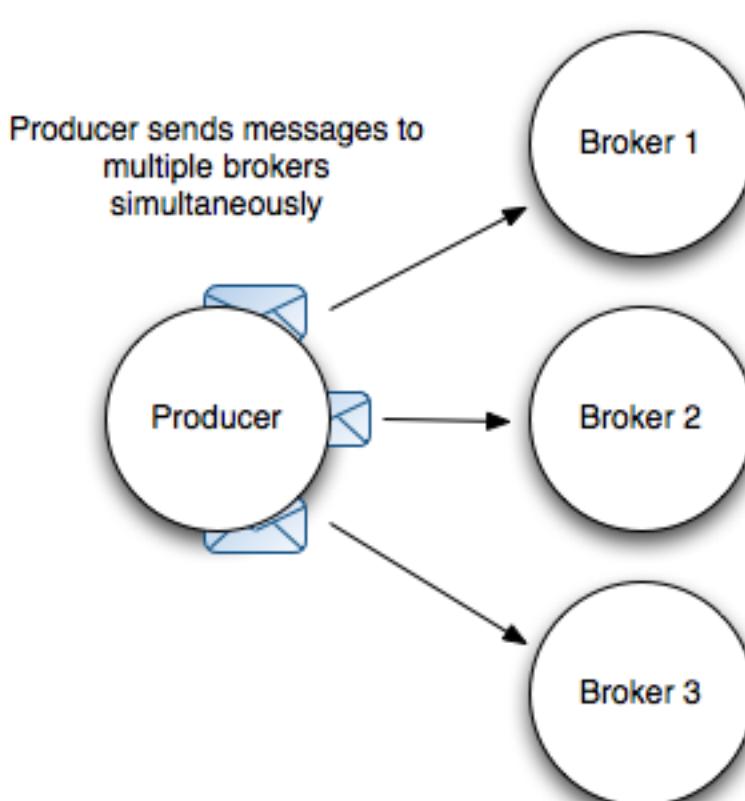


Figure 4.6. Producer sends messages to multiple brokers using fanout protocol

The same effect could be accomplished by simply using the following URI:

```
fanout:(multicast://default)
```

This assumes that the brokers are configured to use multicast to advertise their transport connectors.

By default, the fanout protocol will wait until it connects to at least two brokers

and will not replicate commands to queues (only topics). Both of these functionalities are, of course, configurable with appropriate transport options.

Finally, there are a couple of things you should be aware of if you plan to use the fanout protocol. First of all, it is not recommended to use it for consuming messages. Its only purpose is to produce messages to multiple brokers. Also, if brokers you are using are in the network it is very likely that certain consumers will receive duplicate messages. So basically, it is only recommended to use the fanout protocol to publish messages to multiple non-connected brokers.

4.4. Summary

In this chapter, the various connectivity options in ActiveMQ were discussed. The format of ActiveMQ URIs was explained and you learned a difference between transport and network connectors. Embedded brokers and networks of brokers were also discussed and you saw how to use them to gain certain advantages over the single standalone broker approach. Finally, the reconnection protocols and discovery agents showed the true power of ActiveMQ connectivity options.

Knowing types of connectors and the essence of particular protocols is very important when you choose the overall topology of your JMS system. Various broker topologies and their purpose are explained in details in Chapter 9. In the following chapter message persistence will be discussed. We will see when and how ActiveMQ persists messages and what configuration options are provided for you.

Chapter 5. Message Persistence

The JMS specification supports two types of message delivery, persistent and non-persistent. Not only does ActiveMQ support both of these types, but it also can be configured to support message recovery, an in-between state where messages are cached in memory. ActiveMQ supports a pluggable strategy for message storage and provides storage options for in-memory, file-based and relational databases.

Persistent messages are ideal if you want messages to always be available to a message consumer after they have been delivered to a message broker, or you need messages to survive even if there has been a system failure. Once a message has been consumed and acknowledged by a message consumer, it is typically deleted from the broker's message store.

In this chapter, not only will the message persistence options available in ActiveMQ be explored, but it will also discuss when to use them. This chapter will cover the following items:

- Why messages are stored differently for queues and topics
- How the different message storage options work and when to use them
- How ActiveMQ can temporarily cache messages for retrieval

This chapter will provide a detailed guide to message persistence. In order to lay the groundwork for this, first the storage of messages for JMS destinations will be examined.

5.1. How Are Messages Stored by ActiveMQ?

It is important to gain some basic knowledge of the storage mechanisms for messages in an ActiveMQ message store. Such knowledge will aid in configuration and provide an awareness of what takes place in the ActiveMQ broker during the delivery of persistent messages. Messages sent to queues and

topics are stored slightly differently, and match delivery semantics for the point-to-point and publish/subscribe message domains.

Storage for queues is straightforward where messages are basically stored in first in, first out order (FIFO). See the Figure 5.1 image below for a depiction of this. Only one message is dispatched between one of potentially many consumers. Only when that message has been consumed and acknowledged can it be deleted from the broker's message store.

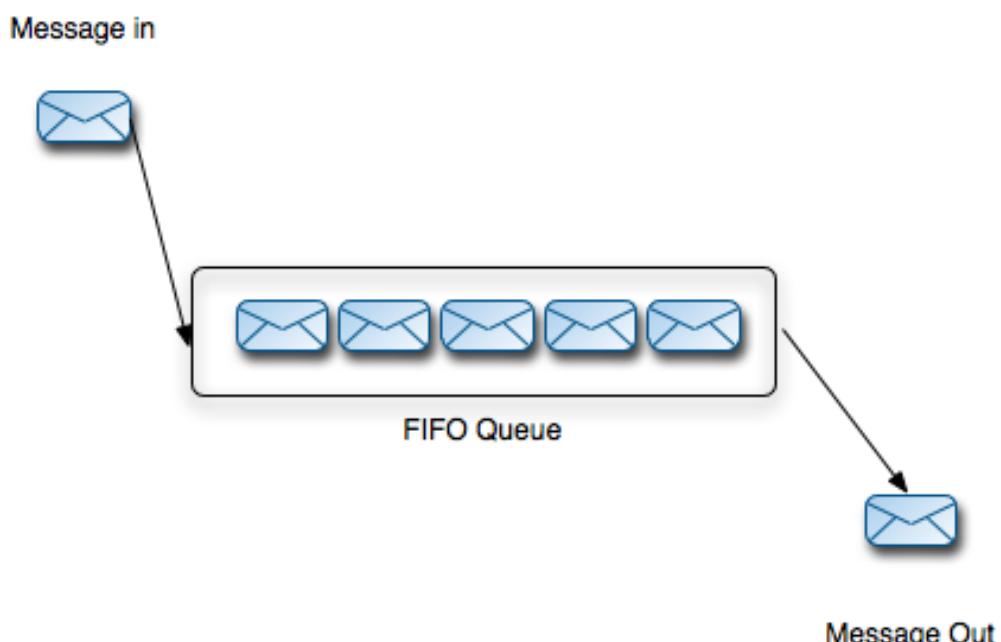


Figure 5.1. FIFO message storage for queues

For durable subscribers to a topic, each consumer gets a copy of the message. In order to save space (some messages can be very large!), only one copy of a message is stored by the broker. A durable subscriber object in the store maintains a pointer to its next stored message and dispatches a copy of it to its consumer as shown in Figure 5.2. The message store is implemented in this manner for durable subscribers because each one will be consuming messages at different rates. Not only that, but not all durable subscribers will be active at the same time. Also,

Please post comments or corrections to the [Author Online Forum](#)

since every message can potentially have many consumers, a message cannot be deleted from the store until it has been delivered to every interested consumer.

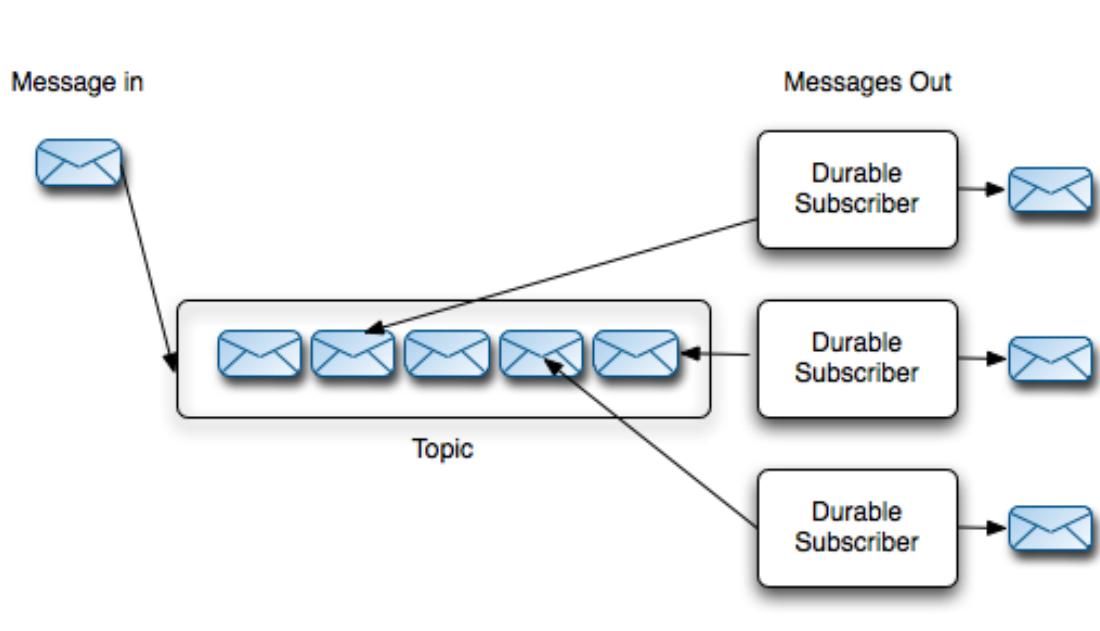


Figure 5.2. Messages Stored for Topics

Every message store implementation in ActiveMQ supports both persisting messages for both queues and topics, though obviously the implementation differs between storage types.

Throughout the rest of this chapter, more details about configuring the different ActiveMQ message stores and their advantages and disadvantages will be examined.

5.2. Available Message Stores in ActiveMQ

ActiveMQ provides a pluggable API for message stores as well as multiple message store implementations including:

- The AMQ Message Store - The default message store designed for performance and reliability

- The KahaDB Message Store - Provides improved scalability and recoverability (available in 5.3 and higher)
- The JDBC Message Store - A message store based on JDBC
- The Memory Message Store - A memory-based message store

These message store implementations provide many different pros and cons that will be touched upon throughout the chapter. As mentioned above, the AMQ store is the first implementation to be examined.

5.2.1. The AMQ Message Store

The AMQ message store is the default message store in ActiveMQ. It is a file-based, transactional store that has been tuned and designed for the very fast storage of messages. The aim of the AMQ message store is to be easy of use and as fast as possible. Its use of a file-based message database makes it easy to use since there is no prerequisite for a third party database. This message store enables ActiveMQ to be downloaded and running in literally minutes. In addition, the structure of the AMQ store has been streamlined especially for the requirements of a message broker.

If a message store is not configured for ActiveMQ, the AMQ store will be used with its default settings. To use an alternative message store or to change the default behavior of the AMQ message store, a persistence adapter must be configured using the `<persistenceAdapter>` element in the activemq.xml file. Below is a minimal configuration for the AMQ message store:

```
...
<broker persistent="true" xmlns="http://activemq.apache.org/schema/core">
...
  <persistenceAdapter>
    <amqPersistenceAdapter/>
  </persistenceAdapter>
...
</broker>
...
```

If ActiveMQ is embedded inside an application, the message store can also be

configured programmatically. Below is an example of a programmatic configuration for the AMQ message store:

```
public class EmbeddedBrokerUsingAMQStoreExample {  
  
    public void main(String[] args) throws Exception {  
  
        BrokerService broker = new BrokerService();❶  
  
        PersistenceAdapterFactory persistenceFactory =  
            new AMQPersistenceAdapterFactory();❷  
        persistenceFactory.setMaxFileLength(1024*16);  
        persistenceFactory.setPersistentIndex(true);  
        persistenceFactory.setCleanupInterval(10000);  
  
        broker.setPersistenceFactory(persistenceFactory);❸  
  
        broker.addConnector("tcp://localhost:61616");❹  
  
        broker.start();❺  
    }  
}
```

- ❶ Initialize the ActiveMQ broker
- ❷ Create an instance of the AMQ message store
- ❸ Instruct the broker to use the AMQ store
- ❹ Create a transport connector to expose the broker to clients
- ❺ Start the broker

Each ActiveMQ message store implementation uses a `PersistenceAdapter` object for access. `PersistenceAdapters` are typically initialized from a `PersistenceAdapterFactory`, hence the use of this interface in the example above. Although the example above seems small, the code is enough to programmatically embed ActiveMQ in a Java application. For more info about embedding ActiveMQ, see Chapter 7. In order to better understand its use and configuration, it's important to examine the internals of the AMQ message store.

5.2.1.1. The AMQ Message Store Internals

The AMQ store is the fastest of all the provided message store implementations. Its speed is the result of the combination of a fast transactional journal, the highly optimized indexing of message id and in-memory message caching. The Figure 5.3

Please post comments or corrections to the [Author Online Forum](#)

provides a high-level diagram of the AMQ message store:

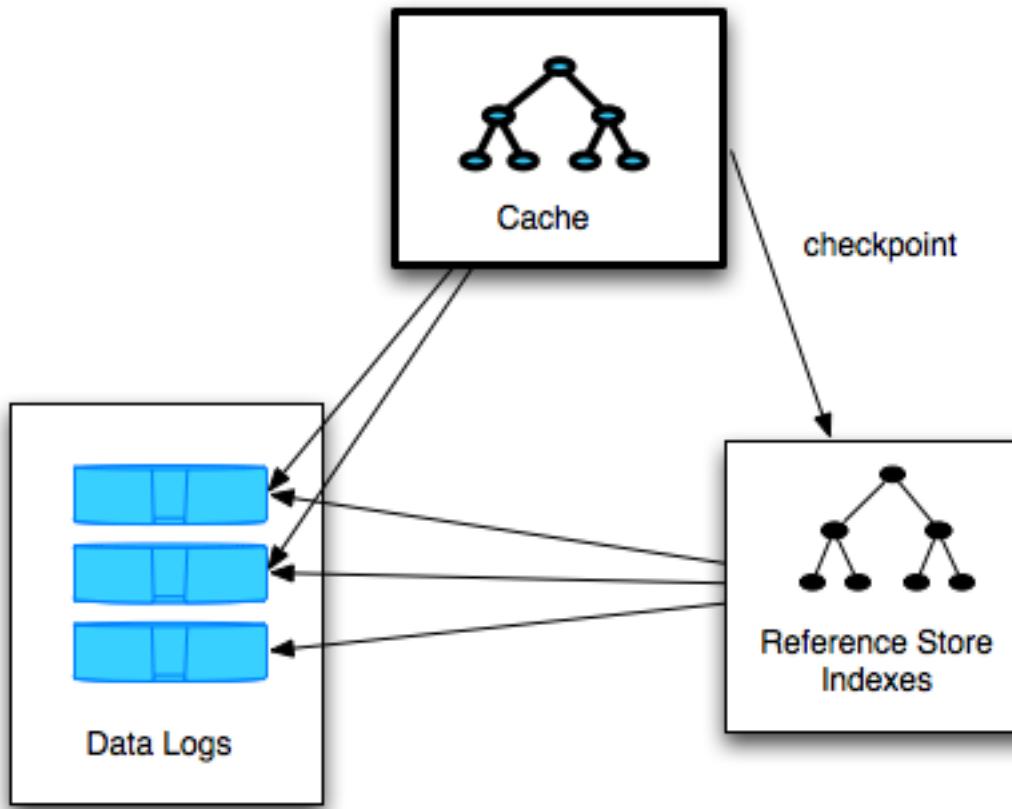


Figure 5.3. The AMQ message Store

The diagram above provides a view of the three distinct parts of the AMQ message store including:

It is important to understand the file-based directory structure used by the ActiveMQ message store. This will help with the configuration and also with problem identification when using ActiveMQ.

- **The Journal** consists of a rolling log of messages and commands (such as transactional boundaries and message deletions) stored in data files of a certain length. When the maximum length of the currently used data file has been reached a new data file is created. All the messages in a data file are reference counted, so that once every message in that data file is no longer

required, the data file can be removed or archived. The journal only appends messages to the end of the current data file, so storage is very fast.

- **The Cache** holds messages for fast retrieval in memory after they have been written to the journal. The cache will periodically update the reference store with its current message ids and location of the messages in the journal. This process is known as performing a checkpoint. Once the reference store has been updated, messages can be safely removed from the cache. The period of time between the cache updates to the reference store is configurable and can be set by the `checkpointInterval` property. A checkpoint will also occur if the ActiveMQ message broker is reaching its memory limit.
- **The Reference Store** holds references to the messages in the Journal that are indexed by their message Id. It is actually the reference store which maintains the FIFO data structure for queues and the durable subscriber pointers to their topic messages. The index type is configurable, the default being a disk based hash Index. It is also possible to use an in-memory hashmap as well. But this is only recommended if the total number of messages expected to be held in the message store is less than 1 million at any given time.

5.2.1.2. The AMQ Message Store Directory Structure

When you start the ActiveMQ with the AMQ message store, a directory will automatically created in which the persistent messages are stored. The AMQ message store directory contains sub-directories for all the brokers that are running on the machine. It is for this reason that it is strongly recommended that each broker use a unique name. In the default configuration for ActiveMQ, the broker name is *localhost* which needs to changed to something unique. This directory structure is represented in Figure 5.4 - the AMQ Store directory structure.

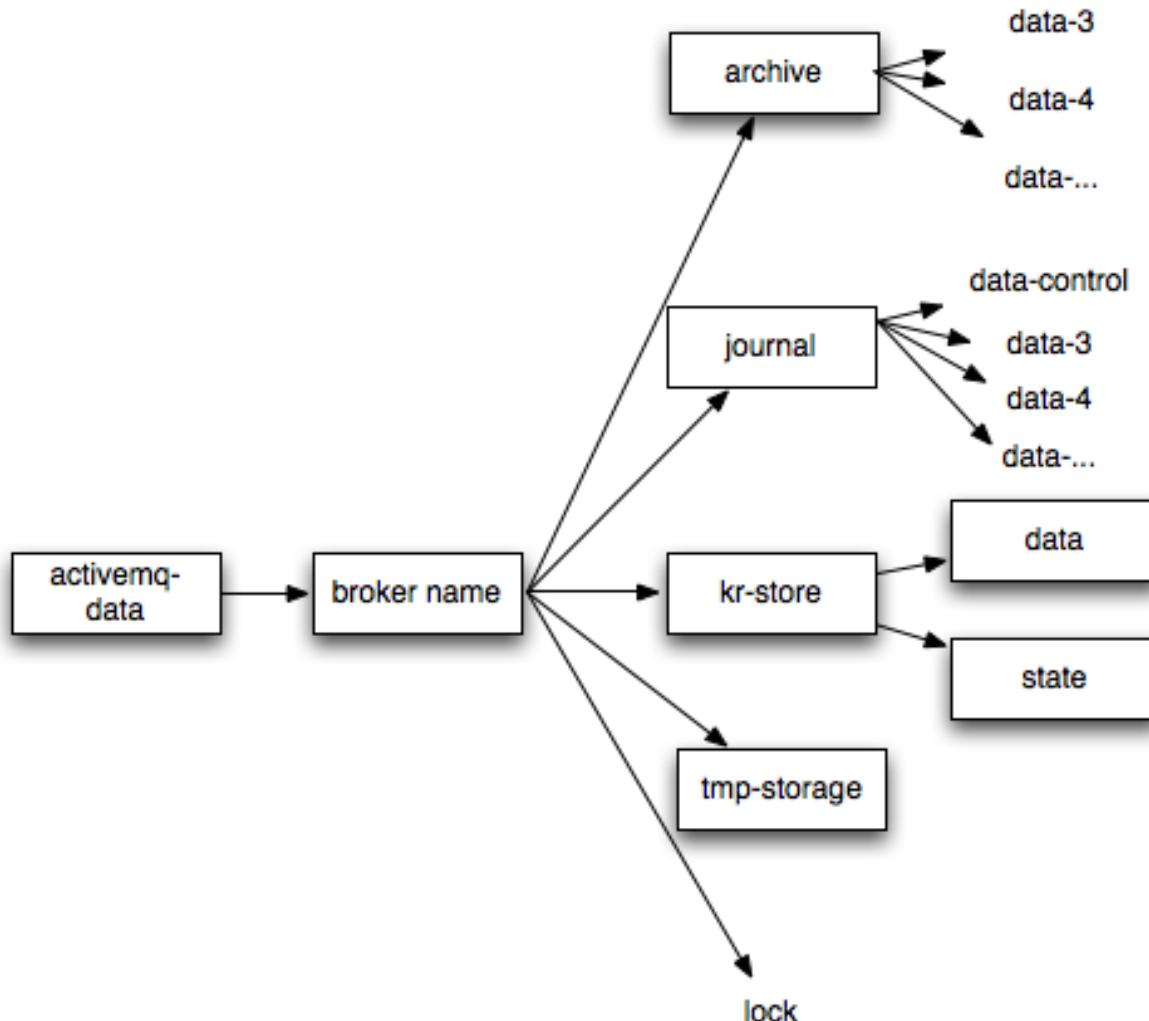


Figure 5.4. The AMQ message store directory structure

Inside of the directory for a given broker, the following items will be found:

- **The data directory** contains the indexes and collections used to reference the messages held in the journal. This data directory is deleted and rebuilt as part of recovery, if the broker has not shut down cleanly. You can force recovery by manually deleting this directory before starting the broker.
- **The state directory** holds information about durable topic consumers. The

journal itself does not hold information about consumers, so when it is recovered it has to retrieve information about the durable subscribers first to accurately rebuild its database.

- **A lock file** to ensure only one broker can access this data at any given time. The lock is commonly used for hot stand-by purposes where more than one broker with the same name will exist on the same system yet only one broker will be able to acquire the lock and start up, while the other broker(s) remain in stand-by mode.
- **A temp-storage directory** is used for storing non-persistent messages that can no longer be stored in broker memory. These messages are typically awaiting delivery to a slow consumer.
- **The kr-store** is the directory structure used by the reference (index) part of the AMQ message store. It uses the Kaha database by default (Kaha is part of the ActiveMQ core library) to index and store references to messages in the journal. There are two distinct parts to the kr-store:
 - **The journal directory** contains the data files for the journal, and a data-control file which holds some meta information. The data files are reference counted, so when all the contained messages are delivered, the data file can be deleted or archived.
 - **The archive directory** exists only if archiving is enabled. Its default location can be found next to the journal. It makes sense, however, to use a separate partition or disk. The archive is used to store data logs from the journal directory which are moved here instead of being deleted. This makes it possible to replay messages from the archive at a later point. To replay messages, move the archived data logs (or a subset) to a new journal directory and start a new broker pointed to the location of this directory. It will automatically replay the data logs in the journal.

Now that the basics of the AMQ message store have been covered, the next step is to review its configuration.

5.2.1.3. Configuring the AMQ Message Store

The AMQ message store is highly configurable and allows the user to change its basic behavior around indexing, checkpoint intervals and the size of the journal data files. These items and many more can be customized through the use of properties. The key properties for the AMQ store are shown in Table 5.1, “”.

Table 5.1.

Property Name	Default Value	Description
directory	activemq-data	The directory path used by the AMQ message store
useNIO	true	NIO provides faster write through to the systems disks
syncOnWrite	false	Sync every write to disk
syncOnTransaction	true	Sync every transaction to disk
maxFileLength	32mb	The maximum size of the message journal data files before a new one is used
persistentIndex	true	Persistent indexes are used. If false an in memory HashMap is used
maxCheckpointMessageAddSize	size	The maximum memory used for a transaction before writing to disk
cleanupInterval	3000(ms)	Time before checking which journal data files are still required

Property Name	Default Value	Description
checkpointInterval	20000(ms)	Time before moving cached message ids to the reference store indexes
indexBinSize	1024	The initial number of hash bins to use for indexes
indexMaxBinSize	16384	The maximum number of hash bins to use
directoryArchive	archive	The directory path used by the AMQ Message Store to place archived journal files
archiveDataLogs	false	If true, journal files are moved to the archive instead of being deleted
recoverReferenceStore	true	Recover the reference store if the broker is not shutdown cleanly; this errs on the side of extreme caution
forceRecoverReferenceStore	false	Force a recovery of the reference store

Below is an example of using the properties from Table 5.1, “” in an ActiveMQ XML configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
  <broker xmlns="http://activemq.apache.org/schema/core">
    <persistenceAdapter>
      <amqPersistenceAdapter
        directory="target/Broker2-data/activemq-data"
        syncOnWrite="true"
        indexPageSize="16kb"
        indexMaxBinSize="100"
```

Please post comments or corrections to the [Author Online Forum](#)

```
    maxFileLength="10mb" />
  </persistenceAdapter>
</broker>
</beans>
```

This is but a very small example of a customized configuration for the AMQ store using the available properties.

5.2.1.4. When to Use the AMQ Message Store

The AMQ Message store is the default message store for ActiveMQ. It provides a balance between ease of use and performance. The fact that this store is embedded in the message broker and already configured for the general message bus scenarios makes it the ideal store for general use.

It's combination of a transactional journal for reliable persistence (to survive system crashes), combined with high performance indexes make this store the best option for a stand alone or embedded ActiveMQ message broker.

The reconfigurability of AMQ message store means that it can be tuned for most usage scenarios, from high throughput applications (trading platforms), to storing very large amounts of messages (GPS tracking).

There are more message store options available, the most flexible being the JDBC message store, which we will go through next.

5.2.2. The KahaDB Message Store

The KahaDB message store is a new message store that has been developed to address some of the limitations in the AMQ message store. The AMQ message store uses two separate files for every index (there is one index per destination) and recovery can be slow if the ActiveMQ broker is not shutdown cleanly. The reason for this is that all the indexes need to be rebuilt, which requires the broker to traverse all its message logs.

To overcome these limitations, the KahaDB message store uses a transactional log for its indexes and only uses one index file for all its destinations. It has been used in production environments with active 10,000 connections, each connection

having a separate queue.

The main components of the KahaDB are very similar to that of the AMQ message store, namely:

- A cache
- Reference Indexes
- A message journal

All index file updates are also recorded in a redo log. This ensures that the indexes can be brought back in a consistent state. In addition, KahaDB uses a B-Tree layout for storage as opposed to AMQ message store which keeps all of its indexes in a persistent hash table. As the hash indexes have to be resized from time to time, KahaDB has more consistent performance characteristics.

The difference in architecture is depicted in figure Figure 5.5 below:

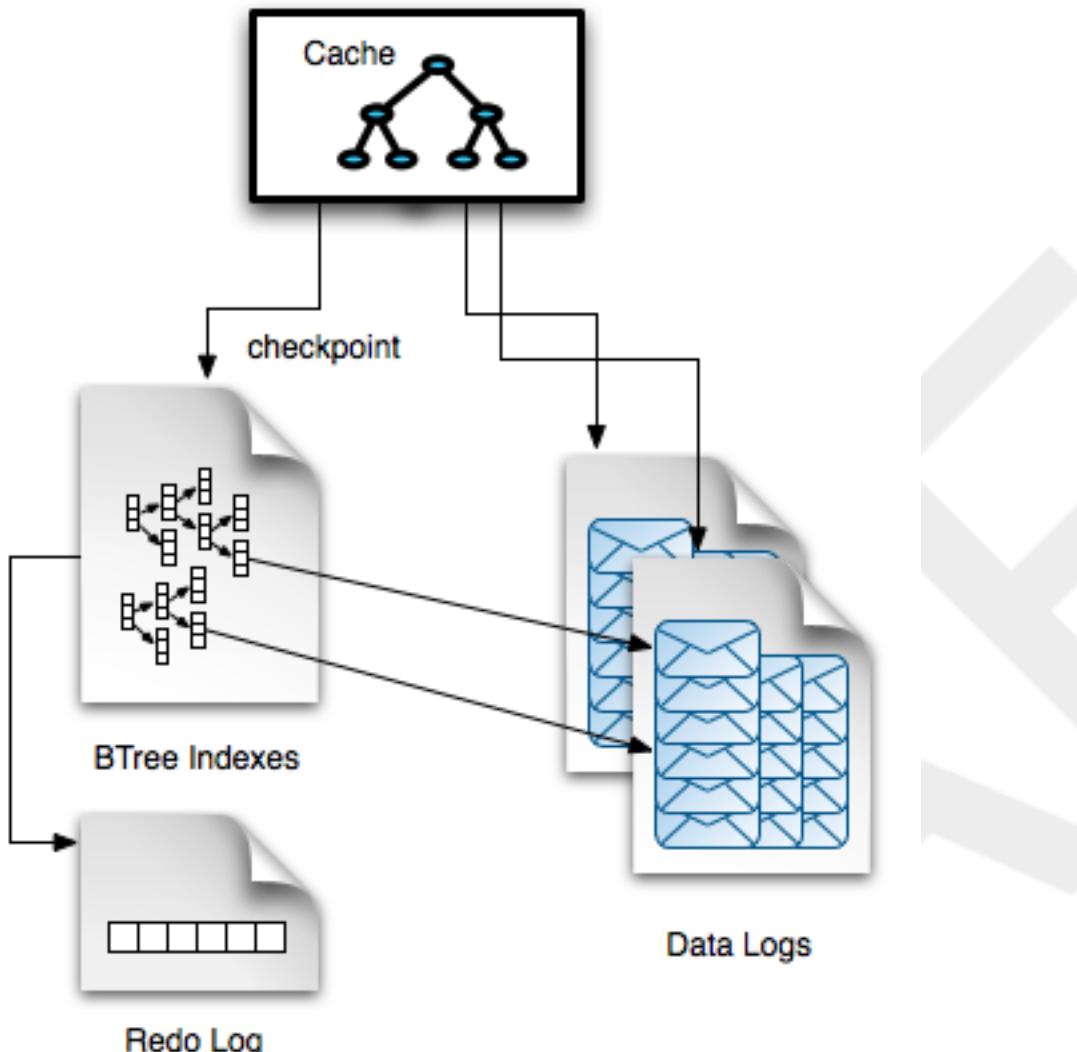


Figure 5.5. The KahaDB message store

The KahaDB store is very straight forward to use as demonstrated in its limited configuration.

5.2.2.1. Configuring the KahaDB Message Store

The KahaDB message store offers fewer options for configuration as shown in Table 5.2, “”.

Table 5.2.

Please post comments or corrections to the [Author Online Forum](#)

property name	default value	description
directory	activemq-data	The directory path used by KahaDB
setIndexWriteBatchSize	1000	The number of indexes to write in a batch to disk
enableIndexWriteAsync	false	If set, will asynchronously write indexes
journalMaxFileLength	32mb	A hint to set the maximum size of the message data logs
maxCheckpointMessageAddSize	1kb	The maximum number of messages to keep in a transaction before automatically committing
cleanupInterval	30000	Time (ms) before checking for a discarding/moving message data logs that are no longer used
checkpointInterval	5000	Time (ms) before checkpointing the journal

Configuration for :

```
<broker brokerName="broker" persistent="true" useShutdownHook="false">
...
<persistenceAdapter>
  <kahaDB directory="activemq-data" journalMaxFileLength="32mb" />
</persistenceAdapter>
...
</broker>
```

5.2.2.2. When to use the KahaDB Store

Please post comments or corrections to the [Author Online Forum](#)

The KahaDB message store can be used in any case where you might use the AMQ message store. If high throughput is required and the number of destinations that will be used in the broker is 500 or less, then the AMQ store may still be the better option.

Please be aware that the KahaDB message store is not compatible with the AMQ message store. There is no conversion tool for moving from one to the other. If you are already using the AMQ message store then stick with it until you get a chance to move over to KahaDB on a new broker.

Although the KahaDB performs better, it's not as common as the JDBC store simply because it is much newer.

5.2.3. The JDBC Message Store

The flexibility of the ActiveMQ pluggable message store API allows for many different implementation choices. The most oldest and more common store implementation uses JDBC for messaging persistence.

When using the JDBC message store the default JDBC driver used in ActiveMQ is Apache Derby. But many other relational databases are supported.

5.2.3.1. Databases supported by the JDBC Message Store

Just about any database with a JDBC driver can be used. Though this is not an exhaustive list, but ActiveMQ has been shown to operate with the following relational databases:

- Apache Derby
- MySQL
- PostgreSQL
- Oracle
- SQLServer

- Sybase
- Informix
- MaxDB

Some users find it useful to use a relational database for message persistence simply because of the ability to query the database to examine messages. The following sections will discuss this topic.

Using Apache Derby With ActiveMQ

As mentioned above, Apache Derby is the default database used with the JDBC store. This is because Derby is a very surprising database. Not only is it written in 100% Java, but it is also designed to be embeddable. Derby offers a very full feature set, performs very well and provides a very small footprint. However, there is one caveat with the use of Derby that should be passed along to ActiveMQ users. Through experience with Derby, it was learned that Derby can be tough on the garbage collector in the JVM. Because so much churn takes place with the storing and deleting of messages in the database, experience has proven that putting Derby in its own JVM instance allows ActiveMQ to perform much better. The reason for this comes down to the fact that ActiveMQ and Derby will no longer be competing for the same JVM resources.

5.2.3.2. The JDBC Message Store Schema

The JDBC message store uses a schema consisting of three tables. Two of the tables are used to hold messages and the third table is used as a lock table to ensure that only one ActiveMQ broker can access the broker at one time. Below is a detailed breakdown of these tables.

The message table by default is named ACTIVEMQ_MSGS defined ???.

Table 5.3.

Please post comments or corrections to the [Author Online Forum](#)

column name	default type	description
ID	INTEGER	The sequence id used to retrieve the message
CONTAINER	VARCHAR(250)	The destination of the message
MSGID_PROD	VARCHAR(250)	The id of the message producer
MSGID_SEQ	INTEGER	The producer sequence number for the message. This together with the MSGID_PROD is equivalent to the JMSMessageID
EXPIRATION	BIGINT	The time in milliseconds when the message will expire
MSG	BLOB	The serialized message itself

Messages are broken down and stored into the ACTIVEMQ_MSGS table for both queues and topics.

There is a separate table for holding durable subscriber information and an id to the last message the durable subscriber received. This information is held in the ACTIVEMQ_ACKS table which is defined in Table 5.4, “”.

Table 5.4.

column name	default type	description
CONTAINER	VARCHAR(250)	The destination
SUB_DEST	VARCHAR(250)	The destination of the

column name	default type	description
		durable subscriber (can be different from the container if using wild cards)
CLIENT_ID	VARCHAR(250)	The clientId of the durable subscriber
SUB_NAME	VARCHAR(250)	The subscriber name of the durable subscriber
SELECTOR	VARCHAR(250)	The selector of the durable subscriber
LAST_ACKED_ID	Integer	The sequence id of last message received by this subscriber

For durable subscribers, the LAST_ACKED_ID sequence is used as a simple pointer into the ACTIVEMQ_MSGS and enables messages for a particular durable subscriber to be easily selected from the ACTIVEMQ_MSGS table.

Now that you understand the database schema, the next task is to walk through some examples of configuring JDBC message stores.

5.2.3.3. Configuring the JDBC Message Store

Its very straightforward to configure the default JDBC message store. As stated previously, the default JDBC store uses Apache Derby in the broker configuration as shown below:

```
<beans>
  <broker brokerName="test-broker" xmlns="http://activemq.apache.org/schema/core">

    <persistenceAdapter>
      <jdbcPersistenceAdapter dataDirectory="activemq-data"/>
    </persistenceAdapter>

  </broker>
</beans>
```

Please post comments or corrections to the [Author Online Forum](#)

One of the key properties on the JDBC persistence adapter (the interface onto the JDBC message store) is the `dataSource` property. This property defines a factory from which connections to a relational database are created. Configuring the `dataSource` object enables the JDBC persistence adaptor to use different physical databases other than the default. Below is an example of an ActiveMQ configuration for the JDBC message store using the MySQL database.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <broker brokerName="test-broker" xmlns="http://activemq.apache.org/schema/core">

        <persistenceAdapter>
            <jdbcPersistenceAdapter dataSource="#mysql-ds" />
        </persistenceAdapter>
    </broker>

    <bean id="mysql-ds" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost/activemq?relaxAutoCommit=true"/>
        <property name="username" value="activemq"/>
        <property name="password" value="activemq"/>
        <property name="maxActive" value="200"/>
        <property name="poolPreparedStatements" value="true"/>
    </bean>

</beans>
```

The example above uses the Apache Commons DBCP `BasicDataSource` to wrap the MySQL JDBC driver for connection pooling.

Just as a point of comparison, below is an example of a configuration to use the Oracle database:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <broker brokerName="test-broker" xmlns="http://activemq.apache.org/schema/core">

        <persistenceAdapter>
            <jdbcPersistenceAdapter dataSource="#oracle-ds" />
        </persistenceAdapter>
    </broker>

    <bean id="oracle-ds" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    </bean>
```

Please post comments or corrections to the [Author Online Forum](#)

```
<property name="url" value="jdbc:oracle:thin:@localhost:1521:AMQDB" />
<property name="username" value="scott" />
<property name="password" value="tiger" />
<property name="maxActive" value="200" />
<property name="poolPreparedStatements" value="true" />
</bean>

</beans>
```

The example above uses the Apache Commons DBCP `BasicDataSource` to wrap the Oracle JDBC driver for connection pooling.

Now that some example configurations for the JDBC message store have been shown, when is it best to use this type of persistence?

5.2.3.4. When to use the JDBC Message Store

The most common reason why the JDBC message store is utilized is because expertise for the administration of a relational database already exists within an organization. JDBC persistence is definitely not superior in performance to the aforementioned message store implementations. The fact of the matter is that many businesses have invested in the use of relational databases so they prefer to make full use of them.

However, the use of a *shared* database is particularly useful for making a redundant master/slave topology out of multiple brokers. When a group of ActiveMQ brokers are configured to use a shared database, they will all try to connect and grab a lock in the lock table, but only one will succeed and become the master. The remaining brokers will be slaves, and will be in a wait state, not accepting client connections until the master fails. This is a common deployment scenario for ActiveMQ.

5.2.3.5. Using the JDBC Message Store With the ActiveMQ Journal

Though the performance of the JDBC message store is not wonderful, it can be improved through the use of the ActiveMQ journal. The journal ensures the consistency of JMS transactions. Because it incorporates very fast message writes with caching technology, it can significantly improve the performance of the ActiveMQ broker.

Below is an example configuration using the journal with JDBC (aka journaled JDBC). In this case, Apache Derby is being used.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <broker brokerName="test-broker" xmlns="http://activemq.apache.org/schema/core">
        <persistenceAdapter>
            <journalJDBC dataDirectory="${activemq.base}/data" dataSource="#derby-ds" />
        </persistenceAdapter>
    </broker>

    <bean id="derby-ds" class="org.apache.derby.jdbc.EmbeddedDataSource">
        <property name="databaseName" value="derbydb" />
        <property name="createDatabase" value="create" />
    </bean>

</beans>
```

The journal can be used with any JDBC datasource, but it's important to know when it should and shouldn't be used.

5.2.3.6. When to Use the JDBC Message Store With the Journal

The journal offers considerable performance advantages over the use of a standard JDBC message store and it is recommended that in general, it is used in conjunction because of this. The only time when it is not possible to use the journal is in a shared database master/slave configuration. As messages from the master may be local in the journal before they have been committed to the database, it means that using the journal in this configuration could lead to lost messages.

The last implementation of the JDBC store is the memory store and it is discussed in the next section.

5.2.4. The Memory Message Store

The Memory Message Store holds all persistent messages in memory. There is no active caching involved, so you have to be careful that both the JVM and the memory limits you set for the broker, are large enough to accommodate all the messages that may exist in this message store at one time.

5.2.4.1. Configuring the Memory Store

Configuration of the memory store is very simple. The memory store is the implementation used when the broker property named persistent is set to false. Below is an example of this:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <broker brokerName="test-broker"
        persistent="false"
        xmlns="http://activemq.apache.org/schema/core">
        <transportConnectors>
            <transportConnector uri="tcp://localhost:61635"/>
        </transportConnectors>
    </broker>
</beans>
```

By setting the persistent attribute on the broker element to false, this effectively tells the broker not to persist messages to long-term storage. Instead, ActiveMQ will hold messages in memory and ...

Embedding an ActiveMQ broker with the memory store is very easy. The following example starts a broker with the memory store:

```
import org.apache.activemq.broker.BrokerService;

public void createEmbeddedBroker() throws Exception {

    BrokerService broker = new BrokerService();
    //configure the broker to use the Memory Store
    broker.setPersistent(false);

    //Add a transport connector
    broker.addConnector("tcp://localhost:61616");

    //now start the broker
    broker.start();
}
```

Notice the bold text above that sets persistence to false on the broker object. This is equivalent to the previous XML configuration example.

5.2.4.2. When to Use the Memory Message Store

The memory message store can be useful if you know that the broker will only store a finite amount of messages, which will typically be consumed very quickly. But it really comes in to its own for small test cases, where you want to prove interaction with a JMS broker, but do not want to incur the cost of a message store start time, or the hassle of cleaning up the message store after the test has finished.

This concludes the discussion of the various message store implementations for message persistent in ActiveMQ. Another topic that bears some discussion regarding message persistence is that of a more specialized case for caching of messages in the ActiveMQ broker for non-durable topic subscribers.

5.3. Caching Messages in the Broker for Consumers

Although one of the most important aspects of message persistence is that the messages will survive in long term storage, there are a number of use cases where messages are required to be available for consumers that were disconnected from the broker, but persisting the messages in a database is too slow. Real-time data delivery of pricing information for a trading platform is a good example use case. But typically real-time data applications use messages that are only valid for a finite amount of time, often less than a minute. So it's rather pointless to persist them to survive a system outage because new messages will arrive very soon.

ActiveMQ supports the caching of messages for these types of systems using message caching in the broker. This configuration uses a configurable policy for deciding which types of messages should be cached, how many and for how long.

5.3.1. How Message Caching for Consumers Works

The ActiveMQ message broker caches messages in memory for every topic that is used. The only types of topics that are not supported are temporary topics and ActiveMQ advisory topics. Caching of messages in this way is not handled for

queues as the normal operation of a queue is to hold every message sent to a queue. Messages that are cached by the broker are only dispatched to a topic consumer if it is retroactive; and never to durable topic subscribers.

Topic consumers are marked as being retroactive by a property set on the destination when the topic consumer is created. Below is an example of this:

```
import org.apache.activemq.ActiveMQConnectionFactory;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.Topic;

public void createRetroactiveConsumer() throws JMSException{
    ConnectionFactory fac = new ActiveMQConnectionFactory();
    Connection connection = fac.createConnection();
    connection.start();

    Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
    Topic topic = session.createTopic("TEST.TOPIC?consumer.retroactive=true");①

    MessageConsumer consumer = session.createConsumer(topic);
}
```

???: Mark that consumers to be retroactive

???: On the broker side, the message caching is controlled by a *Destination Policy* called a *subscriptionRecoveryPolicy*. The default *subscriptionRecoveryPolicy* used in the broker is a *FixedSizedSubscriptionRecoveryPolicy*. The thing to do now is walk through the different *subscriptionRecoveryPolicys* that are available.

5.3.2. The ActiveMQ Subscription Recovery Policies

There are a number of different policies that allow for the fine tuning of the duration and type of messages that are cached for non-durable topic consumers. Each policy type is explained below.

5.3.2.1. The ActiveMQ Fixed Size Subscription Recovery Policy

This policy limits the number of messages cached for the topic based on the amount of memory they use. This is the default subscription recovery policy in ActiveMQ. You can choose to have the cache limit applied to all topics, or on a topic-by-topic basis. The properties available are shown in Table 5.5, “”.

Table 5.5.

property name	default value	description
maximumSize	6553600	The memory size in bytes for this cache
useSharedBuffer	true	If true - the amount of memory allocated will be used across all Topics

5.3.2.2. The ActiveMQ Fixed Count Subscription Recovery Policy

This policy limits the number of messages cached by the topic based on a static count. There is only one property available as listed in Table 5.6, “”.

Table 5.6.

property name	default value	description
maximumSize	100	the number of messages allowed in the Topics cache

5.3.2.3. The ActiveMQ Query Based Subscription Recovery Policy

This policy limits the number of messages cached based on a JMS property selector that is applied to each message. Again, only one property is available as shown in Table 5.7, “”.

Table 5.7.

property name	default value	description
query	null	caches only messages that match the query

5.3.2.4. The ActiveMQ Timed Subscription Recovery Policy

This policy limits the number of messages cached by the topic based on an expiration time that is applied to each message. Note that the expiration time on a message is independent from the timeToLive that set by the `MessageProducer`. See Table 5.8, “” for the one available property.

Table 5.8.

property name	default value	description
recoverDuration	60000	the time in milliseconds to keep messages in the cache

5.3.2.5. The ActiveMQ Last Image Subscription Recovery Policy

This policy holds only the last message sent to a topic. It can be useful for real-time pricing information, whereby a price per topic is used, you might only want the last price that is sent to that topic. There are no configuration properties for this policy.

5.3.2.6. The ActiveMQ No Subscription Recovery Policy

This policy disables message caching for topics. There are no properties to configure for this policy.

5.3.3. Configuring The Subscription Recovery Policy

You can configure the *SubscriptionRecoveryPolicy* for either individual topics, or you can use *wild cards*, in the ActiveMQ broker configuration. An example configuration is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
    <broker brokerName="test-broker"
            persistent="true"
            useShutdownHook="false"
            deleteAllMessagesOnStartup="true"
            xmlns="http://activemq.apache.org/schema/core">
        <transportConnectors>
            <transportConnector uri="tcp://localhost:61635"/>
        </transportConnectors>
        <destinationPolicy>
            <policyMap>
                <policyEntries>
                    <policyEntry topic="Topic.FixedSizedSubs">
                        <subscriptionRecoveryPolicy>
                            <fixedSizeSubscriptionRecoveryPolicy maximumSize="2000000"
                                useSharedBuffer="false"/>
                        </subscriptionRecoveryPolicy>❶
                    </policyEntry>

                    <policyEntry topic="Topic.LastImageSubs">
                        <subscriptionRecoveryPolicy>
                            <lastImageSubscriptionRecoveryPolicy/>
                        </subscriptionRecoveryPolicy>❷
                    </policyEntry>

                    <policyEntry topic="Topic.NoSubs">
                        <subscriptionRecoveryPolicy>
                            <noSubscriptionRecoveryPolicy/>
                        </subscriptionRecoveryPolicy>❸
                    </policyEntry>

                    <policyEntry topic="Topic.TimedSubs">
                        <subscriptionRecoveryPolicy>
                            <timedSubscriptionRecoveryPolicy recoverDuration="25000"/>
                        </subscriptionRecoveryPolicy>❹
                    </policyEntry>

                </policyEntries>
            </policyMap>
        </destinationPolicy>
    </broker>
</beans>
```

Please post comments or corrections to the [Author Online Forum](#)



5.4. Summary

This chapter began by discussing how messages are stored differently for queues and topics. Then the various message store implementations were explained and discussed including their configuration and when to use each. This section provided a good example of the flexibility of message persistence in ActiveMQ.

Finally the special case for caching messages in the broker for non-durable topic consumers was discussed. This section explained why caching is required, when it makes sense to use this feature and the flexibility ActiveMQ provided in configuring the message caches.

Chapter 6. Securing ActiveMQ

Securing access to the message broker and its destinations is a common concern. For this reason, ActiveMQ provides a flexible and customizable security model that can be adapted to the security mechanisms used in your environment.

This chapter will cover the following security topics with ActiveMQ:

- Basic security concepts
- Authentication techniques using simple XML configuration and JAAS
- Authorization techniques
- Message level authorization
- Build a customized security plugin

After reading this chapter, you will be able to secure the broker and integrate it with your existing security infrastructure.

6.1. Introducing Basic Security Concepts

Before a discussion about security with ActiveMQ begins, a brief review of some basic terms related to security and how they fit into the ActiveMQ security model is in order.

Authentication is the process used to verify the integrity of an entity or a user that is requesting access to a secured resource. Some common forms of authentication include plain-text passwords, one time password devices, smart cards or Kerberos just to name a few. ActiveMQ provides simple authentication and JAAS (Java Authentication and Authorization Service) authentication, as well as an API for writing custom authentication plugins. Upon successful authentication, access to the system is granted, but access to perform operations using the system resources may require specific *authorization*.

Please post comments or corrections to the [Author Online Forum](#)

Authorization is the process used to determine the access rights of a user or an entity to a secured resource. Authorization depends upon authentication to prevent unauthorized users from entering the system, but authorization determines if a user has the privileges to perform certain actions. For example, does user X have the necessary permissions to execute program Y on system Z? Such privileges are often referred to as Access Control Lists (ACLs) and determine who or what can access a given resource to perform a given operation. In ActiveMQ, authentication involves restricting access to various operations including the ability to publish to a destination, to consume from a destination, to create a destination or to delete a destination.

Now let's take a look at some practical examples of ActiveMQ security mechanism configurations.

6.2. Authentication

All security concepts in ActiveMQ are implemented as plugins. This allows for easy configuration and customization via the `<plugin>` element of the ActiveMQ XML configuration file. There are two plugins that are available in ActiveMQ to authenticate users:

- Simple authentication plugin - Handles credentials directly in the XML configuration file or in a properties file
- JAAS authentication plugin - Implements the Java Authentication and Authorization Service (JAAS) API and to provide a more powerful and customizable authentication solution

Let's review these two authentication plugins.

6.2.1. Configuring the Simple Authentication Plugin

The easiest way to secure the broker is through the use of authentication credentials placed directly in the broker's XML configuration file. Such functionality is provided by the simple authentication plugin that is part of

Please post comments or corrections to the [Author Online Forum](#)

ActiveMQ. The following snippet provides an example of using the simple authentication plugin:

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

    <transportConnectors>
        <transportConnector name="openwire"
            uri="tcp://localhost:61616" />
    </transportConnectors>
    <plugins>
        <simpleAuthenticationPlugin>
            <users>
                <authenticationUser
                    username="admin"
                    password="password"
                    groups="admins,publishers,consumers"/>
                <authenticationUser
                    username="publisher"
                    password="password"
                    groups="publishers,consumers"/>
                <authenticationUser
                    username="consumer"
                    password="password"
                    groups="consumers"/>
                <authenticationUser
                    username="guest"
                    password="password"
                    groups="guests"/>
            </users>①
        </simpleAuthenticationPlugin>
    </plugins>
</broker>

#A Define Authentication User
```

① Define four authentication users and assign them to four groups

By using this simple configuration snippet, four users can now access ActiveMQ. Obviously, for authentication purposes, each user must have a username and a password. Additionally, the groups attribute provides a comma-separated list of groups to which the user belongs. This information is used for authorization purposes, as will be seen shortly.

The best way to understand this configuration is to use it with the stock portfolio example. First, the broker must be started using the configuration file defined above:

Please post comments or corrections to the [Author Online Forum](#)

```
 ${ACTIVEMQ_HOME}/bin/activemq \
xbean:src/main/resources/org/apache/activemq/book/ch5/activemq-simple.xml
```

Now run the stock publisher and you should see the following exception:

```
...
Exception in thread "main" javax.jms.JMSEException:
  User name or password is invalid.
...
```

The exception above is expected because a security plugin is activated but the authentication credentials have not yet been defined in the publisher client. To fix this exception, modify the publisher to add a username and password. The snippet below provides an example of this:

```
private String username = "publisher";
private String password = "password";

public Publisher() throws JMSEException {
    factory = new ActiveMQConnectionFactory(brokerURL);
    connection = factory.createConnection(username, password);
    connection.start();
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    producer = session.createProducer(null);
}
```

As the snippet above shows, the only necessary change is to define a username and a password that are then used as parameters to the call to the `createConnection()` method. Compiling and running the modified publisher will now yield the proper behavior as shown in the output below:

Notice in the output above...

Unfortunately, with the simple authentication plugin, passwords are stored as clear text which isn't very secure. However, even plain text passwords prevent unauthorized clients from interacting with the broker, and in some environments this is all that is needed. For environments that need a more secure option and/or for environments that already have an existing security infrastructure with which ActiveMQ will need to integrate, the JAAS plugin may be more appropriate.

6.2.2. Configuring the JAAS Plugin

Please post comments or corrections to the [Author Online Forum](#)

A detailed explanation of JAAS is beyond the scope of this book. Instead, this section will briefly introduce the JAAS basic concepts and demonstrate how to create a `PropertiesLoginModule` that can be used to achieve the same functionality as the simple security plugin using JAAS. For more detailed information about JAAS, please refer to the JAAS documentation (<http://java.sun.com/products/jaas/reference/docs/index.html>).

JAAS provides *pluggable authentication* which means ActiveMQ will use the same authentication API regardless of the technique used to verify user credentials (e.g., a text file, a relational database, LDAP, etc.). All that is required is an implementation of the `javax.security.auth.spi.LoginModule` interface (<http://java.sun.com/javase/6/docs/api/javax/security/auth/spi/LoginModule.html>) and a configuration change to ActiveMQ. Fortunately, ActiveMQ comes with implementations of some modules that can authenticate users using properties files, LDAP and SSL certificates which will be enough for many use cases. Because JAAS login modules follow a specification, one advantage of them is that they're relatively straightforward to configure. The best way to understand a login module is by walking through a configuration. For this task, the login module that works with properties files will be used.

The first step in this task is to identify the `PropertiesLoginModule` so that ActiveMQ is made aware of it. To do so, you must create a file named `login.config` that contains a standardized format for configuring JAAS users and groups

(<http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/tutorials/LoginConfigFile.html>)

Below are the contents of the file:

```
activemq-domain {
    org.apache.activemq.jaas.PropertiesLoginModule required
        debug=true
        org.apache.activemq.jaas.properties.user="users.properties"
        org.apache.activemq.jaas.properties.group="groups.properties";
};
```

The `login.config` file shown above contains a few different items for configuring a JAAS module. The `activemq-domain` is the predominant item in this file and it contains all the configuration for the login module. First, is the fully qualified name of the `PropertiesLoginModule` and the trailing notation identifying it as

required. This means that the authentication cannot continue without this login module. Second is a line to enable debug logging for the login module; this is optional. Third, is the `org.apache.activemq.jaas.properties.user` property which points to the `users.properties` file. Fourth, is the `org.apache.activemq.jaas.properties.group` property which points to the `groups.properties` file. Once this is all defined, the two properties files must be created.

Note

The `PropertiesLoginModule` used in this section is an implementation of a JAAS login module and it comes with ActiveMQ.

Defining user credentials in the properties files is simple. The `users.properties` file defines each user in a line-delimited manner along with its password as shown below:

```
admin=password
publisher=password
consumer=password
guest=password
```

The `groups.properties` file defines group names in a line-delimited manner as well. But each group contains a comma-separated list of its users as shown below:

```
admins=admin
publishers=admin,publisher
consumers=admin,publisher,consumer
guests=guest
```

Once these files are created, the JAAS plugin must be defined in the ActiveMQ XML configuration file. Below an example of this necessary change:

```
...
<plugins>
    <jaasAuthenticationPlugin configuration="activemq-domain" />
</plugins>
...
```

The example above is shortened for readability and only shows the necessary change to enable the JAAS login module. As you can see, the JAAS plugin only

needs the name of the JAAS domain in the `login.config` file. ActiveMQ will locate the `login.config` file on the classpath (an alternative to this is to use the `java.security.auth.login.config` system property for the location of the `login.config` file). To test out the JAAS login module that was created above, simply start up ActiveMQ using these changes. Below is the command to use:

```
 ${ACTIVEMQ_HOME}/bin/activemq \
-Djava.security.auth.login.config=\
src/main/resources/org/apache/activemq/book/ch5/login.config \
xbean:src/main/resources/org/apache/activemq/book/ch5/activemq-jaas.xml
```

The broker has been secured just like the previous section where simple authentication was used, only now the JAAS standard was used.

In addition to the ability to authenticate access to the broker services, ActiveMQ also provides the ability to authorize specific operations at a fine-grained level. The next section explores this topic thoroughly.

6.3. Authorization

To build upon authentication, consider a use case requiring more fine-grained control over clients to authorize certain tasks. In most stock trading applications, only specific applications can write to a given destination. After all, you wouldn't want any old random application publishing stock prices to the STOCKS.* destinations. Only an authenticated *and* authorized application should have this ability.

For this reason, ActiveMQ provides two level of authorization, operation level authorization and message level authorization. These two types of authorization provide a more detailed level of control than simple authentication. This section discusses these two types of authorization and walks through some examples to demonstrate each.

6.3.1. Operation Level Authorization

There are three types of user level operations with JMS destinations:

Please post comments or corrections to the [Author Online Forum](#)

- Read - The ability to receive messages from the destination
- Write - The ability to send messages to the destination
- Admin - The ability to administer the destination

Through these well-known operations, the ability to perform the operations can be controlled. Using the ActiveMQ XML configuration file, such authorization can be easily defined. Take a look at the following example to add some operation-specific authorization to some destinations:

```
...
<plugins>
  <jaasAuthenticationPlugin configuration="activemq-domain" />
  <authorizationPlugin>
    <map>
      <authorizationMap>
        <authorizationEntries>
          <authorizationEntry topic="">
            read="admins"
            write="admins"
            admin="admins" />
          <authorizationEntry topic="STOCKS.>">
            read="consumers"
            write="publishers"
            admin="publishers" />#
          <authorizationEntry topic="STOCKS.ORCL">
            read="guests" />
          <authorizationEntry topic="ActiveMQ.Advisory.>">
            read="admins,publishers,consumers,guests"
            write="admins,publishers,consumers,guests"
            admin="admins,publishers,consumers,guests" />
          </authorizationEntries>
        </authorizationMap>
      </map>
    </authorizationPlugin>
  </plugins>
...
```

① An authorization entry for the STOCKS.> destinations

In the snippet above, the JAAS authorization plugin has been defined and pointed at the `activemq-domain` configuration in the `login.config` file. It has also been provided with a map of authorization entries. When configuring the map of

authorization entries, the first task is to define the destination to be secured. This is achieved through the use of either a `topic` or a `queue` attribute on the entry. The next task is to declare which users and/or groups have privileges for operations on that destination.

A really handy feature is the ability to define the destination value using wildcards. For example, the `STOCKS.>` means the entry applies to all destinations in the `STOCKS` path recursively. Also, the authorization operations will accept either a single group or comma-separated list of groups as a value.

Considering this explanation, the configuration used in the previous example can be translated as follows:

- Users from the *admins* group have full access to all topics
- *Consumers* can consume and *publishers* can publish to the destinations in the `STOCKS` path
- *Guests* can only consume from the `STOCKS.ORCL` topic

In order to start the broker to test out both the JAAS authentication plugin as well as the authorization entries, use the following command to start the broker:

```
 ${ACTIVEMQ_HOME}/bin/activemq \
 -Djava.security.auth.login.config= \
 src/main/resources/org/apache/activemq/book/ch5/login.config \
 xbean:src/main/resources/org/apache/activemq/book/ch5/activemq-authorization.xml
```

Note the use of the `java.security.auth.login.config` property to point to the `login.config` file. This ensures that ActiveMQ can locate the file for its use.

In order for the consumer to be able to consume messages from the `STOCKS.ORCL` destination, it must be modified to add an appropriate username and password. Below is a snippet showing the necessary changes that should be made to the `???` file:

```
 ...
     private String username = "guest";
     private String password = "password";

     public Consumer() throws JMSEException {
         factory = new ActiveMQConnectionFactory(brokerURL);
```

Please post comments or corrections to the [Author Online Forum](#)

```
connection = factory.createConnection(username, password);
connection.start();
session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
}
...
```

Credentials have been added so that the consumer can create a connection to the broker using appropriate username and password. Now start the publisher using the following command:

```
<INSERT COMMAND HERE>
```

Next, start the consumer...:

```
<INSERT COMMAND HERE>
```

a consumer now and see what happens when it tries to access particular destinations. For example, if you instruct it to consume messages from STOCKS.CSCO topic, you will get the following exception

```
Exception in thread "main" javax.jms.JMSEException:
User guest is not authorized to read from: topic://STOCKS.CSCO
```

which is exactly what is expected to happen. Consuming from STOCKS.ORCL topic will be performed without any restrictions.

Authorizing operations is just one of the types of authorization that is available in ActiveMQ. Message level authorization will be explored next.

6.3.2. Message Level Authorization

So far in this chapter broker level authentication and authorization has been covered. But as you could see, authorization was granted or denied in the process of creating a connection to the broker. In some situations you might want to authorize access to only particular messages in a destination. In this section, such message level authorization will be examined.

We will implement a simple authorization plugin that allows only applications running on the same host as the broker (i.e., the localhost) to consume messages.

The first thing we need to do is to create an implementation of the `org.apache.activemq.security.MessageAuthorizationPolicy` interface.

```
public class AuthorizationPolicy implements MessageAuthorizationPolicy {  
  
    private static final Log LOG = LogFactory.getLog(AuthorizationPolicy.class);  
  
    public boolean isAllowedToConsume (ConnectionContext context, Message message) {❶  
        LOG.info(context.getConnection().getRemoteAddress());  
        if (context.getConnection().getRemoteAddress().startsWith("/127.0.0.1")) {  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

❶ A method for handling message level authentication

As you can see, the `MessageAuthorizationPolicy` interface is simple and defines only one method named `isAllowedToConsume()`. This method has access to the message in question and the context of the connection in which the message will be consumed. In this example, the remote address property for a connection is used (via the call to the `Connection.getRemoteAddress()` method) to distinguish a remote consumer from a local consumer. The `isAllowedToConsume()` method then determines if the read operation is allowed for the given consumer. Of course, this implementation is arbitrary. You can use any message property or even some message content to make the determination. The implementation of this method is meant to be a very simple example.

Now this policy must be installed and configured in the ActiveMQ broker. The first and most obvious step is to compile this class and package it in the appropriate JAR. Place this JAR into the `lib/` directory of the ActiveMQ distribution and the policy is ready to be used. Second, the policy must be configured to create an instance of the `AuthorizationPolicy` class in the ActiveMQ XML configuration file. Using the Spring beans style XML inside the `<messageAuthorizationPolicy>` element, the `AuthorizationPolicy` class is instantiated when the broker starts up. Below is an example of this configuration:

```
...
```

Please post comments or corrections to the [Author Online Forum](#)

```
<messageAuthorizationPolicy>
  <bean class="org.apache.activemq.book.ch5.AuthorizationPolicy" xmlns="" />
</messageAuthorizationPolicy>
...
```

The only step left is to start up ActiveMQ and test out the new policy. Below is the command to start up the broker using the appropriate configuration file:

```
 ${ACTIVEMQ_HOME}/bin/activemq \
xbean:src/main/resources/org/apache/activemq/book/ch5/activemq-policy.xml
```

If you run chapter 2 examples now on the host on which your broker is running, you'll see that everything works in the same manner as it was with the original configuration. But, when run from another host, messages will be published normally, while no messages will be consumed.

Message level authorization provides some very powerful functionality with endless possibilities. Although a very simple example was used here, anything is possible. Just bear in mind that a message authorization policy is executed for every message that flows through the broker. So be careful not to add functionality that could possibly slow down the flow of messages.

In addition to authorization, ActiveMQ provides a special class for tighter control over broker level operations that is even more powerful. The next section examines and demonstrates just such an example.

6.4. Broker Level Operations

So far this chapter has focused on the built-in security features in ActiveMQ. While these features should provide enough functionality for the majority of users, there's an even more powerful feature available. As stated previously, the ActiveMQ plugin API is extremely flexible and the possibilities are endless. The flexibility in this functionality comes from the `BrokerFilter` class in ActiveMQ. This class provides the ability to intercept many of the available broker level operations. Broker operations includes such items as adding consumers and producers to the broker, committing transactions in the broker, adding and removing connections to the broker just to name a few. Custom functionality can

be added by extending the `BrokerFilter` class and overriding a method for a given operation. For a deeper look at the `BrokerFilter` class, see Chapter 10.

While the ActiveMQ plugin API is not concerned solely with security, implementing a class whose main purpose is to handle a custom security feature is quite achievable. So if you have security requirements that cannot be met using the previous security features, you may want to consider developing a custom solution for your needs. Depending on your needs, there are two choices available:

- Implement a JAAS login module - There is a good chance that you are already using JAAS in your Java applications. In this case, it's only natural that you'll try to reuse all that work for securing ActiveMQ broker, too. Since JAAS is not the main topic of this book, we will not dive any deeper into this topic than we already have.
- Implement a custom plugin for handling security - ActiveMQ provides a very flexible generic plugin mechanism. You can create your own custom plugins for just about anything including the creation of a custom security plugin. So if you have requirements that cannot be met by implementing a JAAS module, writing a custom plugin is the way to go.

In this section we will describe how to write a simple security plugin that authorizes broker connections only from a certain set of IP addresses.

6.4.1. Building A Custom Security Plugin

Using the ActiveMQ plugin API, a custom `BrokerFilter` implementation will be created in this section. This custom plugin will be used to limit connectivity to ActiveMQ based on IP address. The concept is not complex but is good enough to give you a taste of the `BrokerFilter` with an angle toward security.

In order to limit connectivity to the broker based on IP address, a class named `IPAuthenticationBroker` will be created to override the `BrokerFilter.addConnection()` method. The implementation of this method will perform a simple check of the IP address using a regular expression to determine the ability to connect. Below is the implementation of the

Please post comments or corrections to the [Author Online Forum](#)

IPAuthenticationBroker class:

```
public class IPAuthenticationBroker extends BrokerFilter {  
  
    List<String> allowedIPAddresses;  
    Pattern pattern = Pattern.compile("^(?![0-9\\.]*):(.*)$");  
  
    public IPAuthenticationBroker(Broker next, List<String> allowedIPAddresses) {  
        super(next);  
        this.allowedIPAddresses = allowedIPAddresses;  
    }  
  
    public void addConnection(ConnectionString context, ConnectionInfo info)  
        throws Exception { ①  
  
        String remoteAddress = context.getConnection().getRemoteAddress();  
  
        Matcher matcher = pattern.matcher(remoteAddress);  
        if (matcher.matches()) {  
            String ip = matcher.group(1);  
            if (!allowedIPAddresses.contains(ip)) {  
                throw new SecurityException(  
                    "Connecting from IP address " + ip + " is not allowed"  
                );  
            }  
        } else {  
            throw new SecurityException("Invalid remote address " + remoteAddress);  
        }  
  
        super.addConnection(context, info);  
    }  
}
```

- ① The `BrokerFilter.addConnection()` method is being overridden to provide the ability to filter based on IP address

The `BrokerFilter` class defines methods that intercept broker operations such as adding a connection, removing a subscriber, etc. For more information on the `BrokerFilter` class, see Chapter 10. In the `IPAuthenticationBroker` class above, the `addConnection()` method is overridden to create some logic that checks if the address of a connecting client falls within a list of IP addresses that are allowed to connect. If that IP address is allowed to connect, the call is delegated to the `BrokerFilter.addConnection()` method. If that IP address is not allowed to connect, an exception is thrown.

One additional item of note in the `IPAuthenticationBroker` class is that its constructor calls the `BrokerFilter`'s constructor. This call serves to set up the chain of interceptors so that the proper cascading will take place through the chain. Don't forget to do this if you create your own `BrokerFilter` implementation.

After the actual plugin logic has been implemented, the plugin must be configured and installed. For this purpose, an implementation of the `BrokerPlugin` will be created. The `BrokerPlugin` is used to expose the configuration of a plugin and also installs the plugin into the ActiveMQ broker. In order to configure and install the `IPAuthenticationBroker`, the `IPAuthenticationPlugin` class is created as shown below:

```
public class IPAuthenticationPlugin implements BrokerPlugin {  
  
    List<String> allowedIPAddresses;  
  
    public Broker installPlugin(Broker broker) throws Exception {❶  
        return new IPAuthenticationBroker(broker, allowedIPAddresses);  
    }  
  
    public List<String> getAllowedIPAddresses() {  
        return allowedIPAddresses;  
    }  
  
    public void setAllowedIPAddresses(List<String> allowedIPAddresses) {  
        this.allowedIPAddresses = allowedIPAddresses;  
    }  
}
```

- ❶ The `installPlugin()` method implementation is used to create an instance of the custom `IPAuthenticationBroker` class

The `IPAuthenticationBroker.installPlugin()` method is used to instantiate the plugin and return a new intercepted broker for the next plugin in the chain. Notice that the `IPAuthenticationPlugin` class also contains getter and setter methods used to configure the `IPAuthenticationBroker`. These setter and getter methods are then available via a Spring beans style XML configuration in the ActiveMQ XML configuration file. Below is an example of how the `IPAuthenticationPlugin` class is used for configuration:

```
<broker xmlns="http://activemq.org/config/1.0"
```

Please post comments or corrections to the [Author Online Forum](#)

```
brokerName="localhost"
dataDirectory="${activemq.base}/data"
plugins="#ipAuthenticationPlugin"①

<transportConnectors>
<transportConnector name="openwire"
uri="tcp://localhost:61616" />
</transportConnectors>
</broker>

<bean id="ipAuthenticationPlugin"
class="org.apache.activemq.book.ch5.IPAAuthenticationPlugin">
<property name="allowedIPAddresses">
<list>
<value>127.0.0.1</value>
</list>
</property>
</bean>②
```

① The `<broker>` element supports the `plugins` attribute as a shortcut

② The Spring beans style configuration of the `IPAAuthenticationPlugin`

The `<broker>` element provides the `plugins` attribute as a shortcut to declaring all plugins that do not utilize an XBean style of configuration and, therefore, cannot be configured inside the `<plugins>` element (more on this topic in Chapter 10).

Note that the `id` of the bean is used to refer to its definition from the `plugins` attribute. Using this configuration, only those clients connecting from the IP address `127.0.0.1` (i.e., the localhost) can actually connect to the broker.

All that needs to be done now is to test the plugin. Below is the command to start up ActiveMQ using the `IPAAuthenticationPlugin` and the `IPAAuthenticationBroker`:

```
 ${ACTIVEMQ_HOME}/bin/activemq \
xbean:src/main/resources/org/apache/activemq/book/ch5/activemq-custom.xml
```

Now run the client to connect to ActiveMQ from the localhost and everything should be working fine. See the output below

If a connection attempt is made from host other than the localhost you can expect to see the following output including the exception:

```
Exception in thread "main" javax.jms.JMSEException:
Connecting from IP address 127.0.0.1 is not allowed
```

Although this example was a bit more complex, it serves as a good demonstration of the power provided by the `BrokerFilter` class. Just imagine how flexible this plugin mechanism is for integrating with existing custom security requirements. This example focused on a security example, but there are many, many other operations that can be customized by using the pattern illustrated here.

6.5. Summary

In this chapter, the ActiveMQ broker was secured from non-authenticated and non-authorized access. For the most simple purposes, you can use the ActiveMQ simple authentication plugin, allowing you to define security credentials directly into the configuration file. The ActiveMQ JAAS plugins provide the ability to utilize the standardized Java login modules via simple configuration, allowing you to authenticate users from various sources, such as LDAP, properties files and so on. Additionally, custom JAAS login modules could be created for use with other authentication or authorization schemes such as Kerberos, NTLM, NIS, etc.

Operation level authorization was also demonstrated for more fine-grained control over destinations. Next was Message level authorization by creating a custom policy to control consumption of a given message. Finally, the ActiveMQ plugin mechanism was demonstrated briefly through the customized IP-based authentication example.

ActiveMQ provides some powerful security mechanisms as demonstrated in this chapter. Hopefully the process for utilizing these solutions is more clear after walking through the examples.

Part III. Using ActiveMQ to Build Messaging Applications

Now that you have the basics under your belt, it's time to start building applications that utilize ActiveMQ. The asynchronous nature of messaging tends to be foreign to most developers because they're used to using synchronous calls in the applications they build. Using asynchronous style calls requires in your application designs requires a different style of thinking.

Part III explores the topic building applications that utilize messaging with ActiveMQ. Not only are examples provided that use the Java language, but there are examples of connecting to ActiveMQ using other languages as well.

Chapter 7. Creating Java Applications With ActiveMQ

Thus far we were mainly focusing on the ActiveMQ broker as a standalone middleware component used to exchange messages between Java applications. But as was described in Chapter 3, ActiveMQ supports several *in virtual machine* connectors used to communicate with embedded brokers. Being a Java application itself, ActiveMQ could be naturally integrated into other Java applications.

No matter whether you are planning to use your broker only from the same virtual machine (application) or you are planning to allow other applications to send and receive messages from it over the network, embedded brokers support both of these use cases and allow you to have one less Java application to administer. In this chapter we will explore various techniques available for embedding ActiveMQ broker in your Java applications. Also, we will cover in details integration of both brokers and clients in the [Spring Framework](#)

7.1. Integrating Broker

In this section we will go first through basic set of classes used to initialize and configure ActiveMQ broker. Next, we will describe how you can configure your broker using custom configuration XML file. Finally, we will see what options the Spring framework gives developers wanting to initialize the broker as a Spring bean.

7.1.1. Embedding The Broker

7.1.1.1. Broker Service

If you are planning to use plain Java code to set up your broker, the `org.apache.activemq.broker.BrokerService` class should be your starting point. This class is used to configure broker and manage its life cycle. The best way to

Please post comments or corrections to the [Author Online Forum](#)

demonstrate the usage of `BrokerService` class is, of course, with an appropriate example. Let's start with a broker configuration we used in Chapter 5, Securing ActiveMQ, to configure simple authentication plugin and see how we can achieve the same functionality with plain Java code. For starters, let's take a look at this well known XML configuration example:

Example 7.1. Listing 6.1: Example XML configuration

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

  <!-- The transport connectors ActiveMQ will listen to -->
  <transportConnectors>
    <transportConnector name="openwire"
      uri="tcp://localhost:61616" />
  </transportConnectors>
  <plugins>
    <simpleAuthenticationPlugin>
      <users>
        <authenticationUser username="admin" password="password" groups="admins,publishers,consumers" />
        <authenticationUser username="publisher" password="password" groups="publishers,consumers" />
        <authenticationUser username="consumer" password="password" groups="consumers" />
        <authenticationUser username="guest" password="password" groups="guests" />
      </users>
    </simpleAuthenticationPlugin>
  </plugins>
</broker>
```

Here, we have defined a broker with a few properties set, such as name and data directory, one transport connector and one plugin.

Now take a look at the following Java application:

Example 7.2. Listing 6.2: Broker service example

```
public static void main(String[] args) throws Exception {
  BrokerService broker = new BrokerService(); #A
  broker.setBrokerName("localhost"); #A
  broker.setDataDirectory("data/"); #A

  SimpleAuthenticationPlugin authentication =
    new SimpleAuthenticationPlugin();

  List<AuthenticationUser> users =
    new ArrayList<AuthenticationUser>();
```

Please post comments or corrections to the [Author Online Forum](#)

```
users.add(new AuthenticationUser("admin",
                                  "password",
                                  "admins,publishers,consumers"));
users.add(new AuthenticationUser("publisher",
                                  "password",
                                  "publishers,consumers"));
users.add(new AuthenticationUser("consumer",
                                  "password",
                                  "consumers"));
users.add(new AuthenticationUser("guest",
                                  "password",
                                  "guests"));
authentication.setUsers(users);

broker.setPlugins(new BrokerPlugin[]{authentication});          #B

broker.addConnector("tcp://localhost:61616");                  #C

broker.start();                                                 #D
}

#A Instantiate and configure Broker Service
#B Add plugins
#C Add connectors
#D Start broker
```

As you can see, the broker is instantiated by calling the appropriate constructor of the `BrokerService` class. All configurable broker properties are mapped to the properties of this class. Additionally, you can initialize and set appropriate plugins using the `setPlugins()` method, like it is demonstrated for the simple authentication plugin. Finally, you should add connectors you wish to use and start the broker by `addConnector()` and `start()` methods respectively. And there you are, your broker is fully initialized with using just plain Java code, no XML configuration files were used. One important thing to note here is that you should always add your plugins before connectors as they will not be initialized otherwise. Also, all connectors added after the broker has been started will not be properly started.

7.1.1.2. Broker Factory

The `BrokerService` class, described in the previous section, is useful when you want to keep configuration of your code in plain Java code. This method is useful for simple use cases and situation where you don't need to have customizable

configuration for your broker. In many applications, however, you'll want to be able to initialize broker from the same configuration files used to configure standalone instances of the ActiveMQ broker.

For that purpose ActiveMQ provides utility

`org.apache.activemq.broker.BrokerFactory` class. It is used to create an instance of the `BrokerService` class configured from an external XML configuration file. Now, let's see how we can instantiate the `BrokerService` class from the XML configuration file shown in Listing 6.1 using `BrokerFactory`:

Example 7.3. Listing 6.3: Broker factory example

```
public class Factory {  
  
    public static void main(String[] args) throws Exception {  
        System.setProperty("activemq.base", System.getProperty("user.dir"));  
        BrokerService broker = BrokerFactory.createBroker(new URI(  
            "xbean:src/main/resources/org/apache/activemq/book/ch5/activemq-simple.xml"  
        ));  
        broker.start();  
    }  
  
}  
  
#A Creating broker from XML
```

#A
#A
#A

As you can see, the `BrokerFactory` class has the `createBroker()` method which takes a configuration URI as the parameter. In this particular example, we have used `xbean:` URI scheme, which means that broker factory will search for the given XML configuration file in the classpath. If you want to specify XML configuration file not located on your application's classpath you can use the `file:` URI scheme, like:

```
file:/etc/activemq/activemq.xml
```

Finally, you can use the `broker:` URI scheme for simple broker configuration performed completely via configuration URI. Take a look at the following URI for example:

```
broker:(tcp://localhost:61616, network:static:tcp://remotehost:61616)?persistent=false&useJms
```

This single URI contains complete configuration of the broker, including connectors (both transport and network ones) and properties such as persistence and JMX exposure. For a complete reference of the broker URI please take a look at the following address: <http://activemq.apache.org/broker-uri.html>

7.1.2. Integrating With Spring Framework

If you are writing your Java application and want to embed ActiveMQ in it (or just want to send and receive messages), it is highly likely that you are going to use some of the integration frameworks available for Java developers. As one of the most popular integration frameworks in Java community, Spring framework (<http://www.springframework.org>) plays important role in many Java projects. In this section we cover ActiveMQ and Spring integration on both broker and client sides.

7.1.2.1. Integrating The Broker

ActiveMQ broker is developed with Spring in mind (using it for its internal configuration needs), which makes it really easy to embed the broker in Spring-enabled applications. All you have to do is to define a broker bean (as we will see in the moment) in your spring configuration file and initialize the appropriate XML application context. Let's take a look at the following simple Java application:

Example 7.4. Listing 6.1: Integrating ActiveMQ Broker with Spring

```
package org.apache.activemq.book.ch6.spring;

import org.apache.activemq.book.ch5.Publisher;
import org.apache.xbean.spring.context.FileSystemXmlApplicationContext;

public class SpringBroker {

    public static void main(String[] args) throws Exception {
        if (args.length == 0) {
            System.err.println("Please define a configuration file!");
            return;
    }}
```

Please post comments or corrections to the [Author Online Forum](#)

```

String config = args[0];                                #A
System.out.println(
    "Starting broker with the following configuration: " + config
);
System.setProperty("activemq.base", System.getProperty("user.dir")); #B
FileSystemXmlApplicationContext                         #C
    context = new FileSystemXmlApplicationContext(config);      #C

Publisher publisher = new Publisher();                  #D
for (int i = 0; i < 100; i++) {                        #D
    publisher.sendMessage(new String[] {"JAVA", "IONA"}); #D
}

}

}

#A Define configuration file
#B Set base property
#C Initialize application context
#D Send messages

```

In the previous example, we defined an XML configuration file we are going to use, set `activemq.base` system property used in our configuration and instantiated a spring application context from a previously defined XML configuration file. And that's it, your job is done, everything else is done by ActiveMQ and Spring framework, so you can use the `Publisher` class to send stock prices to the running broker. You can also notice, that we have used application context class from the apache xbean project (<http://geronimo.apache.org/xbean/>) used heavily by ActiveMQ. We will discuss XBean in more details soon, but the important thing at this moment is that you have to include it in your classpath in order to successfully execute these examples. The simplest way to do it is by using Maven and adding the following dependency to your project:

```

<dependency>
    <groupId>org.apache.xbean</groupId>
    <artifactId>xbean-spring</artifactId>
    <version>3.3</version>
</dependency>

```

Depending on the version of Spring you are using in your project, you have various options regarding how your configuration file will look like. We will explore these options in more details in the following sections.

7.1.2.1.1. Spring 1.0

In its 1.x versions, Spring framework didn't support custom namespaces and syntaxes in your configuration files. So, Spring XML files were just a set of bean declarations with their properties and references to other beans. It is a common practice for developers of server-side software, like ActiveMQ, to provide so called factory beans for their projects. The purpose of these classes is to provide a configuration mechanism in a bean-like XML syntax. The `org.apache.activemq.xbean.BrokerFactoryBean` class does this job for ActiveMQ. So if you are using Spring 1.x for your project, you can configure a broker using the `BrokerFactoryBean` class, like it is shown in the following example:

Example 7.5. Listing 6.2: Spring 1.0 configuration

```
<beans>
  <bean id="broker"
    class="org.apache.activemq.xbean.BrokerFactoryBean">
    <property name="config"
      value="file:src/main/resources/org/apache/activemq/book/ch5/activemq-simple.xml" /> #1
    <property name="start" value="true" />
  </bean>
</beans>
```

As you can see, there is only one important configuration parameter for this bean, called `config` #1, which is used to point to the standard ActiveMQ XML configuration file we described in earlier chapters. In this particular example, we have pointed our bean to a configuration we have used in Chapter 5, Securing ActiveMQ, for defining a simple authentication plugin.

Another interesting property is `start` #2, which instructs the factory bean to start a broker after its initialization. Optionally, you can disable this feature and start broker manually if you wish.

Now you can run our example shown in Listing 6.1 as follows:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch6.spring.SpringBroker \
-Dlog4j.configuration=file:src/main/java/log4j.properties \
-Dexec.args="src/main/resources/org/apache/activemq/book/ch6/spring-1.0.xml"
```

You should see your broker started and hundred of messages (with stock prices) sent to it.

7.1.2.1.2. Spring 2.0

Since version 2.0, Spring framework allows developers to create and register custom XML schemes for their projects. The idea of having custom XML for configuring different libraries (or projects in general) is to ease configuration by making more human-readable XML. ActiveMQ provides custom XML schema you can use to configure ActiveMQ directly in your Spring configuration file. The example in Listing 6.3 demonstrates how to configure ActiveMQ using custom Spring 2.0 schema.

Example 7.6. Listing 6.3 Spring 2.0 configuration

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:amq="http://activemq.org/config/1.0" #1
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd#
    http://activemq.org/config/1.0 #2
    http://activemq.apache.org/schema/activemq-core.xsd"> #2

    <amq:broker
        brokerName="localhost" dataDirectory="${activemq.base}/data">

        <!-- The transport connectors ActiveMQ will listen to -->
        <amq:transportConnectors>
            <amq:transportConnector name="openwire"
                uri="tcp://localhost:61616" />
        </amq:transportConnectors>
        <amq:plugins>
            <amq:simpleAuthenticationPlugin>
                <amq:users>
                    <amq:authenticationUser username="admin"
                        password="password"
                        groups="admins,publishers,consumers"/>
                    <amq:authenticationUser username="publisher"
                        password="password"
                        groups="publishers,consumers"/>
                    <amq:authenticationUser username="consumer"
                        password="password"
                        groups="consumers"/>
                    <amq:authenticationUser username="guest"
                        password="password"
                        groups="guests"/>
                </amq:users>
            </amq:simpleAuthenticationPlugin>
        </amq:plugins>
    </amq:broker>
```

Please post comments or corrections to the [Author Online Forum](#)

```
</amq:simpleAuthenticationPlugin>
</amq:plugins>
</amq:broker>

</beans>
```

As you can see in the previous example, the first thing we have to do is define a schema that we want to use. For that we have to first specify the prefix we are going to use for ActiveMQ related beans #1 (`amq` in this example) and the location of this particular schema #2.

After these steps, we are free to define our broker-related beans using the customized XML syntax. In this particular example we have configured the broker as it was configured in our previously used Chapter 5 example, with simple authentication plugin.

Now we can start the Listing 6.1 example with this configuration as:

```
$ mvn -e exec:java -Dexec.mainClass=org.apache.activemq.book.ch6.spring.SpringBroker \
-Dlog4j.configuration=file:src/main/java/log4j.properties \
-Dexec.args="src/main/resources/org/apache/activemq/book/ch6/spring-2.0.xml"
```

and expect the same behavior as before.

7.1.2.1.3. XBean

As we said before, ActiveMQ uses Spring and XBean for its internal configuration purposes. XBean provides you with the same ability to define and use custom XML for your projects, just in a bit different way. All `activemq.xml` files we used in previous chapters to configure various features of ActiveMQ are basically Spring configuration files, powered by XBean custom XML schema.

Since we are using customized XBean application context class in our example, there is nothing to stop us from running our example from Listing 6.3 with one of the configuration files we used in previous chapters. For example, the following command:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch6.spring.SpringBroker \
-Dlog4j.configuration=file:src/main/java/log4j.properties \
-Dexec.args="src/main/resources/org/apache/activemq/book/ch5/activemq-simple.xml"
```

will start a broker configured with our original example used in chapter 5, Securing ActiveMQ, and will behave the same as our previously defined examples.

7.1.2.2. Integrating Clients

In the previous section we have seen various mechanisms for embedding brokers into Java Spring-enabled applications. In this section we will provide more details on how Spring can help us write our clients.

ActiveMQ provides good Spring integration for configuring various aspects of client to broker communication and Spring framework, on the other hand, comes with support for easier JMS messaging. Together ActiveMQ and Spring make an excellent JMS development platform, making many common tasks easy to accomplish. Some of those tasks we will cover are:

- Defining connection factory - ActiveMQ provide bean classes that could be used to configure URLs and other parameters of connections to brokers. The connection factory could later be used by your application to get the appropriate connection.
- Defining destinations - ActiveMQ destination classes could be also configured as beans representing JMS destinations used by your producers and consumers
- Defining consumers - Spring JMS support provides helper bean classes that allows you to easily configure new consumers
- Defining producers - Spring also provides helper bean classes for creating new producers

In the following sections, we will go through all of these tasks and use those pieces to rewrite our portfolio application to use all benefits of the ActiveMQ Spring integration.

7.1.2.2.1. Defining Connections

As we have seen in our previous examples, the first step in creating JMS

application is to create an appropriate broker connection. We used `org.apache.activemq.ActiveMQConnectionFactory` class to do this job for us and luckily it could be naturally used in Spring environment. In the following example, we can see how to configure a bean with all important properties, such as broker URL and client credentials.

Example 7.7. Listing 6.4: Spring Connection Factory

```
<bean id="jmsFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
<property name="brokerURL">
  <value>tcp://localhost:61616</value>
</property>
<property name="userName"><value>admin</value></property>
<property name="password"><value>password</value></property>
</bean>
```

As we will see in Section 6.2.2.4 Defining Producers, in certain use cases we need to have a pool of connections in order to achieve desired performances. For that purpose, ActiveMQ provides the `org.apache.activemq.pool.PooledConnectionFactory` class. The example configuration we will use in later examples is shown in Listing 6.5.

Example 7.8. Listing 6.5: Pooled Connection Factory

```
<bean id="pooledJmsFactory" class="org.apache.activemq.pool.PooledConnectionFactory" destroy="<ref local="jmsFactory"/>">
<property name="connectionFactory">
  <ref local="jmsFactory"/>
</property>
</bean>
```

There is only one important property of this bean and that is `connectionFactory`, which defines the actual connection factory that will be used. In this case we have used our previously defined `jmsFactory` bean.

If you plan to use pooled connection factory, you'll have to add an extra dependency to your classpath and that is Apache Commons Pool project. You can find necessary JAR at the project's website (<http://commons.apache.org/pool/>) or if you use Maven for your builds just add something like following:

Please post comments or corrections to the [Author Online Forum](#)

```
<dependency>
<groupId>commons-pool</groupId>
<artifactId>commons-pool</artifactId>
<version>1.4</version>
</dependency>
```

to your project configuration.

7.1.2.2. Defining destinations

In case that you need to predefine destinations you want to use in your application, you can define them using `org.apache.activemq.command.ActiveMQTopic` and `org.apache.activemq.command.ActiveMQQueue` classes. In the following example, you can find definitions of two topics we will use for our portfolio example.

Example 7.9. Listing 6.6: Spring Destinations

```
<bean id="javaDest" class="org.apache.activemq.command.ActiveMQTopic"
  autowire="constructor">
  <constructor-arg value="STOCKS.JAVA" />
</bean>

<bean id="ionaDest" class="org.apache.activemq.command.ActiveMQTopic"
  autowire="constructor">
  <constructor-arg value="STOCKS.IONA" />
</bean>
```

As you can see, these classes use just constructor injection for setting a desired destination name.

7.1.2.3. Defining consumers

Now that we have connections and destinations ready, we can start consuming and producing messages. This is where Spring JMS (<http://static.springframework.org/spring/docs/2.5.x/reference/jms.html>) support shows its value, providing helper beans for easier development. In this and the following sections we will not go into details of Spring JMS since it is out of scope of this book. Instead we will show some of the basic concepts needed to implement

our example. For additional information on Spring JMS support you should consult Spring documentation.

The basic abstraction for receiving messages in Spring is the message listener container. It is practically an intermediary between your message listener and broker, dealing with connections, threading and such, leaving you to worry just about your business logic.

In the following example we will define our portfolio listener used in Chapter 2 and two message listener containers for two destinations defined in the previous section.

Example 7.10. Listing 6.7: Spring Consumers

```
<bean id="portfolioListener" class="org.apache.activemq.book.ch2.portfolio.Listener">
</bean>

<bean id="javaConsumer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
<property name="connectionFactory" ref="jmsFactory"/>
<property name="destination" ref="javaDest" />
<property name="messageListener" ref="portfolioListener" />
</bean>

<bean id="ionaConsumer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
<property name="connectionFactory" ref="jmsFactory"/>
<property name="destination" ref="ionaDest" />
<property name="messageListener" ref="portfolioListener" />
</bean>
```

As you can see, every message listener container needs a connection factory, destination and listener to be used. So all you have to do is to implement your desired listener and leave everything else to Spring. Note that in this examples we have used plain (not pooled) connection factory, since no connection pooling is needed.

In this example we have used the `DefaultMessageListenerContainer` which is commonly used. This container could be dynamically adjusted in terms of concurrent consumers and such. Spring also provides more modest and advanced message containers, so please check its documentation when deciding which one is a perfect fit for your project.

7.1.2.2.4. Defining producers

As it was the case with message consumption, Spring provides a lot of help with producing messages. The crucial abstraction in this problem domain is the `org.springframework.jms.core.JmsTemplate` class (<http://static.springframework.org/spring/docs/2.5.x/api/org/springframework/jms/core/JmsTemplate.html>) which provides helper methods for sending messages (and their synchronous receiving) in many ways.

One of the most common ways to send a message is by using the `org.springframework.jms.core.MessageCreator` interface implementation (<http://static.springframework.org/spring/docs/2.5.x/api/org/springframework/jms/core/MessageCreator.html>) and the appropriate `send()` method of the `JmsTemplate` class. In the following example, we will implement all "message creation" logic from stock portfolio publisher example in the implementation of this interface.

Example 7.11. Listing 6.8: Message Creator

```
public class StockMessageCreator implements MessageCreator {  
  
    private int MAX_DELTA_PERCENT = 1;  
    private Map<Destination, Double> LAST_PRICES = new Hashtable<Destination, Double>();  
  
    Destination stock;  
  
    public StockMessageCreator(Destination stock) {  
        this.stock = stock;  
    }  
  
    public Message createMessage(Session session) throws JMSException {  
        Double value = LAST_PRICES.get(stock);  
        if (value == null) {  
            value = new Double(Math.random() * 100);  
        }  
  
        // lets mutate the value by some percentage  
        double oldPrice = value.doubleValue();  
        value = new Double(mutatePrice(oldPrice));  
        LAST_PRICES.put(stock, value);  
        double price = value.doubleValue();  
  
        double offer = price * 1.001;  
  
        boolean up = (price > oldPrice);  
        MapMessage message = session.createMapMessage();  
        message.setBoolean("up", up);  
        message.setDouble("offer", offer);  
        message.setString("stock", stock.toString());  
        return message;  
    }  
}
```

Please post comments or corrections to the [Author Online Forum](#)

```

        message.setString("stock", stock.toString());
        message.setDouble("price", price);
        message.setDouble("offer", offer);
        message.setBoolean("up", up);
        System.out.println(
            "Sending: " + ((ActiveMQMapMessage)message).getContentMap()
            + " on destination: " + stock
        );
        return message;
    }

    protected double mutatePrice(double price) {
        double percentChange = (2 * Math.random() * MAX_DELTA_PERCENT)
            - MAX_DELTA_PERCENT;

        return price * (100 + percentChange) / 100;
    }
}

```

The `MessageCreator` interface defines only the `createMessage()` method that returns a created message. Here, we have implemented a logic for creating random prices for different stocks and creating appropriate map messages containing all relevant data.

Now we can proceed to writing our publisher class which will use `JmsTemplate` and this message creator to send messages.

Example 7.12. Listing 6.9: Spring Publisher Implementation

```

public class SpringPublisher {

    private JmsTemplate template;
    private int count = 10;
    private int total;
    private Destination[] destinations;
    private HashMap<Destination, StockMessageCreator>
        creators = new HashMap<Destination, StockMessageCreator>();

    public void start() {
        while (total < 1000) {
            for (int i = 0; i < count; i++) {
                sendMessage();
            }
            total += count;
            System.out.println("Published '" + count + "' of '"
                + total + "' price messages");
        }
    }
}

```

Please post comments or corrections to the [Author Online Forum](#)

```
try {
    Thread.sleep(1000);
} catch (InterruptedException x) {
}
}

protected void sendMessage() {
    int idx = 0;
    while (true) {
        idx = (int) Math.round(destinations.length * Math.random());
        if (idx < destinations.length) {
            break;
        }
    }
    Destination destination = destinations[idx];
    template.send(destination, getStockMessageCreator(destination));      #A
}

private StockMessageCreator getStockMessageCreator(Destination dest) {
    if (creators.containsKey(dest)) {
        return creators.get(dest);
    } else {
        StockMessageCreator creator = new StockMessageCreator(dest);
        creators.put(dest, creator);
        return creator;
    }
}

// getters and setters goes here
}

#A Send with JmsTemplate
```

The important thing to note in the previous example is how we used the `send()` method with previously implemented message creator to send messages to the broker. Everything else in this example is the same as in our original stock portfolio publisher.

With this class implemented, we have all components needed to publish messages. All that is left to be done is to configure it properly.

Example 7.13. Listing 6.10: Spring Publisher Configuration

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory">
        <ref local="pooledJmsFactory" />
```

Please post comments or corrections to the [Author Online Forum](#)

```
</property>
</bean>

<bean id="stockPublisher" class="org.apache.activemq.book.ch6.spring.SpringPublisher">
<property name="template">
<ref local="jmsTemplate"/>
</property>
<property name="destinations">
<list>
<ref local="javaDest"/>
<ref local="ionaDest"/>
</list>
</property>
</bean>
```

In the previous XML configuration snippet, we have first configured an instance of `JmsTemplate` we are going to use. Note that we have used pooled connection factory. This is very important thing to do, because `JmsTemplate` is designed for use in EJB containers with their pooling capabilities. So basically, every method call will try to create all context objects (connection, session and producer) which is very inefficient and causes performance issues. So if you are not in an EJB environment, you should use pooled connection factory for sending messages with `JmsTemplate`, just as we have done in this case.

Finally, we have configured our `SpringPublisher` class with an instance of `jmsTemplate` and two destinations we are going to use.

7.1.2.2.5. Putting it all together

After implementing all pieces of our example we are ready to put them all together and run the application.

Example 7.14. Listing 6.11: Spring client

```
public class SpringClient {

    public static void main(String[] args) {
        FileSystemXmlApplicationContext context =
            new FileSystemXmlApplicationContext(
                "src/main/resources/org/apache/activemq/book/ch6/spring-client.xml"
            );
        SpringPublisher publisher = (SpringPublisher)context.getBean("stockPublisher");
        publisher.start();
    }
}
```

Please post comments or corrections to the [Author Online Forum](#)

```
}
```

This simple class just initializes Spring application context and start our publisher. We can run it with the following command:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch6.spring.SpringClient
```

and expect the output similar to the one shown below.

```
Sending: {price=65.958996694717, stock=topic://STOCKS.JAVA, offer=66.0249556914117, up=false  
on destination: topic://STOCKS.JAVA  
topic://STOCKS.IONA 79.97 80.05 down  
Sending: {price=80.67595675108532, stock=topic://STOCKS.IONA, offer=80.7566327078364, up=true  
on destination: topic://STOCKS.IONA  
topic://STOCKS.JAVA 65.96 66.02 down  
Sending: {price=65.63333898492846, stock=topic://STOCKS.JAVA, offer=65.69897232391338, up=false  
on destination: topic://STOCKS.JAVA  
topic://STOCKS.IONA 80.68 80.76 up  
Sending: {price=80.50525969261822, stock=topic://STOCKS.IONA, offer=80.58576495231084, up=false  
on destination: topic://STOCKS.IONA  
topic://STOCKS.JAVA 65.63 65.70 down  
Sending: {price=81.2186806051703, stock=topic://STOCKS.IONA, offer=81.29989928577547, up=true  
on destination: topic://STOCKS.IONA  
topic://STOCKS.JAVA 80.51 80.59 down  
Sending: {price=65.48960846536974, stock=topic://STOCKS.JAVA, offer=65.5550980738351, up=false  
on destination: topic://STOCKS.JAVA  
topic://STOCKS.IONA 81.22 81.30 up  
topic://STOCKS.JAVA 65.49 65.56 down
```

As you can see, both producer and consumer print their messages to standard output.

7.2. Summary

In this chapter we have seen that ActiveMQ could be seen not only as a separate Java infrastructure application, but also as a Java module that can be easily integrated in your Java applications. It can be configured with plain Java code, if you prefer it, or by using the same XML configuration files we use when we use ActiveMQ broker as a standalone application.

We have also seen how ActiveMQ can play well with Spring framework both in terms of integrating brokers in Java applications and simplifying implementation of JMS clients. A final product of this section was our portfolio example, rewritten for usage in Spring environment.

In the following chapter we will focus on ActiveMQ integration options with various J2EE containers. We will see how we can use it in conjunction with other Java containers, such as Apache Geronimo for example, and how we can use JNDI to help us write our clients.

Chapter 8. Embedding ActiveMQ In Other Java Containers

8.1. Introduction

The previous chapter demonstrates some basic examples of embedding ActiveMQ in Java applications. Not only is embedding ActiveMQ possible, it's actually quite popular. Embedding ActiveMQ is actually quite easy to achieve because it was designed from the ground up for this purpose. This means that ActiveMQ can not only be embedded in your Java applications, but it can also be embedded in other Java containers.

In this chapter, readers will:

- Learn what it means to embed ActiveMQ into another application
- Understand how to embed ActiveMQ in a plain old Java application
- Walk through examples of embedding ActiveMQ in other application containers such as:
 - Apache Tomcat
 - Apache Geronimo
 - JBoss
 - Jetty

8.2.

TODO

8.3. Summary

TODO



Please post comments or corrections to the [Author Online Forum](#)

Chapter 9. Connecting to ActiveMQ With Other Languages

Thus far we have been focused on ActiveMQ as a JMS broker and explored various ways of how we can use it in Java environment. But ActiveMQ is more than just a JMS broker. It provides an extensive list of connectivity options, so it can be seen as a general messaging solution for a variety of development platforms. In this chapter we will cover all ActiveMQ aspects related to providing messaging services to different platforms. We'll start by exploring the *STOMP* (*Streaming Text Orientated Messaging Protocol*) protocol, which due to its simplicity plays an important role in messaging for scripting languages. Examples in Ruby, Python, PHP and Perl will demonstrate the ease of messaging with STOMP and ActiveMQ. Next, we will focus on writing clients for C++ and .NET platforms with appropriate examples. Finally, we will see how ActiveMQ could be used in the Web environment through its *REST* and *Ajax* APIs. Even this brief introductory discussion lead us to conclusion that ActiveMQ is not just another message broker, but rather the general messaging platform for various environments. Before we go into details on specific platforms, we have to define examples we will be using throughout this chapter.

9.1. Preparing Examples

In Chapter 2, Introduction to ActiveMQ, we have defined a stock portfolio example that uses map messages to exchange data between producers and consumers. For the purpose of this chapter, however, we will modify this original example a bit and make it a better fit for environments described here. So instead of map messages we will exchange XML data in text messages. One of the primal reasons we are doing this is due to the fact that processing of map messages is not yet supported on some of these platforms (e.g. STOMP protocol).

So we will create a Java message producer that sends text messages with appropriate XML data. Then we will implement appropriate consumers for each of

the platforms described, which will show us how to connect the specified platform with Java in an asynchronous way.

So for starters, we have to modify our publisher to send XML data in text messages instead of map messages. The only thing we have to change from our original publisher is the `createStockMessage()` method. The Listing 8.1 shows the method that creates an appropriate XML representation of desired data and creates a `TextMessage` instance out of it.

Example 9.1. Listing 8.1: Modified publisher

```
protected Message createStockMessage(String stock, Session session)
throws JMSException, XMLStreamException {
    Double value = LAST_PRICES.get(stock);
    if (value == null) {
        value = new Double(Math.random() * 100);
    }

    // lets mutate the value by some percentage
    double oldPrice = value.doubleValue();
    value = new Double(mutatePrice(oldPrice));
    LAST_PRICES.put(stock, value);
    double price = value.doubleValue();

    double offer = price * 1.001;

    boolean up = (price > oldPrice);

    StringWriter res = new StringWriter(); #A
    XMLStreamWriter writer =
        XMLOutputFactory.newInstance().createXMLStreamWriter(res); #A
    writer.writeStartDocument(); #A
    writer.writeStartElement("stock"); #A
    writer.writeAttribute("name", stock); #A
    writer.writeStartElement("price"); #A
    writer.writeCharacters(String.valueOf(price)); #A
    writer.writeEndElement(); #A

    writer.writeStartElement("offer"); #A
    writer.writeCharacters(String.valueOf(offer)); #A
    writer.writeEndElement(); #A

    writer.writeStartElement("up"); #A
    writer.writeCharacters(String.valueOf(up)); #A
    writer.writeEndElement(); #A
    writer.writeEndElement(); #A
    writer.writeEndDocument(); #A

    TextMessage message = session.createTextMessage(); #B
```

Please post comments or corrections to the [Author Online Forum](#)

```

        message.setText(res.toString());
        return message;
    }

#A Create XML data
#B Create Text message

```

As you can see, we have used a simple StAX API (<http://java.sun.com/webservices/docs/1.6/tutorial/doc/SJSXP3.html>) to create an XML representation of our stock data. Next, we created a text message and used the `setText()` method to associate this XML to the message.

Now we can start our publisher in a standard manner:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch8.Publisher -Dexec.args="IONA JAVA"
```

and expect the following output:

```

Sending: <?xml version="1.0" ?>
<stock name="JAVA">
    <price>81.98722521538372</price><offer>82.06921244059909</offer><up>false</up>
</stock>
on destination: topic://STOCKS.JAVA
Sending: <?xml version="1.0" ?>
<stock name="IONA">
    <price>16.220523047943267</price><offer>16.23674357099121</offer><up>false</up>
</stock>
on destination: topic://STOCKS.IONA
Sending: <?xml version="1.0" ?>
<stock name="JAVA">
    <price>82.7035345851213</price><offer>82.78623811970641</offer><up>true</up>
</stock>
on destination: topic://STOCKS.JAVA
Sending: <?xml version="1.0" ?>
<stock name="IONA">
    <price>16.26436632596291</price><offer>16.28063069228887</offer><up>true</up>
</stock>
on destination: topic://STOCKS.IONA
Sending: <?xml version="1.0" ?>
<stock name="JAVA">
    <price>83.34179166698637</price><offer>83.42513345865335</offer><up>true</up>
</stock>
on destination: topic://STOCKS.JAVA
Sending: <?xml version="1.0" ?>
<stock name="JAVA">
    <price>83.89127220511598</price><offer>83.97516347732109</offer><up>true</up>
</stock>
on destination: topic://STOCKS.JAVA

```

Please post comments or corrections to the [Author Online Forum](#)

As expected, the publisher sends a series of XML formatted text messages to different ActiveMQ topics. As they are ready to be consumed, it's time to see how we can consume them using different programming languages and platforms.

9.2. Communicating with the STOMP protocol

In Chapter 3, Understanding Connectors, we have explained various network protocols used for communication between ActiveMQ and clients. But what we haven't discussed there is that choosing the right network protocol is just one side of the story. The equally important aspect of communication is finding the right way to serialize your messages over the network, or picking the *wire protocol*.

ActiveMQ uses *OpenWire* (<http://activemq.apache.org/openwire.html>) as its native wire protocol for exchanging messages between brokers and clients. OpenWire is designed to be an efficient binary protocol in terms of network bandwidth and performance. This makes it an ideal choice for communication with so called native clients usually written in Java, C or C#. But all this efficiency comes at the cost, and in this case it is the complexity of implementation.

Stomp (Streaming Text Orientated Messaging Protocol), on the other hand, is designed with entirely different requirements in mind. It is a simple text-oriented protocol, very similar to HTTP. You can see it as HTTP adopted to the messaging realm. This implies that it is quite easy to implement the Stomp client in an arbitrary programming language. It is even possible to communicate with the broker through the telnet session using Stomp, as we will see in the moment.

We will not explain the Stomp protocol in details here and you are advised to take a look at the protocol specification (<http://stomp.codehaus.org/Protocol>) if you are interested. But let's walk through some basics, just to get the feeling what is happening under the hood.

Clients and brokers communicate with each other by exchanging *frames*, textual representation of messages. Frames could be delivered over any underlying network protocol, but it is usually TCP. Every frame consists of a three basic elements; *command*, *headers* and *body*, like it is shown in the following example:

SEND	#A
------	----

Please post comments or corrections to the [Author Online Forum](#)

```
destination:/queue/a #B  
hello queue a          #C  
^@  
#A Command  
#B Headers  
#C Body
```

The command part of the frame actually identifies what kind of operation should take place. In this example, the `SEND` frame is used to send a message to the broker, but you can also:

- `CONNECT`, `DISCONNECT` from the broker
- `SUBSCRIBE`, `UNSUBSCRIBE` from the destination
- `BEGIN`, `COMMIT` and `ABORT` transaction
- or `ACK` (acknowledge) messages

These commands are self-explanatory and represent common functionalities expected to be found in any messaging system. We will see them in action through examples in the coming sections.

Headers are used to specify additional properties for each command, such as the destination where to send a message in the above example. Headers are basically key-value pairs, separated by a colon (:) character. Every header should be written in the separate line (of course, our example contains only one header).

The blank line indicates the end of the headers section and start of an optional body section. In case of the `SEND` command, the body section contains an actual message we want to send. Finally the frame is ended by the ASCII null character (^@).

After explaining the basic structure of frames, let's go to Stomp sessions. The Listing 8.2 shows how easy it is to create a regular *telnet session* and use it to send and receive messages from the command line.

Example 9.2. Listing 8.2: STOMP session

```
$ telnet localhost 61613
```

Please post comments or corrections to the [Author Online Forum](#)

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^].
CONNECT #1
login:system
passcode:manager

^@
CONNECTED #2
session:ID:dejan-laptop-36961-1221552914772-4:0

SEND #3
destination:/queue/a

hello queue a
^@

SUBSCRIBE #4
destination:/queue/a

^@
MESSAGE #5
message-id:ID:dejan-laptop-36961-1221552914772-4:0:-1:1:1
destination:/queue/a
timestamp:1221553047204
expires:0
priority:0

hello queue a

UNSUBSCRIBE #6
destination:/queue/a

^@

DISCONNECT #7

^@
Connection closed by foreign host.
```

As you can see, the usual session starts by connecting to the broker (with appropriate credentials provided) #1. The broker will acknowledge successful connection, by sending the CONNECTED frame #2 back to the client. After creating a successful connection, the client can send messages #3 using the SEND frame similar to the one we have described above. If it wants to receive messages, it should subscribe to the desired destination #4. From that moment on, messages

from the subscribed destination will be pushed asynchronously to the client #5. When the client is finished with consuming messages, it should unsubscribe from the destination #6. Finally, the client should disconnect from the broker #7 to terminate the session.

You have probably noticed that we started destination name with the `/queue/` prefix, which naturally suggests that the desired destination is a message queue. Stomp protocol does not define any semantics regarding destination names and specifies it only as a string value which is specific to the server implementation. ActiveMQ implements the syntax we have seen in our example, where prefixes `/queue/` or `/topic/` defines the type of the destination, while the rest is interpreted as destination name. So, the value `/queue/a` used in the previous example basically interprets as "queue named a". Having said all this, we can conclude that you should be careful when dealing with destination names starting with the `/` character. For example, you should use value `/queue//a` if you want to access the queue named `/a`.

Now that we have learned basics of the Stomp protocol, let's see how we can configure ActiveMQ to enable this kind of the communication with its clients. The configuration shown in Listing 8.3 defines two transport connectors, the one that allows connections over the TCP connector (and use OpenWire wire protocol) and the other one that uses Stomp.

Example 9.3. Listing 8.3: STOMP transport

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

  <transportConnectors>
    <transportConnector name="openwire"
      uri="tcp://localhost:61616" />
    <transportConnector name="stomp"
      uri="stomp://localhost:61613" />
  </transportConnectors>

</broker>
```

So basically all you have to do is to define a transport connector with `stomp`

keyword for an URI schema and you are ready to go.

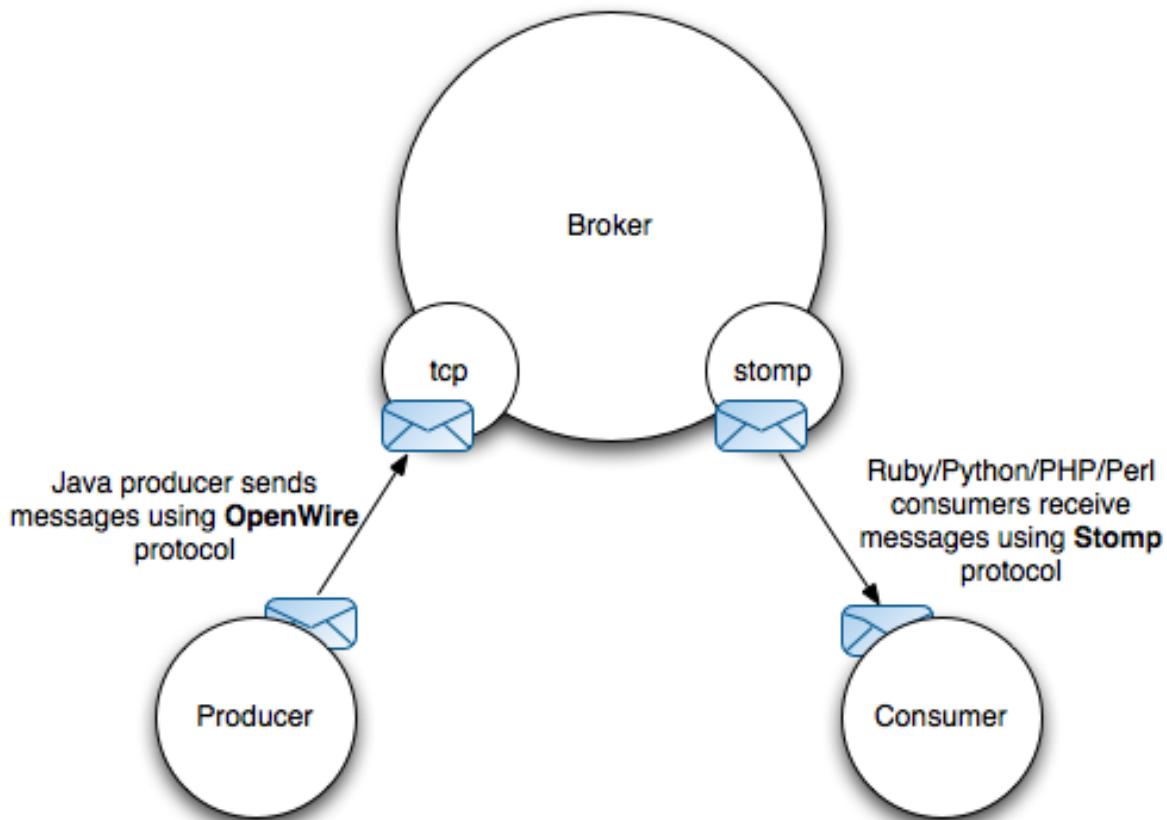


Figure 9.1. Figure 8.1: Stomp Example

Now let's see how to implement consumers of our stock portfolio data in some of the most of the popular scripting languages.

9.2.1. Writing Ruby client

We all witnessed the raising popularity of *Ruby on Rails* Web development framework, which marked Ruby as one of the most popular dynamic languages today. The asynchronous messaging with Stomp and ActiveMQ brings one more tool to the Ruby and Rails developers' toolbox, making it possible to tackle the whole new range of problems. The more information on how to install the Stomp

client for Ruby could be found at the following address:

<http://stomp.codehaus.org/Ruby+Client>. Once you have installed and configured your Ruby environment, you can write the stock portfolio consumer as follows:

Example 9.4. Listing 8.4: Ruby consumer

```
#!/usr/bin/ruby

require 'rubygems'
require 'stomp'
require 'xmlsimple'

@conn = Stomp::Connection.open '', '', 'localhost', 61613, false #A
@count = 0

@conn.subscribe "/topic/STOCKS.JAVA", { :ack =>"auto" } #B
@conn.subscribe "/topic/STOCKS.IONA", { :ack =>"auto" } #B
while @count < 100
  @msg = @conn.receive #C
  @count = @count + 1
  if @msg.command == "MESSAGE"
    @xml = XmlSimple.xml_in(@msg.body)
    $stdout.print "#{@xml['name']}\t"
    $stdout.print "#{%.2f" % @xml['price']} \t"
    $stdout.print "#{%.2f" % @xml['offer']} \t"
    $stdout.print "#{@xml['up'].to_s == 'true'? 'up': 'down'}\n"
  else
    $stdout.print "#{@msg.command}: #{@msg.body}\n"
  end
end
@conn.disconnect

#A Define connection
#B Subscribe
#C Receive the message
```

Basically, all Stomp clients just provide wrapper functions (methods) for creating basic Stomp frames and sending to (or reading from) TCP sockets. In this example, we first created a connection with a broker using the `open()` method. This is equivalent to opening a TCP socket and sending the `CONNECT` frame to the broker. Beside usual connection parameters, such as `username`, `password`, `host` and `port`, we have provided one extra argument at the end of this method call. This parameter specify weather the client will be *reliable*, or in other words, will it try to reconnect until it successfully connects to the broker. In this example, we set the

`reliable` parameter to false, which means it will raise an exception in case of a connection failure.

After the connection is created we can subscribe to desired topics, using the `subscribe()` method. In this example, you can notice that we have passed the additional `ack` header, which defines the way messages are acknowledged. The Stomp protocol defines two acknowledgment modes:

- *auto*, which means the broker will mark the message as delivered right after the client consumes it
- and *client*, which instructs the broker to consider the message delivered only after the client specifically acknowledges it with an `ACK` frame.

The auto mode is a default one and you don't have to include any headers in case you plan to use it. In this example, we have specifically set it just for demonstration purposes. The client acknowledgment mode is explained in examples that follows.

Now let's receive some messages. We can do it by using the `receive()` method which reads frames from the TCP socket and parses them. As you can see we don't have any logic implemented to acknowledge messages as we are using the auto acknowledgment mode.

The rest of the example is dedicated to XML parsing and printing. Now you can run this example and expect the following output (of course, the publisher described at the start of the chapter should be running):

```
$ ruby consumer.rb
IONA 34.53 34.57 up
JAVA 37.61 37.65 down
JAVA 37.55 37.59 down
JAVA 37.56 37.60 up
IONA 34.84 34.88 up
JAVA 37.83 37.87 up
JAVA 37.83 37.87 up
JAVA 38.05 38.09 up
JAVA 38.14 38.18 up
IONA 35.06 35.10 up
JAVA 38.03 38.07 down
JAVA 37.68 37.72 down
JAVA 37.55 37.59 down
JAVA 37.59 37.62 up
```

Please post comments or corrections to the [Author Online Forum](#)

```
IONA 35.21 35.25 up
IONA 35.12 35.15 down
JAVA 37.26 37.30 down
```

If you like Rails-like frameworks, you can also check out the ActiveMessaging project (<http://code.google.com/p/activemessaging/>), which brings "simplicity and elegance of rails development to messaging".

Now let's see how to implement a similar stock portfolio data consumer in Python.

9.2.2. Creating Python client

Python is another extremely popular and powerful dynamic language, often used in a wide range of software projects. As you can see from the list of Stomp Python clients (<http://stomp.codehaus.org/Python>), there is a variety of libraries you can use in your projects. For the basic implementation of our stock portfolio consumer we've chosen the `stomp.py` implementation, you can find at the following web address <http://www.briggs.net.nz/log/projects/stomppy/>.

For starters we will create a helper scripts, called `book.py`, which will contain helper classes and methods for all our Python examples.

Example 9.5. Listing 8.5: Python helper script

```
from xml.etree.ElementTree import XML

def printXml(text):
    xml = XML(text)

    print "%s\t%.2f\t%.2f\t%s" % (
        xml.get("name"),
        eval(xml.find("price").text),
        eval(xml.find("offer").text),
        "up" if xml.find("up").text == "True" else "down"
    )
```

For now, this script only contains a method (called `printxml()`) which parses and prints our stock portfolio XML data.

Now let's take a look at the example in Listing 8.6:

Please post comments or corrections to the [Author Online Forum](#)

Example 9.6. Listing 8.6: Python consumer

```
#!/usr/bin/env python

import time, sys
from elementtree.ElementTree import ElementTree, XML
from book import printXml

import stomp

class MyListener(object):
    def on_error(self, headers, message): #A
        print 'received an error %s' % message

    def on_message(self, headers, message):
        printXml(message)

conn = stomp.Connection() #1
conn.add_listener(MyListener())
conn.start() #2
conn.connect() #3

conn.subscribe(destination='/topic/STOCKS.JAVA', ack='auto') #4
conn.subscribe(destination='/topic/STOCKS.IONA', ack='auto') #4

time.sleep(60);

conn.disconnect()

#A message listener
```

As you can see, the Python client implements an asynchronous JMS-like API with message listeners, rather than using synchronous message receiving philosophy used by other Stomp clients. This code sample defines a simple message listener that parses XML text message (using our `printXml()` function) and prints desired data on the standard output. Then, similarly to Java examples, creates a connection #1, adds a listener #2, starts a connection #3 and finally subscribes to desired destinations #4.

When started, this example will produce the output similar to the following one:

```
$ python consumer.py
IONA 52.21 52.26 down
```

Please post comments or corrections to the [Author Online Forum](#)

```
JAVA 91.88 91.97 down
IONA 52.09 52.14 down
JAVA 92.16 92.25 up
JAVA 91.44 91.53 down
IONA 52.17 52.22 up
JAVA 90.81 90.90 down
JAVA 91.46 91.55 up
JAVA 90.69 90.78 down
IONA 52.33 52.38 up
JAVA 90.45 90.54 down
JAVA 90.51 90.60 up
JAVA 91.00 91.09 up
```

which is consistent with what we had for the Ruby client.

9.2.2.1. Messaging with pyactivemq

As we said before, there are a few other Python clients that could exchange messages with ActiveMQ. One specially interesting is *pyactivemq* project (<http://code.google.com/p/pyactivemq/>), which will be covered in this section. The interesting thing about this project is that it is basically just a Python wrapper around ActiveMQ C++ library (described a bit later) which supports both Stomp and OpenWire protocols and provides an excellent performances for your Python applications. Since it wraps ActiveMQ C++ library, it requires special installation procedure, so be sure to check the project site for information relevant to your platform.

Let's now create a full stock portfolio example using *pyactivemq* and see how it can be used both over Stomp and OpenWire protocols. For starters we need to add a few more helper functions in our helper script.

Example 9.7. Listing 8.7: pyactivemq helper script

```
import random
from xml.etree.ElementTree import Element, SubElement, XML, tostring

def printXml(text):
    xml = XML(text)

    print "%s\t%.2f\t%.2f\t%s" % (
        xml.get("name"),
        eval(xml.find("price").text),
        eval(xml.find("offer").text),
        "up" if xml.find("up").text == "True" else "down"
```

Please post comments or corrections to the [Author Online Forum](#)

```

    )

def mutatePrice(price):                                #A
    MAX_DELTA_PERCENT = 1
    percentChange = (2 * random.random()
                      * MAX_DELTA_PERCENT) - MAX_DELTA_PERCENT

    return price * (100 + percentChange) / 100;

def createXml(oldPrice, price):                       #B
    stock = Element("stock")
    stock.set("name", "JAVA")
    priceElem = SubElement(stock, "price")
    priceElem.text = str(price)

    offer = SubElement(stock, "offer")
    offer.text = str(price * 1.001)

    up = SubElement(stock, "up")
    up.text = str(oldPrice > price)

    return stock

#A mutates the price
#B formats stock data as xml

```

As you can notice, there are two new functions which are basically the same as those we have in our original Java stock portfolio producer example. Now, let's put it all together.

Example 9.8. Listing 8.8: pyactivemq example

```

import pyactivemq, time, sys, random
from pyactivemq import ActiveMQConnectionFactory
from book import mutatePrice, printXml, createXml
from xml.etree.ElementTree import Element, SubElement, XML, tostring

class MessageListener(pyactivemq.MessageListener):          #A

    def onMessage(self, message):
        printXml(message.text)

nmessages = 100

brokerUrl = 'tcp://localhost:61616?wireFormat=openwire'      #1

if(len(sys.argv) == 2 and sys.argv[1] == 'stomp'):

```

Please post comments or corrections to the [Author Online Forum](#)

```

brokerUrl = 'tcp://localhost:61613?wireFormat=stomp'

print 'connecting to: ', brokerUrl

f = ActiveMQConnectionFactory(brokerUrl) #2
conn = f.createConnection()

session = conn.createSession() #3
topic = session.createQueue('stocks')
producer = session.createProducer(topic)

consumer = session.createConsumer(topic) #4
consumer.messageListener = MessageListener()

conn.start()

textMessage = session.createTextMessage()
price = random.uniform(1, 100)

for i in xrange(nmessages):
    oldPrice = price
    price = mutatePrice(price)

    textMessage.text = tostring(createXml(oldPrice, price))
    producer.send(textMessage) #6

time.sleep(5)

conn.close()

#A Message Listener

```

The *CMS API* (described a bit later in this chapter) defines an API very similar to JMS and since the pyactivemq is just a wrapper around a CMS API implementation we can expect JMS-like API for Python. So in this example, we have defined a broker URL #1, created a connection using connection factory #2 and created a session #3 for starters. Next, we creates a producer #4 and consumer with appropriate message listener #5. Finally we can create stock portfolio data (with the help of previously defined functions) and send #6 them to the broker.

Now if we run this example, we will get the output similar to the following:

```

$ python stocks.py
connecting to: tcp://localhost:61616?wireFormat=openwire
JAVA 92.26 92.35 up
JAVA 91.99 92.08 up

```

Please post comments or corrections to the [Author Online Forum](#)

```
JAVA 92.19 92.29 down
JAVA 92.28 92.37 down
JAVA 91.36 91.45 up
JAVA 91.88 91.97 down
JAVA 91.52 91.61 up
JAVA 91.22 91.31 up
```

Please note the URL we are connecting to. You can notice that we're passing a `wireFormat` parameter and in this case it configures the client to use OpenWire protocol to exchange messages. We can change this by passing a `stomp` argument when executing the script.

```
$ python stocks.py stomp
connecting to: tcp://localhost:61613?wireFormat=stomp
JAVA 19.55 19.57 up
JAVA 19.58 19.60 down
JAVA 19.76 19.78 down
JAVA 19.95 19.97 down
JAVA 20.13 20.15 down
JAVA 20.09 20.11 up
JAVA 20.28 20.30 down
JAVA 20.26 20.28 up
JAVA 20.19 20.21 up
JAVA 20.28 20.30 down
```

Now you can see that `wireFormat` parameter value in our connection URL has changed configuring the producer and consumer to use Stomp wire protocol. This example showed how easy is to use both OpenWire and Stomp with just a slight change of connection URL parameter.

After showing Ruby and Python examples, it's time to focus a bit on old-school scripting languages, such as PHP and Perl, and their Stomp clients.

9.2.3. Building PHP client

Despite the tremendous competition in the Web development platform arena, PHP (in combination with Apache web server) is still one of the most frequently used tools for developing web-based applications. *Stompcgi* library (<http://stomp.codehaus.org/PHP>) provides an easy way to use asynchronous messaging in PHP applications. The example shown in Listing 8.9 and explained afterwards, demonstrates how to create a stock portfolio data consumer in PHP:

Please post comments or corrections to the [Author Online Forum](#)

Example 9.9. Listing 8.9: PHP consumer

```
<?

require_once('Stomp.php');

$stomp = new Stomp("tcp://localhost:61613");

$stomp->connect('system', 'manager'); #1

$stomp->subscribe("/topic/STOCKS.JAVA"); #2
$stomp->subscribe("/topic/STOCKS.IONA");

$i = 0;
while($i++ < 100) {

    $frame = $stomp->readFrame(); #3
    $xml = new SimpleXMLElement($frame->body);
    echo $xml->attributes()->name
        . "\t" . number_format($xml->price,2)
        . "\t" . number_format($xml->offer,2)
        . "\t" . ($xml->up == "true"? "up" : "down") . "\n";
    $stomp->ack($frame); #4

}

$stomp->disconnect(); #5

?>
```

Practically, all Stomp examples look alike; the only thing that differs is the language syntax used to write the particular one. So here, we have all basic elements found in Stomp examples: creating a connection #1, subscribing to destinations #2, reading messages #3, and finally disconnecting #5. However, we have one slight modification over the previous examples. Here, we have used the *client acknowledgment* of messages, which means that messages will be considered consumed only after you explicitly acknowledge them. For that purpose we have called the `ack()` method #4 upon processing of each message.

Now we can run the previous script and expect the following result:

```
$ php consumer.php
JAVA 50.64 50.69 down
JAVA 50.65 50.70 up
```

Please post comments or corrections to the [Author Online Forum](#)

```
JAVA 50.85 50.90 up
JAVA 50.62 50.67 down
JAVA 50.39 50.44 down
JAVA 50.08 50.13 down
JAVA 49.72 49.77 down
IONA 11.45 11.46 up
JAVA 49.24 49.29 down
IONA 11.48 11.49 up
JAVA 49.22 49.27 down
JAVA 48.99 49.04 down
JAVA 48.88 48.92 down
JAVA 48.49 48.54 down
IONA 11.42 11.43 down
```

As it was expected, the script produces the output similar to those we have seen in our previous examples. The following section explains the similar example written in another popular old-school scripting language, Perl.

9.2.4. Implementing Perl client

Perl is one of the first powerful dynamic languages and as such have a large community of users. Particular development tasks Perl is used for are pretty wide, but it is probably best known as "an ultimate system administrator tool". Therefore, an introduction of asynchronous messaging for Perl gives developers one more powerful tool in their toolbox.

Implementation of Stomp protocol in Perl could be found in the CPAN Net::Stomp module (<http://search.cpan.org/dist/Net-Stomp/>). The following example contains an implementation of the stock portfolio consumer in Perl.

Example 9.10. Listing 8.10: Perl consumer

```
use Net::Stomp;
use XML::Simple;

my $stomp = Net::Stomp->new( { hostname => 'localhost', port => '61613' } );
$stomp->connect( { login => 'system', passcode => 'manager' } );

$stomp->subscribe(
    {   destination          => '/topic/STOCKS.JAVA',
        'ack'                => 'client',
        'activemq.prefetchSize' => 1
    }
#1
```

Please post comments or corrections to the [Author Online Forum](#)

```
        }  
    );  
    $stomp->subscribe(  
        { 'destination' => '/topic/STOCKS.IONA',  
          'ack' => 'client',  
          'activemq.prefetchSize' => 1  
        }  
    );  
  
    my $count = 0;  
  
    while ($count++ < 100) {  
        my $frame = $stomp->receive_frame;  
        my $xml = XMLin($frame->body);  
        print $xml->{name} . "\t" . sprintf("%.2f", $xml->{price}) . "\t";  
        print sprintf("%.2f", $xml->{offer}) . "\t";  
        print ($xml->{up} eq 'true' ? 'up' : 'down') . "\n";  
  
        $stomp->ack( { frame => $frame } );  
    }  
  
    $stomp->disconnect;
```

The example is practically the same as all our previous examples (especially the PHP one since the syntax is almost the same). However, there is one additional feature we have added to this example, and that is the usage of the `activemq.prefetchSize` value #1 when subscribing to the destination.

ActiveMQ uses *prefetch limit* to determine the number of messages it will pre-send to consumers, so that network is used optimally. This option is explained in more details in Chapter 11, Advanced Client Options, but basically this means that broker will try to send 1000 messages to be buffered on the client side. Once the consumer buffer is full no more messages are sent before some of the existing messages in the buffer gets consumed (acknowledged). While this technique works great for Java consumers, Stomp consumers (and libraries) are usually a simple scripts and don't implement any buffers on the client side, so certain problems (like undelivered messages) could be induced by this feature. Thus, it is advisable to set the prefetch size to 1 (by providing a specialized `activemq.prefetchSize` header to the `SUBSCRIBE` command frame) and instruct the broker to send one message at the time.

Now that we have it all explained, let's run our example:

```
$ perl consumer.pl
IONA 69.22 69.29 down
JAVA 22.20 22.22 down
IONA 69.74 69.81 up
JAVA 22.05 22.08 down
IONA 69.92 69.99 up
JAVA 21.91 21.93 down
JAVA 22.10 22.12 up
JAVA 21.95 21.97 down
JAVA 21.84 21.86 down
JAVA 21.67 21.69 down
IONA 70.60 70.67 up
JAVA 21.70 21.72 up
IONA 70.40 70.47 down
JAVA 21.50 21.52 down
IONA 70.55 70.62 up
JAVA 21.69 21.71 up
```

As you can see, the behavior is the same as with all other Stomp examples.

With Perl we have finished demonstration of Stomp clients and exchanging messages with ActiveMQ using different scripting languages. But Stomp (especially combined with ActiveMQ) is more capable than just simple sending and receiving messages. In the next two sections, we will go through Stomp transactions and how you can create durable topic subscribers for ActiveMQ using your Stomp clients.

9.2.5. Understanding Stomp transactions

Besides sending and acknowledging messages one by one, Stomp protocol introduces a concept of transactions, which group multiple `SEND` and `ACK` commands. Transactions are very well known concept from SQL and JMS and I'm sure you're familiar with atomicity they introduce and transaction-related operations, such as *commit* and *rollback (abort)*.

So if you want to start a transaction, you need to send a `BEGIN` frame to the broker along with the transaction header that contains a transaction id. For example, the following frame

```
BEGIN
transaction:tx1
```

Please post comments or corrections to the [Author Online Forum](#)

```
^@
```

will start a transaction named `tx1`. After you're finished with the transaction, you can either commit it or abort it, by sending the appropriate frame (`COMMIT` or `ABORT`), of course with transaction id passed as the `transaction` header. So, the following frame

```
COMMIT  
transaction:tx1
```

```
^@
```

will commit the previously started `tx1` transaction and mark successful send and acknowledgment of all messages in the transaction.

One more important thing is how you send and acknowledge messages in the transaction. In order to mark that the message is sent or acknowledged in the transaction, you need to add a `transaction` header to `SEND` and `ACK` frames that are sent to the broker. Of course, the value of this header must match the valid (started) transaction id.

So for example, the following frame

```
ACK  
destination:/queue/transactions  
transaction:tx1  
message-id:ID:dejanb.local-62217-1249899600449-6:0:-1:1:3
```

```
^@
```

states that the appropriate message has been acknowledged in transaction `tx1`.

The important thing to note here, is that transactions in Stomp are only related to sending `SEND` and `ACK` frames. So there's no such concept such as *receiving messages in a transaction* as we have in JMS. This basically means that you can only rollback message acknowledgment, but not the message itself, so the message will not be redelivered to the client. The client application (or Stomp client) is responsible for trying to process those messages again and acknowledging them when they do so.

The following PHP example demonstrates sending and acknowledging messages

using transactions. First we will send some messages in a transaction:

Example 9.11. Listing 8.11: Transactions send example

```
<?
require_once( "Stomp.php" );

$con = new Stomp("tcp://localhost:61613");
$con->connect();

$con->begin("tx1");                                     #1
for ($i = 1; $i < 3; $i++) {
    $con->send("/queue/transactions", $i, array("transaction" => "tx1"));
}
$con->abort("tx1");                                     #2

$con->begin("tx2");                                     #3
echo "Sent messages {\n";
for ($i = 1; $i < 5; $i++) {
    $con->send("/queue/transactions", $i, array("transaction" => "tx2"));
    echo "\t$i\n";
}
echo "}\n";

$con->commit("tx2");                                     #5
?>
```

As you can notice, we tried first to send two messages in transaction `tx1` #1. Note that we are passing an additional header to the `send()` method in order to send a message in the transaction. But as we aborted transaction `tx1` #2, those messages were not sent to the broker. Then we started another transaction `tx2` #3 and sent four messages in it #4. Finally we committed transaction `tx2` #5 and thus told the broker to accept those messages. If you execute this script, you can expect the output similar to this one:

```
$ php transactions_send.php
Sent messages {
    1
    2
    3
    4
}
```

and of course, we should expect to have four messages in the queue. Now let's try to consume those messages using transactions.

Example 9.12. Listing 8.12: Transaction ack example

```

<?php
require_once( "Stomp.php" );

$con = new Stomp("tcp://localhost:61613");
$con->connect();
$con->setReadTimeout(1);

$con->subscribe("/queue/transactions",
    array('ack' => 'client', 'activemq.prefetchSize' => 1 )) ;      #1

$con->begin("tx3");
$messages = array();
for ($i = 1; $i < 3; $i++) {
    $msg = $con->readFrame();
    array_push($messages, $msg);
    $con->ack($msg, "tx3");                                         #2
}

$con->abort("tx3");                                                 #3

$con->begin("tx4");                                                 #4
if (count($messages) != 0) {
    foreach($messages as $msg) {
        $con->ack($msg, "tx4");                                     #5
    }
}
for ($i = 1; $i < 3; $i++) {
    $msg = $con->readFrame();
    $con->ack($msg, "tx4");                                         #6
    array_push($messages, $msg);
}
$con->commit("tx4");                                                 #7

echo "Processed messages {\n";
foreach($messages as $msg) {
    echo "\t$msg->body\n";
}
echo "}\n";

$frame = $con->readFrame();                                         #8
if ($frame === false) {
    echo "No more messages in the queue\n";
} else {
    echo "Warning: some messages still in the queue: $frame\n";
}

$con->disconnect();

```

In this script, we have first subscribed to the queue using prefetch size one and client acknowledgment #1. Next we are trying to consume two messages and acknowledge them in a transaction `tx3` #2. Note that we're now passing another parameter to the `ack()` method that identify the transaction in use. Let's now try to abort the transaction and see what happens #3. Those two messages will not be marked as received by the broker and since we are using prefetch size of one, the broker will not send any other messages until we consume these two. So when we start a new transaction `tx4` #4, we need first to acknowledge already received messages #5 before we can start consuming the rest of them #6. Finally, we are ready to commit the transaction #7 and mark all messages consumed. At the end we can verify there are no more messages left in the queue #8.

If you run the above example, you can expect the following output:

```
$ php transactions_receive.php
Processed messages {
    1
    2
    3
    4
}
No more messages in the queue
```

As you can see, transactions enabled us to send and acknowledge messages in atomic operations, which is a crucial requirement for many use cases. Now let's see how ActiveMQ enhances the core Stomp protocol.

9.2.6. Working with Durable Topic Subscribers

As you already know, topic consumers receive messages from the topic while they subscribed. So if they disconnect and connect again, they'll miss all the messages sent to topic in the mean time. ActiveMQ way of dealing with this is by using durable topic subscribers which can receive all messages retroactively. As Stomp protocol doesn't have any notion of queues and topics (and especially durable ones), this is pure ActiveMQ feature and it just an example of Stomp protocol enhancement by ActiveMQ.

In order to create a durable subscriber, we need to do two things. First we need to pass client id of the durable subscriber while we connecting to the broker. We can do that, by passing `client-id` header in the `CONNECT` frame, like this:

```
CONNECT
login:
passcode:
client-id:test
```

Next, we need to use pass the same client id to the `SUBSCRIBE` header, but this time using the `activemq.subscriptionName` header. The following snippet shows the example frame:

```
SUBSCRIBE
destination:/topic/test
ack:client
activemq.subscriptionName:test
activemq.prefetchSize:1
```

Now let's create one durable topic subscriber example.

Example 9.13. Listing 8.13: Durable Topic Subscriber example

```
<?php
require_once("Stomp.php");

$producer = new Stomp("tcp://localhost:61613");
$consumer = new Stomp("tcp://localhost:61613");
$consumer->setReadTimeout(1);
$consumer->clientId = "test";                                #1

$producer->connect();
$consumer->connect();
$consumer->subscribe("/topic/test");                            #2

sleep(1);

$producer->send("/topic/test", "test", array('persistent'=>'true')); #3
echo "Message 'test' sent to topic\n";

$msg = $consumer->readFrame();                                  #4

if ( $msg != null) {
    echo "Message '$msg->body' received from topic\n";
    $consumer->ack($msg);
} else {
    echo "Failed to receive a message\n";
```

Please post comments or corrections to the [Author Online Forum](#)

```

}

sleep(1);

$consumer->unsubscribe("/topic/test"); #5
$consumer->disconnect();
echo "Disconnecting consumer\n";

$producer->send("/topic/test", "test1", array('persistent'=>'true')); #6
echo "Message 'test1' sent to topic\n";

$consumer = new Stomp("tcp://localhost:61613");
$consumer->clientId = "test";
$consumer->connect();
$consumer->subscribe("/topic/test"); #7
echo "Reconnecting consumer\n";

$msg = $consumer->readFrame(); #8

if ( $msg != null) {
    echo "Message '$msg->body' received from topic\n";
    $consumer->ack($msg);
} else {
    echo "Failed to receive a message\n";
}

$consumer->unsubscribe("/topic/test");
$consumer->disconnect();
$producer->disconnect();
?>

```

First of all, we created a producer and consumer. Note that we set `clientId` property to the consumer #1. If set, this property will be passed to both `CONNECT` and `SUBSCRIBE` frames. Next, we subscribe to the topic #2 and send a messages to it #3. As expected, the message is received by the consumer #4. Now we can unsubscribe durable consumer #5 and send another messages while the consumer is offline #6. After subscribing again #7, the consumer will receive the message #8 no matter it was offline while the message was sent.

If you run this example, you can expect the following output:

```

$ php durable.php
Message 'test' sent to topic
Message 'test' received from topic
Disconnecting consumer
Message 'test1' sent to topic
Reconnecting consumer

```

```
Message 'test1' received from topic
```

With durable topic subscribers we are coming to the end of the Stomp section. As we have seen, Stomp protocol is designed to be simple to implement and thus easily usable from scripting languages, such as Ruby or PHP. We also said that ActiveMQ Java clients use, optimized binary OpenWire protocol, which provides better performances than Stomp. So, it is not surprising to see that clients written in languages such as C# or C++ provide more powerful clients using the OpenWire protocol. These clients will be the focus of the following two sections.

9.3. Learning NMS (.Net Message Service) API

Scripting languages covered in previous sections are mostly used for creating server-side software and Internet applications on Unix-like systems. Developers that targets Windows platform, on the other hand, usually choose Microsoft .NET framework as their development environment. Ability to use JMS-like API (and ActiveMQ in particular) to asynchronously send and receive messages can bring a big advantage for .NET developers. The *NMS API (.Net Message Service API)*, an ActiveMQ subproject (<http://activemq.apache.org/nms/nms.html>), provides a standard C# interface to messaging systems. The idea behind NMS is to create a unified messaging API for C#, similar to what JMS API represents to the Java world. Currently, it only supports ActiveMQ and OpenWire protocol, but providers to other messaging brokers could be easily implemented.

In the rest of this section we are going to implement stock portfolio consumer in C# and show you how to compile and run it using the Mono project (<http://www.mono-project.com/>). Of course, you can run this example on standard Windows implementation of .NET as well. For information on how to obtain (and optionally build) the NMS project, please refer to the NMS project site.

Now, let's take a look at the code shown in Listing 8.14:

Example 9.14. Listing 8.14: C# Consumer

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

Please post comments or corrections to the [Author Online Forum](#)

```
using Apache.NMS.Util;
using Apache.NMS.ActiveMQ;

namespace Apache.NMS.ActiveMQ.Book.Ch8
{
    public class Consumer
    {
        public static void Main(string[] args)
        {
            NMSSConnectionFactory NMSFactory =
                new NMSSConnectionFactory("tcp://localhost:61616");
            IConnection connection = NMSFactory.CreateConnection(); #1
            ISession session =
                connection.CreateSession(AcknowledgementMode.AutoAcknowledge); #2
            IDestination destination = session.GetTopic("STOCKS.JAVA"); #3
            IMessageConsumer consumer = session.CreateConsumer(destination); #4
            consumer.Listener += new MessageListener(OnMessage); #5
            connection.Start(); #6
            Console.WriteLine("Press any key to quit.");
            Console.ReadKey();
        }

        protected static void OnMessage(IMessage message) #7
        {
            ITextMessage TextMessage = message as ITextMessage;
            Console.WriteLine(TextMessage.Text);
        }
    }
}
```

As you can see, the NMS API is practically identical to the JMS API which can in great deal simplify developing and porting message-based applications. First, we have created the appropriate connection #1 and session #2 objects. Then we used the session to get desired destination #3 and created an appropriate consumer #4. Finally, we are ready to assign a listener to the consumer #5 and start the connection #6. In this example, we left the listener #7 as simple as possible, so it will just print XML data we received in a message.

To compile this example on the Mono platform, you have to use Mono C# compiler `gmcs` (the one that targets 2.0 runtime). Running the following command:

```
$ gmcs -r:Apache.NMS.ActiveMQ.dll -r:Apache.NMS.dll Consumer.cs
```

assuming that you have appropriate NMS DLLs should produce the `Consumer.exe`

binary. We can run this application with the following command:

```
$ mono Consumer.exe
Press any key to quit.
<?xml version="1.0" ?><stock name="JAVA"><price>43.01361850880874</price><offer>43.05663212
<?xml version="1.0" ?><stock name="JAVA"><price>43.39372871092703</price><offer>43.43712243
<?xml version="1.0" ?><stock name="JAVA"><price>43.31253506864452</price><offer>43.35584760
<?xml version="1.0" ?><stock name="JAVA"><price>43.579419162289334</price><offer>43.62299851
<?xml version="1.0" ?><stock name="JAVA"><price>43.268719403943344</price><offer>43.31198811
<?xml version="1.0" ?><stock name="JAVA"><price>43.03515076051564</price><offer>43.07818591
<?xml version="1.0" ?><stock name="JAVA"><price>42.75679069998244</price><offer>42.799547490
```

As this simple example showed, connecting to ActiveMQ from C# is as simple (and practically the same) as from Java. Now let's see what options C++ developers have if they want to use messaging with ActiveMQ.

9.4. Introducing CMS (C++ Messaging Service) API

Although the focus of software developers in recent years was primarily on languages with virtual machines (such as Java and C#) and dynamic languages (Ruby for examples), there are still a lot of development done in "native" C and C++ languages. Similarly to NMS, *CMS (C++ Messaging Service)* represents a standard C++ interface for communicating with messaging systems.

ActiveMQ-CPP, current implementation of the CMS interface, supports both OpenWire and Stomp protocols. Although having a Stomp in a toolbox could be useful in some use cases, I believe the most of the C++ developers will take the OpenWire route for its better performances.

CMS is also one of the ActiveMQ subprojects and you can find more info on how to obtain and build it on its homepage: <http://activemq.apache.org/cms/>. In the rest of this section we will focus on the simple asynchronous consumer example that comes with the distribution. You can find the original example in the following file `src/examples/consumers/SimpleAsyncConsumer.cpp`. We will modify it to listen and consume messages from one of our stock portfolio topics. Since the overall example is too long for the book format, we will divide it into a few code listings

and explain it section by section.

First of all, our `SimpleAsyncConsumer` class implements two interfaces:

- `MessageListener` - used to receive asynchronously delivered messages
- and `ExceptionListener` - used to handle connection exceptions

Example 9.15. Listing 8.15: C++ Consumer

```
class SimpleAsyncConsumer : public ExceptionListener,  
                           public MessageListener
```

The `MessageListener` interface defines the `onMessage()` method, which handles received messages. In our example it boils down to printing and acknowledging the message.

Example 9.16. Listing 8.16: C++ Message Listener

```
virtual void onMessage( const Message* message ) {  
  
    static int count = 0;  
  
    try  
    {  
        count++;  
        const TextMessage* textMessage =  
            dynamic_cast< const TextMessage* >( message );  
        string text = "";  
  
        if( textMessage != NULL ) {  
            text = textMessage->getText();  
        } else {  
            text = "NOT A TEXTMESSAGE!";  
        }  
  
        if( clientAck ) {  
            message->acknowledge();  
        }  
  
        printf( "Message #%d Received: %s\n", count, text.c_str() );  
    } catch ( CMSEException& e ) {  
        e.printStackTrace();  
    }  
}
```

Please post comments or corrections to the [Author Online Forum](#)

The `ExceptionListener` interface defines the `onException()` method called when connection problems are detected.

Example 9.17. Listing 8.17: C++ Exception Listener

```
virtual void onException( const CMSException& ex AMQCPP_UNUSED) {
    printf("CMS Exception occurred. Shutting down client.\n");
}
```

As you can see, thus far CMS mimics the JMS API completely, which is great for developers wanting to create cross-platform solutions.

The complete code related to creating and running a consumer is located in the `runConsumer()` method. Here, we have all classic elements of creating a consumer with the appropriate message listener as we have seen in our Java examples. We create a connection, session and destination objects first and then instantiate a consumer and adds this object as a message listener.

Example 9.18. Listing 8.18: C++ creating Consumer

```
void runConsumer() {
    try {
        ActiveMQConnectionFactory* connectionFactory =
            new ActiveMQConnectionFactory( brokerURI );
#A
        connection = connectionFactory->createConnection();
#B
        delete connectionFactory;
#C
        connection->start();

        connection->setExceptionListener(this);

        if( clientAck ) {
            session = connection->createSession( Session::CLIENT_ACKNOWLEDGE );
        } else {
            session = connection->createSession( Session::AUTO_ACKNOWLEDGE );
        }
#D
        if( useTopic ) {
            destination = session->createTopic( destURI );
        } else {
            destination = session->createQueue( destURI );
        }
#E
    }
}
```

Please post comments or corrections to the [Author Online Forum](#)

```

        consumer = session->createConsumer( destination );
        consumer->setMessageListener( this );

    } catch (CMSEException& e) {
        e.printStackTrace();
    }
}

#A Define Connection Factory
#B Create Sonnection
#C Start Sonnection
#D Create Session
#E Create Destination
#F Create Consumer
#G Add Message Listener

```

All that is left to be done is to initialize everything and run the application.

Example 9.19. Listing 8.19: C++ main method

```

int main(int argc AMQCPP_UNUSED, char* argv[] AMQCPP_UNUSED) {

    std::cout << "=====\\n";
    std::cout << "Starting the example:" << std::endl;
    std::cout << "-----\\n";

    std::string brokerURI =
        "tcp://127.0.0.1:61616"
        "?wireFormat=openwire"
        "&transport.useAsyncSend=true"
        "&wireFormat.tightEncodingEnabled=true";

    std::string destURI = "STOCKS.JAVA";

    bool useTopics = true;

    bool clientAck = false;

    SimpleAsyncConsumer consumer( brokerURI, destURI, useTopics, clientAck );

    consumer.runConsumer();

    std::cout << "Press 'q' to quit" << std::endl;
    while( std::cin.get() != 'q' ) {}

    std::cout << "-----\\n";
    std::cout << "Finished with the example." << std::endl;
    std::cout << "=====\\n";

```

Please post comments or corrections to the [Author Online Forum](#)

As you can see, we have configured it to listen one of our stock portfolio topics #2. Additionally, you can notice that we have used the OpenWire protocol #1 in this example. If you want to try the Stomp connector, just change the value of the wireFormat query parameter to stomp.

Now, we can rebuild the project with:

```
$ make
```

and run the example with:

```
$ src/examples/simple_async_consumer
=====
Starting the example:
-----
Press 'q' to quit
Message #1 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.33014546680271</price>
Message #2 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.638920307293986</price>
Message #3 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.828934270661314</price>
Message #4 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.41909588529107</price>
Message #5 Received: <?xml version="1.0" ?><stock name="JAVA"><price>53.95595590764363</price>
Message #6 Received: <?xml version="1.0" ?><stock name="JAVA"><price>53.74094054512154</price>
Message #7 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.24485518988984</price>
Message #8 Received: <?xml version="1.0" ?><stock name="JAVA"><price>53.72415991559902</price>
Message #9 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.091504162570935</price>
Message #10 Received: <?xml version="1.0" ?><stock name="JAVA"><price>53.960072785363025</price>
Message #11 Received: <?xml version="1.0" ?><stock name="JAVA"><price>53.82082586259414</price>
Message #12 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.032852016137156</price>
Message #13 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.31012714496037</price>
Message #14 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.76169437095268</price>
Message #15 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.92747610279041</price>
Message #16 Received: <?xml version="1.0" ?><stock name="JAVA"><price>55.074451578258646</price>
Message #17 Received: <?xml version="1.0" ?><stock name="JAVA"><price>55.56804101263522</price>
Message #18 Received: <?xml version="1.0" ?><stock name="JAVA"><price>55.77722621685933</price>
Message #19 Received: <?xml version="1.0" ?><stock name="JAVA"><price>55.58192302489994</price>
Message #20 Received: <?xml version="1.0" ?><stock name="JAVA"><price>55.561645147383295</price>
```

Thus far we have seen how Stomp can be used to create simple messaging clients for practically any programming language. We have also seen how NMS and CMS subprojects help create more complex, JMS-like, APIs for environments that deserve this kind of support. Now let's focus on another very important development platform and that is Web.

9.5. Messaging on the Web

In the last couple of years we witnessed the rebirth of Web, usually called *Web 2.0*. The transformation is taking place in two particular aspects of software development:

- *Service-oriented architecture (SOA)* and *Web services* play increasingly more important role for many software projects. Users demand that software functionality is exposed through some kind of web service interface. One of the ways to achieve this is to introduce RESTful principles to your application architecture, which allows you to expose your application resources over HTTP. ActiveMQ follows these principles by exposing its resources through its *REST API*, as we will see in the moment.
- It's easy to say that *Asynchronous JavaScript and XML (AJAX)* revolutionized the web development as we knew it. The possibility to achieve asynchronous communication between the browser and the server (without page reloading) opened many doors for web developers and provided a way for web applications to become much more interactive. Naturally, you can use ActiveMQ *Ajax API* to communicate directly to the broker from your Web browser, which adds even more asynchronous communication possibilities between clients (JavaScript browser code) and servers (backend server application(s)).

In the rest of this section we will explore REST and Ajax APIs provided by ActiveMQ and how you can utilize them in your projects.

9.5.1. Using REST API

As you probably know, the term *REST* first appeared in Roy T. Fielding's PhD thesis "Architectural Styles and the Design of Network-based Software Architectures" (<http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>). In this work Fielding explains a collection of network architecture principles which defines how to address and manage resources (in general) over the network. In

Please post comments or corrections to the [Author Online Forum](#)

simpler terms however, if application implements a RESTful architecture it usually means that it exposes its resources using HTTP protocol and in a similar philosophy to those used on the World Wide Web (the Web).

The Web is designed as a system for accessing documents over the Internet. Every resource on the Web (HTML page, image, video, etc.) has a unique address defined by its URL (Unified Resource Locator). Resources are mutually interlinked and transferred between clients and servers using the HTTP protocol. HTTP GET method is used to obtain the representation of the resource and shouldn't be used to make any modifications to it. The POST method, on the other hand, is used to send data to be processed by the server. Apply these principles to your application's resources (destinations and messages in case of a JMS broker) and you have defined a RESTful API. Now let's see how ActiveMQ implements its REST API and how you can use it to send and receive messages from the broker.

ActiveMQ comes with the embedded Web server which starts at the same time your broker starts. This web server is used to provide all necessary Web infrastructure for ActiveMQ broker, including the REST API. By default, the demo application is started at:

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

and it is also configured to expose the REST API at the following URL:

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

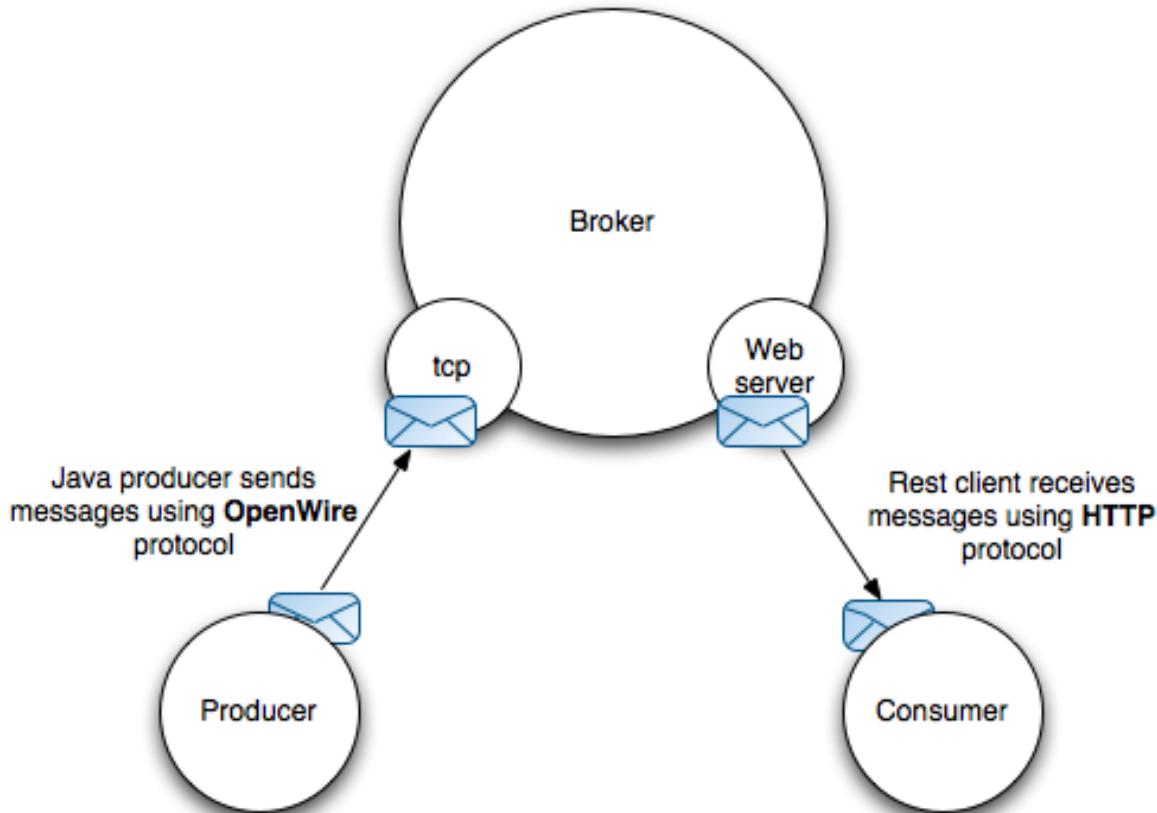


Figure 9.2. Figure 8.2: Rest Example

The API is implemented by the `org.apache.activemq.web.MessageServlet` servlet and if you wish to configure an arbitrary servlet container to expose the ActiveMQ REST API, you have to define and map this servlet in an appropriate `web.xml` file (of course, all necessary dependencies should be in your classpath). The following listing shows how to configure and map this servlet to the `/message` path as it is done in the demo application.

Example 9.20. Listing 8.20: REST configuration

```
<servlet>
  <servlet-name>MessageServlet</servlet-name>
  <servlet-class>org.apache.activemq.web.MessageServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

Please post comments or corrections to the [Author Online Forum](#)

```
<servlet-mapping>
    <servlet-name>MessageServlet</servlet-name>
    <url-pattern>/message/*</url-pattern>
</servlet-mapping>
```

When configured like this, broker destinations are exposed as relative paths under the defined URI. For example, the STOCKS.JAVA topic is mapped to the following URI:

```
http://localhost:8161/demo/message/STOCKS/JAVA?type=topic
```

As you can see there is a path translation in place, so destination name path elements (separated with .) are adjusted to the Web URI philosophy where / is used as a separator. Also, we used the type parameter to define whether we want to access a queue or a topic.

Now we can use GET and POST requests to receive and send messages to destinations (retrospectively). We will now run simple examples to demonstrate how you can use the REST API to communicate with your broker from the command line. For that we will use two popular programs that can make HTTP GET and POST method requests from the command line. First we will use GNU Wget (<http://www.gnu.org/software/wget/>), a popular tool for retrieving files using HTTP, to subscribe to the desired destination.

Example 9.21. Listing 8.21: REST consume

```
$ wget -O message.txt \
--save-cookies cookies.txt --load-cookies cookies.txt --keep-session-cookies \
http://localhost:8161/demo/message/STOCKS/JAVA?type=topic
```

With this command we instructed wget to receive next available message from the STOCKS.JAVA topic and to save it to the message.txt file. You can also notice that we keep HTTP session alive between wget calls by saving and sending cookies back to the server. This is very important because the actual consumer API we use is stored in the particular session. So if you try to receive every message from a

new session, you will spawn a lot of consumers and your requests will be probably left hanging. Also, if you plan to use multiple REST consumers it is advisable to set the prefetch size to 1, just as we were doing with Stomp consumers. To do that, you have to set `consumer.prefetchSize` initialization parameter value of your message servlet. The following example shows how to achieve that:

```
<servlet>
    <servlet-name>MessageServlet</servlet-name>
    <servlet-class>org.apache.activemq.web.MessageServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
    <init-param>
        <param-name>destinationOptions</param-name>
        <param-value>consumer.prefetchSize=1</param-value>
    </init-param>
</servlet>
```

Now, it's time to send some messages to our topic. For that we will use cUrl (<http://curl.haxx.se/>), a popular command line tool for transferring files using HTTP POST method. Take a look at the following command:

Example 9.22. Listing 8.22: REST produce

```
$ curl -d "body=message" http://localhost:8161/demo/message/STOCKS/JAVA?type=topic
```

Here we have used the `-d` switch to specify that we want to POST data to the server. As you can see, the actual content of the message is passed as the `body` parameter. The sent message should be received by our previously ran consumer.

This simple example showed us how easy it is to use the REST API to do asynchronous messaging even from the command line. But generally, you should give Stomp a try (if it is available for your platform) before falling back to the REST API, because it allows you more flexibility and is much more messaging-oriented.

9.5.2. Understanding Ajax API

As we already said, the option to communicate with the web server asynchronously changed a perspective most developers had towards web applications. In this

section we will see how web developers can embrace asynchronous programming even further, by communicating with message brokers directly from JavaScript.

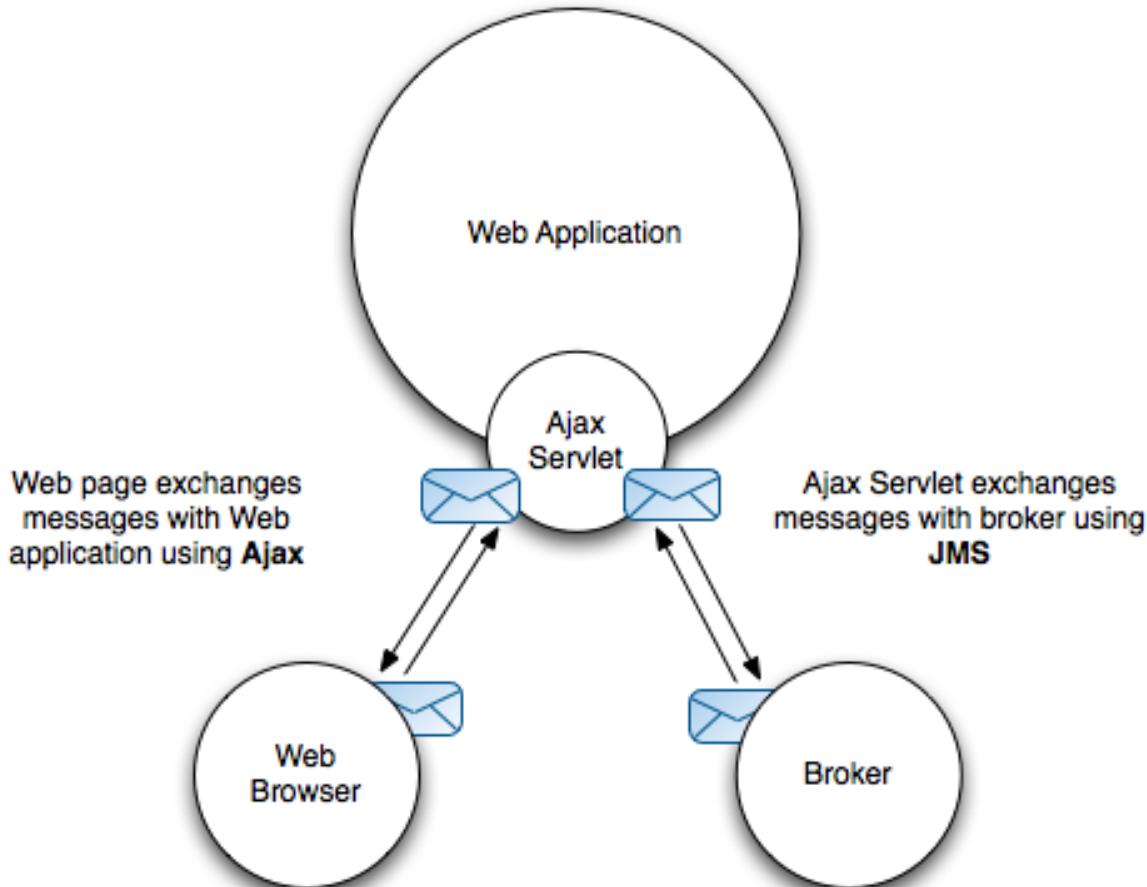


Figure 9.3. Figure 8.3: Ajax API

First of all, of course, we should configure our web server to support ActiveMQ Ajax API. Similarly to the `MessageServlet` class used for implementing the REST API, ActiveMQ provides an `AjaxServlet` that implements Ajax support. The following listing shows how to configure it in your web application's `WEB-INF/web.xml` file.

Example 9.23. Listing 8.23: Ajax server configuration

```
<servlet>
```

Please post comments or corrections to the [Author Online Forum](#)

```
<servlet-name>AjaxServlet</servlet-name>
<servlet-class>org.apache.activemq.web.AjaxServlet</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>AjaxServlet</servlet-name>
<url-pattern>/amq/*</url-pattern>
</servlet-mapping>
```

Of course, in order to make it work properly you have to put ActiveMQ in your web application's classpath. Now that we have a server side configured and a servlet listening to the requests submitted to the URIs starting with /amq/, we can proceed to implementing a client side of our Ajax application.

First of all, we have to include the `amq.js` script, which includes all necessary JavaScript libraries for us. Also, we have to point the `amq.uri` variable to the URI our Ajax servlet listens to. The Listing 8.24 shows how to achieve this.

Example 9.24. Listing 8.24: Ajax client configuration

```
<script type="text/javascript" src="amq/amq.js"></script>
<script type="text/javascript">amq.uri = '/amq';</script>
```

The `amq.js` script defines the JavaScript object named `amq`, which provides an API for us to send messages and subscribe to ActiveMQ destinations. The following example shows how to send a simple message from our Ajax application:

Example 9.25. Listing 8.25: Ajax produce

```
amq.sendMessage("topic://TEST", "message");
```

It can't be much simpler than this, all you have to do is call a `sendMessage()` method and provide a destination and text of the message to be sent.

If you wish to subscribe to the certain destination (or multiple destinations) you have to register a callback function which will be called every time a new message

is available. This is done with the `addListener()` method of the `amq` object, which beside a callback function accepts a destination to subscribe to and `id` which makes further handling of this listener possible.

The ActiveMQ demo application comes with the stock portfolio example we have used throughout the book adopted to the web environment. The example contains a servlet that publishes market data and a web page that uses the Ajax API to consume those data. Using this example, we will show how to consume messages using the Ajax API. Let's take a look at the code shown in Listing 8.26:

Example 9.26. Listing 8.26: Ajax consume

```

var priceHandler =
{
    _price: function(message)                                #1
    {
        if (message != null) {

            var price = parseFloat(message.getAttribute('bid'))
            var symbol = message.getAttribute('stock')
            var movement = message.getAttribute('movement')
            if (movement == null) {
                movement = 'up'
            }

            var row = document.getElementById(symbol)
            if (row) {
                // perform portfolio calculations
                var value = asFloat(find(row, 'amount')) * price
                var pl = value - asFloat(find(row, 'cost'))

                // now lets update the HTML DOM
                find(row, 'price').innerHTML = fixedDigits(price, 2)
                find(row, 'value').innerHTML = fixedDigits(value, 2)
                find(row, 'pl').innerHTML = fixedDigits(pl, 2)
                find(row, 'price').className = movement
                find(row, 'pl').className = pl >= 0 ? 'up' : 'down'
            }
        }
    }
};

function portfolioPoll(first)
{
    if (first)
    {
        amq.addListener('stocks','topic://STOCKS.*',priceHandler._price);  #2
    }
}

```

Please post comments or corrections to the [Author Online Forum](#)

```
    }  
  
    amq.addPollHandler(portfolioPoll);
```

For starters, we have defined a JavaScript object named `priceHandler` with the `_price()` function #1 we will use to handle messages. This function finds an appropriate page element and updates its value (or change its class to show whether it is a positive or negative change). Now we have to register this function to listen to the stock topics #2. As you can see we have named our listener `stocks`, set it to listen to all topics in the `STOCKS` name hierarchy and defined `_price()` as a callback function. You can later remove this subscription (if you wish) by calling the `removeListener()` function of the `amq` object and providing the specified id (`stocks` in this case).

Now we are ready to run this example. First we are going to start the portfolio publisher servlet by entering the following URL in the browser:

```
http://localhost:8161/demo/portfolioPublish?count=1&refresh=2&stocks=IBMW&stocks=BEAS&stocks=GE&stocks=MSFT
```

The Ajax consumer example is located at the following address:

```
http://localhost:8161/demo/portfolio/portfolio.html
```

and after starting it you can expect the page that looks similar to the one shown in Figure 8.4.

My Portfolio

This example displays an example stock portfolio. In a real system this page would be generated dynamically based on the users current stock portfolio

Stock	Description	Amount	Price	Value	Cost	P & L
IBMW	IBM Stock	1000	86.60	86598.74	19000	67598.74
MSFT	Microsoft	6000	21.07	126405.80	22000	104405.80
BEAS	BEA Stock	1100	32.34	35574.76	12342	23232.76
SUNW	Sun Microsystems Inc	3000	0.77	2299.17	7700	-5400.83

Figure 9.4. Figure 8.4: Ajax example

The page will dynamically update as messages come to the broker. This simple example shows how Ajax applications can benefit from asynchronous messaging and thus leverage dynamic web pages to the entirely new level.

9.6. Summary

In this chapter we covered a wide range of technologies (protocols and APIs) which allow developers to connect to ActiveMQ from practically any development platform used today. This implies that ActiveMQ could be seen not only as a JMS broker but the whole development platform as well, especially when you add Enterprise Integration Patterns (EIP) to the mix (as we will see in Chapter 13). These wide range of connectivity options makes ActiveMQ an excellent tool for integrating applications written on different platforms in an asynchronous way.

With this chapter we have finished Part 3 of the book, called *Using ActiveMQ*, in which we described many aspects of how you can employ ActiveMQ in your projects. The next, final, part of the book is called *Advanced ActiveMQ* and it will dive into wide range of topics, such as broker topologies, performance tuning, monitoring, etc. Now that you know all the basics of ActiveMQ, this final part should teach you how to use your ActiveMQ broker instances to the maximum.

We will start by continuing our discussion started in Chapter 3, Understanding Connectors, regarding network connectors. The following chapter discusses various broker topologies and how they can help you implement functionalities such as load balancing and high availability.



Please post comments or corrections to the [Author Online Forum](#)

Part IV. Advanced Features in ActiveMQ

In some environments, it is necessary to utilize multiple brokers in a federated manner. This brings about the need to understand the various topologies supported by ActiveMQ. Or when building applications that utilize messaging, it quickly becomes evident that more advanced features are needed such as various administrative capabilities, tuning to support larger scale and monitoring of different aspects of ActiveMQ.

Part IV begins with the topic of broker topologies followed up by covering advanced broker features. It then moves on to discuss some tuning procedures and finishes up with administration and monitoring techniques for ActiveMQ.

Chapter 10. Broker Topologies

In this chapter we focus on enterprise deployments of ActiveMQ message brokers. We will show you how to configure ActiveMQ for high availability, how to deploy ActiveMQ for wide-area networks and how to scale ActiveMQ to handle thousands of connections and tens of thousands of destinations.

10.2. Broker High Availability

When an application is deployed into a production environment, it is always a good idea to plan for the inevitable - system outages. Such outages as network failures, hardware failures, software failures, you get the idea. When trying to mitigate software failures with ActiveMQ, there are three different types of master/slave configurations to provide high availability.

A master/slave style of configuration in ActiveMQ refers to ...

10.2.1. Shared Nothing Master/Slave

A shared nothing master slave refers to the deployment where both the master and the slave have their own message storage. This is probably the easiest option to set to provide high availability of message brokers. A slave is configured to connect to another broker, the master, but is only the slave that needs additional configuration denoting its special state. There is some optional configuration for the master, but we will cover that later.

All state (messages, acknowledgements, subscriptions, transactions etc) are replicated from the master to the slave - as depicted in Figure 10.1 below:

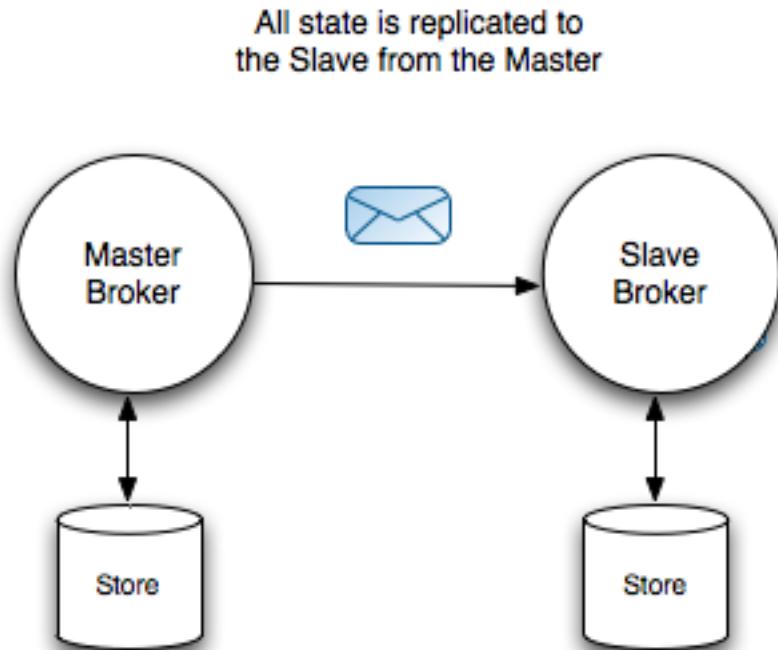


Figure 10.1. 'Shared Nothing' master/slave

A slave broker will connect to the master at startup, so the master needs to be running first. The slave broker will not start any transports (so cannot accept any client or network connections) and will not itself initiate any network connections unless the master fails. Master failure is detected by loss of connectivity from the slave to the master.

A 'Shared Nothing' master/slave configuration does impose extra overhead on message processing. When a Message Producer sends a persistent message to the master, it will wait for a receipt from the master until it can send the next message. The master will not send the receipt until it has passed the message to the slave - and in turn wait for the slave to finish its processing of the message (which will typically involve the slave persisting it to storage). The master will then process the message (persist it to storage and dispatch it to any interesting consumers) before sending back a receipt to the Message Producer that it has successfully processed the message.

When a master fails, the slave can either:

- Shutdown - hence its only acting to preserve the state of the master. In this scenario, an administrator will typically have to configure the slave to be the master - and configure a new message broker to take the role of the slave
- Start up its transports and initiate any network connections - hence the slave now becomes the master

If the slave takes over the role of the master, all clients with failover configured, will failover to the new master. For JMS clients to ActiveMQ the default transport used by the client's Connection is failover - and is typically configured to be able to connect to both the master and the slave:

Example 10.1. Typical failover URI

```
failover://(tcp://masterhost:61616,tcp://slavehost:61616)?randomize=false
```

So there are some limitations with AcitveMQ's 'Shared Nothing' master/slave configuration. A master will only replicate its active state from the time the slave connects to it. So if a clients are using the master before the slave is attached, any messages or acknowledgements that have been processed by the master before the slave has attached itself will be lost. You can avoid this by setting the **waitForSlave** property on the master - see the next section below.

If you already have a running broker that you want to use a 'Shared Nothing' configuration for - it is recommended that you frst stop the broker - copy all message store files (usually from the data directory) to the slave machine - and after configuring, restart the broker (now the master) and the slave.

Having covered the theory, lets look at how to configure a 'Shared Nothing' master/slave.

10.2.1.1. Configuring 'Shared Nothing' master/slave

Designating that a broker is a slave is very straightforward - you configure a masterConnector service that takes parameters for the following:

Please post comments or corrections to the [Author Online Forum](#)

- remoteURI - the transport the master is listening on.
- userName - optional user name if the master is using security
- password - optional password if the master is using security

An example snippet of slave configuration is below

Example 10.2. Configuring a broker to be a slave

```
<services>
  <masterConnector remoteURI= "tcp://localhost:62001" userName="Rob" password="Davies" />
</services>
```

You would normally configure the slave to have duplicate transport and network configurations as the master broker.

There is one additional optional property that you can be useful for a slave in a 'Shared Nothing' configuration:

slave Broker properties are shown in Table 10.1, “”.

Table 10.1.

property name	default value	description
shutdownOnmasterFailure	false	The slave will shut down when the master does.

You can designate a broker to be a master without any additional configuration - but there is some optional properties that may be useful:

master Broker properties are shown in Table 10.2, “”.

Table 10.2.

property name	default value	description
waitForSlave	false	The master will not allow any client or network connections until a slave has attached itself.
shutdownOnSlaveFailure	false	IF true, the master will shutdown if a slave becomes detached. This ensures that a slave is only ever in sync with the master.

Having exposed the configuration options, we should look at when you should use a 'Shared Nothing' master/slave configuration.

10.2.1.2. When to use 'Shared Nothing' master/slave

So 'Shared Nothing' master/slave is cheap and cheerful - and setup and recovery can involve a lot of manual intervention, especially if you want to use it to add high availability to an existing running broker.

It is a useful configuration to know about though - it can be used in situations where using a shared database or shared filesystem isn't an option. We are going to cover shared storage configuration in the next section.

10.2.2. Shared Database Master/Slave

If you are already using an enterprise relational database for message storage, then providing broker high availability is extremely straightforward. When an ActiveMQ Message Broker uses a relational database, it grabs an exclusive lock on a table - ensuring that no other ActiveMQ broker can access the database at the same time - as depicted in Figure 10.2 below:

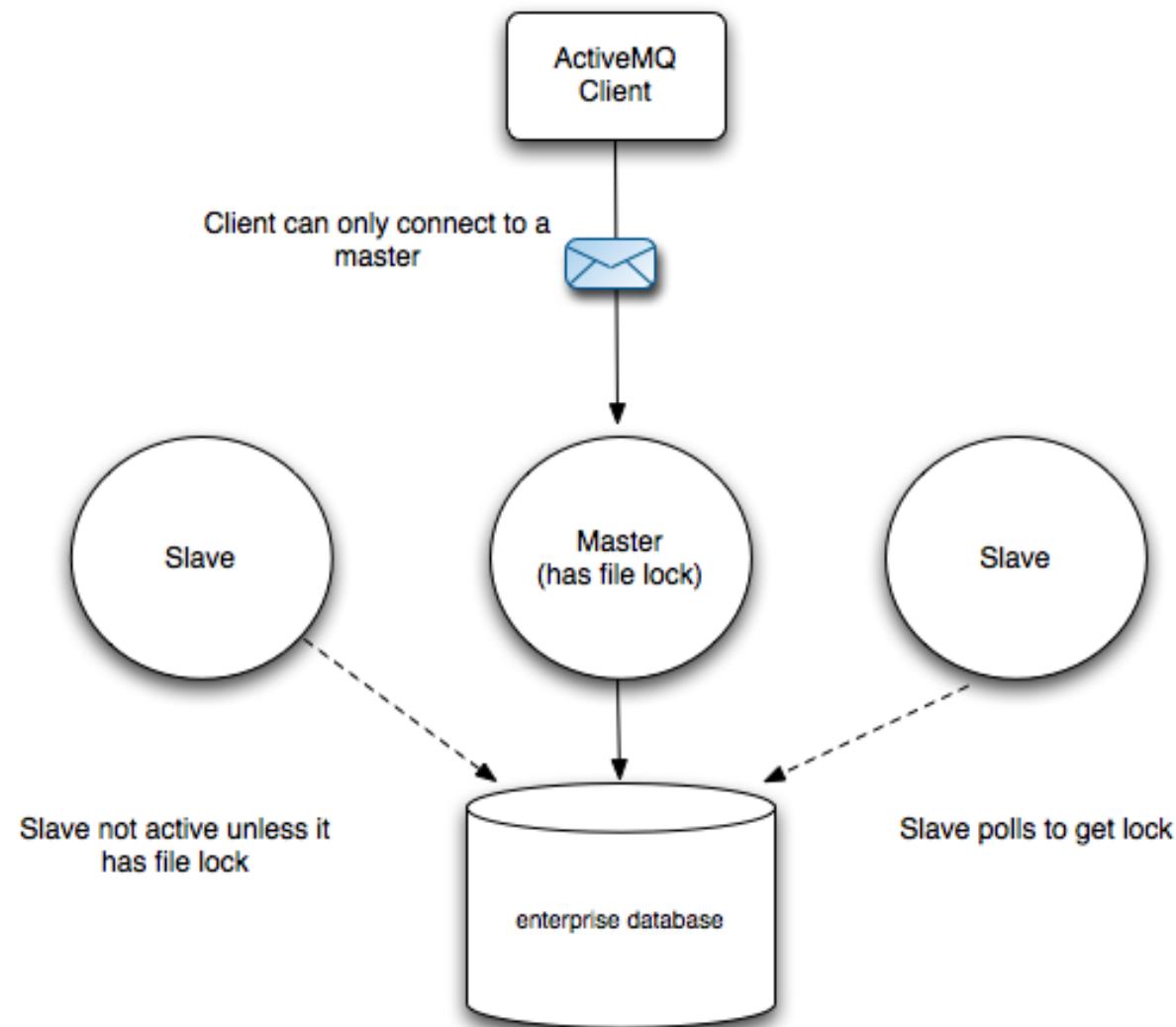


Figure 10.2. Shared Database master/slave

If you are running more than one broker that is trying to access the same database, only the first one to connect will grab the lock, any other brokers will poll until they can get access to the lock - and whilst in this polling state, the ActiveMQ broker assumes its a slave, so will not start any transport connections or network connections.

So you can run multiple brokers (there is not a restriction on how many), one and only one broker will ever be the master. All the brokers in this configuration can use the same configuration file - making setup extremely straightforward.

However, because of the way the JDBC Journal works, you cannot use a Journal with JDBC with this configuration.

10.2.2.1. When to use shared database master/slave

Shared database master/slave is an ideal configuration if you are already using an enterprise relational database. Although generally slower than a 'Shared Nothing' configuration, its requires no additional configuration and there is no limitations on the number of slave brokers that can be run.

However for a green field site, where using an enterprise relational database is not an option, then there is an alternative to both 'Shared Nothing' and shared database, using a shared file system, which we will cover next.

10.2.3. Shared File system Master/Slave

An alternative to using a shared enterprise database is to use a shared filesystem. The setup is very similar to the shared database Master/Slave in that no additional configuration is required. You have to use either the default message store, or KahaDB however. When either of these two message stores start, they grab a file lock, to prevent any other broker accessing them at the same time- as shown in Figure 10.3

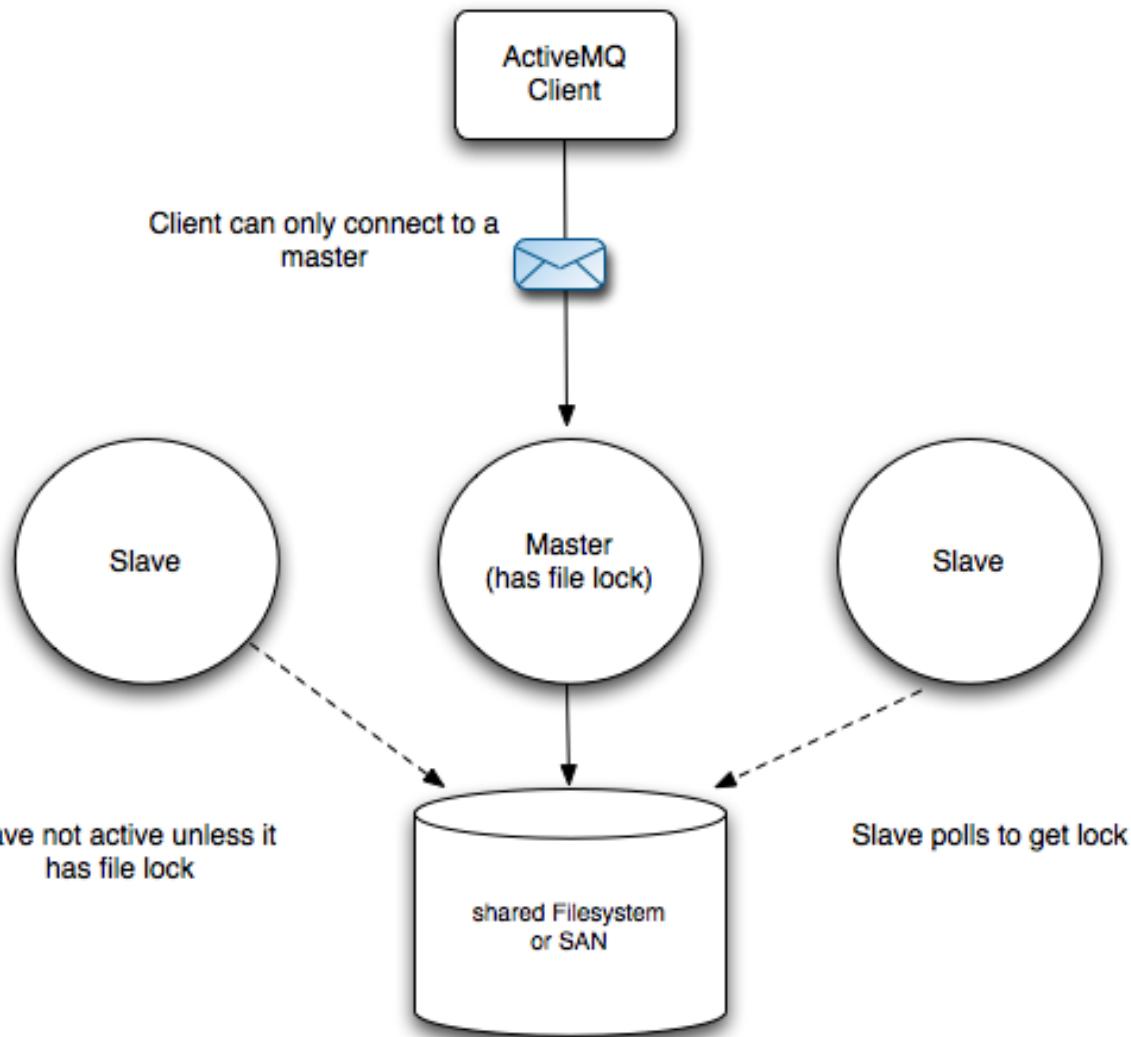


Figure 10.3. Shared File System master/slave

This means that just like the shared database Master/Slave configuration, there is no restriction on the number of slaves that can be started, the first becomes the master and rest become slaves, awaiting for the master to die.

There are some technical restrictions on where you can run a shared file system Master/Slave configuration. The shared file system needs to support the semantics of a distributed shared file lock, so if you are not using a storage area network (SAN) there are some alternatives like Network File System (NFS) version 4 and above and Global File System (GFS) 2 - part of RedHat Enterprise 5.3 and above.

10.2.3.1. When to use shared file system master/slave

Using a shared file system is probably the best solution for providing high availability for ActiveMQ to date. It combines the high throughput of the AMQMessageStore or KahaDB (though will be limited by the underlying performance of the shared file system) and the simplicity that you get from using a shared resource.

The only caveat is that you are restricted to environments that support distributed locking on a shared file system.

10.3. Networks of Brokers

ActiveMQ supports the concept of linking ActiveMQ message brokers together into different topologies or networks. Often it's the requirement that geographically dispersed applications need to communicate in a reliable way, where having a centralized broker architecture will not be the optimal way of passing messages. So you can use ActiveMQ networks, to pass messages from one domain to another.

ActiveMQ networks use store and forward; messages are always stored at the local broker before being passed across a network to another broker. This means that if messages can't be delivered because connectivity might not be available (you could be using a satellite link for example) - when the connection is re-established, a broker will be able to send any unsent messages.

In the next section we will walk through one very common usage pattern for ActiveMQ networks, store and forward.

10.3.1. Store and Forward

By default, a network is only operate in one direction, and logically pushes messages across its network connection - as shown in Figure 10.4

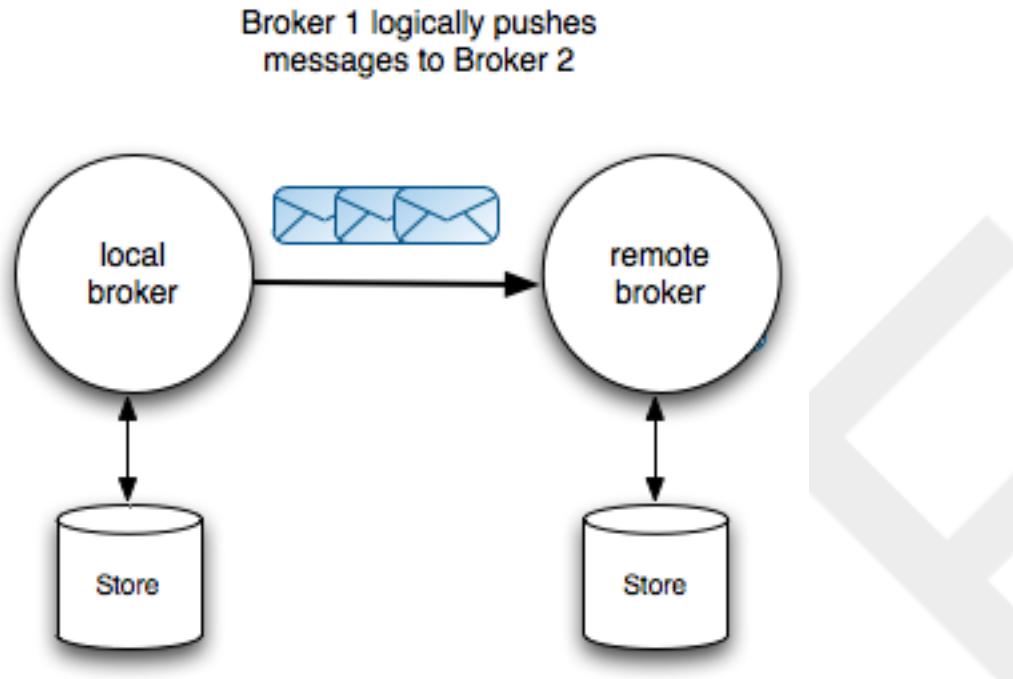


Figure 10.4. Network Connection

When a network is established from a local broker to a remote broker, the remote broker will pass information containing all its durable and active consumers destinations to the local broker. The local broker uses this information to determine what messages the remote broker is interested and forward them to the remote broker. It is possible to define filters on the network connection - and to always include or exclude messages for a particular destination, but we will cover that in the configuration section later in this chapter.

Having networks operate in one direction allows for networks to be configured for message passing one way only. If you want networks to work bi-directionally, you can either configure the remote broker with a network connector to point to the local broker, or configure a network to be duplex, to work in both directions.

Suppose you have a deployment scenario where you have a lot of remote stores that need to connect to a back office order system. It would be hard to configure new stores and inflexible for the the broker(s) at the back office to know about all the remote brokers. In addition, the back office brokers would typically be behind

firewall, with only a limited number of ports open to accept connections inwards - as depicted below Figure 10.5

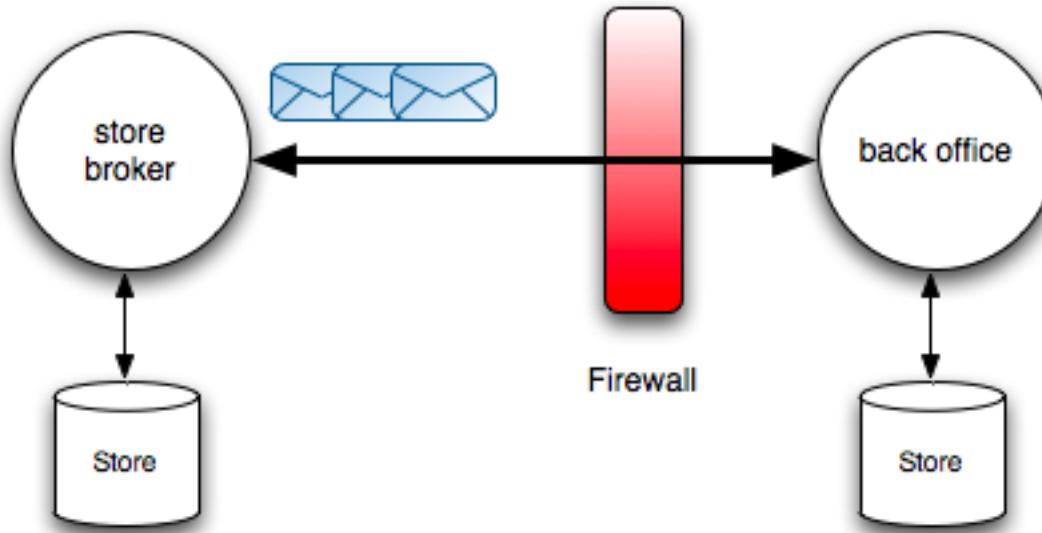


Figure 10.5. Store and Back office

Configuring the network used by the store broker in this case needs to be duplex. The single network connection, established from the store broker to the back office would be able to pass messages in both directions - and would behave in the same way as if the back office broker had established a normal network connection back to the store broker.

The configuration for the store broker would include configuration for the network connector, that would look something like this:

Example 10.3. Configuring a Store Network Broker

```
<networkConnectors>
  <networkConnector uri="static://(tcp://backoffice:61617)"
    name="bridge"
    duplex="true"
    conduitSubscriptions="true"
    decreaseNetworkConsumerPriority="false">
  </networkConnector>
</networkConnectors>
```

Please be aware that the order in which you specify the network connections and the persistence you use in the ActiveMQ broker configuration is important. Always configure networks, persistence and transports in the following order:

1. networks - they need to be established before the message store
2. message store - should be configured before transports
3. transports - should be the last in the broker configuration

An example broker configuration, in correct order is shown below:

Example 10.4. Correct Configuration Order

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://activemq.org/config/1.0">

    <broker brokerName="receiver" persistent="true" useJmx="true">
        <networkConnectors>
            <networkConnector uri="static:(tcp://backoffice:61617)"/>
        </networkConnectors>

        <persistenceAdapter>
            <kahaDB/>
        <kahaDB></kahaDB>

        </persistenceAdapter>

        <transportConnectors>
            <transportConnector uri="tcp://localhost:62002"/>
        </transportConnectors>
    </broker>

</beans>
```

In large deployment scenarios it makes sense to combine high availability and network configurations, as below in Figure 10.6

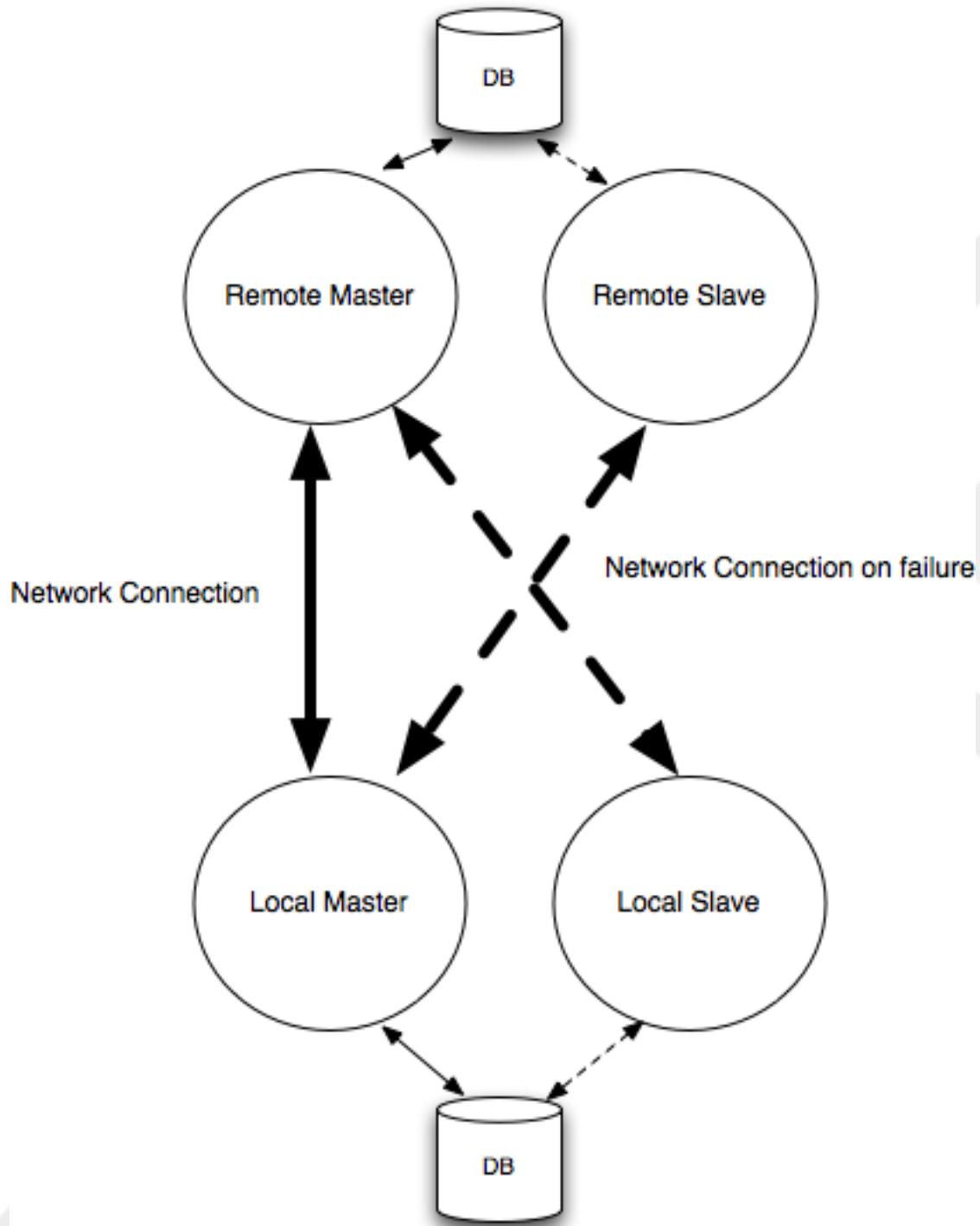


Figure 10.6. Combining High Availability and networks

Please post comments or corrections to the [Author Online Forum](#)

In this scenario, the local master and slave broker both need to be configured to create network connections to both the remote master and the slave to cater for the remote master failing. Only when a slave becomes active (becomes the new master), will it start its transports and network connections.

If a network cannot establish a connection to a remote broker (as in the case of a remote slave), or the network connection fails, the network will poll the connection until it can establish a connection.

10.3.2. Network Discovery

When a network connection is established to a remote broker, it uses a discovery agent to locate the remote broker and re-establish that connection if the connection fails (for example, a network outage).

There are two main types of network discovery that come out of the box with ActiveMQ:

1. dynamic - search for brokers to connect (by using multicast or rendevous)
 2. static - configured with list of urls to connect to, to establish a connection
- Using multicast discovery is to create network connections is extremely straightforward. When you start an ActiveMQ broker with multicast, it will search for any other broker using multicast and establish a connection. A network configuration for multicast discovery looks something like this:

Example 10.5. Multicast discovery for Networks

```
<networkConnectors>
    <networkConnector uri="multicast://default"/>
</networkConnectors>
```

The "default" name in the **multicast://** uri denotes the group that the broker belongs too. Its strongly recommended that when using multicast discovery, you use a unique group name so that your brokers do not connect to other application

brokers - which can lead to surprising results!

One limitation for mmulticast discovery is control, the other is that you are usually restricted to discovery brokers on your local network segment (as multicast invariably does not extend through routers).

The example configurations we have used previously have used static discovery for establishing networks. Although they require a little more configuration - and they wouldn't be suitable for a large number of networks, they are typically used for most deployments.

static discovery can take a list of uris, and will try and connect to the remote brokers in the order they are determined in the list.

So for example, to configure the local Master and local Slave to connect to the remote Master, but fail over to the remote Slave (see Figure 10.6), you would configure the local brokers like this:

Example 10.6. Network configuration for high availability

```
<networkConnectors>
    <networkConnector uri="static:(tcp://remote-master:61617,tcp://remote-slave:61617)"/>
</networkConnectors>
```

The static network discovery agent can be configured to control the frequency that it will try and re-establish a connection on failure. The **static://** configuration properties are as follows:

Table 10.3.

property name	default value	description
initialReconnectDelay	1000	The time in milliseconds before attempting to reconnect the network. This is only used if useExponentialBackOff is not enabled.

Please post comments or corrections to the [Author Online Forum](#)

property name	default value	description
maxReconnectDelay	30000	The maximum time in milliseconds that the network will wait before trying to establish a connection after failure. This is only used if useExponentialBackOff is enabled.
useExponentialBackOff	true	IF this is enabled the network will increase the time to wait between each failed attempt to establish a connection
backOffMultiplier	2	Used in conjunction with useExponentialBackOff, the multiplier to use to increase the time to wait between each new attempt to establish a network connection.

A network connection will always try and establish a connection to a remote broker, so there isn't a concept of just giving up!

You can set the **static://** configuration options as part of the uri - e.g:

Example 10.7. Setting options with static discovery for networks

```
<networkConnectors>
    <networkConnector uri="static:(tcp://remote:61617)?useExponentialBackOff=false"/>
</networkConnectors>
```

Having explained how to combine networks and high availability, we will cover general network configuration in the next section.

10.3.3. Network Configuration

Networks by default rely on the remote broker informing it of to determine if a remote broker has an interest in its local messages. For existing active and new Message Consumers, the destination the Message Consumer is listening to is propagated to the local broker's network connection. The local network connection will then subscribe on behalf of the remote broker's Message Consumers for messages to be forwarded across the network. In order for networks to function properly, the broker property **advisorySupport** needs to be enabled, because advisory messages are used to forward information about changing Message Consumers across networks as well as clients.

There may not be any active Durable TopicSubscribers or consumers for an existing Queue on the remote broker. So when the network connection is initialized to the remote broker, the remote broker will read through its Message Store for existing Destinations and pass those to the local broker, so the local broker can forward messages for those Destinations as well.

It is important to note that a network uses both its name and the name of the broker to create a unique durable subscription proxy on behalf of a remote broker. Hence, if at a later point in time, you change either the network name or the name the broker, you could loose messages over networks for durable topic subscribers.

Now having a basic understanding of how networks operate, you will be able to comprehend some of the side affects if you change the default network configuration. A Network has several important configuration properties, in addition to the **duplex** property, which will explain in a little more detail:

10.3.3.1. Network Property: **dynamicOnly**

This is false by default, but if enabled, messages will only be forwarded if there is an active (connected) consumer on the remote broker.

10.3.3.2. Network Property: `prefetchSize`

The default is 1000, and is the number of messages that will be forwarded across the network for before there is an acknowledgement from the remote broker.

10.3.3.3. Network Property: `conduitSubscriptions`

An ActiveMQ message broker will send a copy of a message to every interested consumer that it is aware of, even across networks. However, this can be a problem, as the the remote broker will send every message receives to any of its interested consumers. So its possible to end up with duplicate messages on the remote broker. The `conduitSubscription` property, when enabled, informs the network connection to treat multiple matching destinations as one to avoid this problem.

10.3.3.4. Network Property: `excludedDestinations`

You can ask the network to exlude destinations from being passed across a network. This can be used, for example to prevent destinaions that should only used by local consumers from being propagated to a remote broker. Excluded destinations are denoted as either a `queue` or a `topic` and use a `physicalName` attribute for the name of the `queue` or `topic` to exclude. You can combine a list of excluded destinations, and use wild cards to denote the names of the destinations to exclude too.

Excluded destinations take priority over both `staticallyIncludedDestinations` and `dynamicallyIncludedDestinations`. So if you have matching destinations in either of those lists, they will be excluded. Here's an example configuration using `excludedDestinations`:

Example 10.8. Setting options for excluded destinations

```
<networkConnectors>
    <networkConnector uri="static:(tcp://remote:61617)?useExponentialBackOff=false"/>
        <excludedDestinations>
            <queue physicalName="audit.queue-1"/>
            <queue physicalName="audit.queue-2"/>
            <queue physicalName="local.>"/>
```

Please post comments or corrections to the [Author Online Forum](#)

```
<topic physicalName="local.>" />
</excludedDestinations>
</networkConnectors>
```

10.3.3.5. Network Property: dynamicallyIncludedDestinations

You can ask the network to only pass messages to the remote broker for active Message Consumers that match the list of destinations for **dynamicallyIncludedDestinations**. The format is the same as for **excludedDestinations**. An empty list denotes that all messages will be passed to the remote broker, as long as they are not in the **excludedDestinations** list.

10.3.3.6. Network Property: staticallyIncludedDestinations

You can ask the network to only pass messages to the remote broker if they match the list of destinations for **staticallyIncludedDestinations**. The format is the same as for **excludedDestinations**, e.g.

Example 10.9. Setting options for included destinations

```
<networkConnectors>
  <networkConnector uri="static:(tcp://remote:61617)?useExponentialBackOff=false">
    <staticallyIncludedDestinations>
      <queue physicalName="management.queue-1" />
      <queue physicalName="management.queue-2" />
      <queue physicalName="global.>" />
      <topic physicalName="global.>" />
    </staticallyIncludedDestinations>
  </networkConnector>
</networkConnectors>
```

10.3.3.7. Network Property: decreaseNetworkConsumerPriority

This property is false by default - if it is set, then the algorithm used to determine which Message Consumer for a queue should receive the next dispatched message, will give a network consumer the lowest priority (which means in reality that messages from a local broker queue will only be sent to a remote broker if

there are no local consumers or they are all busy).

10.3.3.8. Network Property: **networkTTL**

The default value is 1. The **networkTTL** property denotes the maximum number of remote brokers a message can pass through before being discarded. This is useful for ensuring messages are not forwarded needlessly, if you have a cyclic network of connected brokers.

10.3.3.9. Network Property: **name**

The default name is "bridge" - and mostly you can ignore this property - just don't decide to change it after using your broker with networks for a while, because it forms part of the unique key used when proxying durable subscribers across a network.

However, there are cases when it makes sense to have more than one network connection between the same local and remote brokers, in which case, they do require unique names. So why have more than one network connection between the two same brokers ? It comes down to performance. A network connection uses a single transport connection, and if you are anticipating a heavy load across a network, it would make sense to have more than one transport connection. You do need to be careful that you do not get duplicates messages, so you have to setup the network connections with the appropriate filters. Using one for queues and one for topics can often improve throughput for general messages cases - as depicted in Figure 10.7 below:

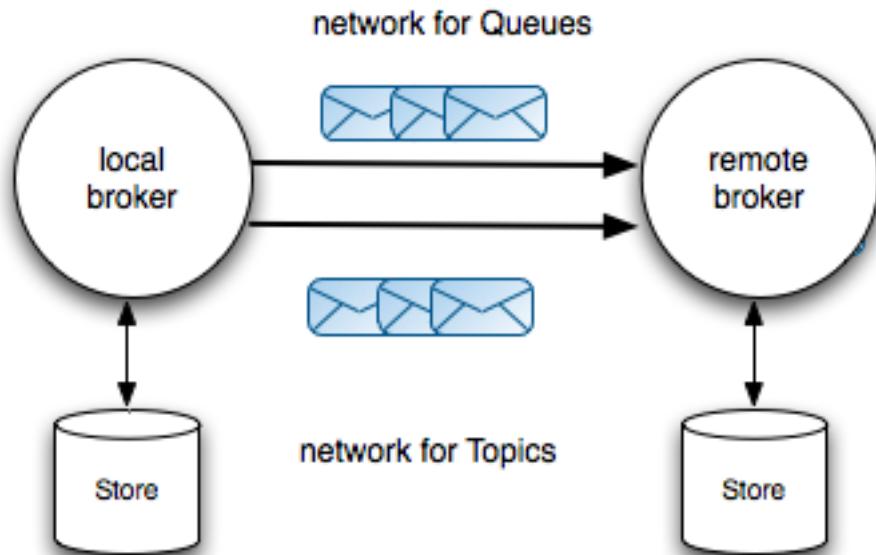


Figure 10.7. 'Shared Nothing' master/slave

The corresponding configuration would look like this:

Example 10.10. Setting options for included destinations

```

<networkConnectors>
    <networkConnector uri="static://(tcp://remotehost:61617)"
        name="queues_only"
        duplex="true"
        <excludedDestinations>
            <topic physicalName=""/>
        </excludedDestinations>
    </networkConnector>
    <networkConnector uri="static://(tcp://remotehost:61617)"
        name="topics_only"
        duplex="true"
        <excludedDestinations>
            <queue physicalName=""/>
        </excludedDestinations>
    </networkConnector>
</networkConnectors>
  
```

Having looked at how networks work and how to configure them - we can now use

this knowledge as one way to help scale our ActiveMQ applications - which we will be covering in the next section.

10.4. Scaling Applications

In this section we are going to look at scaling your ActiveMQ applications, and examine three techniques to allow you to do that. We will start with scaling connections and numbers of Queues on a single broker to achieve thousands of connections and Queues. Then we will look at scaling connections to tens of thousands of connections by using techniques for horizontally scaling your applications by using networks. Finally we will examine traffic partitioning, which will balance scaling and performance, but will add more complexity to your ActiveMQ application.

10.4.1. Vertical Scaling

Vertical scaling increases is a technique to increase the number of connections and load that a single ActiveMQ broker can handle. By default, the ActiveMQ broker is designed to move messages as efficiently as possible to ensure low latency and good performance. However, there are some configuration decisions that you can make to ensure that the ActiveMQ broker can handle both a large number of concurrent connections and a large number of Queues. We will examine each in turn.

By default, ActiveMQ will use blocking I/O to handle transport connections, which results in a thread being used per connection. You can use non-blocking I/O on the ActiveMQ broker (and still use the default transport on the client) to reduce the number of threads used. You can configure `nio` to be the transport connector for the broker in the ActiveMQ configuration file, an example of how to do this is shown below:

Example 10.11. Setting `nio` as the default ActiveMQ transport connector

```
<broker>
  <transportConnectors>
```

Please post comments or corrections to the [Author Online Forum](#)

```
<transportConnector name="nio" uri="nio://localhost:61616"/>
</transportConnectors>
</broker>
```

In addition to using a thread per connection for blocking I/O, the ActiveMQ broker can use a thread for dispatching messages per client connection. You can ensure uses a thread pool instead, by setting the system property, org.apache.activemq.UseDedicatedTaskRunner to false, or by setting the ACTIVEMQ_OPTS property in the start-up script in the bin directory - e.g:

Example 10.12. Configure the ActiveMQ Broker to use Thread Pooling

```
ACTIVEMQ_OPTS="-Dorg.apache.activemq.UseDedicatedTaskRunner=false"
```

Ensure your ActiveMQ broker has enough memory to handle lots of concurrent connections is a two step process. First you need to ensure that the JVM the broker is running in is configured for for enough memory - again you can use the ACTIVEMQ_OPTS property in the start-up script - e.g.

Example 10.13. Configure the ActiveMQ Broker JVM for more memory

```
ACTIVEMQ_OPTS="-Xmx1024M -Dorg.apache.activemq.UseDedicatedTaskRunner=false"
```

Secondly, ensure you configure enough memory for your ActIveMQ broker to use by setting the System Usage memory limit. 512mb should be the minimum for an ActiveMQ broker with more than a few hundred active connections. You can configure the memory limit in the activemq configuration file - like so:

Example 10.14. Setting memory limit for the ActiveMQ broker

```
<systemUsage>
  <systemUsage>
    <memoryUsage>
      <memoryUsage limit="512 mb"/>
    </memoryUsage>
  </systemUsage>
</systemUsage>
```

Please post comments or corrections to the [Author Online Forum](#)

```
</memoryUsage>
<storeUsage>
    <storeUsage limit="10 gb" name="foo" />
</storeUsage>
<tempUsage>
    <tempUsage limit="1 gb" />
</tempUsage>
</systemUsage>
</systemUsage>
```

Its also advisable to reduce the CPU load per connection. If you are using the OpenWire wire format, turn off tight encoding - which can be CPU intensive. You can turn off tight encoding on a client by client basis, parameters for OpenWire can be set as part of the URI used to connect to the ActiveMQ Broker. For example,

Example 10.15. Disabling tight encoding for OpenWire

```
String uri = "failover://(tcp://localhost:61616?wireFormat.tightEncodingEnabled=false)";
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory(url);
```

So we have looked at some tuning aspects for scaling an ActiveMQ broker to handle thousands of connections, so now we can look at tuning the broker to handle thousands of Queues.

The default Queue configuration uses a separate thread for paging messages from the store into the Queue to be dispatched to interested Message Consumers. For a large number of Queues its advisable to disable this, by enabling the optimizeDispatch property for all Queues - like in the configuration snippet below:

Example 10.16. Setting optimize Dispatch

```
<destinationPolicy>
    <policyMap>
        <policyEntries>
            <policyEntry queue=">" optimizedDispatch="true" />
        </policyEntries>
    </policyMap>
</destinationPolicy>
```

Note that we use the wildcard '>' character to denote all Queues.

In addition, the default message store for ActiveMQ is built for speed - but not designed for high scalability. The reason is that the index system it uses requires two file descriptors per destination. So ensure you can scale not only to thousands of connections, but also to tens of thousands of Queues, use either a JDBC Message Store or the new KahaDB Message Store.

So far we have looked at scaling connections, reducing thread usage and selecting the right Message Store to scale A example configuration for ActiveMQ, tuned for scaling, is shown below:

Example 10.17. Configuration for Scaling

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

    <persistenceAdapter>
        <kahaDB directory="${activemq.base}/data" journalMaxFileLength="32mb" />
    </persistenceAdapter>

    <destinationPolicy>
        <policyMap>
            <policyEntries>
                <policyEntry queue=">" optimizedDispatch="true" />
            </policyEntries>
        </policyMap>
    </destinationPolicy>

    <systemUsage>
        <systemUsage>
            <memoryUsage>
                <memoryUsage limit="512 mb" />
            </memoryUsage>
            <storeUsage>
                <storeUsage limit="10 gb" name="foo" />
            </storeUsage>
            <tempUsage>
                <tempUsage limit="1 gb" />
            </tempUsage>
        </systemUsage>
    </systemUsage>

    <!-- The transport connectors ActiveMQ will listen to -->
    <transportConnectors>
        <transportConnector name="openwire">
```

Please post comments or corrections to the [Author Online Forum](#)

```
    uri="nio://localhost:61616" />
</transportConnectors>

</broker>
```

Having looked at how to scale an ActiveMQ broker, we should look at using networks to increase horizontal scaling.

10.4.2. Horizontal Scaling

As well as scaling a single broker - you can use networks to increase the number of ActiveMQ brokers available for your applications. As networks automatically pass messages to connected brokers that have interested consumers, you can configure your clients to connect to a cluster of brokers, selecting one at random to connect to, for example:

Example 10.18. spreading the load with failover URI

```
failover://(tcp://broker1:61616,tcp://broker2:61616)?randomize=true
```

In order to make sure that messages for Queues or durable Topic subscribers are not "orphaned" on a broker, configure the networks to use **dynamicOnly** and a low network **prefetchSize**, e.g:

Example 10.19. Network configuration for horizontal scaling

```
<networkConnector uri="static://(tcp://remotehost:61617)"
  name="bridge"
  dynamicOnly="true"
  prefetchSize="1"
/>
```

Its possible to scale to a very large number of clients, by combining vertical scaling of a broker with application level splitting of destinations, or traffic partitioning,

which we cover in the next session.

10.4.3. Traffic Partitioning

Client side traffic partitioning is a hybrid of vertical and horizontal partitioning previously described. Networks are typically not used, as the client application decides what traffic should go to which broker(s). The client application has to maintain multiple JMS Connections, and decide which JMS Connection should be used for the destinations used.

The advantages of not directly using network connections is that you reduce the overhead of forwarding messages between brokers. You do need to balance that with the additional complexity that result in your application.

A representation of using traffic partitioning can be shown below in figure Figure 10.8

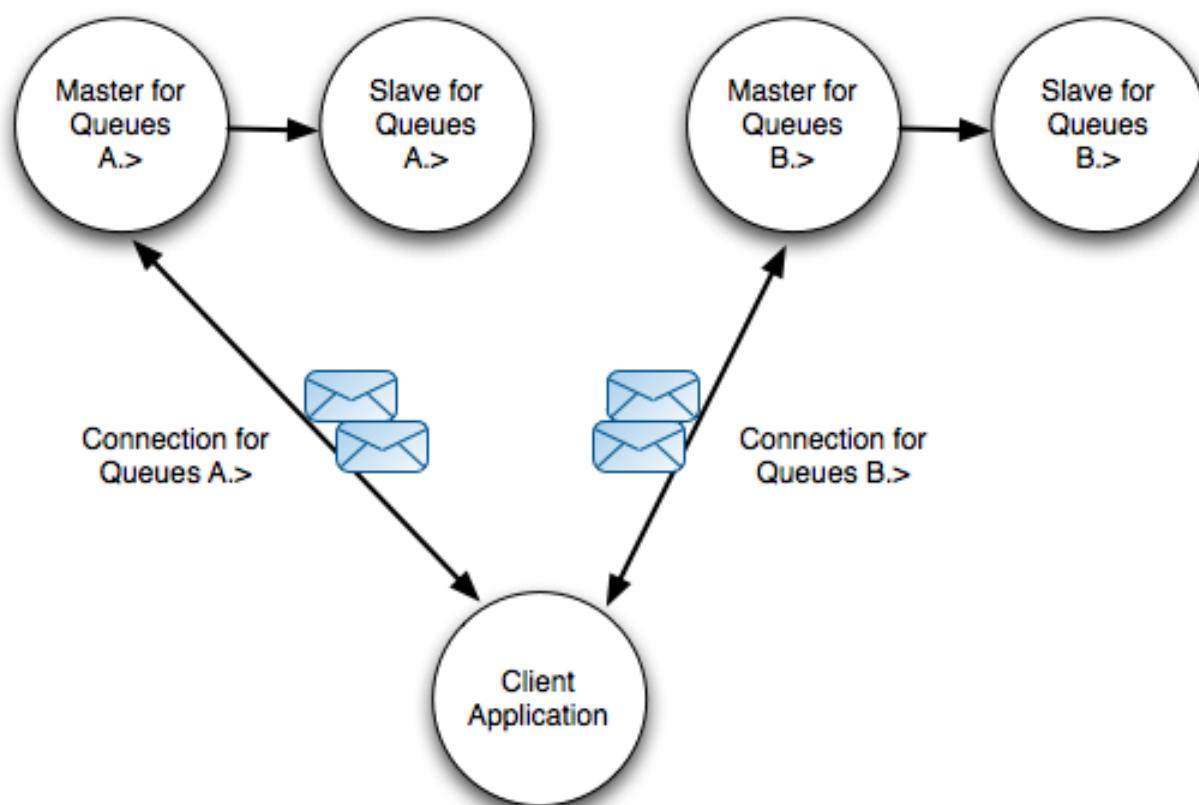


Figure 10.8. Combining High Availability and networks

We have covered vertical and horizontal scaling and traffic partitioning with ActiveMQ. You should now have a good understanding of how to use ActiveMQ to provide connectivity for thousands of concurrent connections and tens of thousands of destinations.

10.5. Summary

IN this chapter we have learned how to configure ActiveMQ brokers to provide high availability, utilising either a "shared nothing" or shared database or file system. We have covered options for failover for ActiveMQ clients and the performance trade-offs for having high availability configured for your applications.

We have also learned about how to use ActiveMQ for store and forward, to provide global distribution of messages across wide-area networks and how ActiveMQ can be configured to filter out or filter in destinations that are required to be global.

Finally we have shown how to configure ActiveMQ to scale for thousands of connections and tens of thousands of destinations!

Chapter 11. Advanced ActiveMQ Broker Features

11.1. Introduction

This chapter explains some advanced configurations options for the ActiveMQ Message Broker, including wildcards and composite destinations, advisory messages virtual topics and dead letter queues.

11.2. Wildcards and Composite Destinations

In this section we will be looking at two useful features of ActiveMQ, subscribing to multiple destinations using wildcards, and publishing to multiple destinations, using composite destinations.

11.2.1. Subscribing to Wildcard Destinations

ActiveMQ supports the concept of destination hierarchies - where the name of a destination can be used to organize messages into hierarchies, an element in the name is delimited by a DOT - ". ". Destination hierarchies applies to both Topics and Queues.

For example, if you had an application that subscribed to the latest results for sports on a Saturday afternoon, you could use the following naming convention for your Topics:

<Sport><League>.<Team> -

e.g. to subscribe to the latest result for a team called Leeds in an english football game, you would subscribe to the Topic: football.division1.leeds. Now Leeds play both soccer and Rugby, and for convenience, you would want to see all results for Leeds for both Soccer and Rugby for the same MessageConsumer. This is where

the usage of wildcards is useful.

There are three special characters that are reserved for destination names:

- . - A DOT - used to separate elements in the destination name
- * - used to match one element
- > - matches one or all trailing elements

So to subscribe to the latest scores that all Leeds teams are playing in, you can subscribe to the Topic named *.*.Leeds -e.g.

Example 11.1. Subscribing to wildcards

```
String brokerURI = ActiveMQConnectionFactory.DEFAULT_BROKER_URL;
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerURI);
Connection connection = connectionFactory.createConnection();
connection.start();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

Topic allLeeds = session.createTopic("*.*.Leeds");

MessageConsumer consumer = session.createConsumer(allLeeds);
Message result = consumer.receive();
```

If you wanted to find out the results of all the soccer games in Division 1, you would subscribe to: soccer.division1.* and if you wanted to find out the latest scores for all Rugby games, you could subscribe to rugby.>.

Wildcards and destination hierarchies are a useful tool for adding flexibility to your applications, by allowing for a Message Consumer to subscribe to more than destination at a time. As the ActiveMQ broker will scan any destination name for a match using wildcards, generally the shorter the destination name, the better the performance.

However, wildcards only work for Consumers, if you publish a Message to a Topic named rugby.>, the message will only be sent to the Topic named rugby.>, and not all Topics that start with the name rugby. There is a way for a Message Producer to send a message to multiple destinations, by using composite destinations, which

we will cover in the next section.

11.2.2. Sending a Message to Multiple Destinations

It can be very useful to send the same message to different destinations at once. For example, an application used by a store might want to send a message to an order queue -so that it can be processed securely in-order but also broadcast the order store monitoring systems. Usually you would have to do this by sending the message twice, and use two Message Producers, one for the Queue and one for the Topic. However, ActiveMQ supports a feature called composite destinations, which you can use to send the same message to multiple destinations at once.

A composite destination uses a comma separated name space in the destination name, so for example, if you created a Queue with the name `store.order.backoffice,store.order.warehouse` - then messages sent to that composite destination would be sent to the two Queues at once, one named `store.order.backoffice` and one named `store.order.warehouse`.

Composite destinations can support a mixture of Queues and Topics, you have to prepend the destination name with either `queue://` or `topic://`. So for our store application scenario, where you want to send an order message to an order queue, but also broadcast on a Topic, you would set up your message producer as follows:

Example 11.2. Sending to Composite Destinations

```
String brokerURI = ActiveMQConnectionFactory.DEFAULT_BROKER_URL;
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerURI);
Connection connection = connectionFactory.createConnection();
connection.start();

Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue ordersDestination = session.createQueue("store.orders, topic://store.orders");
MessageProducer producer = session.createProducer(ordersDestination);
Message order = session.createObjectMessage();
producer.send(order);
```

Wildcards and composite destinations are powerful tools for building less complicated and flexible applications with ActiveMQ.

Next we will look at the system messages that ActiveMQ broadcasts, and how you can subscribe to them.

11.3. Advisory Messages

Advisory messages are system messages, generated by the ActiveMQ broker to notify of changes to its system. Typically, an advisory message will be generated every time a new administered object (Connection, Destination, Consumer, Producer) joins or leaves the broker, but Advisory messages can be generated for to warn about the ActiveMQ broker reaching system limits too. Advisory messages are regular JMS messages, that are generated on system defined Topics, which enables ActiveMQ applications to get notified asynchronously over JMS in changes in the ActiveMQ brokers state. They can be a good alternative to using JMX to find out about the running state of an ActiveMQ broker.

ActiveMQ uses Advisory messages internally too, to notify Connections about the availability of temporary Destinations and Networks about the availability of Consumers, so you should take care if you want to disable them.

Every advisory message generated has a JMSType of "Advisory" and predefined JMS String properties, identifying the broker where the Advisory was generated:

- originBrokerId - the id of the broker that generated the Advisory
- originBrokerName - the name of the broker that generated the Advisory
- originBrokerURL - the first transport connector URL of the broker that generated the Advisory

Advisory messages for changes in state to the administered objects usually use ActiveMQ specific internal commands as the payload, but they do carry useful information - lets look at how to listen for Connections starting and stoping with ActiveMQ:

Example 11.3. Subscribing for Connection Advisories

Please post comments or corrections to the [Author Online Forum](#)

```

String brokerURI = ActiveMQConnectionFactory.DEFAULT_BROKER_URL;
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerURI);
Connection connection = connectionFactory.createConnection();
connection.start();

Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Topic connectionAdvisory = org.apache.activemq.advisory.AdvisorySupport.CONNECTION_ADVISORY;
MessageConsumer consumer = session.createConsumer(connectionAdvisory);

ActiveMQMessage message = (ActiveMQMessage) consumer.receive();

DataStructure data = (DataStructure) message.getDataStructure();
if (data.getDataStructureType() == ConnectionInfo.DATA_STRUCTURE_TYPE) {
    ConnectionInfo connectionInfo = (ConnectionInfo) data;
    System.out.println("Connection started: " + connectionInfo);
} else if (data.getDataStructureType() == RemoveInfo.DATA_STRUCTURE_TYPE) {
    RemoveInfo removeInfo = (RemoveInfo) data;
    System.out.println("Connection stopped: " + removeInfo.getObjectId());
} else {
    System.err.println("Unknown message " + data);
}

```

You can see from the above example, we just use a regular JMS constructs to start listening to Advisory Topics. Its worth noting the use of `AdvisorySupport` class, which contains the definition of all the Advisory Topic definitions. Things get a little harder when we start using ActiveMQ specific command objects - although a `ConnectionInfo` is sent when a Connection starts, a `RemoveInfo` is sent when a Connection stops. The `RemoveInfo` does carry the `connectionId` (set as the `RemoveInfo`'s `objectId`) - so its possible to correlate which connection has stopped.

Most advisory messages are specific to destinations, but the `AdvisorySupport` class does have some helper methods for determining which Topic it listen to. You can also use wild cards - so for example, if created an Advisory Topic for the Queue named ">", you would get information for all Queues.

Lets look at an example for listening for Consumers coming and going for a Queue named "test.Queue":

Example 11.4. Subscribing for Consumer Advisories

```

String brokerURI = ActiveMQConnectionFactory.DEFAULT_BROKER_URL;
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerURI);

```

Please post comments or corrections to the [Author Online Forum](#)

```
Connection connection = connectionFactory.createConnection();
connection.start();

//Lets first create a Consumer to listen too
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
//Lets first create a Consumer to listen too
Queue queue = session.createQueue("test.Queue");

MessageConsumer testConsumer = session.createConsumer(queue);

//so lets listen for the Consumer starting and stoping
Topic advisoryTopic = org.apache.activemq.advisory.AdvisorySupport.getConsumerAdvisoryTopic();
MessageConsumer consumer = session.createConsumer(advisoryTopic);
consumer.setMessageListener(new MessageListener(){

    public void onMessage(Message m) {
        try {
            System.out.println("Consumer Count = " + m.getStringProperty("consumerCount"));
            DataStructure data = (DataStructure) message.getDataStructure();
            if (data.getDataStructureType() == ConsumerInfo.DATA_STRUCTURE_TYPE) {
                ConsumerInfo consumerInfo = (ConsumerInfo) data;
                System.out.println("Consumer started: " + consumerInfo);
            } else if (data.getDataStructureType() == RemoveInfo.DATA_STRUCTURE_TYPE) {
                RemoveInfo removeInfo = (RemoveInfo) data;
                System.out.println("Consumer stopped: " + removeInfo.getObjectId());
            } else {
                System.err.println("Unknown message " + data);
            }
        } catch (JMSEException e) {
            e.printStackTrace();
        }
    }
});

testConsumer.close();
```

You will notice in the example above, that we create a test Consumer on the Queue "test.queue" before we create the listener for Consumer advisories on "test.queue". This is to demonstrate that the ActiveMQ broker will also send advisory messages for Consumers that all ready exist when you start to listen for them.

There are some advisories on destinations that are not enabled by default, these are advisories on message delivery, slow consumers, fast producer etc. To enable these advisories, you have to configure them on the a destination policy in the ActiveMQ broker configuration file. For example, to configure an advisory message for slow consumers on all Queues, you need to add the following to your configuration:

Example 11.5. Enabling Slow Consumer Advisories

```
<destinationPolicy>
    <policyMap><policyEntries>
        <policyEntry queue="" advisoryForSlowConsumers="true" />
    </policyEntries></policyMap>
</destinationPolicy>
```

You can use Advisory messages to supplement your application behavior (e.g. slow message production if you have slow consumers) or to supplement JMX monitoring of the ActiveMQ broker. We will now look at another useful advanced feature of ActiveMQ, Virtual Destinations.

11.4. Virtual Topics

If you want to broadcast a message to multiple consumers, then you use a Topic. If you want a pool of Consumers to consume from a Destination you would use a Queue. However, there isn't a satisfactory way to broadcast a message to a Topic and then have multiple load balanced Consumers consume from that Topic, as Queues do.

Virtual Topics are a convenient way of combining Queue semantics with a Topic, and have persistent messages consumed by a pool of consumers, each taking one message at time from the Topic. Below is a diagram - ???of how Virtual Topics are structured in the ActiveMQ broker below:

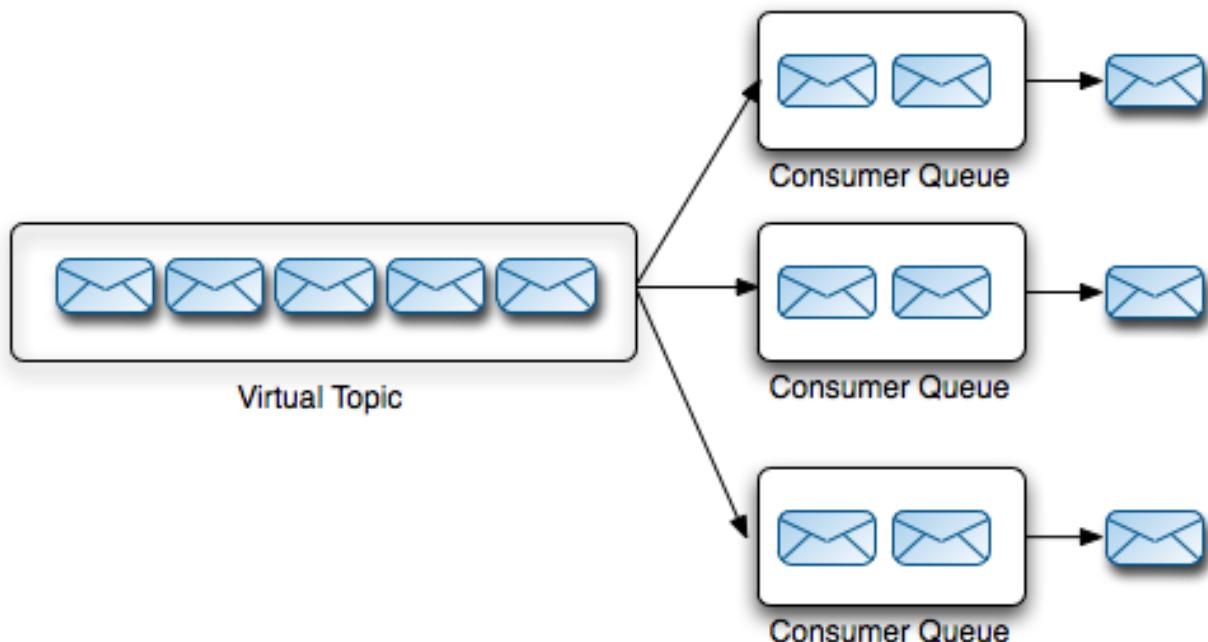


Figure 11.1. Virtual Topics

There are some naming conventions that need to be used for Virtual Topics. Firstly, to identify that a Topic is Virtual - it should always start with the element **VirtualTopic**. So if you want to create a VirtualTopic for orders, you need to create a destination with the name `VirtualTopic.orders`.

In order to setup and consume from the Queue component of a VirtualTopic - you need to call your Queue Consumer.`<consumer name>.VirtualTopic.<VirtualTopic Name>`. So suppose you want to consume messages on two separate Queues, A and B from our orders VirtualTopic, you would create a two Queue receivers - one consuming from a Queue called `Consumer.A.VirtualTopic.orders`, and one subscribing from a Queue called `Consumer.B.VirtualTopic.orders` - e.g.:

Example 11.6. Setting up VirtualTopics

```
String brokerURI = ActiveMQConnectionFactory.DEFAULT_BROKER_URL;  
  
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerURI);  
Connection consumerConnection = connectionFactory.createConnection();  
consumerConnection.start();
```

Please post comments or corrections to the [Author Online Forum](#)

```
Session consumerSessionA = consumerConnection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue consumerAQueue = consumerSessionA.createQueue("Consumer.A.VirtualTopic.orders");
MessageConsumer consumerA = consumerSessionA.createConsumer(consumerAQueue);

Session consumerSessionB = consumerConnection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue consumerBQueue = consumerSessionB.createQueue("Consumer.B.VirtualTopic.orders");
MessageConsumer consumerB = consumerSessionB.createConsumer(consumerAQueue);

//setup the sender
Connection senderConnection = connectionFactory.createConnection();
senderConnection.start();
Session senerSession = senderConnection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Topic ordersDestination = senerSession.createTopic("VirtualTopic.orders");
MessageProducer producer = senerSession.createProducer(ordersDestination);
```

VirtualTopics are a convenient mechanism to combine the load balancing and failover aspects of Queues, with the distribution mechanism of Topics.

In the next section we will look at using ActiveMQ to combine the longevity of durable subscribers, with the performance of normal Topic subscribers.

11.5. Retroactive Consumers

For applications that require messages to be sent and consumed as fast as possible, for example, a real-time data feed - its recommend that you send messages with persistence turned off.

However, there is a down side to consuming non-persistent messages, in that you will only be able to consume messages from the point that your Message Consumer starts. You can miss messages if your MessageConsumer might start behind the MessageProducer, or there is a network glitch and your MessageConsumer needs to reconnect to the broker (or another one in a network).

In order to provide a limited method of retroactive consumption of messages without requiring Message persistence, ActiveMQ has the ability to cache a configurable size or number of messages sent on a Topic. There are two parts to this- your Message Consumers need to inform the ActiveMQ broker that it is interested in retroactive messages, and there is the configuration of the destination

in the broker to say how many messages should be cached for consumption at a later point.

To mark a consumer as being retroactive, you need to set the retroactive flag for the Message Consumer - the easiest way to do that is to set the property on the Topic name you use - e.g.

Example 11.7. Setting A Retroactive Consumer

```
String brokerURI = ActiveMQConnectionFactory.DEFAULT_BROKER_URL;
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerURI);
Connection connection = connectionFactory.createConnection();connection.start();

Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Topic topic = session.createTopic("soccer.division1.leeds?consumer.retroactive=true");
MessageConsumer consumer = session.createConsumer(topic);
Message result = consumer.receive();
```

On the broker side, there are a number of recover policies that you can configure on a topic by topic basis. The default is called the `FixedSizedSubscriptionRecoveryPolicy`, which holds a number of messages in a Topic, based on the calculated size the messages will take from the broker memory. The default size being 64 kb.

You can configure the subscription recovery policy on a named Topic, or use wildcards to apply them to hierarchies. Below is an example configuration snippet of how to change the default cache size for the `FixedSizedSubscriptionRecoveryPolicy` for all Topics created in the ActiveMQ broker.

Example 11.8. Configuring a Subscription Recovery Policy

```
<destinationPolicy>
<policyMap>
    <policyEntries>
        <policyEntry topic="" >
            <subscriptionRecoveryPolicy >
                <fixedSizedSubscriptionRecoveryPolicy />
            </subscriptionRecoveryPolicy maximumSize = "8mb">
        </policyEntry>
```

Please post comments or corrections to the [Author Online Forum](#)

```
</policyEntries>  
</policyMap>  
</destinationPolicy>
```

Retroactive Message Consumers are a convenient mechanism to improve the reliability of your applications without incurring the overhead of message persistence. We have seen how enable Retroactive Consumers and how to configure their broker-side counterpart, the SubscriptionRecoveryPolicy.

In the next section we are going to look at how ActiveMQ stores messages that cannot be delivered to Message Consumers - dead letter Queues.

11.6. Message Redelivery and Dead-letter Queues

When messages expire on the ActiveMQ broker (they exceed their time to live, if set) - or the message cannot be redelivered, they are moved to a Dead Letter Queue, so they can be consumed or browsed by an administrator at a later point.

Messages are normally redelivered to a client for the following scenarios:

- A client is using transactions and calls rollback() on the Session.
- A client is using transactions, and closes before calling commit,
- A client is using CLIENT_ACKNOWLEDGE on a Session and calls recover() on that Session.

A client application usually has a good reason to rollback a transacted Session or call recover(), it may not be able to complete the processing of the message(s) because of its inability to negotiate with a third party resource for example. However, sometimes an application may decide to not accept delivery of a message because the message is poorly formatted. For such a scenario it does not make sense for the ActiveMQ broker to attempt re-delivery for ever.

There is a configurable POJO that is associated with the ActiveMQ Connection that you can tune to set different policies. You can configure the amount of time the ActiveMQ broker should wait before trying to resend the message, if that time should increase after every failed attempt (use an exponential back-off and back-off multiplier) and set the maximum number of redelivery attempts before the message(s) are moved to a Dead Letter Queue.

Here's an example of how to configure a clients redelivery policy:

Example 11.9. Configuring a Subscription Recovery Policy

```
RedeliveryPolicy policy = connection.getRedeliveryPolicy();
policy.setInitialRedeliveryDelay(500);
policy.setBackOffMultiplier(2);
policy.setUseExponentialBackOff(true);
policy.setMaximumRedeliveries(2);
```

By default, there is one Dead Letter Queue for all messages, called AcitveMQ.DLQ, that expired messages, or messages that have exceeded their redelivery attempts get sent to. You can configure configure a Dead Letter Queue for a hierarchy, or an individual Destination in the ActiveMQ broker Configuration like in the example below, where we set an IndividualDeadLetterStrategy.

Example 11.10. Configuring a Dead Letter Strategy

```
<destinationPolicy>
    <policyMap>
        <policyEntries>
            <policyEntry queue="">
                <deadLetterStrategy>
                    <individualDeadLetterStrategy queuePrefix="DLQ."
                        useQueueForQueueMessages="true"
                        processExpired="false"
                        processNonPersistent="false"/>
                </deadLetterStrategy>
            </policyEntry>

        </policyEntries>
    </policyMap>
</destinationPolicy>
```

Please post comments or corrections to the [Author Online Forum](#)

Notice that we configure this Dead Letter Strategy to ignore non-persistent and expired messages, which can prevent overwhelming the ActiveMQ broker with messages, if you are using time to live on non-persistent messages.

11.7. Summary

In this chapter you have learned how to use wildcard and composite destinations, to improve the flexibility of your ActiveMQ applications to receive and send messages with multiple destinations. You will now have an understanding of Advisory messages generated by the ActiveMQ broker.

We have also covered the benefits of using Virtual Topics and retroactive consumers and when to use them. Finally we have explained when Dead Letter Queues are used and how to configure them.

Chapter 12. Advanced Client Options

The power of ActiveMQ resides in its flexibility and its ability to be configured - and a lot of this configuration resides in client-side configuration. While the next chapter will look in depth at some of the options to trade off performance and reliability, this chapter will focus on some of the features of ActiveMQ that are configured by the client.

In this chapter we will cover advanced use of Queues (exclusive consumers and message groups), and how to handle large messages.

12.1. Exclusive Consumer

Consumption of messages from a Queue will be in-order at the point at which they are dispatched from the ActiveMQ Message Broker. However, if you cannot guarantee the order in which the messages will be consumed if you have more than one MessageConsumer - unless all the MessageConsumers reside in the same session and you are using a MessageListener for delivery. The reason for this is that you never have control over the scheduling of threads used to deliver the messages on the client - even if all your MessageConsumers are using the same connection.

The reason ordered message consumption can be guaranteed for a single session and MessageListeners - is that only one thread (the session thread will be used to dispatch messages.

For applications where message order is important, or you need to ensure that there will be only one MessageConsumer for a Queue, ActiveMQ offers a client side option to have only one active MessageConsumer process messages. The ActiveMQ Message Broker will select one consumer on the Queue to process messages. The benefit of allowing the Broker to make the choice, is that if the consumer stops or fails, then another MessageConsumer can be selected to be active - as depicted in Figure 12.1 below:

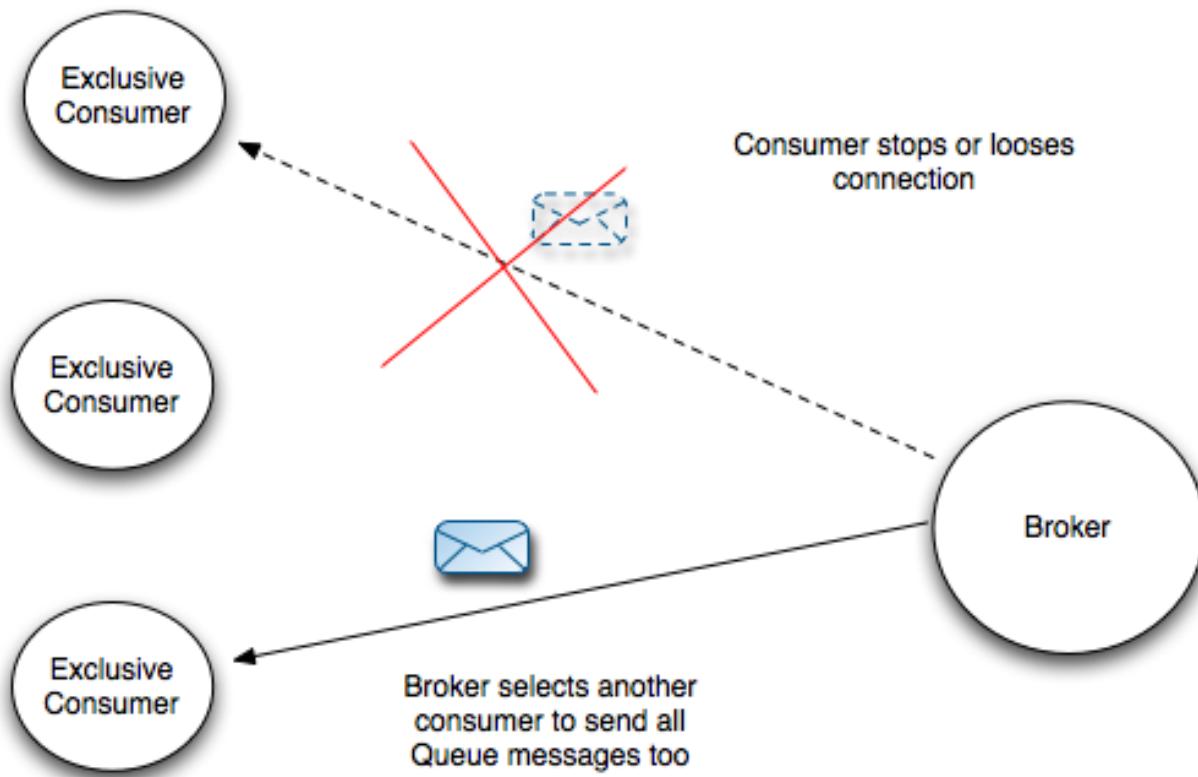


Figure 12.1. Exclusive Consumer

If you mix standard consumers and exclusive consumers on the same Queue, the ActiveMQ Message Broker will still only deliver messages to one of the exclusive consumers. If all the exclusive consumers become inactive, and there is still a standard MessageConsumer - then consumption of Queue messages will return to the normal mode of delivery - the messages will be delivered in a round robin fashion between all the remaining active standard MessageConsumers.

You can create an Exclusive Consumer using a destination option on the client, like in the code extract in ???below:

Example 12.1. Creating an Exclusive Consumer

```
queue = new ActiveMQQueue( "TEST.QUEUE?consumer.exclusive=true" );
consumer = session.createConsumer(queue);
```

Please post comments or corrections to the [Author Online Forum](#)

12.1.1. Exclusive Consumer Example

Often you use messaging to broadcast data from an external resource, be that changes to records in a database, cvs lines appended to a file or a raw real-time data feed. You might wish to build in redundancy, so if an instance of the application reading and broadcasting the changing data fails, another can take over. Often you can rely on locking a resource (row lock or file lock) to ensure that only one process will be accessing the data and broadcasting over a topic at a time. However, when you don't want the overhead of using a database, or want to run processes across more than one machine (and don't want to use NFS 4.x) - using exclusive consumers to get simulate a lock might seem appropriate. In Figure 12.2 below we outline the what we want to achieve for the implementation of an Exclusive Producer - using exclusive consumers to get a lock:

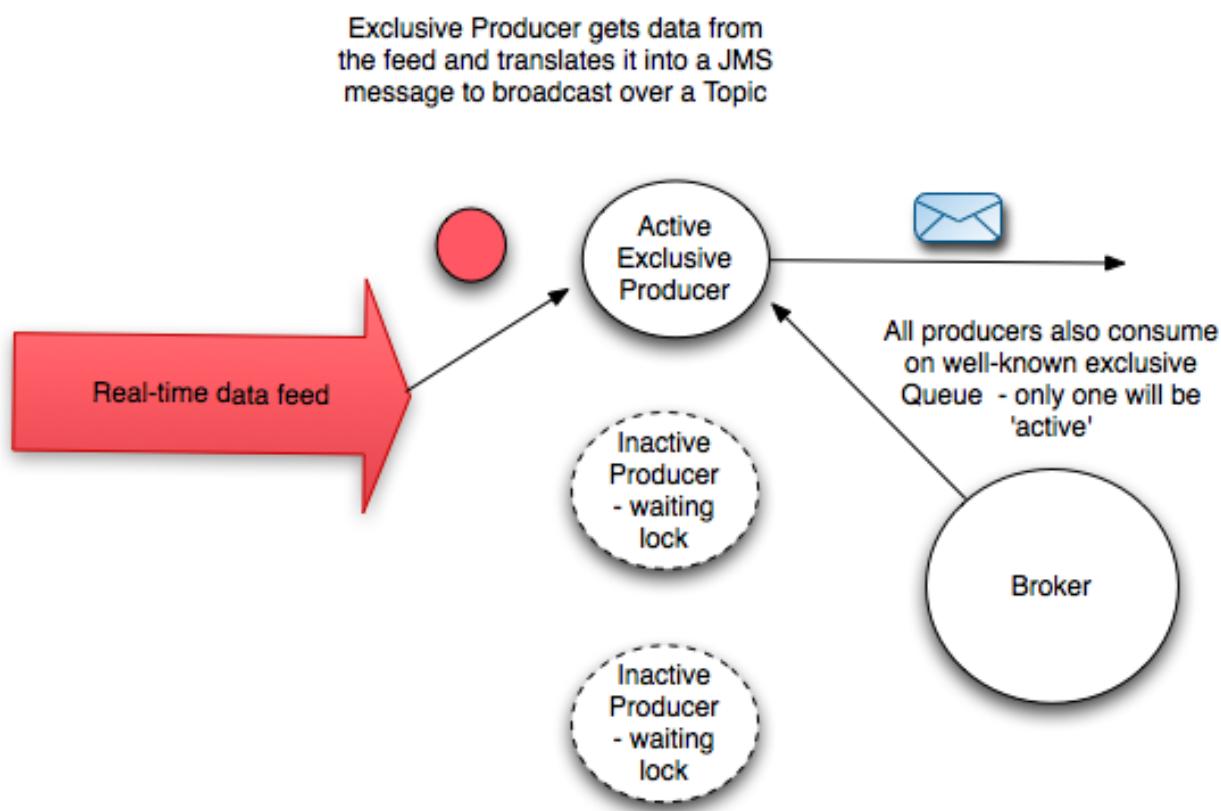


Figure 12.2. An Exclusive Producer scenario

To implement an Exclusive Producer - use a well known queue and create an exclusive consumer as in ??? below:

Example 12.2. Creating an Exclusive Consumer

```
public void start() throws JMSException {
    this.connection = this.factory.createConnection();
    this.connection.start();
    this.session = this.connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
    Destination destination = this.session.createQueue(this.queueName + "?consumer.exclu
    Message message = this.session.createMessage();
    MessageProducer producer = this.session.createProducer(destination);
    producer.send(message);
    MessageConsumer consumer = this.session.createConsumer(destination);
    consumer.setMessageListener(this);
}
```

In this example - we always send a message to the well-known queue - to start off consumption - this step could always be done externally by a management process. Note that we use Session.CLIENT_ACKNOWLEDGE mode to consume the message ? Although we want to be notified we are an exclusive consumer - and hence have the lock - we don't want to remove the message from the well-known queue. In this way, if we fail - another exclusive producer will be activated.

In the MessageListener for this shown in ??? - below - we consume the message - and start processing the real-time feed.

Example 12.3. MessageListener for Exclusive Producer

```
public void onMessage(Message message) {
    if (message != null && this.active==false) {
        this.active=true;
        startProducing();
    }
}
```

12.2. Message Groups

We can refine the Exclusive Consumer concept further. Rather than all messages going to a single message consumer, message consumption is defined by the Message header **JMSXGroupID** - with all messages of with the same **JMSXGroupID** being sent to the same consumer. This is a common concept in messaging and called message groups. The ActiveMQ broker will ensure that all messages of the same group are sent to the same consumer - as in Figure 12.3 below. If the consumer closes or is unavailable, messages of the same group will all be routed to a different consumer.

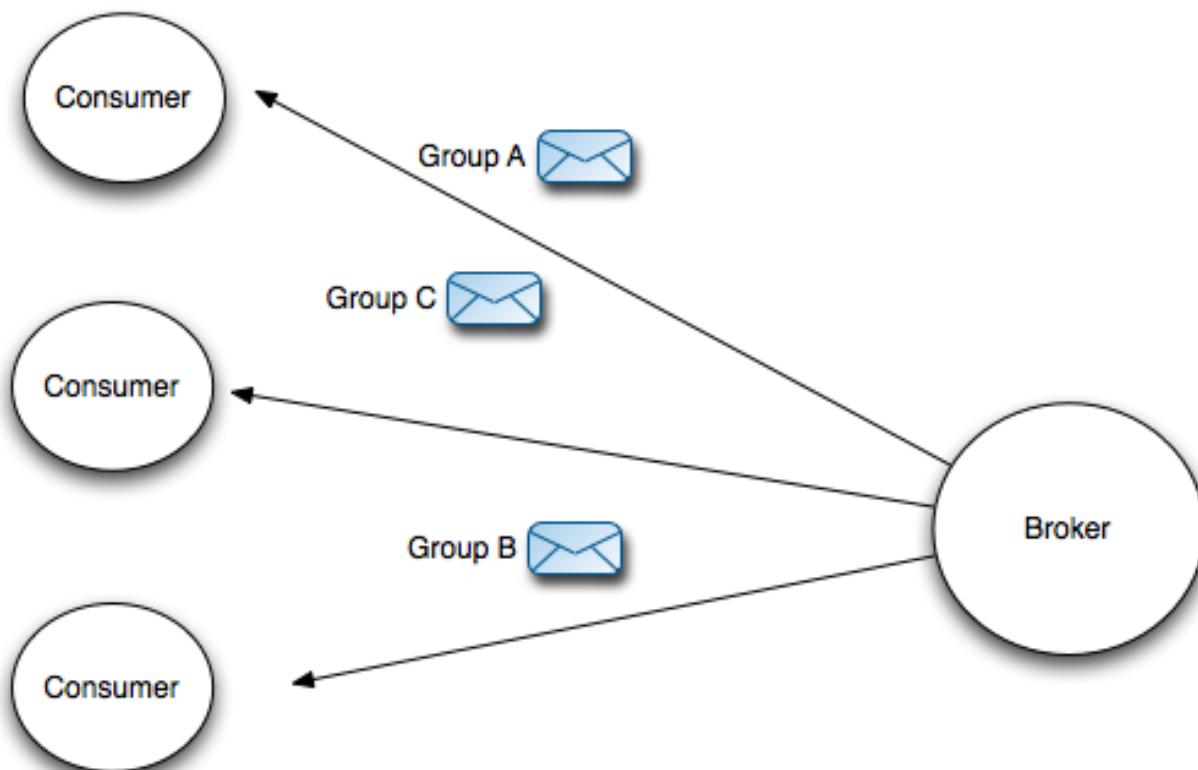


Figure 12.3. Message Groups

It's extremely straightforward to start using message groups - as ??? below shows. The definition of a group is left up to a user and is done on the Message Producer - it just has to be unique for a particular Queue. All the routing is done in the

ActiveMQ Message Broker - the message consumers involved are just normal Queue consumers.

Example 12.4. Create a Message Group

```
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue queue = session.createQueue("group.queue");
MessageProducer producer = session.createProducer(queue);
Message message = session.createTextMessage(
<foo>test</foo>
");
message.setStringProperty("JMSXGroupID", "TEST_GROUP_A");
producer.send(message);
```

ActiveMQ will add a sequence number to each message in a group , using the standard **JMSXGroupSeq** message header property.

You may wish to explicitly close a message group - you can by explicitly setting the **JMSXGroupSeq** to **-1**, just like in ???

Example 12.5. Close a Message Group

```
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue queue = session.createQueue("group.queue");
MessageProducer producer = session.createProducer(queue);
<foo></foo>

Message message = session.createTextMessage(
<foo>close</foo>
");
message.setStringProperty("JMSXGroupID", "TEST_GROUP_A");
message.setIntProperty("JMSXGroupSeq", -1);
producer.send(message);
```

You can simply recreate the particular group after it has been closed by sending a new message to the group. However, the group may be assigned to another consumer.

You cannot always assume that the JMSXGroupSeq will start at 1 for a message group. If an existing message group consumer closes or dies, any messages being

routed to its group will be assigned a new consumer. To help identify that a consumer is receiving messages to a new group, or a group that it hasn't seen before - a property, **JMSXGroupFirstForConsumer** is set for the first message sent to the new consumer. You can check to see if a message is being sent to your consumer for the first time by seeing if this property has been set - like in ???below:

Example 12.6. Check for start of new Group Messages

```
Session session = MessageConsumer consumer = session.createConsumer(queue);
Message message = consumer.receive();
String groupId = message.getStringProperty("JMSXGroupId");
if (message.getBooleanProperty("JMSXGroupFirstForConsumer")) {
    // do processing for new group
}
```

It is often the case that you start a number of message consumers to start processing messages at the same time. The ActiveMQ Message Broker will allocate all message groups evenly across all consumers - but if there are already messages waiting to be dispatched - the message groups will typically be allocated to the first consumer. To ensure there is an even distributed load, it is possible to give the Message Broker a hint, to wait for more message consumers to start. The configuration can be set on a destination policy - e.g.

Example 12.7. Wait for consumers before allocating Message Groups

```
<destinationPolicy>
<policyMap>
    <policyEntries>
        <policyEntry queue="" consumersBeforeDispatchStarts="2" timeBeforeDispatchStarts="5000">
    </policyEntries>
</policyMap>
</destinationPolicy>
```

The configuration in ??? will wait for for 2 consumers - or 5 seconds before selecting which consumer should be attached to which message group.

Having looked at Exclusive Consumers and Message Groups, we are going to look at how to transport large messages with ActiveMQ, using either JMS Streams or Blob Messages.

12.3. ActiveMQ Streams

ActiveMQ Streams are an advanced feature (so use with caution) that allows you to use a file or a socket stream to pass data over through ActiveMQ. This works well if you only use one Consumer on a Queue (or an exclusive Consumer) - else message order could become a problem.

The benefit of using JMS Streams is that ActiveMQ will break the stream into manageable chunks and re-assemble them for you at the consumer. So it's possible to transfer very large files of data using this functionality - as depicted in Figure 12.4.

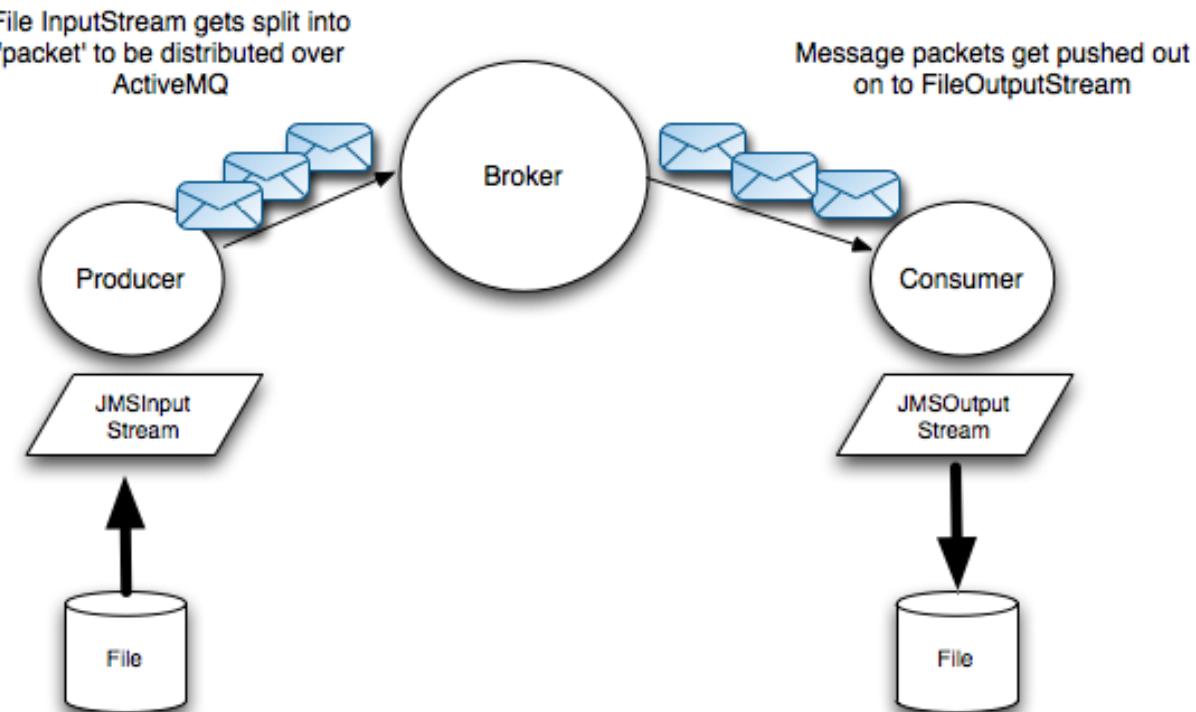


Figure 12.4. Transferring a File using Streams

To demonstrate using Streams, here is an example of reading a large file and writing it out over ActiveMQ:

Example 12.8. Sending data over an ActiveMQ Stream

```
//source of our large data
FileInputStream in = new FileInputStream("largetextfile.txt");

String brokerURI = ActiveMQConnectionFactory.DEFAULT_BROKER_URL;
ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerURI);
ActiveMQConnection connection = (ActiveMQConnection) connectionFactory.createConnection();
connection.start();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue destination = session.createQueue(QUEUE_NAME);

OutputStream out = connection.createOutputStream(destination);

//now write the file on to ActiveMQ
byte[] buffer = new byte[1024];
while(true){
    int bytesRead = in.read(buffer);
    if (bytesRead==-1){
        break;
    }
    out.write(buffer,0,bytesRead);
}
//close the stream so the receiving side knows the steam is finished
out.close();
```

Note that we close the stream - this is important so that the receiving side can determine the stream is finished. It is recommended that you use a new stream for each file you send. For completeness, below is the receiving end of an ActiveMQ Stream.

Example 12.9. Receiving data from an ActiveMQ Stream

```
//destination of our large data
FileOutputStream out = new FileOutputStream("copied.txt");

String brokerURI = ActiveMQConnectionFactory.DEFAULT_BROKER_URL;
ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerURI);
ActiveMQConnection connection = (ActiveMQConnection) connectionFactory.createConnection();
connection.start();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

Please post comments or corrections to the [Author Online Forum](#)

```
//we want to be an exclusive consumer
String exclusiveQueueName= QUEUE_NAME + "?consumer.exclusive=true";
Queue destination = session.createQueue(exclusiveQueueName);

InputStream in = connection.createInputStream(destination);

//now write the file from ActiveMQ
byte[ ] buffer = new byte[1024];
while(true){
    int bytesRead = in.read(buffer);
    if (bytesRead== -1){
        break;
    }
    out.write(buffer,0,bytesRead);
}
out.close();
```

We use an exclusive consumer, to ensure that only one consumer will be reading the stream at a time. You can use streams with Topics too - though if a Consumer for a Topic starts part way through the delivery of a Stream, it won't receive any data that was sent before it was started.

ActiveMQ breaks the stream into manageable chunks of data, and sends each chunk of data as a separate message, which means that you have to be careful when using them (like making sure you use an exclusive consumer for example).

There is an alternative and more robust method of sending large payloads, and that is to use Blob Messages.

12.4. Blob Messages

ActiveMQ introduced the concept of Blob Messages so that users can take advantage of ActiveMQ message delivery semantics (transactions, load balancing and smart routing) but use those in conjunction with very large messages. A Blob Message does not contain the data being sent, but is a notification that a Blob (Binary large Object) is available. The Blob itself is transferred out of bounds, by either ftp or http. In fact, all an ActiveMQ BlobMessage contains is the URL to the data itself, with a helper method to grab an InputStream to the real data. Lets work through an example.

Firstly creating and sending a BlobMessage. We'll assume that a file already exists on a shared web site, so we have to create a BlobMessage to notify any consumers that it exists:

Example 12.10. Sending a BlobMessage

```
String brokerURI = ActiveMQConnectionFactory.DEFAULT_BROKER_URL;
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerURI);
Connection connection = connectionFactory.createConnection();
connection.start();
ActiveMQSession session = (ActiveMQSession) connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue destination = session.createQueue(QUEUE_NAME);
MessageProducer producer = session.createProducer(destination);
BlobMessage message = session.createBlobMessage(new URL("http://some.shared.site.com"));
producer.send(message);
```

When we receive a BlobMessage, we have been notified that there is a new URI to process (on the shared web site).

Example 12.11. Processing a Blob message

```
// destination of our Blob data
FileOutputStream out = new FileOutputStream("blob.txt");

String brokerURI = ActiveMQConnectionFactory.DEFAULT_BROKER_URL;
ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(brokerURI);
Connection connection = (ActiveMQConnection) connectionFactory.createConnection();
connection.start();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue destination = session.createQueue(QUEUE_NAME);

MessageConsumer consumer = session.createConsumer(destination);
BlobMessage blobMessage = (BlobMessage) consumer.receive();

InputStream in = blobMessage.getInputStream();
// now write the file from ActiveMQ
byte[] buffer = new byte[1024];
while (true) {
    int bytesRead = in.read(buffer);
    if (bytesRead == -1) {
        break;
    }
    out.write(buffer, 0, bytesRead);
}
out.close();
```

Please post comments or corrections to the [Author Online Forum](#)

Using BlobMessages is certainly more robust than StreamMessages - but they do rely on using external servers for delivery of the actual data.

12.5. Summary

In this chapter we have learned about some of the advanced features that an ActiveMQ client can use above and beyond the JMS specification.

We have learned about Exclusive Consumers, and walked through an example of using them to ensure (paradoxically) that only one producer will be running for a distributed application. We have seen the power of using Message Groups and examined some of ways that ActiveMQ can be used to transport very large messages across applications.

Chapter 13. Tuning ActiveMQ For Performance

The performance of ActiveMQ is highly dependent on a number of different factors - including the network topology, the transport used, the quality of service and speed of the underlying network, hardware, operating system and the Java Virtual Machine.

However, there are some performance techniques you can apply to ActiveMQ to improve performance regardless of its environment. Your application may not need guaranteed delivery, in which case reliable, non-persistent messaging would yield much better performance for your application. It may make sense to use embedded brokers - reducing the paths of serialization that your messages needs to pass through - and finally there are a multitude of tuning parameters that can be applied, each of which have benefits and caveats. In this chapter we will walk through all the standard architectural tweaks, tuning tricks and more so that you have the best information to tune your application to meet your goals for performance.

Before we get to the complex tuning tweaks, we'll walk through some general, but simple messaging techniques - using non-persistent message delivery and batching messages together. Either one of these can really reap large performance benefits - definitely the first thing to consider if performance is going to be critical for you.

13.1. General Techniques

There are two simple things you can do to improve JMS messaging performance: use non-persistent messaging or if you really need guaranteed messaging - use transactions to batch up large groups of messages. Usually non-persistent message delivery gets discounted for all applications except where you don't care that a message will be lost (e.g. real-time data feeds - as the status will be sent repeatedly) and batching messages in transactions won't always be applicable. ActiveMQ however, incorporates fail-safes for reliable delivery of non-persistent messages - so only catastrophic failure would result in message loss. In this section

Please post comments or corrections to the [Author Online Forum](#)

we will explain why non-persistent message delivery and batching messages are faster, and why they could be applicable to use in your application - if you don't need to absolutely guarantee that messages will never, ever be lost.

13.1.1. Persistent vs Non-Persistent Messages

The JMS specification allows for two message delivery options, persistent and non-persistent delivery. When you send a message that is persistent (this is the default JMS delivery option), the message broker will always persist it to its message database to either mitigate against catastrophic failure or to deliver to consumers who might not yet be active. If you are using non-persistent delivery, then the JMS specification allows the messaging provider to make best efforts to deliver the message to currently active message consumers. ActiveMQ provides additional reliability around this - which we cover later in this section.

Non-persistent messages are significantly faster than sending messages persistent messages - there are two reasons for this:

- Messages are sent asynchronously from the message producer - so the producer doesn't have to wait for a receipt from the broker - Figure 13.1
- Persisting messages to the message store (which typically involves writing to disk) is slow compared to messaging over a network

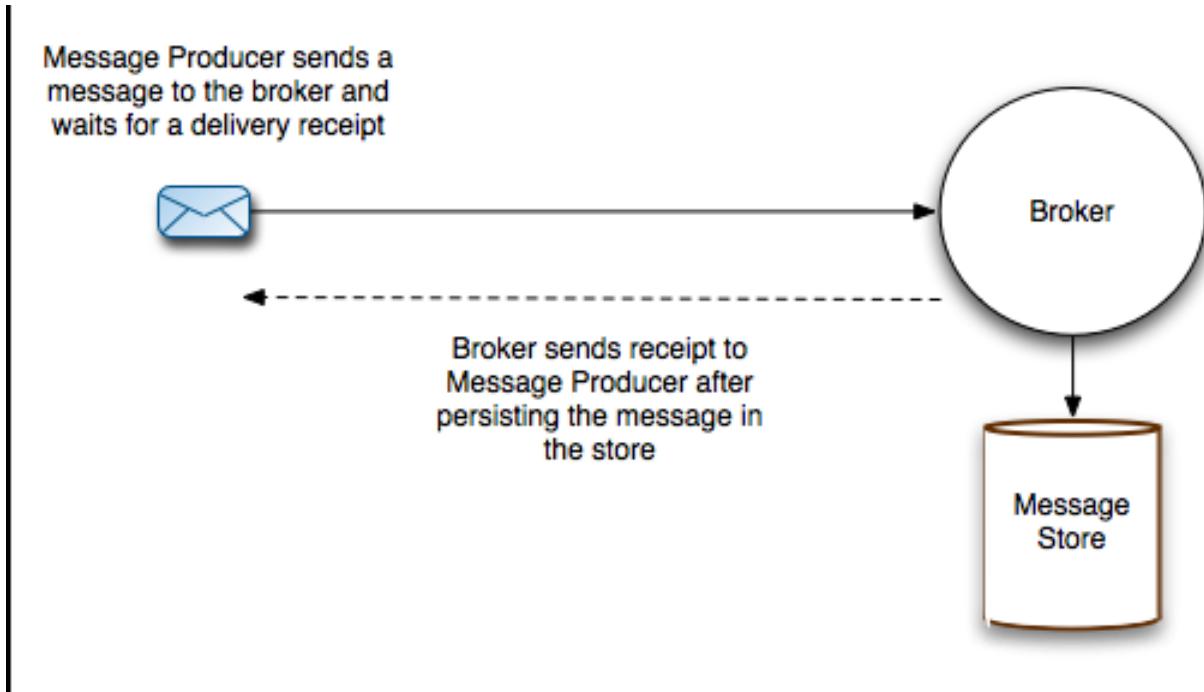


Figure 13.1. Persistent Message delivery

The main reason for using persistence is to negate message loss in the case of a system outage. However, ActiveMQ incorporates reliability to prevent this. By default, the fault tolerant transport caches asynchronous messages to resend again on a transport failure - and to stop duplicates - both a broker and a client use message auditing, to filter out duplicate messages. So for usage scenarios where only reliability is required, as opposed to guaranteed message delivery, using a non-persistent delivery mode will meet your needs.

As by default the message delivery mode is persistent, you have to explicitly set the delivery mode on the MessageProducer to send non-persistent messages - as can be seen in ???:

Example 13.1. Setting the delivery mode

```
MessageProducer producer = session.createProducer(topic);  
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
```

Please post comments or corrections to the [Author Online Forum](#)

We have seen the reasons why there is such a big performance difference between persistent and non-persistent delivery of messages - and the steps that ActiveMQ takes to improve reliability of non-persistent messages. The benefit of reliable message delivery allows non-persistent messages to be used in many more cases than would be typical of a JMS provider.

Having covered non-persistent messages, we will explain the second generalized technique for improving performance of delivering messages in your application - by batching messages together. The easiest way to batch messages is to use transaction boundaries - which we will explain below.

13.1.2. Transactions

When you send messages using a transaction - only the transaction boundary (the commit() call on the Session) results in synchronous communication with the message broker. So its possible to batch up the producing and or consuming of messages to improve performance of sending persistent messages - why not try the example below in ??? :

Example 13.2. Transacted and non-transacted example

```
public void sendTransacted() throws JMSEException {  
  
    //create a default connection - we'll assume a broker is running  
    //with its default configuration  
    ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();  
    Connection connection = cf.createConnection();  
    connection.start();  
  
    //create a transacted session  
  
    Session session = connection.createSession(true, Session.SESSION_TRANSACTED);  
    Topic topic = session.createTopic("Test.Transactions");  
    MessageProducer producer = session.createProducer(topic);  
    int count = 0;  
    for (int i = 0; i < 1000; i++) {  
        Message message = session.createTextMessage("message " + i);  
        producer.send(message);  
  
        //commit every 10 messages  
  
        if (i != 0 && i % 10 == 0) {  
            session.commit();  
        }  
    }  
}
```

Please post comments or corrections to the [Author Online Forum](#)

```
        session.commit();
    }
}

public void sendNonTransacted() throws JMSEException {

    //create a default connection - we'll assume a broker is running
    //with its default configuration

    ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
    Connection connection = cf.createConnection();
    connection.start();

    //create a default session (no transactions)

    Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Topic topic = session.createTopic("Test.Transactions");
    MessageProducer producer = session.createProducer(topic);
    int count = 0;
    for (int i = 0; i < 1000; i++) {
        Message message = session.createTextMessage("message " + i);
        producer.send(message);
    }
}
```

So we've covered some of the easy pickings in terms of performance, use non-persistent messaging where you can and now use transaction boundaries for persistent messages if it makes sense for your application. We are now going to start (slowly!) delving in to some ActiveMQ specifics covered under general techniques which can aid performance. The first of which is to use an embedded broker. Embedded brokers cut down on the amount of serialization and network traffic that ActiveMQ uses as messages can be passed around in the same JVM.

13.1.3. Embedding Brokers

It is often a requirement to co-locate applications with a broker, so that any service that is dependent on a message broker will only be available at the same time the message broker - see Figure 13.2. Its really straight forward to create an embedded broker, but one of the advantages of using the `vm://` transport is that message delivered through a broker do not incur the cost of being serialized on the wire to

be transported across the network, making it ideal for applications that have service lots of responses very quickly.

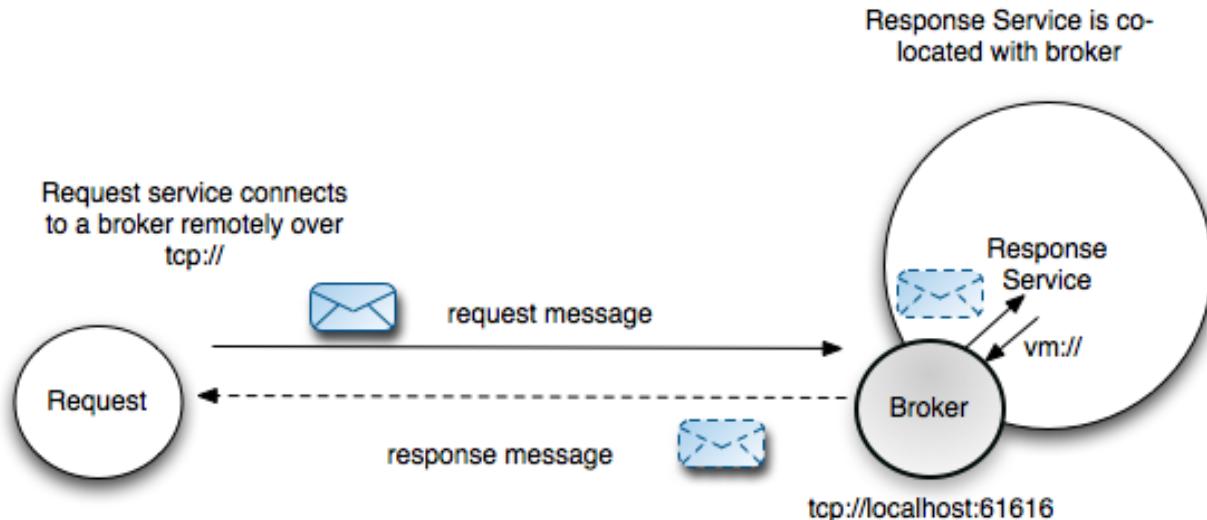


Figure 13.2. Co-locate with a Broker

You can create an embedded broker with a transport connector to listen to `tcp://` connections - but still connect to it using the `vm://` transport. By default, a broker always listens for transport connections on `vm://<broker name>`. Below in ??? is an example of setting up a service using an embedded broker to listen for requests on a Queue named `service.queue`

Example 13.3. Creating a Queue Service

```
//By default a broker always listens on vm://<broker name>
//so we don't need to set up an explicit connector for
//vm:// connections - just the tcp connector

BrokerService broker = new BrokerService();
broker.setBrokerName("service");
broker.setPersistent(false);
broker.addConnector("tcp://localhost:61616");
broker.start();

ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory("vm://service");
cf.setCopyMessageOnSend(false);
Connection connection = cf.createConnection();
connection.start();
```

Please post comments or corrections to the [Author Online Forum](#)

```
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

//we will need to respond to multiple destinations - so use null
//as the destination this producer is bound to

final MessageProducer producer = session.createProducer(null);

//create a Consumer to listen for requests to service

Queue queue = session.createQueue("service.queue");
MessageConsumer consumer = session.createConsumer(queue);
consumer.setMessageListener(new MessageListener() {
    public void onMessage(Message msg) {
        try {
            TextMessage textMsg = (TextMessage)msg;
            String payload = "REPLY: " + textMsg.getText();
            Destination replyTo;
            replyTo = msg.getJMSReplyTo();
            textMsg.clearBody();
            textMsg.setText(payload);
            producer.send(replyTo, textMsg);
        } catch (JMSEException e) {
            e.printStackTrace();
        }
    }
});
```

You can test out the above service, with a QueueRequestor that connects to the service's embedded broker by its tcp:// transport connector - as shown in ??? below:

Example 13.4. Connecting a QueueRequestor

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory("tcp://localhost:61616");
QueueConnection connection = cf.createQueueConnection();
connection.start();
QueueSession session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
Queue queue = session.createQueue("service.queue");
QueueRequestor requestor = new QueueRequestor(session,queue);
for(int i =0; i < 10; i++) {
    TextMessage msg = session.createTextMessage("test msg: " + i);
    TextMessage result = (TextMessage)requestor.request(msg);
    System.err.println("Result = " + result.getText());
}
```

As an aside, ActiveMQ by default will always copy the real message sent by a message producer to insulate the producer to changes to the message as it passes through the broker and is consumed by the consumer, all in the same Java virtual machine. If you intend to never re-use the sent message, you can reduce the overhead of this copy by setting the `copyMessageOnSend` property on the ActiveMQ ConnectionFactory to false - as seen below in ???:

Example 13.5. Reducing copying of messages

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
cf.setCopyMessageOnSend(false);
```

We have looked at some relatively easy to implement techniques to improve messaging performance; and in this section using an embedded broker co-located with an application is a relatively trivial change to make. The performance gains and atomicity of the service co-located with its broker can be an attractive architectural change to make too. Having gone through some of the easier 'quick wins' we are going to start moving into some harder configuration areas. So the next section is going to touch on the OpenWire protocol and list some of the parameters that you can tune to boost the performance of your messaging applications. These are very dependent on both hardware and the type of network you use.

13.1.4. Tuning the OpenWire protocol

Its worth covering some of the options available on the open wire protocol used by ActiveMQ Java and C++ clients. The OpenWire protocol is the binary format used for transporting commands over a transport (e.g. tcp) to the broker. Commands include messages and message acknowledgements, as well as management and control. Below in Table 13.1, “”. are some OpenWire wire format parameters that are relevant to performance

Example 13.6. OpenWire Tuning Parameters

Please post comments or corrections to the [Author Online Forum](#)

Table 13.1.

parameter name	default value	description
tcpNoDelayEnabled	false	provides a hint to the peer transport to enable/disable tcpNoDelay. If this is set, it may improve performance where you are sending lots of small messages across a relatively slow network.
cacheEnabled	true	commonly repeated values (like producerId and destination) are cached - enabling short keys to be passed instead. This decreases message size - which can make a positive impact on performance where network performance is relatively poor. The cache lookup involved does add an overhead to cpu load on both the clients and the broker machines - so take this into account.
cacheSize	1024	maximum number of items kept in the cache - shouldn't be bigger than Short.MAX_VALUE/2. The larger the cache, the better the performance

Please post comments or corrections to the [Author Online Forum](#)

parameter name	default value	description
		where caching is enabled. However, one cache will be used with every transport connection - so bear in mind the memory overhead on the broker - especially if its loaded with a large number of clients.
tightEncodingEnabled	true	cpu intensive way to compact messages. We would recommend that you turn this off if the broker starts to consume all the available cpu :)

You can add these parameters to the URI used to connect to the broker in the following way - ??? demonstrates disabling tight encoding - using the tightEncodingEnabled parameter:

Example 13.7. Setting OpenWire options

```
String uri = "failover://(tcp://localhost:61616?wireFormat.cacheEnabled=false&wireFormat.tig
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory(url);
cf.setAlwaysSyncSend(true);
```

These parameters are very dependent on the type of application, type of machine(s) used to run the clients and the broker and type of network used. Unfortunately is is not an exact science, so some experimentation is recommended. As we have lightly introduced some of the tuning parameters available on the OpenWire protocol, in the next section we will be looking at some of the tuning parameters available on the TCP Transport protocol.

Please post comments or corrections to the [Author Online Forum](#)

13.1.5. Tuning the TCP Transport

The most commonly used transport for ActiveMQ is the TCP transport - and there are two parameters that directly affect performance for this transport:

- `socketBufferSize` - the size of the buffers used to send and receive data over the TCP transport. Usually the bigger the better (though this is very operating system dependent - so worth testing!). The default value is 65536 - which is the size in bytes.
- `tcpNoDelay` - the default is false. Normally a TCP socket buffers up small sizes of data before being sent. By enabling this option - messages will be sent as soon as possible. Again, its worth testing this out - as it can be operating system dependent if this boosts performance or not.

Below is in ??? is an example transport URI where `tcpNoDelay` is enabled:

Example 13.8. Setting the TCP no delay option

```
String url = "failover://(tcp://localhost:61616?tcpNoDelay=true)";  
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory(url);  
cf.setAlwaysSyncSend(true);
```

We have covered some general techniques to improve performance at the application level, and looked at tuning the wire protocol and the TCP Transport. In the next two parts of this chapter we will look at tuning message producers and then message consumers. ActiveMQ is very flexible in its configuration and its producers can be configured to optimize their message exchanges with the broker which can boost throughput considerably.

13.2. Optimizing Message Producers

The rate that producers send messages to an ActiveMQ message broker before they are dispatched to consumers is a fundamental element of overall application

performance. We will cover some tuning parameters that affect the throughput and latency of messages sent from a message producer to an ActiveMQ broker.

13.2.1. Asynchronous send

We have already covered the performance gains that can be if you use non-persistent delivery for persistent messages. In ActiveMQ non-persistent delivery is reliable, in that delivery of messages will survive network outages and system crashes (as long as the producer is active, as it holds messages for re-delivery in its failover transport cache). However, you can also get the same reliability for persistent messages - by setting the `useAsyncSend` property on the message producer's `ConnectionFactory` - eg:

Example 13.9. Setting the delivery mode

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
cf.setUseAsyncSend(true);
```

This will set a property that is used by the `MessageProducer` to not expect a receipt for messages it sends to the ActiveMQ broker. Setting this property allows a user to benefit from the improvement in performance, allows for messages to be delivered at a later point to consumer(s), even if they aren't active whilst still being reliable.

If your application requires guaranteed delivery, it is recommended that you use the defaults with persistent delivery, and preferably use transactions too.

Why using asynchronous message delivery as an option for gaining performance should be well understood, and setting a property on the ActiveMQ `ConnectionFactory` is a really straight forward way of achieving that. What we are going to cover next is one of the most frequently run into gotchas with ActiveMQ - producer flow control. We see a lot of questions about producers slowing down - or pausing - and understanding flow control will allow you to negate that happening to your applications.

13.2.2. Producer Flow Control

Producer Flow Control allows the message broker to slow the rate of messages that are passed through it when resources are running low. This typically happens when consumers are slower than the producers - and messages are using memory in the broker awaiting dispatch.

A producer will wait until it receives a notification from the broker that it has space for more messages - as outlined in Figure 13.3

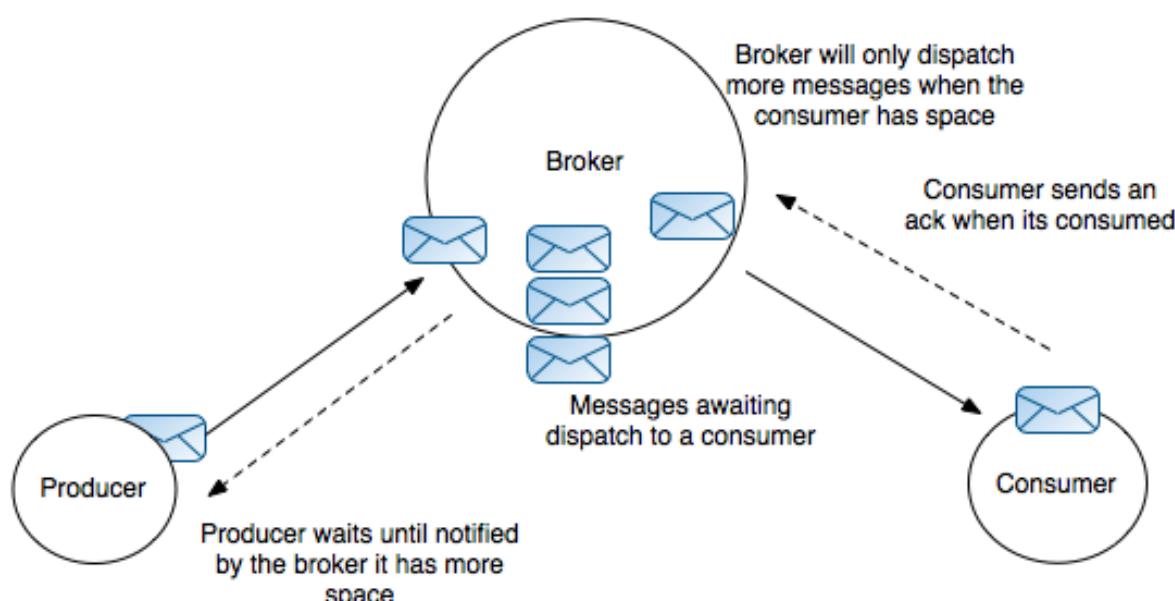


Figure 13.3. Producer Flow Control enabled

Producer flow control is a necessity to prevent a broker's limits for memory, temporary disk or store space being overrun, especially for wide area networks.

Producer flow control works automatically for persistent messages but has to be enabled for asynchronous publishing (persistent messages, or for connections configured to always send asynchronously). You can enable flow control for asynchronous publishing by setting the `producerWindowSize` property on the connection factory - as in ???:

Example 13.10. Setting the producer window size

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
cf.setProducerWindowSize(1024000);
```

The producerWindowSize is the number bytes allowed in the producers send buffer before it will be forced to wait for a receipt from the broker that it is still within its limits.

If this isn't enabled for an asynchronous publisher, the broker will still pause message flow, defaulting to simply blocking the message producers transport (which is inefficient and prone to deadlocks).

Although protecting the broker from typically running low on memory is a noble aim, it doesn't aid our cause for performance when everything slows down to the slowest consumer! So lets see what happens if you disable producer flow control, and you can do that in the Broker configuration on a destination policy like below in ???:

Example 13.11. How to disable flow control

```
<destinationPolicy>
  <policyMap>
    <policyEntries>

      <policyEntry topic="FOO.>" producerFlowControl="false" memoryLimit="10mb">
        <dispatchPolicy>
          <strictOrderDispatchPolicy/>
        </dispatchPolicy>
        <subscriptionRecoveryPolicy>
          <lastImageSubscriptionRecoveryPolicy/>
        </subscriptionRecoveryPolicy>
      </policyEntry>

    </policyEntries>
  </policyMap>
</destinationPolicy>
```

With producer flow control disabled, messages for slow consumers will be off-lined to temporary storage by default, enabling the producers and the rest of the consumers to run at a much faster rate - as outlined in ???

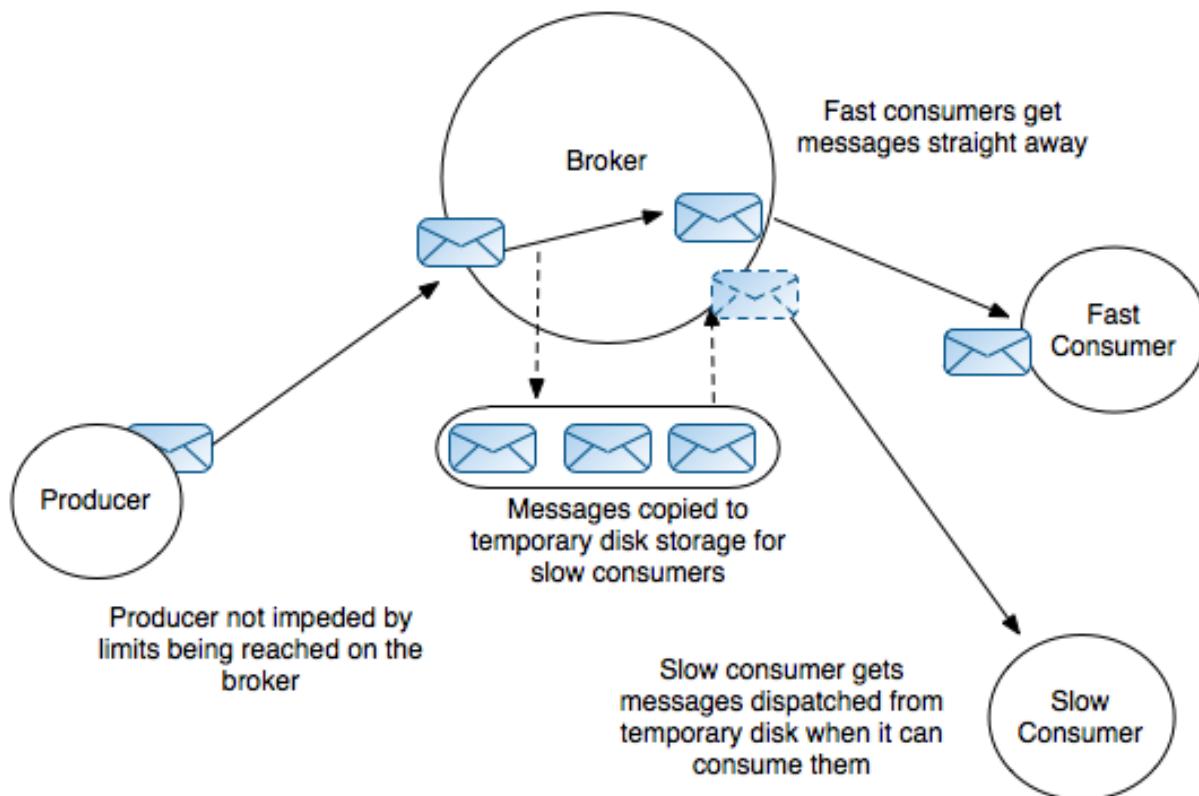


Figure 13.4. Producer Flow control disabled

Disabling flow control enables messaging applications to run at a pace independent of the slowest consumer, though there is a slight performance hit in off-lining messages. In an ideal world, consumers would always be running as fast as the fastest producer, which neatly brings us to the next section - optimizing Message Consumers.

13.3. Optimizing Message Consumers

In order to maximize application performance you have to look at all the

participants - and as we have seen so far, Consumers play a very big part in the overall performance of ActiveMQ. Message Consumers typically have to work twice as hard as a message Producer, because as well as consuming messages, they have to acknowledge the message has been consumed to the broker. We will explain some of the biggest performance gains you can get with ActiveMQ, by tuning your Consumers.

Typically the ActiveMQ broker will delivery messages as fast as possible to consumer connections. Messages once they are delivered over the transport from the ActiveMQ broker, are typically queued in the Session associated with the consumer where they wait to be delivered. In the next section we will explain why and how the rate that messages are pushed to consumers is controlled and how to tune that rate for better throughput.

13.3.1. Prefetch Limit

ActiveMQ uses a push-based model for delivery - delivering messages to Consumers when they are received by the ActiveMQ broker. To ensure that a Consumer won't exhaust its memory, there is a limit (prefetch limit) to how many messages will be delivered to a Consumer before the broker waits for an acknowledgement that the messages have been consumed by the application. Internally in the Consumer, messages are taken off the transport when they are delivered, and placed into an internal queue associated with the Consumer's session - Figure 13.5.

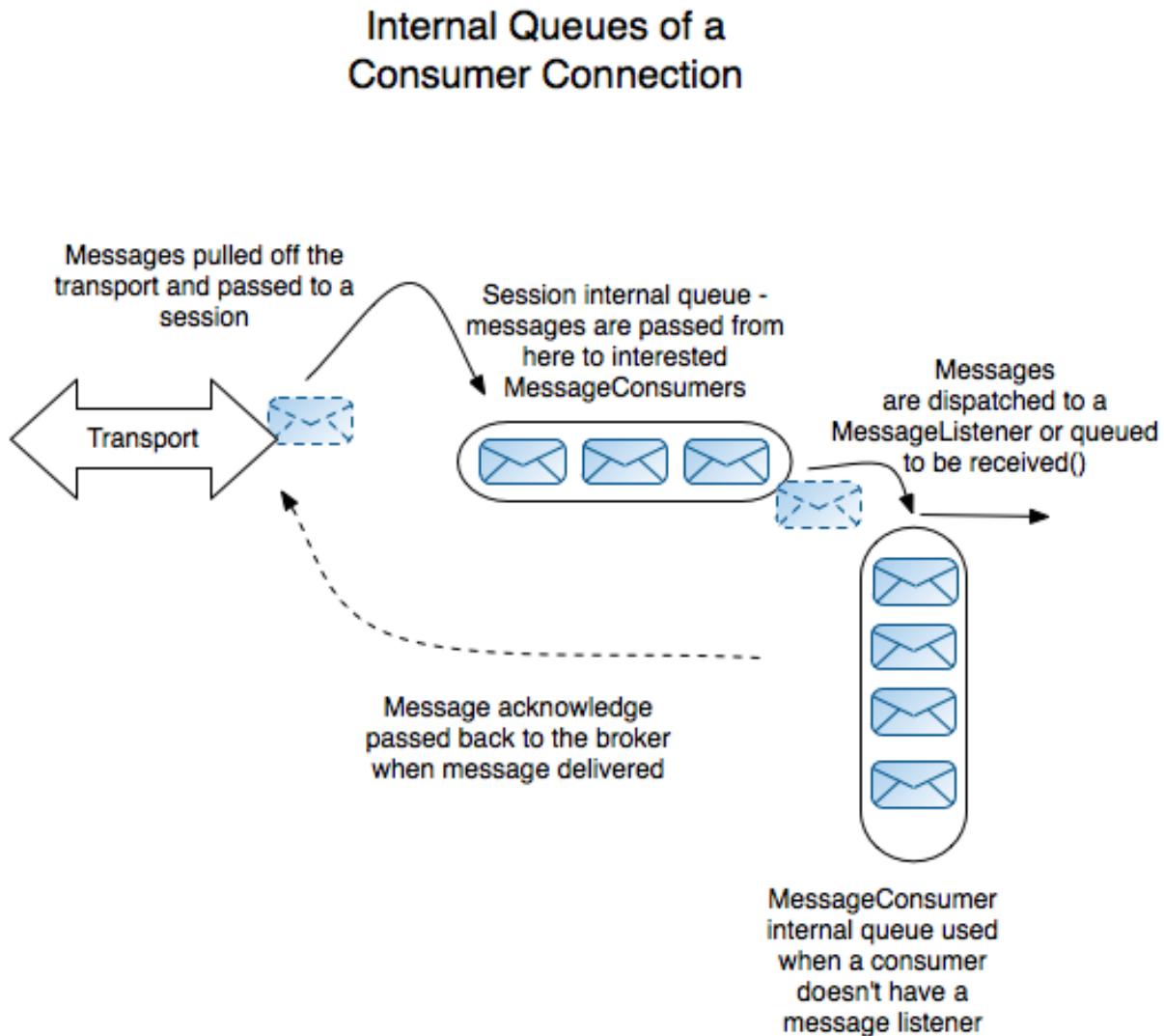


Figure 13.5. Connection internals

A Consumer connection will queue messages to be delivered internally, and the size of these queues (plus messages inflight - or on the transport between the broker and the Consumer is limited by the prefetch limit for that consumer. In general, the larger the prefetch, the faster the consumer will work.

However, this isn't always ideal for Queues, where you might want to ensure messages are evenly distributed across all consumers of a queue. In this case with a large prefetch, a slow consumer could have pending messages waiting to be

processed that could have been worked on by a faster consumer. In this case a lower prefetch number would work better. If the prefetch is zero - the consumer will pull messages from the broker - and no push will be involved.

There are different default prefetch sizes for different consumers - these are as follows:

- Queue Consumer default prefetch size = 1000
- Queue Browser Consumer - default prefetch size = 500
- persistent Topic Consumer default prefetch size = 100
- non-persistent Topic Consumer default prefetch size = 32766

The prefetch size is the number of outstanding messages that your Consumer will have waiting to be delivered - not the memory limit. You can set the prefetch size for your connection by configuring the ActiveMQConnectionFactory - as shown below in ???:

Example 13.12. Setting the prefetch policy

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();

Properties props = new Properties();
props.setProperty("prefetchPolicy.queuePrefetch", "1000");
props.setProperty("prefetchPolicy.queueBrowserPrefetch", "500");
props.setProperty("prefetchPolicy.durableTopicPrefetch", "100");
props.setProperty("prefetchPolicy.topicPrefetch", "32766");

cf.setProperties(props);
```

or you can pass the prefetch size as a destination property when you create a destination - as seen in ???:

Example 13.13. Setting prefetch policy on a Destination

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

Prefetch limits are an easy mechanism to boost performance, but should be used with caution. For Queues you should consider the impact on your application if you have a slow consumer and for Topics factor how much memory your messages will consume on the client before they are delivered.

Controlling the rate that messages are delivered to a consumer is only part of the story. Once the message reaches the consumer's connection, the method of message delivery to the consumer and the options chosen for acknowledging the delivery of that message back to the ActiveMQ broker have an impact on performance. We will be covering these in the next section.

13.3.2. Delivery and Acknowledgement of messages

Something that should be apparent from Figure 13.5 is that delivery of messages via a javax.jms.MessageListener will always be faster with ActiveMQ than calling receive(). If a MessageListener is not set for a MessageConsumer - then its messages will be queued for that consumer, waiting for a receive() to be called. Not only will maintaining the internal queue for the consumer be expensive, but so will the context switch by the application thread calling the receive().

As the ActiveMQ broker keeps a record of how many messages have been consumed to maintain its internal prefetch limits a MessageConsumer has to send a message acknowledgement for every message it has consumed. When you use transactions, this happens at the Session commit() call, but is done individually for each message if you are using auto acknowledgement.

There are some optimizations used for sending message acknowledgements back to the broker which can drastically improve the performance when using the DUPS_OK_ACKNOWLEDGE session acknowledgment mode. In addition you can set the optimizeAcknowledge property on the ActiveMQ ConnectionFactory to give a hint to the consumer to roll up message acknowledgements - as seen in ???:

Example 13.14. Setting optimize acknowledge

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
cf.setOptimizeAcknowledge(true);
```

Please post comments or corrections to the [Author Online Forum](#)

When using optimizeAcknowledge or the DUPS_OK_ACKNOWLEDGE acknowledgment mode on a session, the MessageConsumer can send one message acknowledgement to the ActiveMQ message broker containing a range of all the messages consumed. This reduces the amount of work the MessageConsumer has to do, enabling it to consume messages at a much faster rate.

Table 13.2, “”. below outlines the different options for acknowledging messages and how often they send back a message acknowledgement to the ActiveMQ message broker.

Example 13.15. Different acknowledgment modes

Table 13.2.

Acknowledgement Mode	Sends an Acknowledgement	description
Session.SESSION_TRANSACTED	Up acknowledgements with Session.commit()	Reliable way for message consumption - and performs well providing you consume more than one message in a commit.
Session.CLIENT_ACKNOWLEDGE	Messages upto when a message is acknowledged are consumed.	Can perform well, providing the application consumes a lot of messages before calling acknowledge.
Session.AUTO_ACKNOWLEDGE	A message acknowledgement back to the ActiveMQ broker for every message consumed	This can be slow - but is often the default mechanism for message consumers.
Session.DUPS_OK_ACKNOWLEDGE	The consumer to send one acknowledgement back to the ActiveMQ	An acknowledgement will be sent back when the prefetch limit has reached

Acknowledgement Mode	Sends an Acknowledgement	description
	broker for a range of messages consumed.	50%. The fastest standard way of consuming messages.
ActiveMQSession.INDIVIDUAL	Sends ACKNOWLEDGE acknowledgement for every message consumed.	Allows great control by enabling messages to be acknowledged individually - but can be slow.
optimizeAcknowledge	Allows the consumer to send one acknowledgement back to the ActiveMQ broker for a range of messages consumed.	A hint that works in conjunction with Session.AUTO_ACKNOWLEDGE. An acknowledgement will be sent back when the prefetch limit has reached 65%. The fastest way of consuming messages.

The downside to not acknowledging every message individually is that if the MessageConsumer were to lose its connection with the ActiveMQ broker or die - then your messaging application could receive duplicate messages. However for applications that require fast throughput (e.g. real time data feeds) and are less concerned about duplicates - using optimizeAcknowledge is the recommended approach.

The ActiveMQ MessageConsumer incorporates duplicate message detection, which helps minimize the risk of receiving the same message more than once.

13.3.3. Asynchronous dispatch

Every Session maintains an internal queue of messages to be dispatched to interested MessageConsumers (as can be seen from Figure 13.5). The usage of an

internal queue together with an associated thread to do the dispatching to MessageConsumers can add considerable overhead to the consumption of messages.

There is a property called alwaysSessionAsync you can disable on the ActiveMQ ConnectionFactory to turn this off - allowing messages to be passed directly from the Transport to the MessageConsumer. This property can be disabled as below in ???:

Example 13.16. Setting always use asynchronous dispatch

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
cf.setAlwaysSessionAsync(false);
```

Disabling asynchronous dispatch allows messages to bypass the internal queueing and dispatching done by the Session - as shown below in Figure 13.6

Internal Queues of a Consumer Connection with alwaysSessionAsync=false

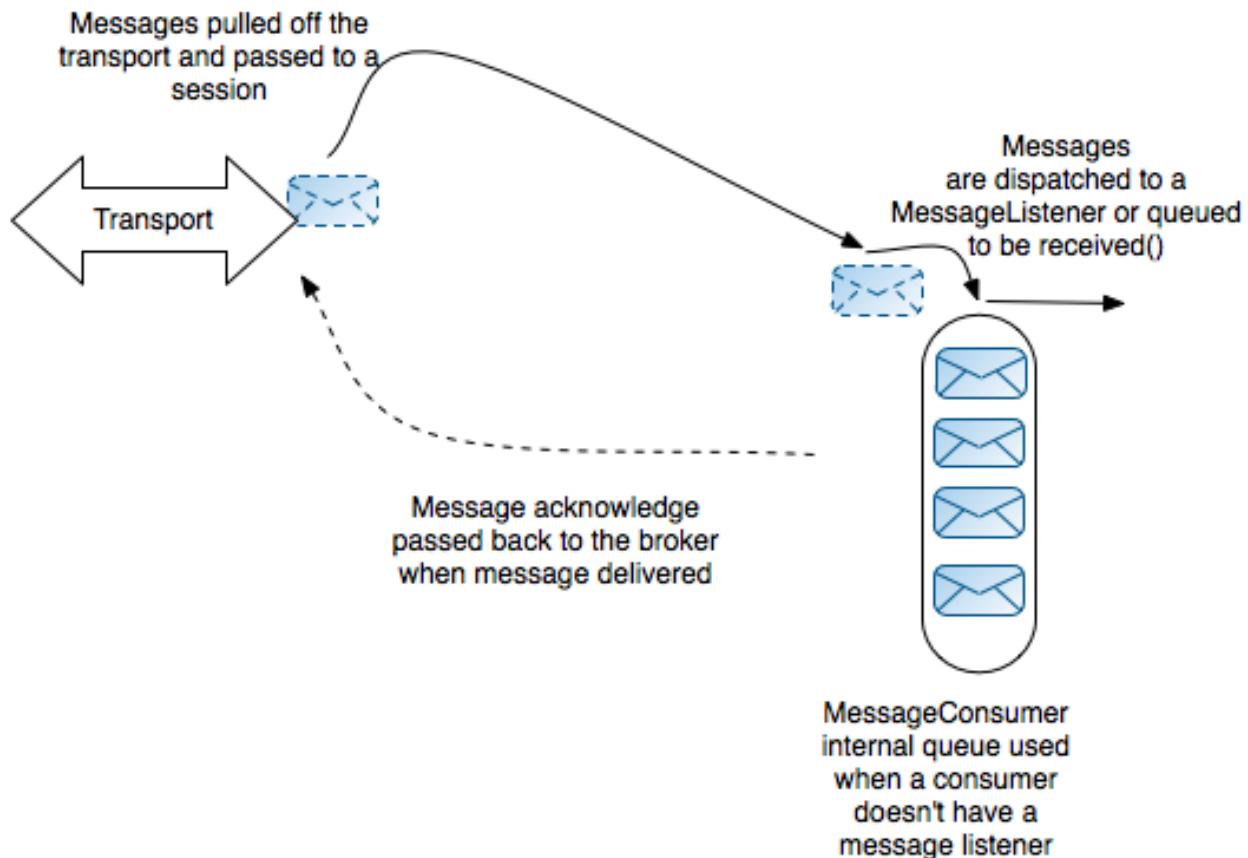


Figure 13.6. Optimized Connection internals

So far we've looked at some general techniques you can use to improve performance, like using reliable messaging instead of guaranteed and co-locating an ActiveMQ broker with a service. We have covered different tuning parameters for transports, producers and consumers.

As using examples are the best way to demonstrate something, in the next section we are going to demonstrate improving performance with an example application -

a real-time data feed.

13.4. Putting it all Together

Lets demonstrate pulling some of these performance tuning options together with an example application. We will simulate a real-time data feed, where the producer is co-located with an embedded broker and an example consumer listens for messages remotely - as shown in Figure 13.7.

What we will be demonstrating is using an embedded broker to reduce the overhead of publishing the data to the ActiveMQ broker. We will show some additional tuning on the message producer to reduce message copying. The embedded broker itself will be configured with flow control disabled and memory limits set to allow for fast streaming of messages through the broker.

Finally the message consumer will be configured for straight- through message delivery, coupled with high prefetch limit and optimized message acknowledgment.

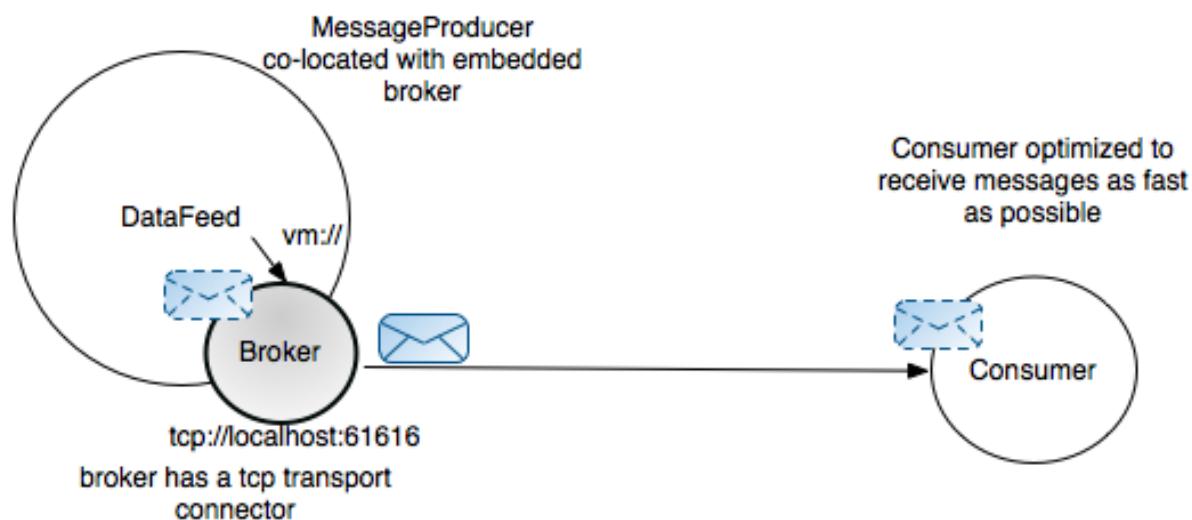


Figure 13.7. Data feed application

Firstly we setup the broker to be embedded, with the memory limit set to a

reasonable amount (64mb), limits set on to each destination and flow control disabled. The policies for the destinations in the broker are set up using a default PolicyEntry - as seen in the following code snippet in ????. A PolicyEntry is a holder for configuration information for a destination used within the ActiveMQ broker. You can have a separate policy for each destination, create a policy to only apply to destinations that match a wild-card (e.g. naming a PolicyEntry "foo.>" will only apply to destinations starting with "foo."). For our example, we are only setting memory limits and disabling flow control. For simplicity, we will only configure the default entry - which will apply to all destinations.

Example 13.17. Creating the embedded broker

```
import org.apache.activemq.broker.BrokerService;
import org.apache.activemq.broker.region.policy.PolicyEntry;
import org.apache.activemq.region.policy.PolicyMap;
...
//By default a broker always listens on vm://<broker name>

BrokerService broker = new BrokerService();
broker.setBrokerName("fast");
broker.getSystemUsage().getMemoryUsage().setLimit(64*1024*1024);

//Set the Destination policies

PolicyEntry policy = new PolicyEntry();

//set a memory limit of 4mb for each destination
policy.setMemoryLimit(4 * 1024 *1024);

//disable flow control

policy.setProducerFlowControl(false);

PolicyMap pMap = new PolicyMap();

//configure the policy

pMap.setDefaultEntry(policy);

broker.setDestinationPolicy(pMap);
broker.addConnector("tcp://localhost:61616");
broker.start();
```

This broker is uniquely named "fast" so that the co-located data feed producer can bind to it using the vm:// transport.

Apart from using an embedded broker, the producer is very straight forward, except its configured to send non-persistent messages and not use message copy. The example producer is configured as in ??? below:

Example 13.18. Creating the producer

```
//tell the connection factory to connect to an embedded broker named fast.  
//if the embedded broker isn't already created, the connection factory will  
//create a default embedded broker named "fast"  
  
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory("vm://fast");  
  
//disable message copying  
  
cf.setCopyMessageOnSend(false);  
  
Connection connection = cf.createConnection();  
connection.start();  
  
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);  
Topic topic = session.createTopic("test.topic");  
final MessageProducer producer = session.createProducer(topic);  
  
//send non-persistent messages  
  
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);  
for (int i =0; i < 1000000;i++) {  
    TextMessage message = session.createTextMessage("Test:"+i);  
    producer.send(message);  
}
```

The consumer is configured for straight through processing (having disabled asynchronous session dispatch) and using a javax.jms.MessageListener. The consumer is set to use optimizeAcknowledge to gain the maximum consumption - as can be seen in ??? below:

Example 13.19. Creating the Consumer

```
//set up the connection factory to connect to the the producer's embedded broker  
//using tcp://
```

Please post comments or corrections to the [Author Online Forum](#)

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory("failover://(tcp://localhost:61616, tcp://localhost:61617)?maxConnections=1000");

//configure the factory to create connections
//with straight through processing of messages
//and optimized acknowledgement

cf.setAlwaysSessionAsync(false);
cf.setOptimizeAcknowledge(true);

Connection connection = cf.createConnection();
connection.start();

//use the default session
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

//set the prefetch size for topics - by parsing a configuration parameter in
// the name of the topic

Topic topic = session.createTopic("test.topic?consumer.prefetchSize=32766");

MessageConsumer consumer = session.createConsumer(topic);

//setup a counter - so we don't print every message

final AtomicInteger count = new AtomicInteger();

consumer.setMessageListener(new MessageListener() {
    public void onMessage(Message message) {
        TextMessage textMessage = (TextMessage)message;
        try {
            //only print every 10,000th message
            if (count.incrementAndGet()%10000==0)
                System.err.println("Got = " + textMessage.getText());
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
});
```

In this section we have pulled together an example for distributing real-time data using ActiveMQ. We have created a demo producer and configured it to pass messages straight through to an embedded broker. We have created the embedded broker, and disabled flow control. Finally we have configured a message consumer to receive messages as fast as possible.

We would recommend trying changing some of the configuration parameters we have set (like optimize acknowledge) to see what impact that has on performance.

13.5. Summary

In this chapter we have learned about some of the general principals for improving performance with any JMS based application. We have also dived into some of the internals of ActiveMQ and how changes to configuration can increase performance. We have learned when and when not to use those options and their side-affects.

We have brought the different aspects of performance tuning together in an example real-time data feed application.

Finally we have seen the special case of caching messages in the broker for non-durable Topic consumers. Why caching is required, when it makes sense to use this feature and the flexibility ActiveMQ provides in configuring the message caches.

In general, message performance can be improved by asking ActiveMQ to do less. So reducing the cost of transport of a message from a producer or consumer to an ActiveMQ broker by co-locating them together and using embedded brokers. If possible use reliable messaging or batching of messages in transactions to reduce the overhead of passing back a receipt from the broker to the producer that it has received a message. You can reduce the amount of work the ActiveMQ broker does by setting suitable memory limits (more is better) and deciding if producer flow control is suitable for your application. The message consumer has to work twice as hard as the message producer, so optimizing delivery with a MessageListener and using straight through message processing together with an acknowledgement mode or transactions that allow acknowledgements to be batched can reduce this load.

You should now have a better understanding of where the performance bottle necks may occur when using ActiveMQ and when and why to alleviate them. We have shown how to tune your message producers and message consumers and the configuration parameters and their impact on your architecture. You should be able to make the right architectural decisions for your application to help performance, whilst have a good understanding of the down sides in terms of guaranteeing delivery and how ActiveMQ can be used to mitigate against them.

Chapter 14. Administering and Monitoring ActiveMQ

The final topic left to be covered is the management and monitoring of ActiveMQ broker instances. As with any other infrastructure software, it is very important for developers and administrators to be able to monitor broker metrics during runtime and notice any suspicious behavior that could possibly impact messaging clients. Also, you might want to interact with your broker in other ways. For example, changing broker configuration properties or sending test messages from administration consoles. ActiveMQ implements some features beyond the standard JMS API that allows for administration and monitoring both programatically and by using a well-known administration tools.

We will start this chapter with the explanation of various APIs you can use to communicate with the broker. First, we will explain the *Java Management Extension API (JMX)*, a standard API of managing Java applications. Next, we will explain the concept of *Advisory messages* which allow you to receive important notifications from the broker in more messaging-like manner.

In later sections we will focus on administrator tools for interacting with brokers. We will explore some of the tools embedded in the ActiveMQ distribution such as the *Command Agent* and the *Web Console* as well as some of the external tools such as *JConsole*.

Finally, we will explain how to adjust the ActiveMQ logging mechanism to suit your needs and demonstrate how to use it to track down potential problems. We will also show you how to change the ActiveMQ logging preferences during runtime.

Now, let's get started with APIs.

14.1. APIs

The natural way to communicate with the message broker is through the JMS API.

Please post comments or corrections to the [Author Online Forum](#)

In addition to messaging client applications, you may have to the need to create Java applications that will monitor the broker during runtime. Some of those monitoring tasks may include:

- Obtaining broker statistics, such as number of consumers (total or per destination)
- Adding new connectors or removing existing ones
- Changing some of the broker configuration properties

For this purpose, ActiveMQ provides some mechanisms that can be used to manage and monitor your broker during runtime.

14.1.1. JMX

Nearly every story on management and monitoring in the Java world begins with *Java Management Extensions (JMX)*. The JMX API allows you to implement *management interfaces* for your Java applications by exposing functionality to be managed. These interfaces consist of *Management Beans*, usually called *MBeans*, which expose resources of your application to external management applications. To access the MBeans for ActiveMQ, the JMX agent in the JVM must first be enabled. In order to understand the separation between the JMX agent and the MBeans for ActiveMQ, you must first understand some basics about JMX and how it's used by ActiveMQ.

14.1.1.1. Local vs. Remote JMX Access

The JVM provides what is known as the JMX agent. The JMX agent is comprised of the MBean server, some agent services and some protocol adapters and connectors. It is the JMX agent that is used to expose the ActiveMQ MBeans. In order to control various aspects of the JMX agent, a set of properties is provided. Through the use of these properties, various features in the JMX agent can be enabled and disabled. For more information, see the document titled [Monitoring and Management for the Java Platform](#).

Below is a snippet from the ActiveMQ startup script for Linux/Unix concerning the JMX capabilities:

```
if [ -z "$SUNJMX" ] ; then
    #SUNJMX="-Dcom.sun.management.jmxremote.port=1099 -Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false"
    SUNJMX="-Dcom.sun.management.jmxremote"
fi
```

Below is the same snippet from the ActiveMQ startup script for Windows concerning the SUNJMX variable:

```
if "%SUNJMX%" == "" set SUNJMX=-Dcom.sun.management.jmxremote
REM set SUNJMX=-Dcom.sun.management.jmxremote.port=1099 -Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

Note that each of the snippets demonstrates the use of a variable named SUNJMX. This is a variable that is specific to the ActiveMQ startup scripts that is used to hold the JMX properties that are recognized by the JVM. Each of these snippets shows that the only JMX property that is enabled by default is the com.sun.management.jmxremote property. Don't let the name of this property fool you, as it simply enables the JMX agent for *local access* only. This can be easily tested by starting up Jconsole and seeing ActiveMQ in the list of locally accessible objects as shown in Figure 14.1, “Accessing ActiveMQ locally from Jconsole”.

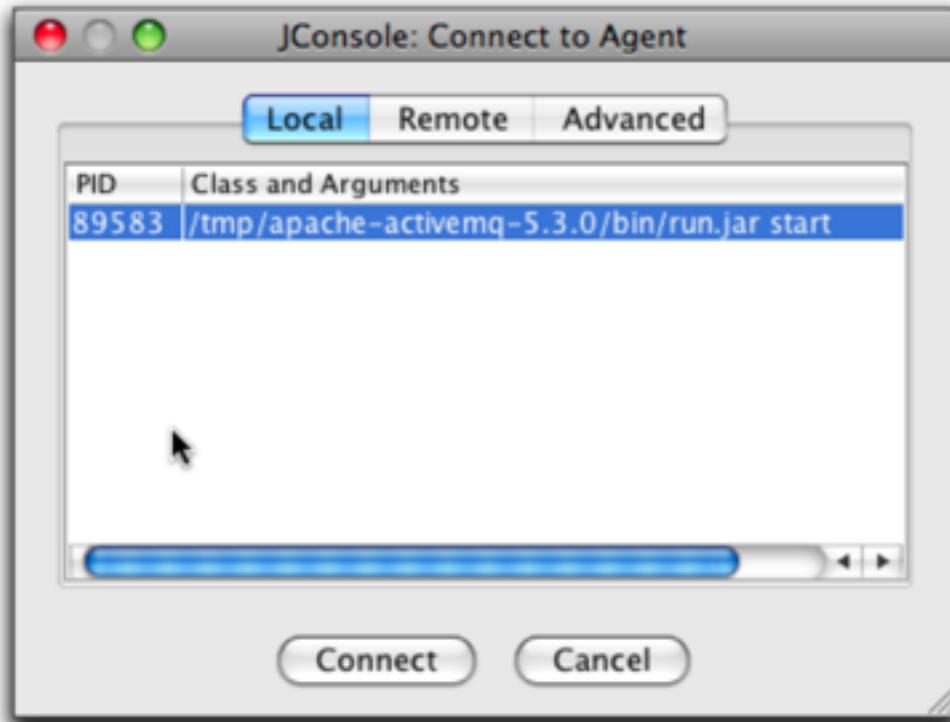


Figure 14.1. Accessing ActiveMQ locally from Jconsole

Upon selecting the ActiveMQ run.jar and clicking the Connect button, the main screen will appear in Jconsole as shown in Figure 14.2, “The main jconsole screen with the ActiveMQ domain included”.

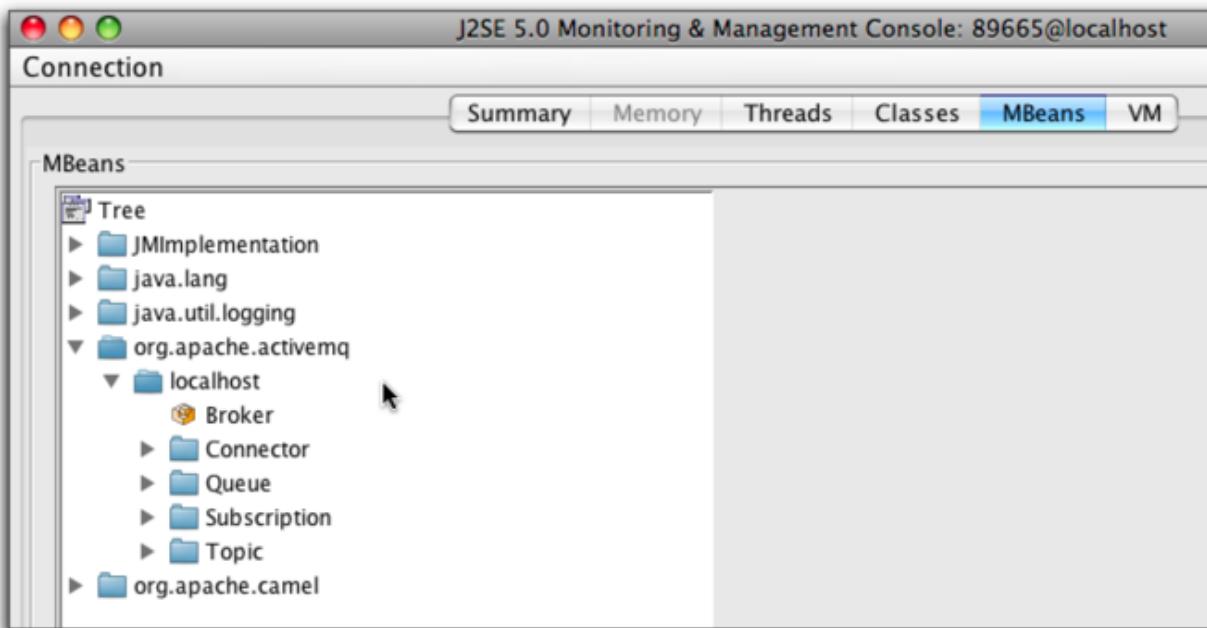


Figure 14.2. The main jconsole screen with the ActiveMQ domain included

Notice in Figure 14.2, “The main jconsole screen with the ActiveMQ domain included” near the cursor is the ActiveMQ JMX domain (listed as `org.apache.activemq`). Using the default configuration and startup script in ActiveMQ, this is what will appear in Jconsole, indicating that the JMX agent is enabled for *local* access. However, attempts to access ActiveMQ *remotely* (i.e., from a remote host) via Jconsole will fail as the JMX agent has not been exposed on a specific port number using the `com.sun.management.jmxremote.port` property. More on this in Section 14.1.1.4, “Advanced JMX Configuration”.

Below is a snippet from the default broker configuration with the `useJmx` attribute explicitly enabled:

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

By simply changing the `useJmx` attribute from true to false, the ActiveMQ domain

will no longer be available for access:

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

Upon making this small configuration change, you will be able to access the JMX agent via the ActiveMQ run.jar as shown in Figure 14.1, “Accessing ActiveMQ locally from Jconsole”, but the ActiveMQ domain will not be available as shown in Figure 14.3, “The main Jconsole screen without the ActiveMQ domain included”.

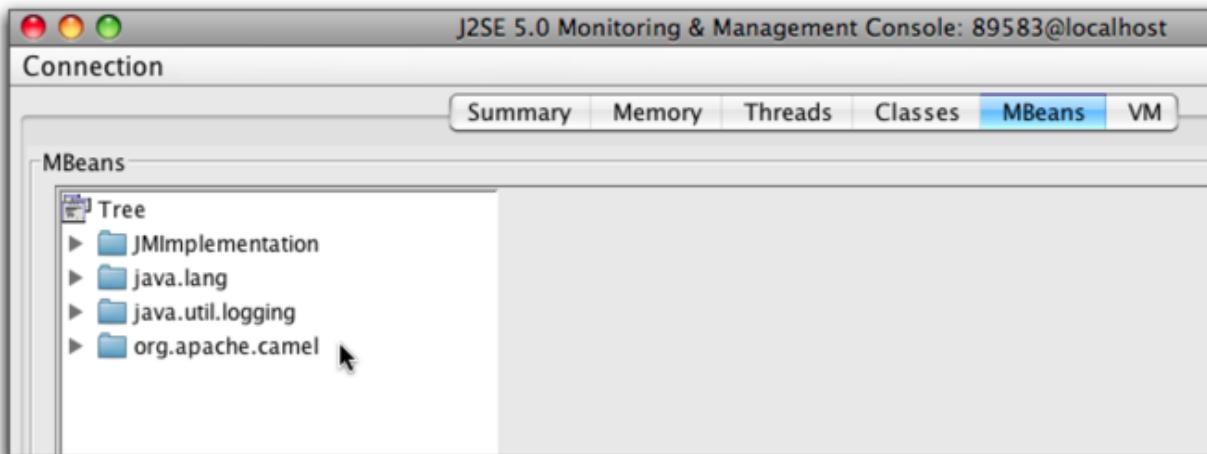


Figure 14.3. The main Jconsole screen without the ActiveMQ domain included

The reason for this is that the JMX agent for the JVM and the domain for ActiveMQ are distinct. The JMX agent in the JVM is controlled by the `com.sun.management.jmxremote` property whereas the ActiveMQ domain is controlled by the `useJmx` attribute in the broker configuration file.

14.1.1.2. Exposing the JMX MBeans For ActiveMQ

By default, the the MBeans for ActiveMQ are enabled to be exposed for ease of

use. In order to fully utilize the MBeans, there are additional properties in the broker configuration file to enable additional functionality. The following configuration shows what needs to be changed in the broker configuration to enable JMX support:

Example 14.1. ActiveMQ JMX configuration

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

    <managementContext>
        <managementContext connectorPort="2011" jmxDomainName="my-broker" />
    </managementContext>

    <!-- The transport connectors ActiveMQ will listen to -->
    <transportConnectors>
        <transportConnector name="openwire" uri="tcp://localhost:61616" />
    </transportConnectors>

</broker>
```

There are two important items in the configuration file above that are related to the JMX configuration. The first is the `useJmx` attribute of the `<broker>` element that enables/disables JMX support. The value of this attribute is `true` by default so the broker uses JMX by default, but it is included in this example configuration in order to be explicit for demonstration purposes.

By default, ActiveMQ starts a connector which enables remote management on port 1099 and exposes MBeans using the `org.apache.activemq` domain name. These default values are sufficient for most use cases, but if you need to customize the JMX context further, you can do so using the `<managementContext>` element. In our example we changed the port to 2011 and the domain name to `my-broker`. You can find all management context properties at the following reference page:

<http://activemq.apache.org/jmx.html#JMX-ManagementContextPropertiesReference>

Now we can start the broker with the following command:

```
$ ${ACTIVEMQ_HOME}/bin/activemq \
xbean:${EXAMPLES}src/main/resources/org/apache/activemq/book/ch14/activemq-jmx.xml
```

Please post comments or corrections to the [Author Online Forum](#)

Among the usual log messages shown during the broker startup, you can notice the following line:

```
INFO ManagementContext - JMX consoles can connect to  
service:jmx:rmi:///jndi/rmi://localhost:2011/jmxrmi
```

This is the JMX URL we can use to connect to the broker using a utility such as JConsole as discussed later in the chapter. As you can see from the output, the port number for accessing the broker via JMX has been changed from 1099 to 2011.

Now that the JMX support has been enabled in ActiveMQ, you can begin utilizing the JMX APIs to interact with the broker.

14.1.1.3. Using the JMX APIs With ActiveMQ

Using the JMX API, statistics can be obtained from a broker at runtime. The example shown in Example 14.2, “ActiveMQ broker statistics” connects to the broker via JMX and prints out some of the basic statistics such as total number of messages, consumers and queues. Next it iterates through all available queues and print their current size and number of consumers subscribed to them.

Example 14.2. ActiveMQ broker statistics

```
public class Stats {  
  
    public static void main(String[] args) throws Exception {  
  
        JMXServiceURL url = new JMXServiceURL(  
            "service:jmx:rmi:///jndi/rmi://localhost:2011/jmxrmi");  
        JMXConnector connector = JMXConnectorFactory.connect(url, null);  
        connector.connect();  
        MBeanServerConnection connection = connector.getMBeanServerConnection();❶  
  
        ObjectName name = new ObjectName(  
            "my-broker:BrokerName=localhost,Type=Broker");  
        BrokerViewMBean mbean = (BrokerViewMBean) MBeanServerInvocationHandler  
            .newProxyInstance(connection, name, BrokerViewMBean.class, true);❷  
  
        System.out.println("Statistics for broker " + mbean.getBrokerId()  
            + " - " + mbean.getBrokerName());  
        System.out.println("\n-----\n");  
        System.out.println("Total message count: " + mbean.getTotalMessageCount() + "\n");  
        System.out.println("Total number of consumers: " + mbean.getTotalConsumerCount());  
        System.out.println("Total number of Queues: " + mbean.getQueues().length);❸  
    }  
}
```

Please post comments or corrections to the [Author Online Forum](#)

```
for (ObjectName queueName : mbean.getQueues()) {
    QueueViewMBean queueMbean = (QueueViewMBean) MBeanServerInvocationHandler
        .newProxyInstance(connection, queueName,
            QueueViewMBean.class, true);
    System.out.println("\n-----\n");
    System.out.println("Statistics for queue " + queueMbean.getName());
    System.out.println("Size: " + queueMbean.getQueueSize());
    System.out.println("Number of consumers: " + queueMbean.getConsumerCount());❸
}
}
```

- ❶ This block of code creates a connection to the MBean server
- ❷ This block of code queries for the broker MBean
- ❸ This block of code grabs some broker statistics from the MBean
- ❹ This block of code grabs some queue statistics from the queue MBeans

The example above is using the standard JMX API to access and use broker and request information. For starters, we have to create an appropriate connection to the broker's MBean server. Note that we have used the URL previously printed in the ActiveMQ startup log. Next, we will use the connection to obtain the MBean representing the broker. The MBean is referenced by its name, which in this case has the following form:

```
<jmx domain name>:BrokerName=<name of the broker>,Type=Broker
```

The JMX object name for the ActiveMQ MBean using the default broker configuration is as follows:

```
org.apache.activemq:BrokerName=localhost,Type=Broker
```

But recall that back in Example 14.1, “ActiveMQ JMX configuration” that the JMX domain name was changed from `localhost` to `my-broker`. Therefore the JMX object name for the changed broker configuration looks like the following:

```
my-broker:BrokerName=localhost,Type=Broker
```

Using this object name to fetch the broker MBean, now the methods on the MBean

can be used to acquire the broker statistics as shown in Example 14.2, “ActiveMQ broker statistics”. In this example, we print the total number of messages (`getTotalMessageCount()`), the total consumer count (`getTotalConsumerCount()`) and the total number of queues(`getQueues().length()`).

The `getQueues()` method returns the object names for all the queue MBeans. These names have the similar format as the broker MBean object name. For example, one of the queues we are using in the jobs queue named JOBS.suspend and it has the following MBean object name:

```
my-broker:BrokerName=localhost,Type=Queue,Destination=JOBS.suspend
```

The only difference between this queue's object name and broker's object name is in the portion marked in bold. This portion of the object name states that this MBean represents a a type of Queue and has an attribute named Destination with the value JOBS.suspend.

Now it's time to begin to examine the job queue example to see how to capture broker runtime statistics using the example from Example 14.2, “ActiveMQ broker statistics”. But first the consumer must be slowed down a bit to be sure that some messages exist in the system before the statistics are gathered. For this purpose the following broker URL is used:

```
private static String brokerURL =
    "tcp://localhost:61616?jms.prefetchPolicy.all=1";
```

Notice the parameter on the URI for the broker (the bold portion). This parameter ensures that only one message is dispatched to the consumer at the time. (For more information about the ActiveMQ prefetch policy, see ???).

Additionally, the consumer can be slowed down by adding a one second sleep to the thread for every message that flows through the `Listener.onMessage()` method. Below is an example of this:

```
public void onMessage(Message message) {
    try {
        //do something here
        System.out.println(job + " id:" + ((ObjectMessage)message).getObject());
        Thread.sleep(1000);
    } catch (Exception e) {
```

Please post comments or corrections to the [Author Online Forum](#)

```
    e.printStackTrace();
}
```

The consumer (and listener) modified in this manner have been placed into package `org.apache.activemq.book.ch14.jmx` and we will use it in the rest of this section.

Now the producer can be started just like it was started in Chapter 2, *Understanding Message-Oriented Middleware and JMS*:

```
mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch2.jobs.Publisher
```

And the modified consumer can be run as well:

```
mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch14.jmx.Consumer
```

Finally, run the JMX statistics class using the following command:

```
mvn -e exec:java -Dexec.mainClass=org.apache.activemq.book.ch14.jmx.Stats
```

The `org.apache.activemq.book.ch14.jmx.Stats` class output is shown below:

```
Statistics for broker ID:dejanb-52630-1231518649948-0:0 - localhost
-----
Total message count: 670

Total number of consumers: 2
Total number of Queues: 2

-----
Statistics for queue JOBS.suspend
Size: 208
Number of consumers: 1

-----
Statistics for queue JOBS.delete
Size: 444
Number of consumers: 1
```

Notice that the statistics from the `Stats` class are output to the terminal. There are

many more statistics on the MBeans from ActiveMQ. The example shown here is meant only to be an introduction.

As you can see, it is very easy to access ActiveMQ using the JMX API. This will allow you to monitor the broker status, which is very useful in both development and production environments. But what if you want to restrict access to the JMX capabilities?

14.1.1.4. Advanced JMX Configuration

There are situations where some advanced configuration of the JMX agent is necessary. This includes remote access, restricting access to a specific host and restricting access to particular users via authentication. Most of these tasks are fairly easy to achieve through the use of the JMX agent properties and by slightly modifying the ActiveMQ startup script.

Again, remember the snippet from the ActiveMQ startup script for Linux/Unix concerning the JMX capabilities:

```
if [ -z "$SUNJMX" ] ; then
    #SUNJMX="-Dcom.sun.management.jmxremote.port=1099 \
-Dcom.sun.management.jmxremote.authenticate=false \
-Dcom.sun.management.jmxremote.ssl=false"
    SUNJMX="-Dcom.sun.management.jmxremote"
fi
```

As well as the same snippet from the ActiveMQ startup script for Windows concerning the SUNJMX variable:

```
if "%SUNJMX%" == "" set SUNJMX=-Dcom.sun.management.jmxremote
REM set SUNJMX=-Dcom.sun.management.jmxremote.port=1099 \
-Dcom.sun.management.jmxremote.authenticate=false \
-Dcom.sun.management.jmxremote.ssl=false
```

Notice the portions that are commented out. These three additional properties will be covered in this section as well as a fourth, related property.

14.1.1.4.1. Enabling Remote JMX Access

Sometimes it's necessary to allow access to the JMX agent from a remote host. Enabling remote access to the JMX agent is actually quite easy. The default

ActiveMQ startup scripts include a configuration for remote access to the JMX agent using the `com.sun.management.jmxremote.port` property, but it is commented out. By simply adding this property to the uncommented portion of the `SUNJMX` variable, remote access will be enabled on the specified port number.

Below is a snippet from the ActiveMQ startup script for Linux/Unix with the `com.sun.management.jmxremote.port` property enabled:

```
if [ -z "$SUNJMX" ] ; then
  SUNJMX="-Dcom.sun.management.jmxremote.port=1099 \
-Dcom.sun.management.jmxremote.authenticate=false \
-Dcom.sun.management.jmxremote.ssl=false"
fi
```

Below is the same snippet from the ActiveMQ startup script for Windows with the `com.sun.management.jmxremote.port` property enabled:

```
if "%SUNJMX%" == "" set SUNJMX=-Dcom.sun.management.jmxremote.port=1099 \
-Dcom.sun.management.jmxremote.authenticate=false \
-Dcom.sun.management.jmxremote.ssl=false
```

Notice that the `com.sun.management.jmxremote.port` property is now enabled and port number 1234 has been specified. Also, two additional JMX properties related to JMX security have been enabled as well. Specifically the `com.sun.management.jmxremote.authentication` property and the `com.sun.management.jmxremote.ssl` property and both have been set to false. These two properties are included and set to false to disable security because otherwise they are both true by default for remote monitoring. The JMX agent in the JVM where ActiveMQ is started will be available for access via port number 1234. After enabling remote access, you can test this using Jconsole's Remote tab as shown in Figure 14.4, "Accessing the JMX Agent remotely on port 1234".

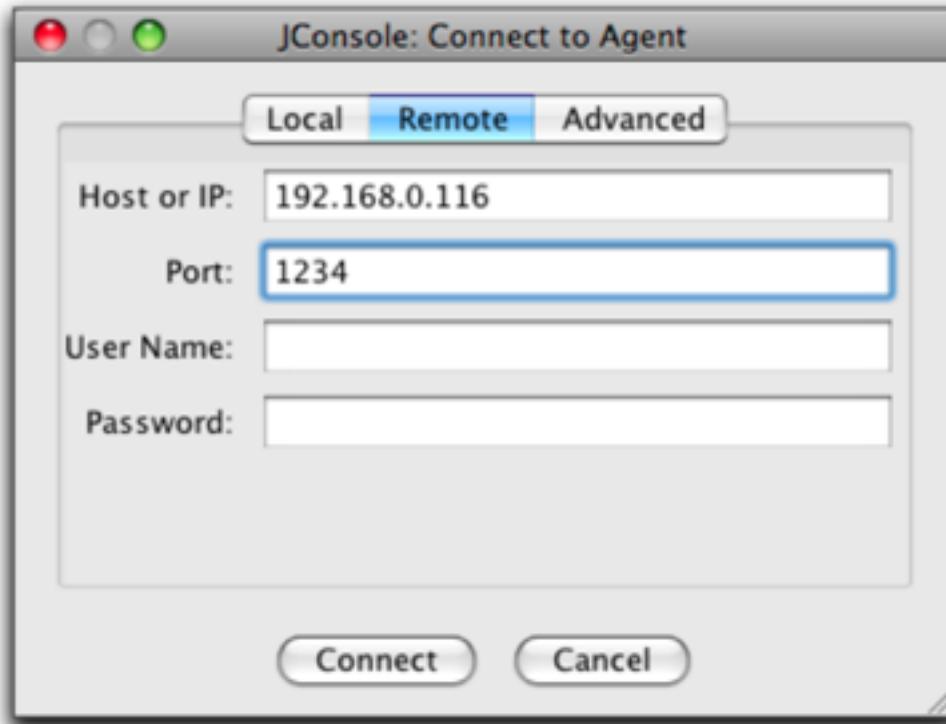


Figure 14.4. Accessing the JMX Agent remotely on port 1234

Upon successfully connecting to the remote ActiveMQ instance, you will be able to remotely manage and monitor ActiveMQ.

Note

In order for the JMX remote access to work successfully, the `/etc/hosts` file must be in order. Specifically, the `/etc/hosts` file must contain more than just the entry for the localhost on 127.0.0.1. The `/etc/hosts` file must also contain an entry for the real IP address and the hostname for a proper configuration. Below is an example of a proper configuration:

```
127.0.0.1      localhost
192.168.0.23    krustykrab.bsnnyder.org  krustykrab
```

Notice the portion of the `/etc/hosts` file above that contains an entry for the localhost and an entry for the proper hostname and IP address.

14.1.1.4.2. Restricting JMX Access to a Specific Host

Sometimes there is a need to restrict the use of JMX to specific host in the environment, such as the host on which ActiveMQ is running. The fourth related property mentioned above is one that will provide this type of restriction via the Java SE, not by ActiveMQ itself. The `java.rmi.server.hostname` property is used to provide just such a restriction. As defined in the Java SE documents on the section about [RMI properties](#):

The value of this property represents the host name string that should be associated with remote stubs for locally created remote objects, in order to allow clients to invoke methods on the remote object. In 1.1.7 and later, the default value of this property is the IP address of the local host, in "dotted-quad" format.

This property must be added to the `SUNJMX` variable in the ActiveMQ startup script. Below is an example of adding this property to the ActiveMQ startup script for Linux/Unix:

```
if [ -z "$SUNJMX" ] ; then
    SUNJMX="-Dcom.sun.management.jmxremote -Djava.rmi.server.hostname=localhost"
fi
```

And here an example of adding this property to the ActiveMQ startup script for Windows:

```
if "%SUNJMX%" == "" set SUNJMX=-Dcom.sun.management.jmxremote -Djava.rmi.server.hostname=loc
```

This slight change to use the `java.rmi.server.hostname` property simply notes the name of the host to which the access should be restricted. After making this change, the ActiveMQ MBeans can only be accessed from this host.

14.1.1.4.3. Configuring JMX Password Authentication

Password authentication for the JMX agent is controlled by an access file and a password file. The access file is used to define roles and assign those roles permissions. The password file is used to map roles to passwords. The JVM provides examples of each of these files so the best way to begin is to take a look

Please post comments or corrections to the [Author Online Forum](#)

these files. The files we want to see are located in the `$JAVA_HOME/jre/lib/management/` directory and are named `jmxremote.access` and `jmxremote.password.template`. Each of these files is intended to provide a starting point for you to define your own values.

Below is the contents of the default `jmxremote.access` file:

```
monitorRole❶ readonly❷
controlRole   readwrite
```

- ❶ The role name
- ❷ The access level

The format of this file is noted above where the `monitorRole` is the role name and `readonly` is the access type. The role name needs to correspond to a role name in the password file.

The contents of the default `jmxremote.password.template` file is empty but provides the following suggestion:

```
monitorRole ❶QED❷
controlRole R&D
```

- ❶ The role name
- ❷ The password

The format for this file is noted above where the `monitorRole` is the role name and `QED` is the password. This role name corresponds to the role name in the `jmxremote.access` file above. The idea with the `jmxremote.password.template` file is that it should be used as a template. That is, you should make a copy of the file to use as your password file and make changes specific to your needs.

To make a copy of the `jmxremote.password.template` file for Linux/Unix, use the following command:

```
$ cp $JAVA_HOME/jre/lib/management/jmxremote.password.template \
$JAVA_HOME/jre/lib/management/jmxremote.password
```

To make a copy of the `jmxremote.password.template` file for Windows, use the following command:

```
> copy %JAVA_HOME%\jre\lib\management\jmxremote.password.template \
%JAVA_HOME%\jre\lib\management\jmxremote.password
```

Edit the new `jmxremote.password` file so that the contents match the following:

```
myRole foo
yourRole bar
```

Notice that two new roles have been defined with their own passwords.

Now edit the `jmx.access` file to include the two new roles so that the contents match the following:

```
myRole readwrite
yourRole readonly
```

Notice how the roles in the `jmxremote.access` and `jmxremote.password` files correspond to one another.

The last requirement is to enable password authentication to the JMX agent is to remove the `com.sun.management.jmxremote.authenticate` property from the `SUNJMX` variable in the ActiveMQ startup script. Below is an example of removing this property from the ActiveMQ startup script for Linux/Unix:

```
if [ -z "$SUNJMX" ] ; then
  SUNJMX="-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=1099
-Dcom.sun.management.jmxremote.ssl=false"
fi
```

And here an example of removing this property from the ActiveMQ startup script for Windows:

```
if "%SUNJMX%" == "" set SUNJMX=-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.ssl=false
```

After removing the `com.sun.management.jmxremote.authenticate` property, the last thing to do is make sure that the user running the JVM has access to the `jmxremote.password` file. Also make sure the file is not anonymously readable.

Note

After copying the `jmxremote.password.template` file, chances are that you will need to change the permissions on that file to disallow read

access. If read access is allowed on the `jmxremove.password` file, the following error will potentially rear its head when starting up ActiveMQ:

```
$ ./bin/activemq
Error: Password file read access must be restricted: \
/opt/jdk1.5.0_15/jre/lib/management/jmxremote.password
```

The only thing left to do is start up ActiveMQ and make sure that there are no errors:

```
$ ./bin/activemq
Java Runtime: Sun Microsystems Inc. 1.5.0_15 /opt/jdk1.5.0_15/jre
  Heap sizes: current=4992k free=3865k max=504896k
    JVM args: -Xmx512M -Dorg.apache.activemq.UseDedicatedTaskRunner=true
-Djava.util.logging.config.file=logging.properties -Dcom.sun.management.jmxremote.port=1099
-Dcom.sun.management.jmxremote.ssl=false -Dactivemq.classpath=/home/bsnyder/apache-activemq-5.3.0
-Dactivemq.home=/home/bsnyder/apache-activemq-5.3.0 -Dactivemq.base=/home/bsnyder/apache-activemq-5.3.0
ACTIVEMQ_HOME: /home/bsnyder/apache-activemq-5.3.0
ACTIVEMQ_BASE: /home/bsnyder/apache-activemq-5.3.0
Loading message broker from: xbean:activemq.xml
INFO | Using Persistence Adapter: org.apache.activemq.store.kahadb.KahaDBPersistenceAdapter
INFO | Replayed 1 operations from the journal in 0.014 seconds.
INFO | ActiveMQ 5.3.0 JMS Message Broker (localhost) is starting
INFO | For help or more information please see: http://activemq.apache.org/
INFO | JMX consoles can connect to service:jmx:rmi:///jndi/rmi://localhost:2011/jmusrmi
INFO | Listening for connections at: tcp://krustykrab:61616
INFO | Connector openwire Started
INFO | ActiveMQ JMS Message Broker (localhost, ID:krustykrab-8119-1255404256224-0:0) started
INFO | Logging to org.slf4j.impl.JCLLoggerAdapter(org.mortbay.log) via org.mortbay.log.Slf4jLog
INFO | jetty-6.1.9
INFO | ActiveMQ WebConsole initialized.
INFO | Initializing Spring FrameworkServlet 'dispatcher'
INFO | ActiveMQ Console at http://0.0.0.0:8161/admin
INFO | Initializing Spring root WebApplicationContext
INFO | Connector vm://localhost Started
INFO | Camel Console at http://0.0.0.0:8161/camel
INFO | ActiveMQ Web Demos at http://0.0.0.0:8161/demo
INFO | RESTful file access application at http://0.0.0.0:8161/fileserver
INFO | Started SelectChannelConnector@0.0.0.0:8161
```

Now start up Jconsole on a remote machine and attempt to use Jconsole to remotely connect to ActiveMQ. Figure 14.5, “Attempting to remotely connect to the JMX agent with the wrong role name and password” shows what will happen when making an attempt to remotely log in to the JMX agent using a role name and password that do not exist in the `jmxremote.access/jmxremote.password` file

pair.

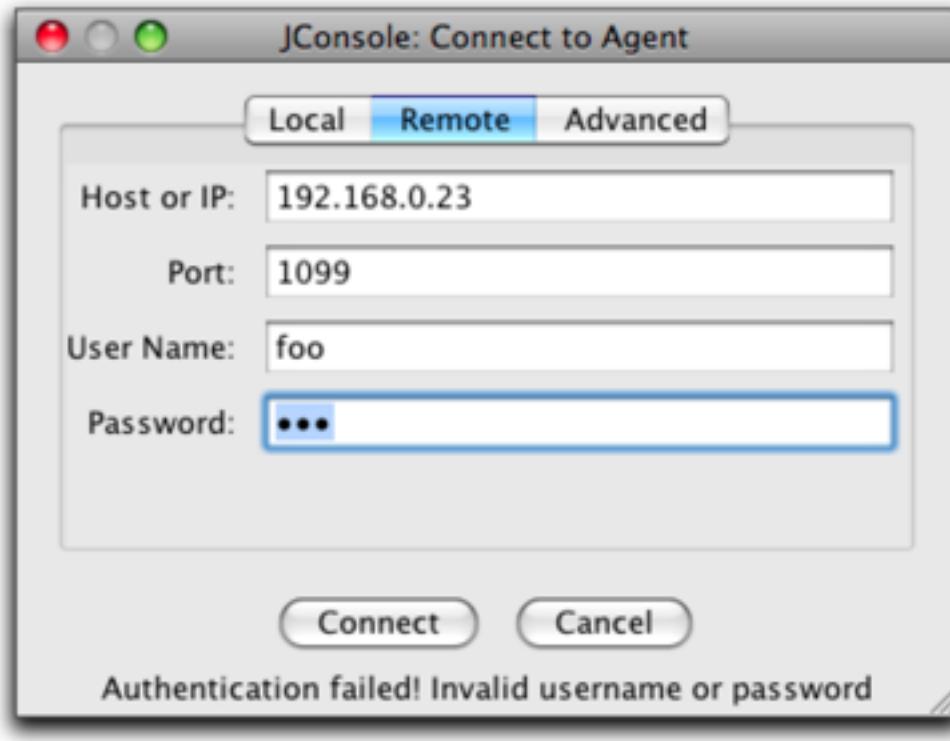


Figure 14.5. Attempting to remotely connect to the JMX agent with the wrong role name and password

To successfully make a remote connection to the JMX agent, you must use the correct role name and password as shown in Figure 14.6, “Making a successful remote connection to the JMX agent requires the correct role name and password”.

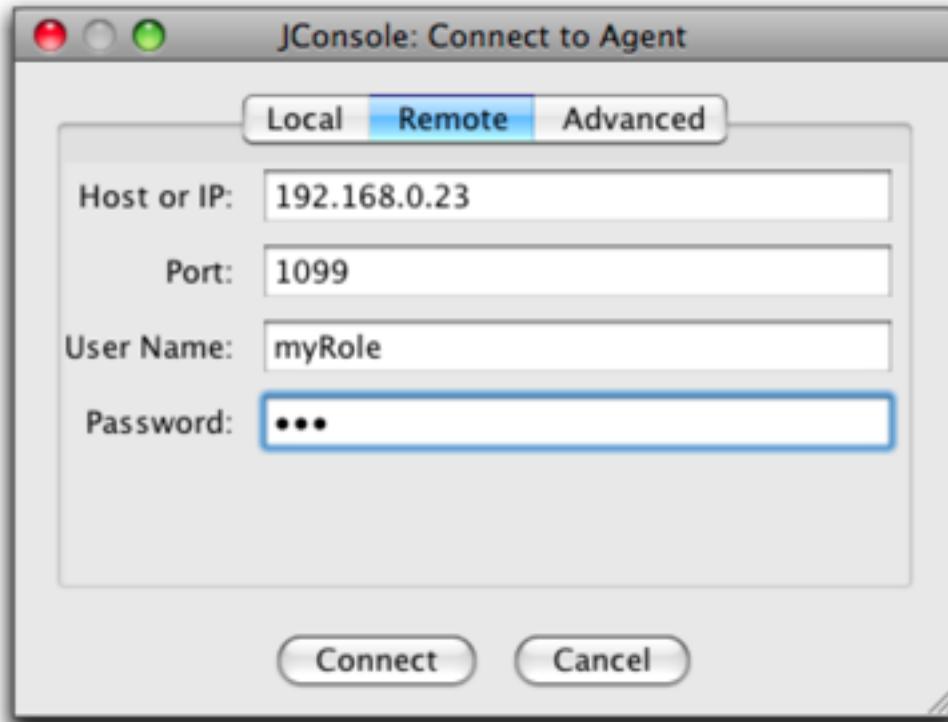


Figure 14.6. Making a successful remote connection to the JMX agent requires the correct role name and password

Hopefully these advanced JMX configurations will help you with your ActiveMQ configurations and your ability to properly configure access.

Beyond the JMX features for monitoring and managing ActiveMQ, there are additional means of monitoring the inner workings of ActiveMQ via what are known as advisory messages.

14.1.2. Advisory Messages

The JMX API is a well known mechanism often used to manage and monitor a wide range of Java applications. But since you're already building a JMS application using ActiveMQ, shouldn't it be natural to receive messages regarding important broker events using the same JMS API? Fortunately, ActiveMQ provides what are known as *Advisory Messages* to represent administrative

commands that can be used to notify messaging clients about important broker events.

Advisory messages are delivered to topics whose names use the prefix `ActiveMQ.Advisory`. For example, if you are interested to know when connections to the broker are started and stopped, you can see this activity by subscribing to the `ActiveMQ.Advisory.Connection` topic. There are variety of advisory topics available depending on what broker events of interest to you. Basic events such as starting and stopping consumers, producers and connections trigger advisory messages by default. But for more complex events such as sending messages to a destination without a consumer, advisory messages must be explicitly enabled. Let's take a look at how to enable advisory messages for this purpose:

Example 14.3. Configuring advisory support

```
<broker xmlns="http://activemq.org/config/1.0" useJmx="true"
    brokerName="localhost" dataDirectory="${activemq.base}/data"
    advisorySupport="true">❶

    <destinationPolicy>
        <policyMap>
            <policyEntries>
                <policyEntry topic="">
                    sendAdvisoryIfNoConsumers="true" />❷
                </policyEntries>
            </policyMap>
        </destinationPolicy>

        <!-- The transport connectors ActiveMQ will listen to -->
        <transportConnectors>
            <transportConnector name="openwire" uri="tcp://localhost:61616" />
        </transportConnectors>

    </broker>
```

- ❶ Advisory support can be enabled using the `advisorySupport` attribute of the `<broker>` element. Please note that advisory support is enabled by default, so technically there is no need to set the `advisorySupport` attribute unless you want to be very explicit about the configuration.
- ❷ The second and more important item above is the use of a *destination policy* to enable more complex advisories for your destinations. In example above,

Please post comments or corrections to the [Author Online Forum](#)

the configuration instructs the broker to send advisory messages if the destination has no consumers subscribed to it. One advisory message will be sent for every message that is sent to the destination.

To demonstrate this functionality, start the broker using the example configuration from above (named `activemq-advisory.xml`) via the following command:

```
$ ./bin/activemq \
xbean:src/main/resources/org/apache/activemq/book/ch14/activemq-advisory.xml
```

To actually demonstrate this functionality we need to create a simple class that makes use of the advisory messages. This Java class will make use of the advisory messages to print log messages to standard output (stdout) whenever a consumer subscribes/unsubscribes or a message is sent to a topic that has no consumers subscribed to it. This example can be run along with the stock portfolio example to make use of the advisory messages (and therefore, certain broker events).

To complete this demonstration, the stock portfolio producer must be modified a bit. ActiveMQ will send an advisory message when a message is sent to a topic with no consumers, but only when those messages are non-persistent. Because of this, we need to modify the producer to send non-persistent messages to the broker by simply setting the delivery mode to non-persistent like this. We'll take a publisher used in Chapter 3, Understanding Connectors, and make this simple modification (marked as bold):

```
public Publisher(String brokerURL) throws JMSEException {
    factory = new ActiveMQConnectionFactory(brokerURL);
    connection = factory.createConnection();
    connection.start();
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    producer = session.createProducer(null);
    producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
}
```

The consumer modified in this manner has been placed into package `org.apache.activemq.book.ch14.advisory` and we will use it in the rest of this section.

Now let's take a look at our advisory messages example application shown in Example 14.4, “Advisory example”.

Example 14.4. Advisory example

```
public class Advisory {

    protected static String brokerURL = "tcp://localhost:61616";
    protected static transient ConnectionFactory factory;
    protected transient Connection connection;
    protected transient Session session;

    public Advisory() throws Exception {
        factory = new ActiveMQConnectionFactory(brokerURL);
        connection = factory.createConnection();
        connection.start();
        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);❶
    }

    public static void main(String[] args) throws Exception {
        Advisory advisory = new Advisory();
        Session session = advisory.getSession();
        for (String stock : args) {

            ActiveMQDestination destination =
                (ActiveMQDestination)session.createTopic("STOCKS." + stock);

            Destination consumerTopic =
                AdvisorySupport.getConsumerAdvisoryTopic(destination);❷
            System.out.println("Subscribing to advisory " + consumerTopic);
            MessageConsumer consumerAdvisory = session.createConsumer(consumerTopic);
            consumerAdvisory.setMessageListener(new ConsumerAdvisoryListener());

            Destination noConsumerTopic =
                AdvisorySupport.getNoTopicConsumersAdvisoryTopic(destination);❸
            System.out.println("Subscribing to advisory " + noConsumerTopic);
            MessageConsumer noConsumerAdvisory = session.createConsumer(noConsumerTopic);
            noConsumerAdvisory.setMessageListener(new NoConsumerAdvisoryListener());

        }
    }

    public Session getSession() {
        return session;
    }
}
```

- ❶ This code block initializes the JMS connection and the JMS session co-use ~~This advisory block ported from the~~ `AdvisorySupport` helper class to obtain the ??? consumer advisory topic

- ③ This code block uses the `AdvisorySupport` helper class to obtain the no consumer advisory topic

Example 14.4, “Advisory example” provides a demonstration using standard JMS messaging. In the main method, all topics of interest are traversed and consumers are created for the appropriate advisory topics. Note the use of the `AdvisorySupport` class, which you can use as a helper class for obtaining an appropriate advisory destination. In this example, subscriptions were created for the *consumer* and the *no topic consumer* advisory topics. For the topic named `topic://STOCKS.CSCO`, a subscription is created to the advisory topics named `topic://ActiveMQ.Advisory.Consumer.Topic.STOCKS.CSCO` and `topic://ActiveMQ.Advisory.NoConsumer.Topic.STOCKS.CSCO`.

Note

Wildcards can be used when subscribing to advisory topics. For example, by subscribing to `topic://ActiveMQ.Advisory.Consumer.Topic.>` an advisory message is received when a consumer subscribes and unsubscribes to all topics in the namespace recursively.

Now let's take a look at the consumer listeners and how they process advisory messages. First the listener that handles consumer start and stop events will be explored as it is shown in Example 14.5, “Consumer advisory listener”.

Example 14.5. Consumer advisory listener

```
public class ConsumerAdvisoryListener implements MessageListener {  
  
    public void onMessage(Message message) {  
        ActiveMQMessage msg = (ActiveMQMessage) message;  
        DataStructure ds = msg.getDataStructure();  
        if (ds != null) {  
            switch (ds.getDataStructureType()) {  
                case CommandType.CONSUMER_INFO:  
                    ConsumerInfo consumerInfo = (ConsumerInfo) ds;  
                    System.out.println("Consumer '" + consumerInfo.getConsumerId()  
                        + "' subscribed to '" + consumerInfo.getDestination()  
                        + "'");  
                    break;  
                case CommandType.REMOVE_INFO:  
                    RemoveInfo removeInfo = (RemoveInfo) ds;  
            }  
        }  
    }  
}
```

Please post comments or corrections to the [Author Online Forum](#)

```
ConsumerId consumerId = ((ConsumerId) removeInfo.getObjectId());
System.out.println("Consumer '" + consumerId + "' unsubscribed");
break;
default:
    System.out.println("Unknown data structure type");
}
} else {
    System.out.println("No data structure provided");
}
}
```

- ❶ The `CommandTypes.CONSUMER_INFO` type indicates that the consumer created a new subscription
- ❷ The `CommandTypes.REMOVE_INFO` type indicates that the consumer unsubscribed

Every advisory is basically a regular instance of a `ActiveMQMessage` object. In order to get more information from the advisory messages, the appropriate data structure must be used. In this particular case, the message data structure denotes whether the consumer is subscribed or unsubscribed. If we receive a message with the `ConsumerInfo` as data structure it means that it is a new consumer subscription and all the important consumer information is held in the `ConsumerInfo` object. If the data structure is an instance of `RemoveInfo`, it means that this is a consumer that just unsubscribed from the destination. The call to

`removeInfo.getObjectId()` method will identify which consumer it was.

In addition to the data structure, some advisory messages may contain additional properties that can be used to obtain important information that couldn't be included in the data structure. The complete reference of available advisory channels, along with appropriate data structures and properties you can expect on each of them could be found at the following page:
<http://activemq.apache.org/advisory-message.html>

Next is an example of a consumer that handles messages sent to a topic with no consumers. It is shown in Example 14.6, “No consumer advisory listener”.

Example 14.6. No consumer advisory listener

Please post comments or corrections to the [Author Online Forum](#)

```
public class NoConsumerAdvisoryListener implements MessageListener {  
  
    public void onMessage(Message message) {  
        try {  
            System.out.println("Message " + ((ActiveMQMapMessage)message).getContentMap()  
                + " not consumed by any consumer");  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

In this example, the advisory message is the actual message sent to the destination. So the only action to take is to print the message to standard output (stdout).

To run the example from the command line, use the command shown below:

```
$ mvn -e exec:java -Dexec.mainClass=org.apache.activemq.book.ch14.jmx.Advisory \  
-Dexec.args="tcp://localhost:61616 CSCO ORCL"  
  
...  
  
Subscribing to advisory topic://ActiveMQ.Advisory.Consumer.Topic.STOCKS.tcp://localhost:61616  
Subscribing to advisory topic://ActiveMQ.Advisory.NoConsumer.Topic.STOCKS.tcp://localhost:61616  
Subscribing to advisory topic://ActiveMQ.Advisory.Consumer.Topic.STOCKS.CSCO  
Subscribing to advisory topic://ActiveMQ.Advisory.NoConsumer.Topic.STOCKS.CSCO  
Subscribing to advisory topic://ActiveMQ.Advisory.Consumer.Topic.STOCKS.ORCL  
Subscribing to advisory topic://ActiveMQ.Advisory.NoConsumer.Topic.STOCKS.ORCL  
  
...
```

Notice that the example application has subscribed to the appropriate advisory topics, as expected.

In a separate terminal, run the stock portfolio consumer using the following command:

```
$ mvn -e exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Consumer \  
-Dexec.args="tcp://localhost:61616 CSCO ORCL"
```

Upon running this command, the Advisory application will print the following output to the terminal:

```
Consumer 'ID:dejan-bosanacs-macbook-pro.local-64609-1233592052313-0:0:1:1'  
subscribed to 'topic://STOCKS.CSCO'  
Consumer 'ID:dejan-bosanacs-macbook-pro.local-64609-1233592052313-0:0:1:2'
```

Please post comments or corrections to the [Author Online Forum](#)

```
subscribed to 'topic://STOCKS.ORCL'
```

This means that two advisory messages were received, one for each of the two consumers that subscribed.

Now the stock portfolio publisher can be started, the one that was modified above to send non-persistent messages. This application can be started in another terminal using the following command:

```
$ mvn -e exec:java -Dexec.mainClass=org.apache.activemq.book.ch14.advisory.Publisher  
-Dexec.args="tcp://localhost:61616 CSCO ORCL"
```

Notice that the messages are being sent and received as expected. But if the stock portfolio consumer is stopped the Advisory application output will print messages similar to those listed below:

```
...  
Consumer 'ID:dejan-bosanacs-macbook-pro.local-64609-1233592052313-0:0:1:2' unsubscribed  
Consumer 'ID:dejan-bosanacs-macbook-pro.local-64609-1233592052313-0:0:1:1' unsubscribed  
Message {up=false, stock=ORCL, offer=11.817656439151577, price=11.805850588563015}  
not consumed by any consumer  
Message {up=false, stock=ORCL, offer=11.706856077241527, price=11.695160916325204}  
not consumed by any consumer  
Message {up=false, stock=ORCL, offer=11.638181080673165, price=11.62655452614702}  
not consumed by any consumer  
Message {up=true, stock=CSCO, offer=36.51689387339347, price=36.480413459933544}  
not consumed by any consumer  
Message {up=false, stock=ORCL, offer=11.524555643871604, price=11.513042601270335}  
not consumed by any consumer  
Message {up=true, stock=CSCO, offer=36.58309487095556, price=36.54654832263293}  
not consumed by any consumer  
Message {up=false, stock=ORCL, offer=11.515997849703322, price=11.504493356346975}  
not consumed by any consumer  
Message {up=true, stock=ORCL, offer=11.552511335860867, price=11.540970365495372}  
not consumed by any consumer  
...
```

The first two messages indicate that the two consumers unsubscribed. The rest of the messages sent to the stock topics are not being consumed by any consumer, and that's why they are delivered to the Advisory application.

Although it took some time to dissect this simple example, it's a good demonstration of how advisory messages can be used to act on broker events asynchronously, just as is standard procedure in message-oriented applications.

So far we have shown how the ActiveMQ APIs can be used to create applications to monitor and manage broker instances. Luckily, you won't have to do that often as there are many tools provided for this purpose already. In fact, the following section takes a look at some of these very tools.

14.2. Tools

This section will focus on tools for administration and monitoring that are included in the ActiveMQ binary distribution. These tools allow you to easily query the ActiveMQ broker status to diagnose possible problems. Most of these tools use the JMX API to communicate with the broker, so be sure to enable JMX support as explained in 14.1.1.2: “Exposing the JMX MBeans For ActiveMQ”.

14.2.1. Command-Line Tools

You already know how to use the `bin/activemq` script to start the broker. In addition to this script, the `bin/activemq-admin` script can be used to monitor the broker state from the command line. The `activemq-admin` script provides the following functionality:

- *Start and stop* the broker
- *List* available brokers
- *Query* the broker for certain state information
- *Browse* broker destinations

In the following sections, this functionality and the command used to expose it will be explored through the use of examples. For the complete reference and explanation of all available command options, you should refer to <http://activemq.apache.org/activemq-command-line-tools-reference.html>

14.2.1.1. Starting and Stopping the Broker

The standard method for starting ActiveMQ is to use the following command on

Please post comments or corrections to the [Author Online Forum](#)

the command line:

```
$ cd apache-activemq-5.3.0  
$ ./bin/activemq
```

In addition, the following command using the `bin/activemq` script can also be used:

```
$ ./bin/activemq-admin start
```

Using the same script, ActiveMQ can also be used stopped using the following command:

```
$ ./bin/activemq-admin stop
```

The `bin/activemq` script is a nice alternative for stopping the broker. It will attempt to use the JMX API to do this, so be sure to enable JMX support if you plan to use this script. Please note that the `bin/activemq` script connects to the default ActiveMQ JMX URL to send commands, so if you made some modifications to the JMX URL (as we did for the JMX examples above) or the JMX domain, be sure to provide the correct JMX URL and domain to the script using the appropriate parameters. For example, to stop the previously defined broker, that starts the JMX connector on port 2011 and uses the `my-broker` domain, the following command should be used:

```
$ ./bin/activemq-admin stop \  
--jmxurl service:jmx:rmi:///jndi/rmi://localhost:2011/jmxrmi --jmxdomain my-broker
```

This command will connect to ActiveMQ via JMX to send a command to the broker telling it to stop.

The standard ActiveMQ startup script (`bin/activemq`) can be used with rc.d style scripts to start and stop ActiveMQ automatically when an operating system is started or stopped. The following is an example of just such a script that can be used from a service script for most Red Hat Linux-based operating systems. Though with some adaptation, can be used in other LInux distributions:

```
1#!/bin/bash  
#####
```

Please post comments or corrections to the [Author Online Forum](#)

```
# Customize the following variables for your environment
5 #
PROG=activemq
PROG_USER=activemq
DAEMON_HOME=/opt/activemq
DAEMON=$DAEMON_HOME/bin/$PROG
10 LOCKFILE=/var/lock/subsys/$PROG
PIDFILE=/var/run/$PROG.pid
#####
#####

test -x $DAEMON || exit 0
15
# Source function library (Red Hat-specific)
. /etc/rc.d/init.d/functions

RETVAL=0
20
usage () {
    echo "Usage: service $PROG {start|stop|restart|status}"
    RETVAL=1
}
25
start () {
    echo -n $"Starting $PROG: "
    if [ ! -e $LOCKFILE ]; then
        cd $DAEMON_HOME
30    sudo -i -u $PROG_USER $DAEMON > >(logger -t $PROG) 2>&1 &
    else
        echo -n "Lockfile exists"
        false
    fi
35    RETVAL=$?
    if [ $RETVAL -eq 0 ]; then
        logger -t activemq "starting $PROG."
        echo $! > $PIDFILE
        touch $LOCKFILE
    40 else
        logger -t activemq "unable to start $PROG."
    fi
    [ $RETVAL -eq 0 ] && success $$PROG startup" || failure $$PROG startup"
    echo
45 }

stop () {
    echo -n "Shutting down $PROG: "
    killproc -p $PIDFILE -d 20
50    RETVAL=$?
    echo
    [ $RETVAL = 0 ] && rm -f $LOCKFILE
}

55 case "$1" in
      start) start ;;
```

Please post comments or corrections to the [Author Online Forum](#)

```
stop) stop ;;
restart|reload)
      stop
60      start
      ;;
status)
      status $PROG -p $PIDFILE
      RETVAL=$?
65      ;;
*) usage ;;
esac

exit $RETVAL
70
```

Please note that this script will require some customization of variables as noted near the top before it can be used successfully. Upon customizing the necessary variables for your environment, this script can be used to start and stop ActiveMQ automatically when the operating system is cycled.

Now it's time to see how to get information from ActiveMQ using the command line.

14.2.1.2. Listing Available Brokers

In some situations, there may be multiple brokers running in the same JMX context. Using the `bin/activemq` script you can use the `list` command to list all the available brokers as shown in Example 14.7, “activemq-admin list”.

Example 14.7. activemq-admin list

```
$ ./bin/activemq-admin list
Java Runtime: Apple Inc. 1.5.0_16 /System/Library/Frameworks/JavaVM.framework/Versions/1.5.
  Heap sizes: current=1984k free=1709k max=65088k
    JVM args: -Dactivemq.classpath=/tmp/apache-activemq-5.3.0/conf;
-Dactivemq.home=/tmp/apache-activemq-5.3.0 -Dactivemq.base=/tmp/apache-activemq-5.3.0
ACTIVEMQ_HOME: /tmp/apache-activemq-5.3.0
ACTIVEMQ_BASE: /tmp/apache-activemq-5.3.0
Connecting to pid: 99591
BrokerName = localhost
```

As you can see in the example above, we have only one broker in the given context

and its name is localhost.

14.2.1.3. Querying the Broker

Starting, stopping and listing all available brokers are certainly useful features, but what you'll probably want to do more often is query various broker parameters. Let's take a look at the following example demonstrating the query command being used to grab information about destinations:

Example 14.8. activemq-admin query

```
$ ./bin/activemq-admin query -QQueue=*
Java Runtime: Apple Inc. 1.5.0_16 /System/Library/Frameworks/JavaVM.framework/Versions/1.5.0_16
Heap sizes: current=1984k free=1709k max=65088k
JVM args: -Dactivemq.classpath=/tmp/apache-activemq-5.3.0/conf;
-Dactivemq.home=/tmp/apache-activemq-5.3.0 -Dactivemq.base=/tmp/apache-activemq-5.3.0
ACTIVEMQ_HOME: /tmp/apache-activemq-5.3.0
ACTIVEMQ_BASE: /tmp/apache-activemq-5.3.0
Connecting to pid: 99591
DequeueCount = 0
Name = TEST.FOO
MinEnqueueTime = 0
CursorMemoryUsage = 0
MaxAuditDepth = 2048
Destination = TEST.FOO
AverageEnqueueTime = 0.0
InFlightCount = 0
MemoryLimit = 1048576
Type = Queue
EnqueueCount = 0
MaxEnqueueTime = 0
MemoryUsagePortion = 0.0
ProducerCount = 0
UseCache = true
MaxProducersToAudit = 32
CursorFull = false
BrokerName = localhost
ConsumerCount = 0
ProducerFlowControl = true
Subscriptions = []
QueueSize = 0
MaxPageSize = 200
CursorPercentUsage = 0
MemoryPercentUsage = 0
DispatchCount = 0
ExpiredCount = 0
DequeueCount = 0
```

Please post comments or corrections to the [Author Online Forum](#)

```
Name = example.A
MinEnqueueTime = 0
CursorMemoryUsage = 0
MaxAuditDepth = 2048
Destination = example.A
AverageEnqueueTime = 0.0
InFlightCount = 0
MemoryLimit = 1048576
Type = Queue
EnqueueCount = 0
MaxEnqueueTime = 0
MemoryUsagePortion = 0.0
ProducerCount = 0
UseCache = true
MaxProducersToAudit = 32
CursorFull = false
BrokerName = localhost
ConsumerCount = 1
ProducerFlowControl = true
Subscriptions = [org.apache.activemq:BrokerName=localhost,Type=Subscription,persistentMode=destinationType=Queue,destinationName=example.A,clientId=ID_mongoose.local-59784-1255450207356-2_0_1_1]
consumerId=ID_mongoose.local-59784-1255450207356-2_0_1_1]
QueueSize = 0
MaxPageSize = 200
CursorPercentUsage = 0
MemoryPercentUsage = 0
DispatchCount = 0
ExpiredCount = 0
```

In the example above, the `bin/activemq` script was used with the `query` command and a query of `-QQueue=*`. This query will print all the state information about all the queues in the broker instance. In the case of a broker using a default configuration, the only queue that exists is one named `example.A` (from the Camel configuration example in the `conf/activemq.xml` file) and these are its properties.

The command line tools reference page contains the full description of all available query options, so I'd advise you to study it and find out how it can fulfill your requests. If you call the `query` command without any additional parameters, it will print all available broker properties, which can you can use to get quick snapshot of broker's state.

14.2.1.4. Browsing Destinations

Browsing destinations in the broker is another fundamental administrative task.

Please post comments or corrections to the [Author Online Forum](#)

This functionality is also exposed in the bin/activemq-admin script. Below is an example of browsing one of the queues we are using in our job queue example:

Example 14.9. activemq-admin browse

```
$ {ACTIVEMQ_HOME}/bin/activemq-admin browse --amqurl tcp://localhost:61616 JOBS.delete
ACTIVEMQ_HOME: /workspace/apache-activemq-5.2.0
ACTIVEMQ_BASE: /workspace/apache-activemq-5.2.0
JMS_HEADER_FIELD:JMSDestination = JOBS.delete
JMS_HEADER_FIELD:JMSDeliveryMode = persistent
JMS_HEADER_FIELD:JMSMessageID =
ID:dejan-bosanacs-macbook-pro.local-64257-1234789436483-0:0:1:1:2
JMS_BODY_FIELD:JMSObjectClass = java.lang.Integer
JMS_BODY_FIELD:JMSObjectString = 1000001
JMS_HEADER_FIELD:JMSExpiration = 0
JMS_HEADER_FIELD:JMSPriority = 4
JMS_HEADER_FIELD:JMSRedelivered = false
JMS_HEADER_FIELD:JMSTimestamp = 1234789436702

JMS_HEADER_FIELD:JMSDestination = JOBS.delete
JMS_HEADER_FIELD:JMSDeliveryMode = persistent
JMS_HEADER_FIELD:JMSMessageID =
ID:dejan-bosanacs-macbook-pro.local-64257-1234789436483-0:0:1:1:3
JMS_BODY_FIELD:JMSObjectClass = java.lang.Integer
JMS_BODY_FIELD:JMSObjectString = 1000002
JMS_HEADER_FIELD:JMSExpiration = 0
JMS_HEADER_FIELD:JMSPriority = 4
JMS_HEADER_FIELD:JMSRedelivered = false
JMS_HEADER_FIELD:JMSTimestamp = 1234789436706

JMS_HEADER_FIELD:JMSDestination = JOBS.delete
JMS_HEADER_FIELD:JMSDeliveryMode = persistent
JMS_HEADER_FIELD:JMSMessageID =
ID:dejan-bosanacs-macbook-pro.local-64257-1234789436483-0:0:1:1:4
JMS_BODY_FIELD:JMSObjectClass = java.lang.Integer
JMS_BODY_FIELD:JMSObjectString = 1000003
JMS_HEADER_FIELD:JMSExpiration = 0
JMS_HEADER_FIELD:JMSPriority = 4
JMS_HEADER_FIELD:JMSRedelivered = false
JMS_HEADER_FIELD:JMSTimestamp = 1234789436708
```

The `browse` command is different from the previous commands as it does not use JMX, but browses queues using the JMS API. For that reason, you need to provide it with the broker URL using the `--amqurl` switch. The final parameter provided to this command is the name of the queue to be browsed.

As you can see, there are a fair number of monitoring and administration operations that can be achieved from the command line. This functionality and help you to easily check the broker's state and can be very helpful for diagnosing possible problems. But this is not the end of the administrative tools for ActiveMQ. There are still a few more advanced administrative tools and they are explained in following sections.

14.2.2. Command Agent

Sometimes issuing administration commands to the broker from the command line is not easily achievable, mostly in situations when you don't have a shell access to the machines hosting your brokers. In these situations you'll want to administer your broker using some of the existing administrative channels. The *command agent* allows you to issue administration commands to the broker using plain old JMS messages. When the command agent is enabled, it will listen to the ActiveMQ.Agent topic for messages. All commands like `help`, `list` and `query` submitted in form of JMS text messages will be processed by the agent and the result will be posted to the same topic.

In this section we will demonstrate how to configure and use the command agent with the ActiveMQ broker. We will also go one step further and introduce the XMPP transport connector and see how you can use practically any instant messaging client to communicate with the command agent.

Let's begin by looking at the following configuration example:

Example 14.10. Command Agent configuration

```
...
<broker xmlns="http://activemq.apache.org/schema/core" brokerName="localhost"
       dataDirectory="${activemq.base}/data">

    <transportConnectors>
        <transportConnector name="openwire" uri="tcp://localhost:61616"/>
        <transportConnector name="xmpp" uri="xmpp://localhost:61222"/>
    </transportConnectors>

</broker>

<commandAgent xmlns="http://activemq.apache.org/schema/core" brokerUrl="vm://localhost"/>
```

Please post comments or corrections to the [Author Online Forum](#)

...

There are two important details in this configuration fragment. First we have started the XMPP transport connector on port 61222 to expose the broker to clients via XMPP (the *Extensible Messaging and Presence Protocol*). This was achieved by using the appropriate URI scheme, like we do for all supported protocols. XMPP is an open XML-based protocol mainly used for instant messaging and developed by the Jabber project (<http://jabber.org/>). Since it's open and very widespread, there are a lot of *chat* clients that already support this protocol and you can use these clients to communicate with ActiveMQ.

For the purpose of this book, we chose to use the Adium (<http://www.adiumx.com/>), instant messaging client. This client runs on MacOS X and speaks many different protocols including XMPP. Any XMPP client can be used here. The first step is always to provide the details to connect to ActiveMQ to the XMPP client such as server host, port, username and password. Of course, you should connect to your broker on port 61222 since that's where the XMPP transport connector is running and you can use any user and password.

After successfully connecting to the broker you have to join the appropriate chat room which basically means that you'll subscribe to the topic with the same name. In this example we will subscribe to the `ActiveMQ.Agent` topic, so we can use the command agent.

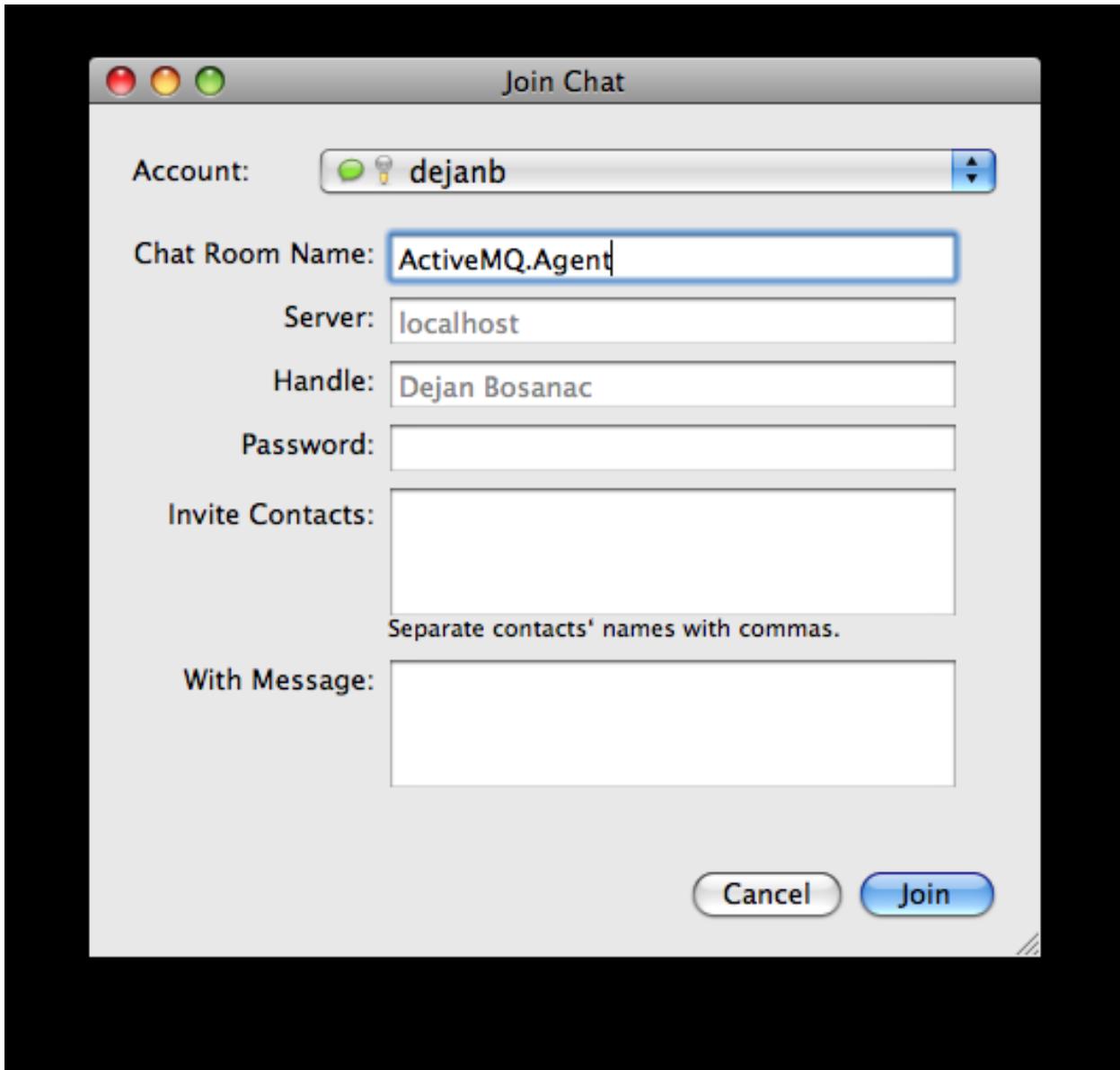


Figure 14.7. XMPP subscribe to agent topic

Typing a message in the chat room sends a message to the topic, so you can type your commands directly into the messaging client. An example of the response for the `help` command is shown in Figure 14.8, “Command Agent help” below:

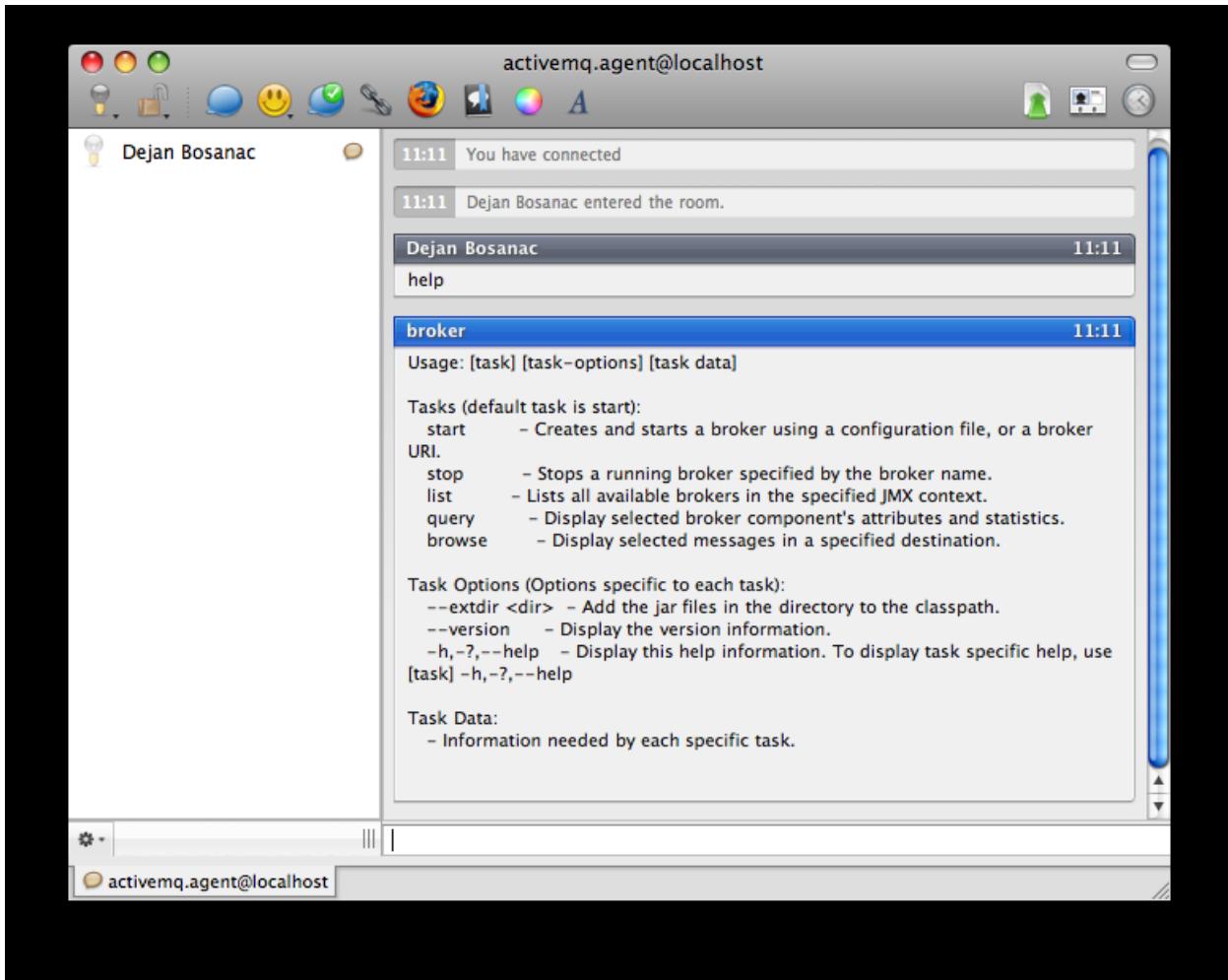


Figure 14.8. Command Agent help

Of course, more complex commands are supported as well. Figure 14.9, “Command Agent query” shows how you can query the topic named TEST.FOO using the `query -QTopic=TEST.FOO` command.

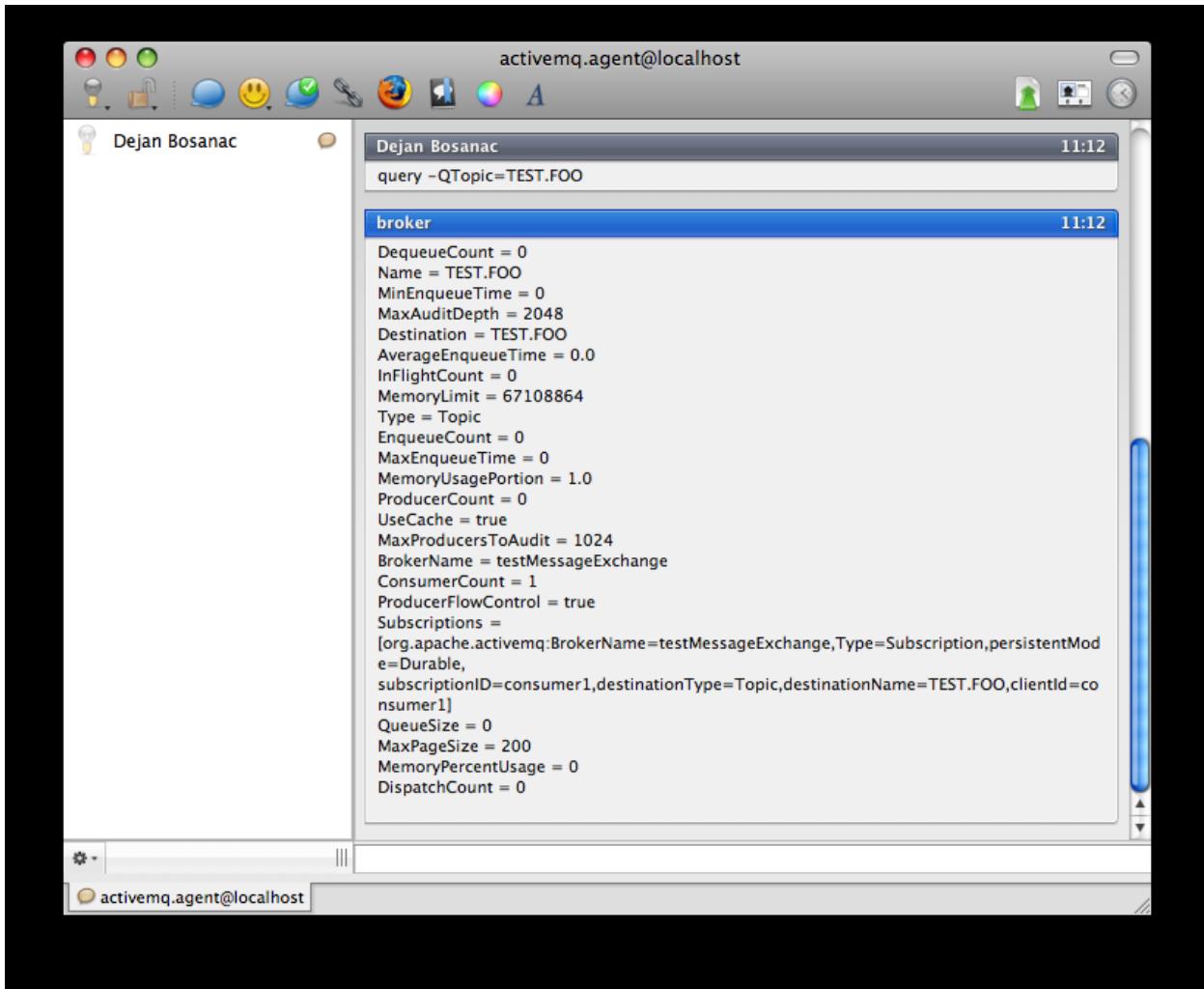


Figure 14.9. Command Agent query

The example shown in this section introduced two very important concepts: the use of XMPP protocol which allows you to use instant messaging applications to interact with the ActiveMQ command agent to administer the broker using standard JMS messages. Together, these two concepts provide a powerful tool for administering remote brokers. Now let's return to some classic administration tools such as JConsole.

14.2.3. JConsole

Please post comments or corrections to the [Author Online Forum](#)

As we said earlier, the JMX API is the standardized API used to by developers to manage and monitor Java applications. But the API is not so useful without a client tool. That's why the Java SE comes with a tool named JConsole, the Java Monitoring and Management Console. JConsole is a client application that allows you browse and call methods of exposed MBeans. Because ActiveMQ requires the Java SE to run, JConsole should be available and is very handy for quickly viewing broker state. In this section, we will cover some of its basic operation with ActiveMQ.

The first thing you should do after starting JConsole (using the `jconsole` command on the command line) is to choose or locate the application you want to monitor (Figure 14.10, “JConsole connect”).

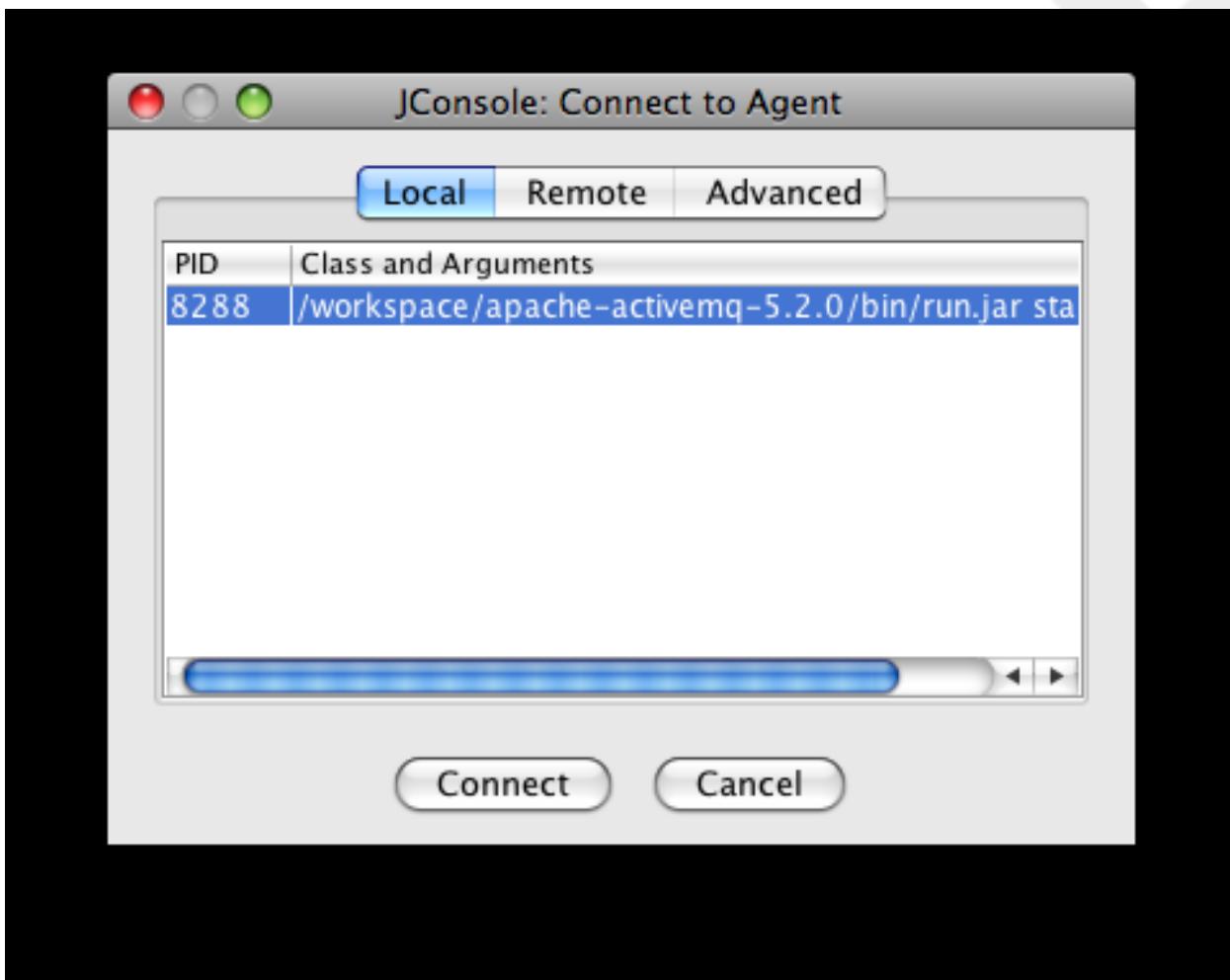
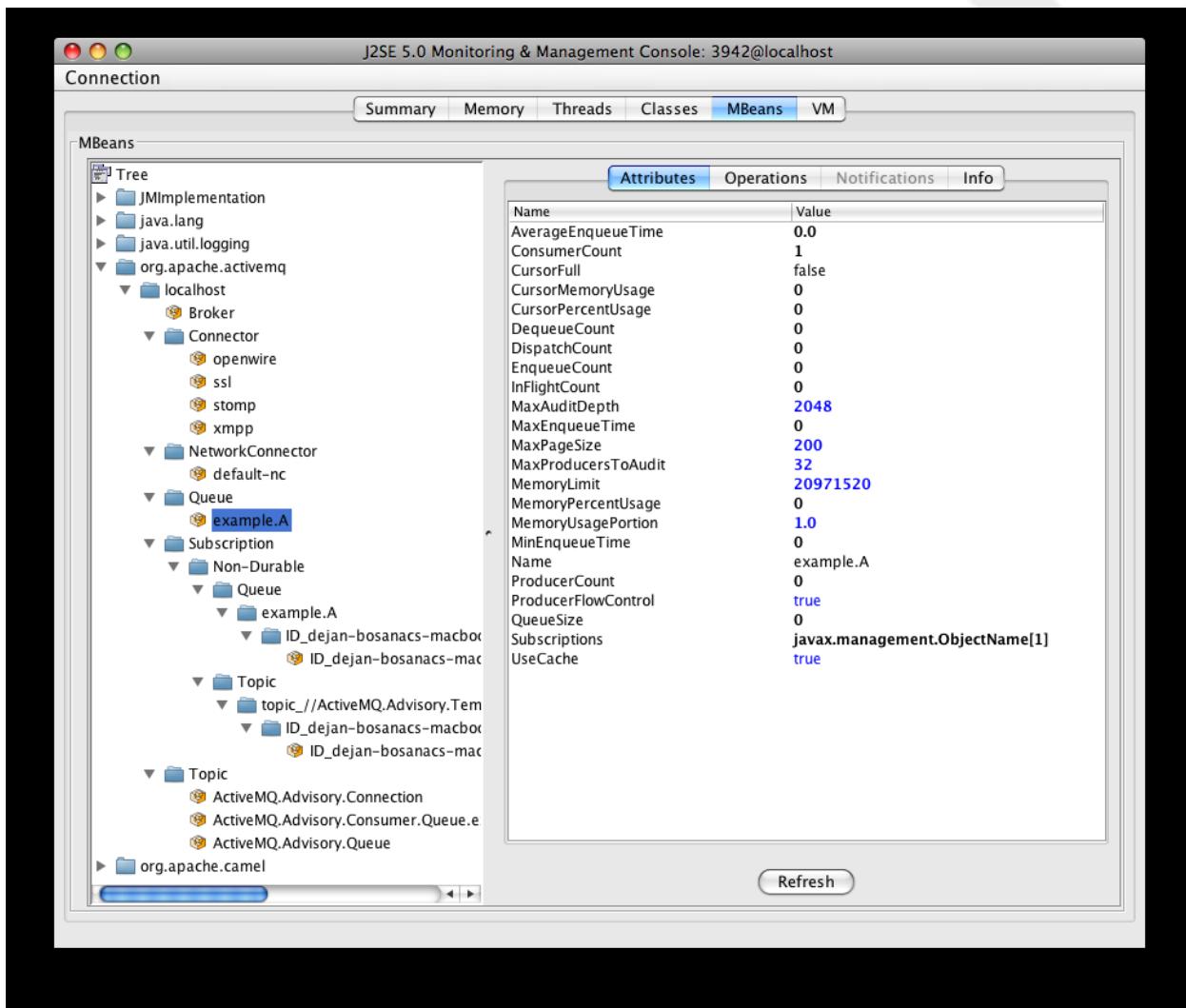


Figure 14.10. JConsole connect

In this figure, we see a local Java process running. This is the case when ActiveMQ and JConsole are started on the same machine. To monitor ActiveMQ on a remote machine, be sure to start a JMX connector from the ActiveMQ configuration file (via the `createConnector` attribute from the `<managementContext>` element). Then you can enter *host* and *port* information (such as `localhost` and `1099` in case of a local broker) in the *Remote* tab, or the full URL (such as `service:jmx:rmi:///jndi/rmi://localhost:1099/jmjaxrmi`) in the *Advanced* tab.

Upon successfully connecting to the local ActiveMQ broker, Figure 14.11, “JConsole Queue View” demonstrates some of what you are able to see.



Please post comments or corrections to the [Author Online Forum](#)

Figure 14.11. JConsole Queue View

As you can see in Figure 14.11, “JConsole Queue View” above, the ActiveMQ broker exposes information about all of its important objects (connectors, destinations, subscriptions, etc.) via JMX. In this particular example, all the attributes for `queue://example.A` can be easily viewed. Such information as queue size, number of producers and consumers can be valuable debugging information for your applications or the broker itself.

Besides peaking at the broker state, you can also use JConsole (and the JMX API) to execute MBean methods. If you go to the *Operations* tab for the destination named `queue://example.A`, you will see all available operations for that particular queue as shown in ???.

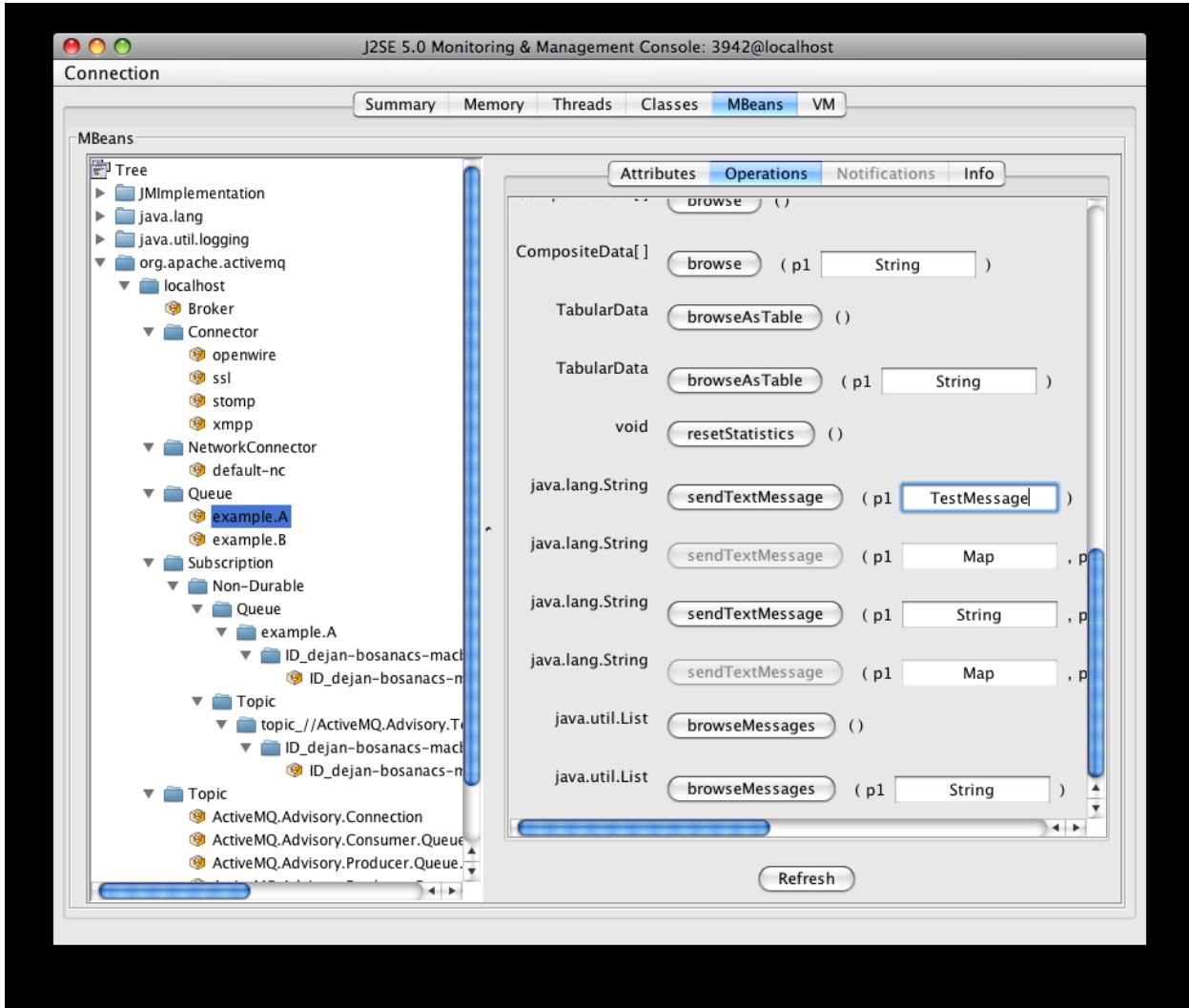


Figure 14.12. JConsole Queue Operations

As you can see in Figure 14.12, “JConsole Queue Operations”, the *sendTextMessage* button allows you to send a simple message to the queue. This can be a very simple test tool to produce messages without writing the code.

Now let's look at another similar tool that is distributed with ActiveMQ.

14.2.4. Web Console

In ???, we saw how an internal web server is used to expose ActiveMQ resources

via REST and Ajax APIs. The same web server is used to host the *web console*, which provides basic management functions via a web browser. Upon starting ActiveMQ using the default configuration, you can visit <http://localhost:8161/admin/> to view the web console.

The web console is far more modest in capabilities compared to JConsole, but it allows you to do some of the most basic management tasks using an user interface adapted to ActiveMQ management. ??? shows a screenshot of the web cosonle viewing a list of queues with some basic information.

The screenshot shows the Apache ActiveMQ Web Console interface. At the top, there's a navigation bar with links for Home, Queues, Topics, Subscribers, and Send. A search bar labeled 'Queue Name' and a 'Create' button are also present. The main content area is titled 'Queues' and displays a table of queue information. The table has columns for Name, Number Of Pending Messages, Number Of Consumers, Messages Sent, Messages Received, Views, and Operations. Three rows are listed:

Name	Number Of Pending Messages	Number Of Consumers	Messages Sent	Messages Received	Views	Operations
JOBS.suspend	86	0	86	0	Browse atom rss	Send To Purge Delete
example.A	0	1	0	0	Browse atom rss	Send To Purge Delete
JOBS.delete	144	0	144	0	Browse atom rss	Send To Purge Delete

On the right side, there's a sidebar with sections for 'Queue Views' (Graph, XML) and 'Useful Links' (Documentation, FAQ, Downloads, Forums). At the bottom of the page, there's a copyright notice: 'Copyright 2005-2007 The Apache Software Foundation.' and a link to the 'printable version'.

Figure 14.13. Web console

For every destination you can also execute certain management operations. For example, you can browse, purge and delete queues or send, copy and move messages to various destinations. Figure 14.14, “Web console” shows the page that displays basic message properties.

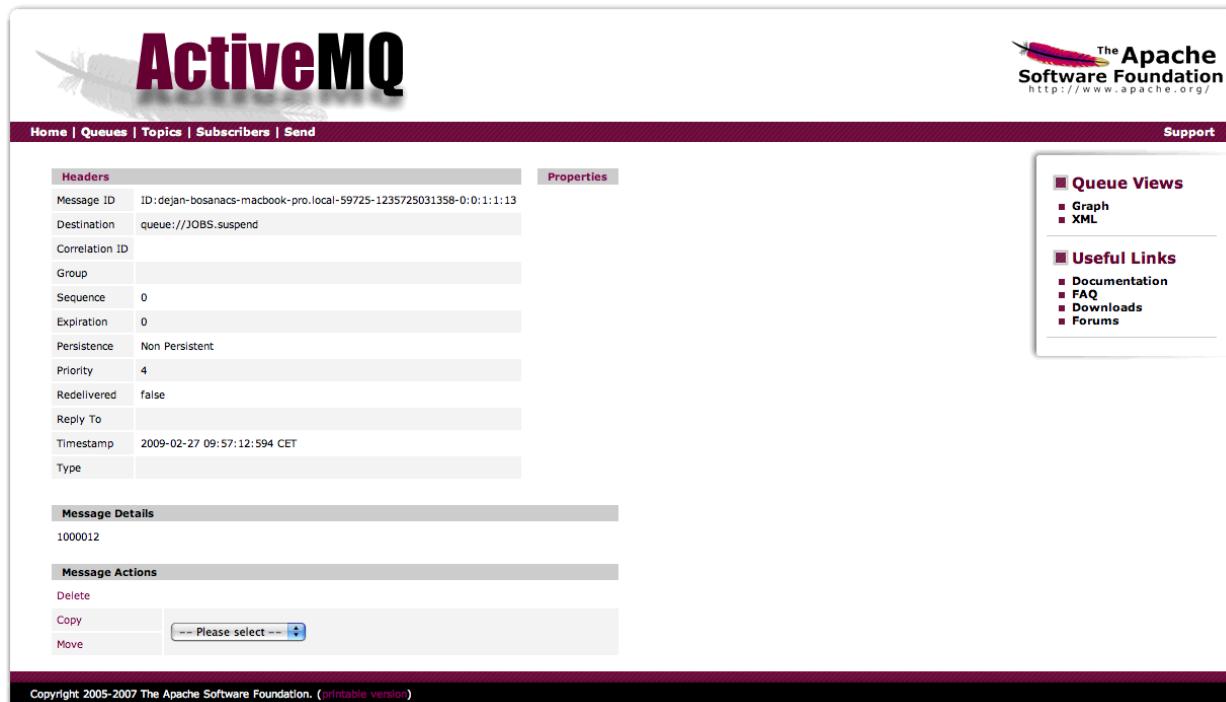


Figure 14.14. Web console

The ActiveMQ web console provides some additional pages for viewing destinations and sending messages. As stated above, this functionality is fairly basic and is meant to be used for development environments, not production environments.

14.3. Logging

So far we have seen how you can monitor ActiveMQ either programmatically or using tools such as JConsole. But there is, of course, one more way you can use to peek at the broker status and that's through its internal logging mechanism. When you experience problems with broker's behavior the first and most common place to begin looking for a potential cause of the problem is the `data/activemq.log` file. In this section you'll learn how you can adjust the logging to suit your needs and how it can help you in detecting potential broker problems.

ActiveMQ uses *Apache Commons Logging API*

Please post comments or corrections to the [Author Online Forum](#)

<http://commons.apache.org/logging/>) for its internal logging purposes. So if you embed ActiveMQ in your Java application, it will fit whatever logging mechanisms you already use. The standalone binary distribution of ActiveMQ uses *Apache Log4J* (<http://logging.apache.org/log4j/>) library as its logging facility.

The ActiveMQ logging configuration can be found in the `conf/log4j.properties` file. By default, it defines two log appenders, one that prints to standard output and other that prints to the `data/activemq.log` file. Example 14.11, “Default logger configuration” shows the standard Log4J logger configuration:

Example 14.11. Default logger configuration

```
log4j.rootLogger=INFO, stdout, out
log4j.logger.org.apache.activemq.spring=WARN
log4j.logger.org.springframework=WARN
log4j.logger.org.apache.xbean.spring=WARN
```

As you can see, by default ActiveMQ will only print messages with log level `INFO` or above, which should be enough for you to monitor its usual behavior. In case you detect a problem with your application and want to turn on more detailed debugging, you should change the level for the root logger to `DEBUG`. Just be aware that the `DEBUG` logging level that ActiveMQ will output considerably more logging information, so you'll probably want to narrow debug messages to a particular Java package. To do this, you should leave the root logger at the `INFO` level and add a line that turns on debug logging on the desired class or even package. For example to turn tracing on the TCP transport you should add the following configuration:

```
log4j.logger.org.apache.activemq.transport.tcp=TRACE
```

After making this change in the `conf/log4j.properties` file and restarting ActiveMQ, you will begin to see the following log output:

TRACE TcpTransport	- TCP consumer thread for tcp://127.0.0.1:49383 start
DEBUG TcpTransport	- Stopping transport tcp://127.0.0.1:49383
TRACE TcpTransport	- TCP consumer thread for tcp://127.0.0.1:49392 start
DEBUG TcpTransport	- Stopping transport tcp://127.0.0.1:49392

In addition to starting/stopping ActiveMQ after changing the logging

configuration, one common question is about how to change the logging configuration at runtime. This is a very reasonable request since, in many cases, you don't want to stop ActiveMQ to change the logging configuration. Luckily, you can use JMX API and JConsole to achieve this. Just make the necessary changes to the `conf/log4j.properties` file and save them. Then open JConsole and select the Broker MBean as shown in Figure 14.15, “Reload Log4J properties”:

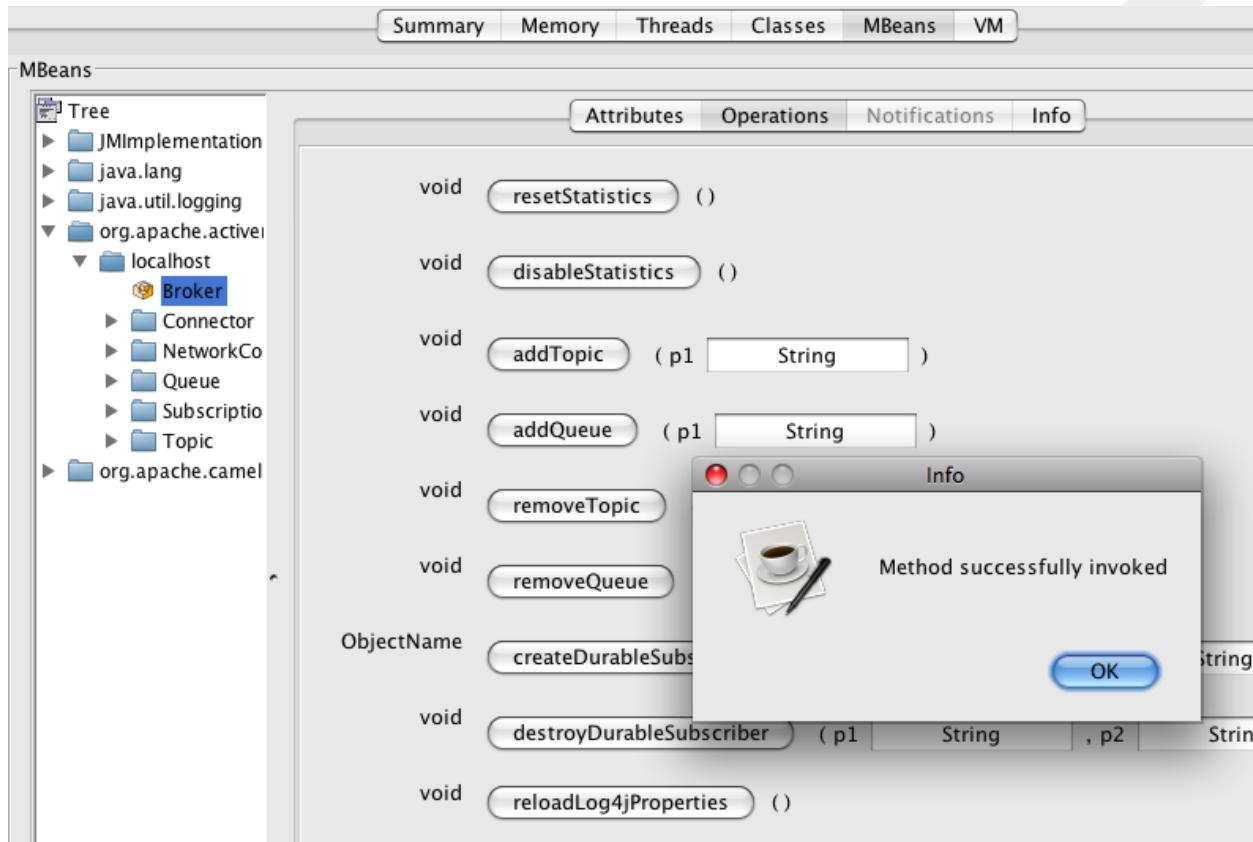


Figure 14.15. Reload Log4J properties

Locate the method `reloadLog4jProperties()` on the Broker MBean's Operations tab. Clicking the button named `reloadLog4jProperties` and the `conf/log4j.properties` file will be reloaded and your changes will be applied. In addition to logging from the broker-side, there is also logging client-side.

14.3.1. Client Logging

Logging on the broker-side is definitely necessary, but how do you debug problems on the client side, in your Java applications? The ActiveMQ Java client APIs use the same logging approach as the broker, so you can use the same style of Log4J configuration file in your client application as well. In this section we will show you a few tips on how you can customize client-side logging and see more information about what's going on inside the client-to-broker communication.

For starters a Log4J configuration file must be made available to the client-side application. Example 14.12, “Client logging” shows an example Log4J configuration file that will be used in this section.

Example 14.12. Client logging

```
log4j.rootLogger=INFO, out, stdout  
  
log4j.logger.org.apache.activemq.spring=WARN  
log4j.logger.org.springframework=WARN  
log4j.logger.org.apache.xbean.spring=WARN  
  
log4j.logger.org.apache.activemq.transport.failover.FailoverTransport=DEBUG  
log4j.logger.org.apache.activemq.transport.TransportLogger=DEBUG
```

As you can see the standard `INFO` level is being used for the root logger. Additional configuration has been added (marked in bold) to monitor the failover transport and TCP communication.

Now, let's run our stock portfolio publisher example, but with some additional properties that will allow us to use logging settings previously defined.

```
$ mvn exec:java \  
-Dlog4j.configuration=file:src/main/resources/org/apache/activemq/book/ch14/log4j.properties  
-Dexec.mainClass=org.apache.activemq.book.ch14.advisory.Publisher \  
-Dexec.args="failover:(tcp://localhost:61616?trace=true) CSCO ORCL"
```

The `log4j.configuration` system property is used to specify the location of the Log4J configuration file. Also note that the `trace` parameter has been set to `true`.

via the transport connection URI. Along with setting the `TransportLogger` level to `DEBUG` will allow all the commands exchanged between the client and the broker to be easily viewed.

Let's say an application is started while the broker is down. What will be seen in the log output are messages like the following:

```
2009-03-19 15:47:56,699 [publisher.main()] DEBUG FailoverTransport
- Reconnect was triggered but transport is not started yet. Wait for start to connect the
transport.
2009-03-19 15:47:56,829 [publisher.main()] DEBUG FailoverTransport
- Started.
2009-03-19 15:47:56,829 [publisher.main()] DEBUG FailoverTransport
- Waking up reconnect task
2009-03-19 15:47:56,830 [ActiveMQ Task ] DEBUG FailoverTransport
- Attempting connect to: tcp://localhost:61616?trace=true
2009-03-19 15:47:56,903 [ActiveMQ Task ] DEBUG FailoverTransport
- Connect fail to: tcp://localhost:61616?trace=true, reason:
java.net.ConnectException: Connection refused
2009-03-19 15:47:56,903 [ActiveMQ Task ] DEBUG FailoverTransport
- Waiting 10 ms before attempting connection.
2009-03-19 15:47:56,913 [ActiveMQ Task ] DEBUG FailoverTransport
- Attempting connect to: tcp://localhost:61616?trace=true
2009-03-19 15:47:56,914 [ActiveMQ Task ] DEBUG FailoverTransport
- Connect fail to: tcp://localhost:61616?trace=true, reason:
java.net.ConnectException: Connection refused
2009-03-19 15:47:56,915 [ActiveMQ Task ] DEBUG FailoverTransport
- Waiting 20 ms before attempting connection.
2009-03-19 15:47:56,935 [ActiveMQ Task ] DEBUG FailoverTransport
- Attempting connect to: tcp://localhost:61616?trace=true
2009-03-19 15:47:56,937 [ActiveMQ Task ] DEBUG FailoverTransport
- Connect fail to: tcp://localhost:61616?trace=true, reason:
java.net.ConnectException: Connection refused
2009-03-19 15:47:56,938 [ActiveMQ Task ] DEBUG FailoverTransport
- Waiting 40 ms before attempting connection.
```

With debug level logging enabled, the failover transport provides a detailed log of its attempts to establish a connection with the broker. This can be extremely helpful in situations where you experience connection problems from a client application.

Once a connection with the broker is established, the TCP transport will start tracing all commands exchanged with the broker to the log. An example of such traces is shown below.

```
2009-03-19 15:48:02,038 [ActiveMQ Task ] DEBUG FailoverTransport
- Waiting 5120 ms before attempting connection.
```

Please post comments or corrections to the [Author Online Forum](#)

```
2009-03-19 15:48:07,158 [ActiveMQ Task ] DEBUG FailoverTransport
- Attempting connect to: tcp://localhost:61616?trace=true
2009-03-19 15:48:07,162 [ActiveMQ Task ] DEBUG Connection:11
- SENDING: WireFormatInfo {...}
2009-03-19 15:48:07,183 [127.0.0.1:61616] DEBUG Connection:11
- RECEIVED: WireFormatInfo { ... }
2009-03-19 15:48:07,186 [ActiveMQ Task ] DEBUG Connection:11
- SENDING: ConnectionControl { ... }
2009-03-19 15:48:07,186 [ActiveMQ Task ] DEBUG FailoverTransport
- Connection established
2009-03-19 15:48:07,187 [ActiveMQ Task ] INFO FailoverTransport
- Successfully connected to tcp://localhost:61616?trace=true
2009-03-19 15:48:07,187 [127.0.0.1:61616] DEBUG Connection:11
- RECEIVED: BrokerInfo { ... }
2009-03-19 15:48:07,189 [publisher.main()] DEBUG Connection:11
- SENDING: ConnectionInfo { ... }
2009-03-19 15:48:07,190 [127.0.0.1:61616] DEBUG Connection:11
- RECEIVED: Response {commandId = 0, responseRequired = false, correlationId = 1}
2009-03-19 15:48:07,203 [publisher.main()] DEBUG Connection:11
- SENDING: ConsumerInfo { ... }
2009-03-19 15:48:07,206 [127.0.0.1:61616] DEBUG Connection:11
- RECEIVED: Response { ... }
2009-03-19 15:48:07,232 [publisher.main()] DEBUG Connection:11
- SENDING: SessionInfo { ... }
2009-03-19 15:48:07,239 [publisher.main()] DEBUG Connection:11
- SENDING: ProducerInfo { ... }
Sending: {offer=51.726420585933745, price=51.67474584009366, up=false, stock=CSCO}
on destination: topic://STOCKS.CSCO
2009-03-19 15:48:07,266 [publisher.main()] DEBUG Connection:11
- SENDING: ActiveMQMapMessage { ... }
2009-03-19 15:48:07,294 [127.0.0.1:61616] DEBUG Connection:11
- RECEIVED: Response { ... }
Sending: {offer=94.03931872048393, price=93.94537334713681, up=false, stock=ORCL}
on destination: topic://STOCKS.ORCL
```

For the purpose of readability, some details of specific commands have been left out except for one log message which is marked bold. These traces provide a full peek into the client-broker communication which can help to narrow application connection problems further.

This simple example shows that with just a few minor configuration changes, many more logging details can be viewed. But beyond standard Log4J style logging, ActiveMQ also provides a special logger for the broker.

14.3.2. Logging Interceptor

The previous section demonstrated how the client side can be monitored through

the use of standard Log4J logging. Well, similar functionality is available on the broker side using a *logging interceptor*. ActiveMQ plugins were introduced in ??? where you saw how they can be used to authenticate client applications and authorize access to broker resources. The logging interceptor is just a simple plugin, which uses the broker's internal logging mechanism to log messages coming from and going to the broker. To install this plugin, just add the `<loggingBrokerPlugin/>` element to the list of your plugins in the `conf/activemq.xml` configuration file. Below is an example of this:

```
<plugins>
  <loggingBrokerPlugin/>
</plugins>
```

After the restarting the broker, you will see message exchanges being logged. Below is an example of the logging that is produced by the logging interceptor:

INFO Send	- Sending: ActiveMQMapMessage { ... }
INFO Send	- Sending: ActiveMQMapMessage { ... }
INFO Send	- Sending: ActiveMQMapMessage { ... }

Of course, message properties are again intentionally left out, for the sake of clarity. This plugin along with other logging techniques could help you gain a much better perspective of the broker activities while building message-oriented systems.

14.4. Summary

With this discussion, we came to the end of the topics planed for this book. We hope you enjoyed reading it and that it helped you get your ActiveMQ (and messaging in general) knowledge to the next level. This should be by no means the end of your journey into ActiveMQ, since it's a project that is continuously developed and improved. So, be sure to stay up to date with its development and new features.