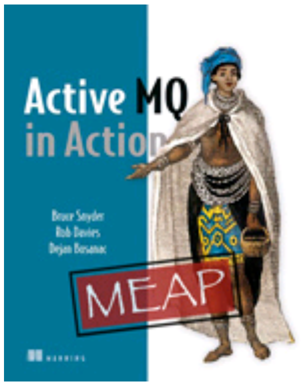
MANNING PUBLICATIONS

Introduction to Apache ActiveMQ

Green Paper from



ActiveMQ in Action
EARLY ACCESS EDITION

Bruce Snyder, Dejan Bosanac, and Rob Davies
MEAP Release: August 2008
Softbound print: August 2010 | 375 pages
ISBN: 1933988940

This green paper is taken from the book ActiveMQ in Action from Manning Publications. The authors discuss ActiveMQ, 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. For the table of contents, the author forum, and other resources, go to http://manning.com/snyder/.

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 among applications. However, it's only within the last 10 years 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 green paper will introduce you to 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 that use ActiveMQ in a strategic manner. 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 dozens of additional features and value on top of this spec. 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 paper will walk you through the task of becoming familiar with ActiveMQ in a simple manner. You will gain not only a high level understanding of the ActiveMQ feature set but you will also be taken through a discussion of why and where to use ActiveMQ in your application development. Then, you will be prepared enough to install and begin using ActiveMQ.

ActiveMQ Features

ActiveMQ provides an abundance of features created through hundreds of man-years of effort. Following is a high-level list of many of those 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. 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. 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, multicast, SSL, Stomp, TCP, UDP, XMPP, and others. 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 important.
- **Pluggable persistence and security**—ActiveMQ provides multiple flavors of persistence and you can choose your favorite. Also, security in ActiveMQ can be completely customized for the type of authentication and authorization that's best for your needs.
- **Client APIs**—ActiveMQ also provides a client API for many languages besides just Java including C/C++, .NET, Perl, PHP, Python, Ruby, and others. 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.
- **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.
- **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.
- **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.

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.

Using ActiveMQ: why and when?

Back around 2003, a group of open-source developers got together to form Apache Geronimo. In doing, it was discovered that there was not a good message broker available that utilized a BSD style of license. Geronimo needed a JMS implementation for reasons of J2EE compatibility so a few of the developers started discussing the possibilities. Possessing vast experience with many commercial MOMs and even having built a few MOMs themselves previously, these developers set out to create the next great open-source message broker. Additional inspiration for ActiveMQ came from the fact that most of the MOMs in the market were commercial, closed source, and very costly to buy and support. The commercial MOMs were certainly popular with businesses, but some businesses could not afford the steep costs required by them. This helped to increase the motivation to build an open-source alternative even further. There was clearly a market available for an open source MOM using an Apache License. What evolved over time is Apache ActiveMQ.

ActiveMQ was intended to be used, as the JMS spec intended, for remote communications between distributed applications. To better understand what this means, the best thing to do is totake a look at a few of the ideas behind distributed application design; specifically, communications.

Loose coupling and ActiveMQ

ActiveMQ provides the benefits of loose coupling for application architecture. Loose coupling is commonly introduced into an architecture to mitigate the classic tight coupling of Remote Procedure Calls (RPC). Such a loosely coupled design is considered to be asynchronous; where the calls from either application have no bearing on one another there is no interdependence or timing requirements. The applications can rely upon ActiveMQ's ability to guarantee message delivery. Because of this, it is often said that the applications sending messages just fire and forget; that is, they send the message to ActiveMQ and are not concerned with how or when the message is delivered. At the same time, the consuming applications have no concerns with the message's original location or how the message was sent to ActiveMQ. This is an especially powerful benefit in heterogeneous environments allowing clients to be written using different languages and, possibly, different wire protocols. ActiveMQ acts as the middleman allowing heterogeneous integration and interaction in an asynchronous manner. More on this in the next section.

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 coupling is to consider the effect of changes on any application in the system, such as the implications across the other applications in the architecture as new features are added. 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 more loosely coupled. The overall lesson here is that tightly coupled applications are more difficult to maintain compared with loosely coupled applications. Said another way, loosely coupled applications can easily deal with unforeseen changes.

Technologies such as COM, CORBA, DCE and EJB, which use techniques called Remote Procedure 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 depicts this concept.

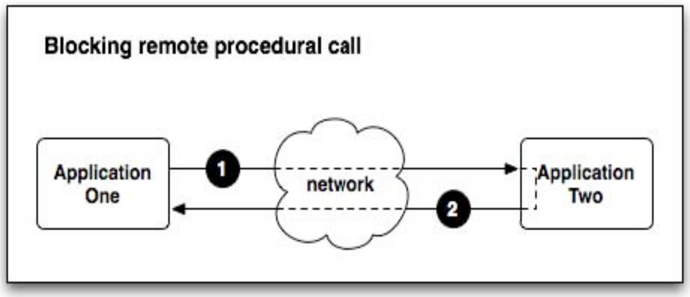


Figure 1 Two tightly-coupled applications using remote procedure calls to communicate

The caller (application one) in figure 1 is blocked until the callee (application two) returns control. Many system architectures use RPC and are very successful. However, there are many disadvantages to such a tightly coupled design; most notable is the higher amount of maintenance that is required because even small changes ripple throughout the system architecture. Correct timing is a necessity between the two applications. Both applications must be available at the same time for the request from application one to reach application two (label #1) and for the response to travel from application two to application one (label #2). Such timing requirements can be very cumbersome. Compare such a tightly coupled design with a design where two applications are completely unaware of one another such as that depicted in figure 2.

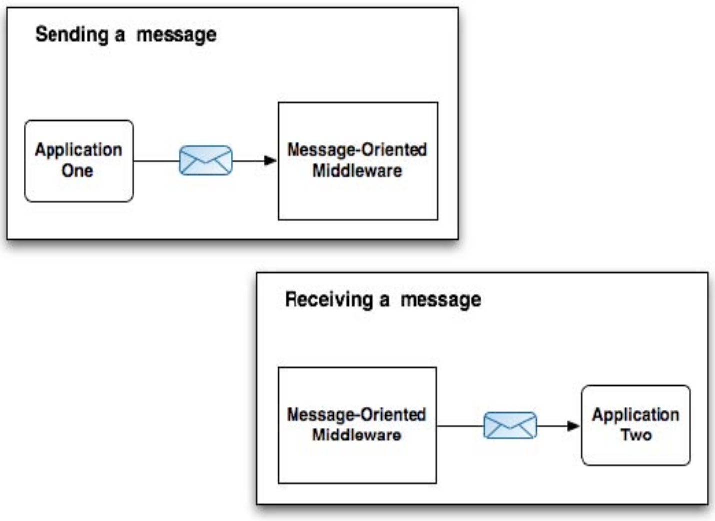


Figure 2 Two loosely-coupled applications using JMS messaging to communicate

Application one in figure 2 sends a message to the MOM in a one-way fashion. Then, possibly sometime later, application two receives a message from the MOM in a one-way fashion. Neither application has any knowledge that the other 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 big advantages over tightly coupled architectures when considering distributed application design. This is where ActiveMQ enters the picture.

Consider the changes necessary when an application must move to a new location. This can happen when new hardware is necessary or machines simply need to be moved. With a tightly coupled system design, such movement is difficult because all segments of the application must experience an outage. With an application designed using loose coupling, different segments of the system can be moved independently of one another.

Consider a scenario where there are multiple instances of application one and multiple instances of application two, where each instance resides on a different machine. ActiveMQ is installed on still another machine independently of either application one or application two. In this scenario, any one of the application one or application two instances can be moved around without affecting one another. In fact, multiple instances of ActiveMQ could be used in what's known as a *network of brokers* configuration. This would allow the ActiveMQ instances to be moved around without affecting either application one or application two. This means that any segment of this architecture can be taken down for maintenance at any time without taking down the entire system.

So ActiveMQ provides an incredible amount of flexibility in application architecture and for the concepts surrounding loose coupling to become a reality. But when should ActiveMQ be used to introduce these benefits?

When to use ActiveMQ

There are many occasions where ActiveMQ and asynchronous messaging can have a meaningful impact on a system architecture. Here are just a few example scenarios:

- **Heterogeneous application integration**—The ActiveMQ broker is written using the Java language so, naturally, a Java client API is provided. But ActiveMQ also provides clients for C/C++, .NET, Perl, PHP, Python, Ruby, and a few other languages. This is a huge advantage when considering how you might integrate applications written in different languages on different platforms. In cases such as this, the various client APIs make it possible to send and receive messages via ActiveMQ no matter what language is used. In addition to the cross-language capabilities provided by ActiveMQ, the ability to integrate such applications without the use of remote procedure calls (RPC) is definitely a big benefit because messaging truly helps to decouple the applications.

- **As a replacement for RPC**—Applications using RPC-style synchronous calls are very widespread. Consider that the vast majority of client-server applications use RPC including ATMs, most web applications, credit card systems, point-of-sale systems, and many others. Even though many of these systems are very successful, conversion to the use of asynchronous messaging can bring about many benefits without giving up the guarantee of a response. Systems that rely upon synchronous requests typically have a limited ability to scale because, eventually, requests will begin to back up, thereby slowing the whole system. Instead of experiencing this type of a slowdown, using asynchronous messaging, additional message receivers can be easily added so that messages are consumed concurrently and, therefore, handled faster. This, of course, assumes that your applications can be decoupled.
- **To loosen the coupling between applications**—As already discussed, tightly coupled architectures can be problematic for many reasons, especially if they are distributed. Loosely coupled architectures, on the other hand, exhibit fewer dependencies, making them better at handling unforseen changes. Not only will a change to one component in the system not ripple across the entire system but component interaction is also dramatically simplified. Instead of using a synchronous scheme for component interaction (where one method calls another and the caller waits for a response from the callee), components utilize asynchronous communications (where they simply send a message without waiting for a response, also known as fire and forget). Such loose coupling throughout a system can lead to what is known as an event-driven architecture (EDA).
- **As the backbone of an event-driven architecture**—The decoupled, asynchronous style of architecture described in the previous point allows the software itself to scale much further (known as horizontal scalability) instead of only relying upon the ability of the hardware to scale (known as vertical scalability). Consider an incredibly high-traffic, e-commerce site such as Amazon. When a user makes a purchase on Amazon, that order must travel through many separate stages, including order placement, invoice creation, payment processing, order fulfillment, shipping, and probably more. However, when a user actually places an order, the user is immediately taken to a page stating, "Thanks for your order". Not only that, but, without delay, the user also receives an email stating that the order was received. The order placement process employed by Amazon is a good example of the first stage in a much larger, asynchronous process. Each stage of the order is handled discretely by a separate service. When the user places the order, there is a synchronous call to submit the order, but the entire order process does not take place behind a synchronous call via the web browser. Instead, the order is accepted and acknowledged immediately. The rest of the steps in the process are handled asynchronously. If a problem prevents the process from proceeding, the user is notified. Such asynchronous processes afford massive scalability.
- **To improve application scalability**—Many applications utilize an event-driven architecture in order to provide massive scalability, including such domains as e-commerce, government, manufacturing, and online gaming, just to name a few. By separating an application along the business domain lines using asynchronous messaging, many other possibilities begin to emerge. Consider the ability to design an application using a service for a specific task. This is the backbone of service-oriented architecture (SOA). Each service fulfills a discrete function and only that function. Applications are then built through the composition of these services, and the communication among services is achieved using asynchronous messaging. This style of application design makes it possible to introduce such concepts as complex event processing (CEP). Using CEP, the interactions among the components in a system are tracked for further analysis. Such possibilities are truly endless when you consider that asynchronous messaging is simply adding a level of indirection between components in a system. Now that you've been offered some examples of where to use ActiveMQ, it's time to install ActiveMQ and begin using it.

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 capable of accepting connections and sending messages. ActiveMQ comes with some simple examples that will help you with this task, but, first, we need to install Java and download ActiveMQ.

In this section, you will download and install the Java SE, download and install ActiveMQ, examine the ActiveMQ directory, and start up ActiveMQ for the first time.

Downloading and installing 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 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 a similar output, then you'll need to rectify this situation before moving on to the next section.

Downloading 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're ready to take a peek at the ActiveMQ directory.

Examining the ActiveMQ directory

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

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

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/Home
   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-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
Loading message broker from: xbean:activemq.xml
   INFO | Using Persistence Adapter:
org.apache.activemq.store.kahadb.KahaDBPersistenceAdapter@5bc68c
   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) 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
```

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. 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.

The best way to begin sending and receiving messages is by using some of the examples that come with ActiveMQ. The next section walks you through this in a step-by-step manner.

Running your first examples with ActiveMQ

The previous section walked you through the startup of ActiveMQ in one terminal. For this verification, you should open a couple more terminals to run the ActiveMQ examples. In the second terminal, move into the example directory and look at its contents:

```
[apache-activemq-5.3.0]$ cd ./example/
bsnyder@mongoose [example]$ ls -1
build.xml
conf
perfharness
ruby
src
transactions
```

The example directory contains a few different items. Below is a quick description of each item in that directory:

- **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.
- **perfharness**—The perfharness 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:
     [mkdir] Created dir:
/Users/bsnyder/amq/apache-activemq-5.3.0/example/target/classes

compile:
     [javac] Compiling 10 source files to
/Users/bsnyder/amq/apache-activemq-5.3.0/example/target/classes
     [copy] Copying 2 files to
/Users/bsnyder/amq/apache-activemq-5.3.0/example/target/classes

consumer:
     [echo] Running consumer against server at $url = tcp://localhost:61616
for subject $subject = TEST.FOO
     [java] Connecting to URL: tcp://localhost:61616
     [java] Consuming queue: TEST.FOO
     [java] Using a non-durable subscription
     [java] We are about to wait until we consume: 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 = TEST.FOO
     [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: 1213991298653 description: Number of messages exchanged }
     [java] messageRateTime{ count: 0 maxTime: 0 minTime: 0
totalTime: 0 averageTime: 0.0 averageTimeExMinMax: 0.0
averagePerSecond: 0.0 averagePerSecondExMinMax: 0.0 unit: millis
startTime: 1213991298654 lastSampleTime: 1213991298654 description:
Time taken to process a message (thoughtput rate)
}
     [java] pendingMessageCount{ count: 0 unit: count startTime:
1213991298654 lastSampleTime: 1213991298654 description: Number of
pending messages }
     [java] expiredMessageCount{ count: 0 unit: count startTime:
1213991298654 lastSampleTime: 1213991298654 description: Number of
expired messages }
```

```
     [java] messageWaitTime{ count: 0 maxTime: 0 minTime: 0
totalTime: 0 averageTime: 0.0 averageTimeExMinMax: 0.0
averagePerSecond: 0.0 averagePerSecondExMinMax: 0.0 unit: millis
startTime: 1213991298654 lastSampleTime: 1213991298654 description:
Time spent by a message before being delivered }
     [java] durableSubscriptionCount{ count: 0 unit: count
startTime: 1213991298654 lastSampleTime: 1213991298654 description:
The number of durable subscriptions }

     [java] producers {
     [java] producer queue://TEST.FOO {
     [java] messageCount{ count: 0 unit: count startTime:
1213991298662 lastSampleTime: 1213991298662 description: Number of
messages processed }
     [java] messageRateTime{ count: 0 maxTime: 0 minTime: 0
totalTime: 0 averageTime: 0.0 averageTimeExMinMax: 0.0
averagePerSecond: 0.0 averagePerSecondExMinMax: 0.0 unit:
millis startTime: 1213991298662 lastSampleTime: 1213991298662
description: Time taken to process a message (thoughtput rate)
}
     [java] pendingMessageCount{ count: 0 unit: count
startTime: 1213991298662 lastSampleTime: 1213991298662 description:
Number of pending messages }
     [java] messageRateTime{ count: 0 maxTime: 0 minTime: 0
totalTime: 0 averageTime: 0.0 averageTimeExMinMax: 0.0
averagePerSecond: 0.0 averagePerSecondExMinMax: 0.0 unit: millis
startTime:
1213991298662 lastSampleTime: 1213991298662 description: Time taken
to process a message (thoughtput rate)
}
     [java] expiredMessageCount{ count: 0 unit: count
startTime: 1213991298662 lastSampleTime:
1213991298662 description: Number of expired messages }
     [java] messageWaitTime{ count: 0 maxTime: 0 minTime: 0
totalTime: 0 averageTime: 0.0 averageTimeExMinMax: 0.0
averagePerSecond: 0.0 averagePerSecondExMinMax: 0.0 unit: millis startTime:
1213991298662 lastSampleTime: 1213991298662 description: Time
spent by a message before being delivered }
     [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 connector (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 in terminal two:

```
     [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.

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 ActiveMQ mailing lists is the best way to find help. 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.

Summary

ActiveMQ is clearly a very versatile, easy to use messaging middleware. You learned about some of the ActiveMQ features that will be covered throughout this book and about some scenarios where ActiveMQ can be applied. The scenarios introduced in this paper are real-world use cases that are deployed in businesses throughout the world. The JMS spec was designed for use in business applications with these scenarios in mind.