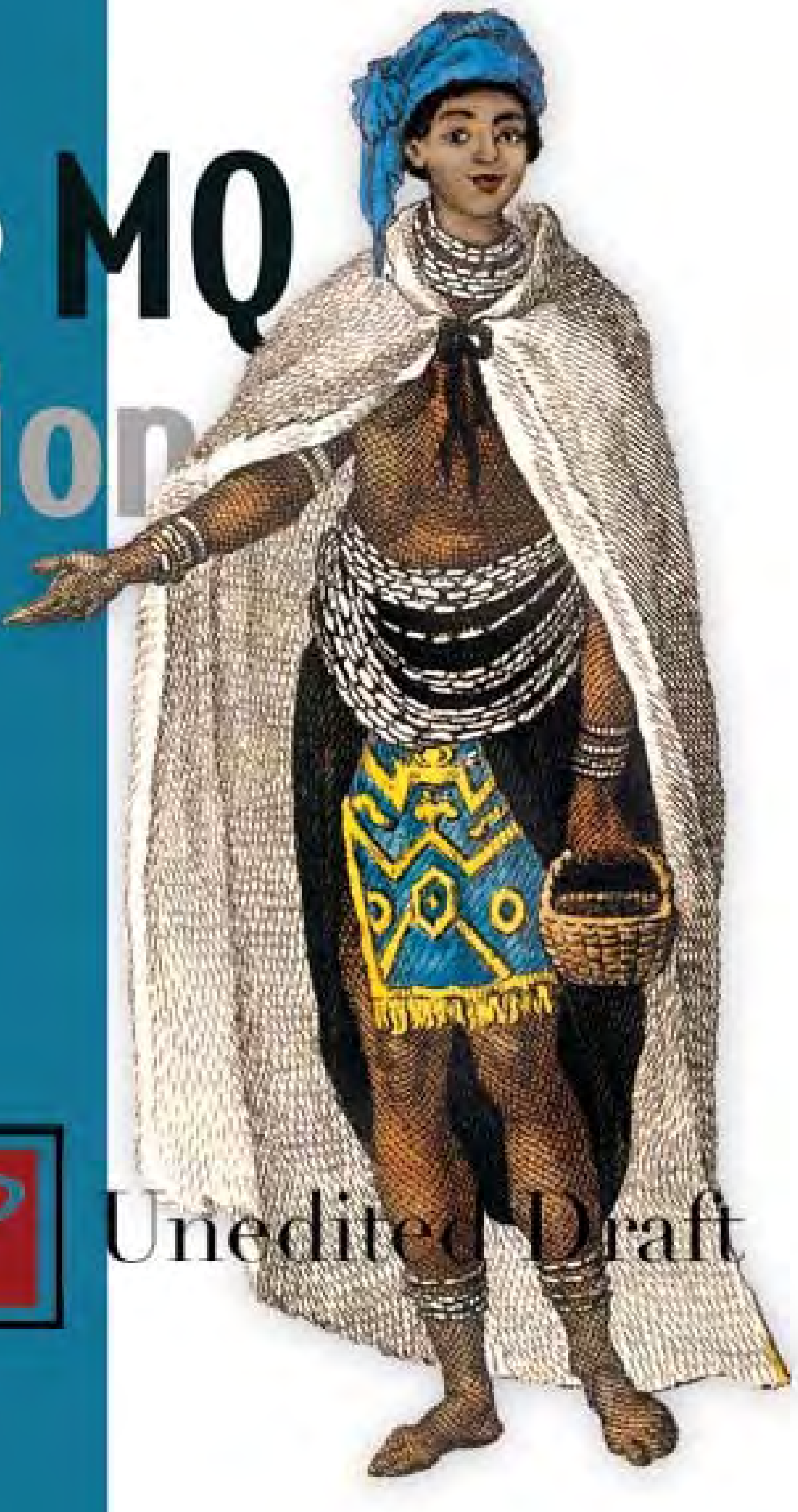
ActiveMQ in Action

Bruce Snyder
Rob Davies
Dejan Bosanac

MEAP Unedited Draft



MANNING

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

MEAP Edition
Manning Early Access Program

Copyright 2009 Manning Publications

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Contents

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Part I. An Introduction to Messaging and ActiveMQ

[Intro goes here]

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Chapter 1. Understanding Message-Oriented Middleware and JMS

1.1. Introduction

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)

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

- Some examples of using the JMS API

1.2. 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 1.1 shows a diagram demonstrating how terminals connect to a mainframe.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

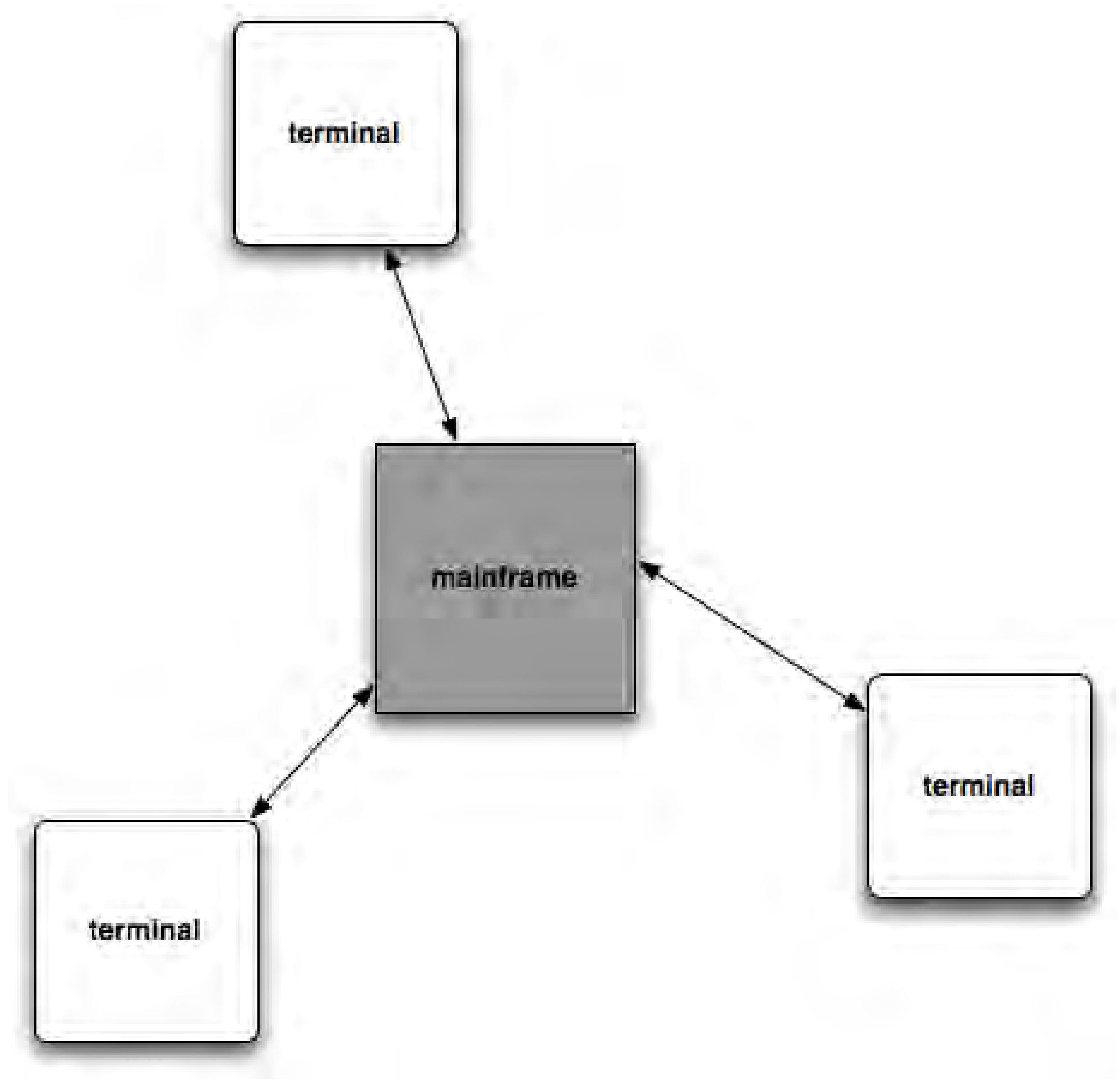


Figure 1.1. Terminals connecting to a mainframe

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

even more important because applications needing access to the mainframe were being developed to run on PCs and workstations. Figure 1.2 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.

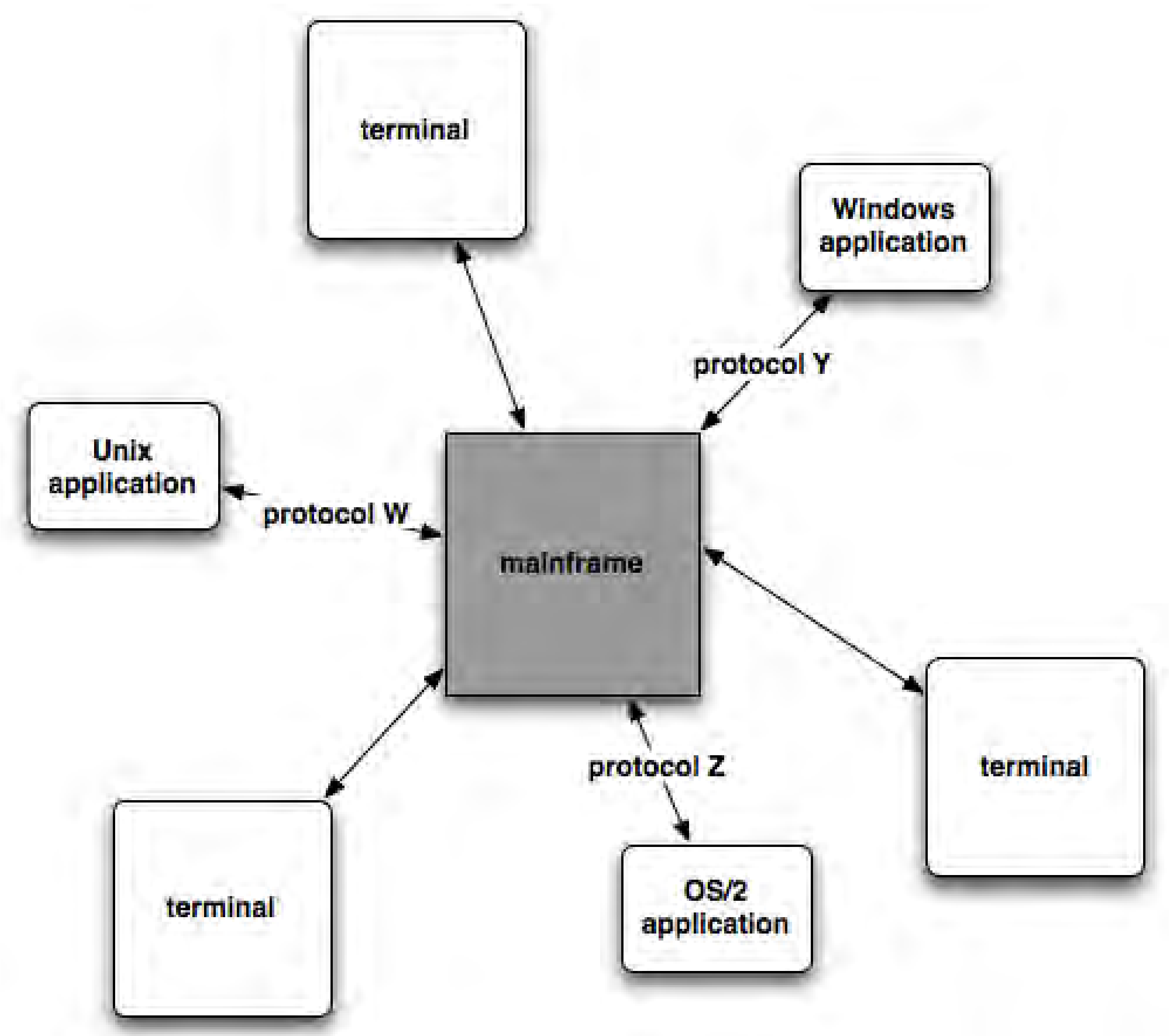


Figure 1.2. Terminals and applications connecting to a mainframe

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

1.3. What is Message Oriented Middleware?

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 1.3 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.

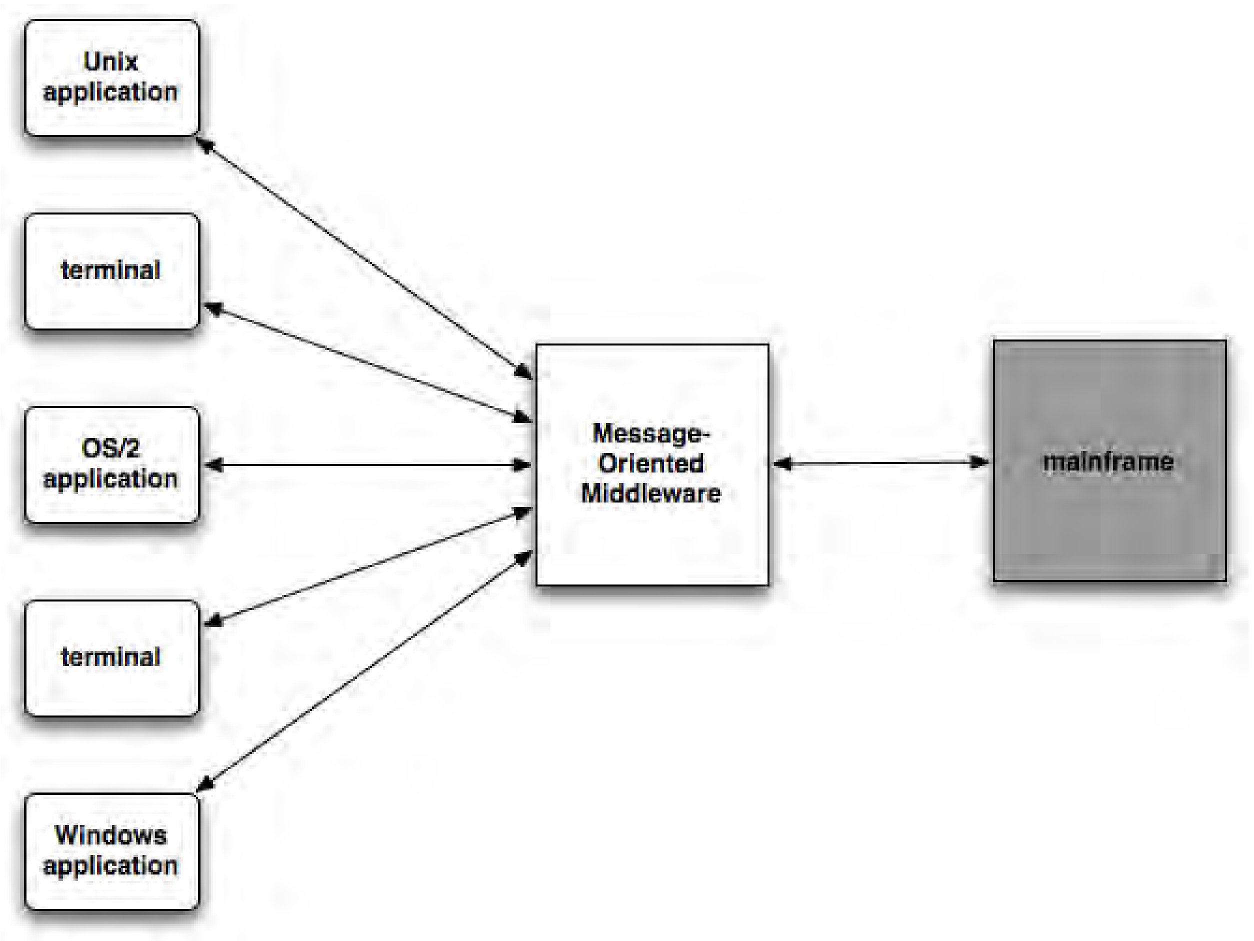


Figure 1.3. Introducing message-oriented middleware

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

1.4. What is the Java Message Service?

The Java Message Service (JMS) moved beyond vender-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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 1.4 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.

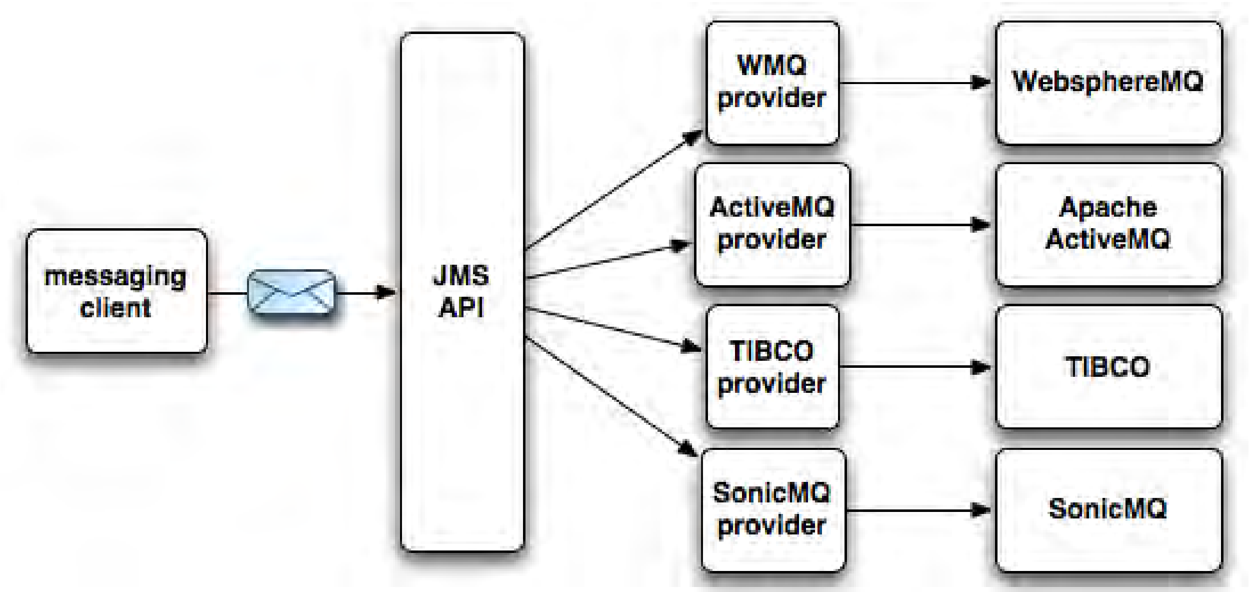


Figure 1.4. JMS allows a single client to easily connect to many JMS providers

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

1.4.1. 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.

1.4.1.1. 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.

1.4.1.1.1. 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 by for individual messages by using the `MessageProducer.send()` method. The `MessageProducer` interface is shown below:

```
public interface MessageProducer {
    void setDisableMessageID(boolean value) throws JMSException;

    boolean getDisableMessageID() throws JMSException;

    void setDisableMessageTimestamp(boolean value) throws JMSException;

```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
    boolean getDisableMessageTimestamp() throws JMSException;

    void setDeliveryMode(int deliveryMode) throws JMSException;

    int getDeliveryMode() throws JMSException;

    void setPriority(int defaultPriority) throws JMSException;

    int getPriority() throws JMSException;

    void setTimeToLive(long timeToLive) throws JMSException;

    long getTimeToLive() throws JMSException;

    Destination getDestination() throws JMSException;

    void close() throws JMSException;

    void send(Message message) throws JMSException;

    void send(Message message, int deliveryMode, int priority,
              long timeToLive)
        throws JMSException;

    void send(Destination destination, Message message)
        throws JMSException;

    void send(
        Destination destination,
        Message message,
        int deliveryMode,
        int priority,
        long timeToLive)
        throws JMSException;
}
```

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 1.4.3.

1.4.1.1.2. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 JMSException;

    MessageListener getMessageListener() throws JMSException;

    void setMessageListener(MessageListener listener) throws JMSException;

    Message receive() throws JMSException;

    Message receive(long timeout) throws JMSException;

    Message receiveNoWait() throws JMSException;

    void close() throws JMSException;
}
```

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

1.4.1.2. 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.

1.4.2. 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).

1.4.3. Anatomy of a JMS Message

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 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 1.5, 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.

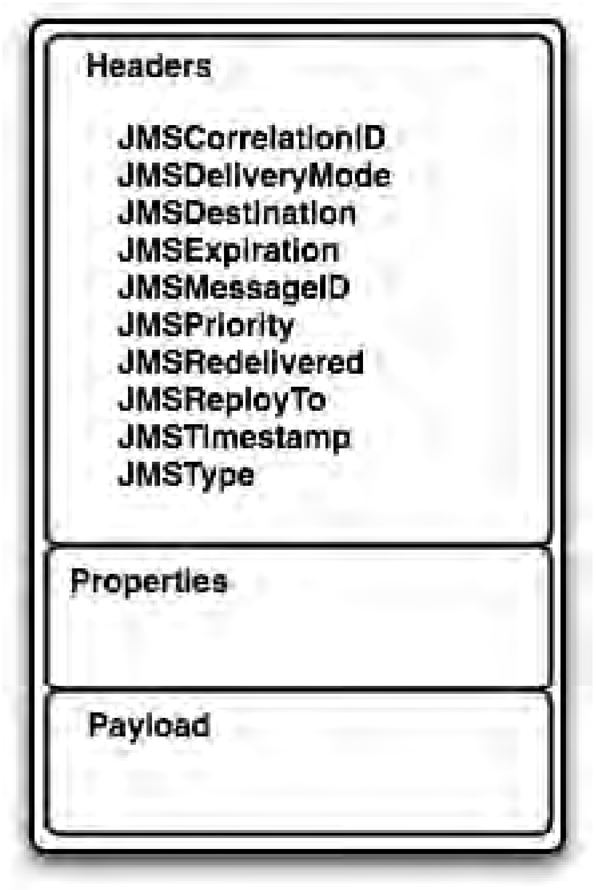


Figure 1.5. A graphical representation of a JMS message

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

1.4.3.1. JMS Message Headers

As shown in Figure 1.5, 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 be 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:

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

  - 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 are 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

- **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 a 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.

1.4.3.2. 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() throws JMSException;

    byte getByteProperty() throws JMSException;

    short getShortProperty() throws JMSException;

    int getIntProperty() throws JMSException;

    long getLongProperty() throws JMSException;

```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
    float getFloatProperty() throws JMSException;

    double getDoubleProperty() throws JMSException;

    String getStringProperty() throws JMSException;

    Object getObjectProperty(String name) throws JMSException;

...

    Enumeration getPropertyNames() throws JMSException;

    boolean propertyExists(String name) throws JMSException;

...

    void setBooleanProperty(String name, boolean value)
        throws JMSException;

    void setByteProperty(String name, byte value) throws JMSException;

    void setShortProperty(String name, short value) throws JMSException;

    void setIntProperty(String name, int value) throws JMSException;

    void setLongProperty(String name, long value) throws JMSException;

    void setFloatProperty(String name, float value) throws JMSException;

    void setDoubleProperty(String name, double value) throws JMSException;

    void setStringProperty(String name, String value) throws JMSException;

    void setObjectProperty(String name, Object value) throws JMSException;

....

}
```

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 the properties on a given message to easily iterate through all the properties on a message 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

properties and provider-specific properties.

1.4.3.2.1. 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.)

1.4.3.2.2. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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

1.4.3.2.3. 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.

1.4.4. 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 utlize 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 identfiers, literals and operators taken directly

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

from the SQL92 syntax and are defined in the Table 1.1 table below:

Table 1.1. 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 1.1 are used to create queries against message headers and properties. Consider the message in defined in Example 1.1 below. This message defines a couple properties that will be used for filtering messages in our example below.

Example 1.1. A JMS message with custom properties

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

    TextMessage textMessage = session.createTextMessage();
    textMessage.setText(payload);
    textMessage.setStringProperty("SYMBOL", symbol); #1
    textMessage.setDoubleProperty("PRICE", price);   #1

    producer.send(destination, textMessage);
}
 #1 Custom properties are added to the message
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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

Example 1.2. Filter messages using the SYMBOL header

```
...
String selector = "SYMBOL == 'AAPL'"; #1
MessageConsumer consumer = session.createConsumer(destination, selector);
...
 #1 Select messages containing a property named
 SYMBOL whose value is AAPL
```

Example 1.2 defines a selector to filter only messages for Apple, Inc. This consumer receive only messages matching the query defined in the selector.

Example 1.3. Filter messages using both the SYMBOL and PRICE headers

```
...
String selector = "SYMBOL == 'AAPL' AND PRICE > " + previousPrice ; #1
MessageConsumer consumer = session.createConsumer(destination, selector);
...
 #1 Select messages containing a property named
 PRICE whose value is great than the previous price
```

x specifies a selector that filters only messages for Apple, Inc. whose price property 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 1.4. Filter messages using the SYMBOL header, the PRICE header and the JMSExpiration header

```
...
String selector = "SYMBOL IN ('AAPL', 'CSCO') AND PRICE > " +
        previousPrice + " AND JMSExpiration > " +
        new Date().getTime(); #1
MessageConsumer consumer = session.createConsumer(destination, selector);
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
...
 #1 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 1.4 defines a more complext 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.

1.4.4.1. 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

1.4.5. 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.

1.4.5.1. 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

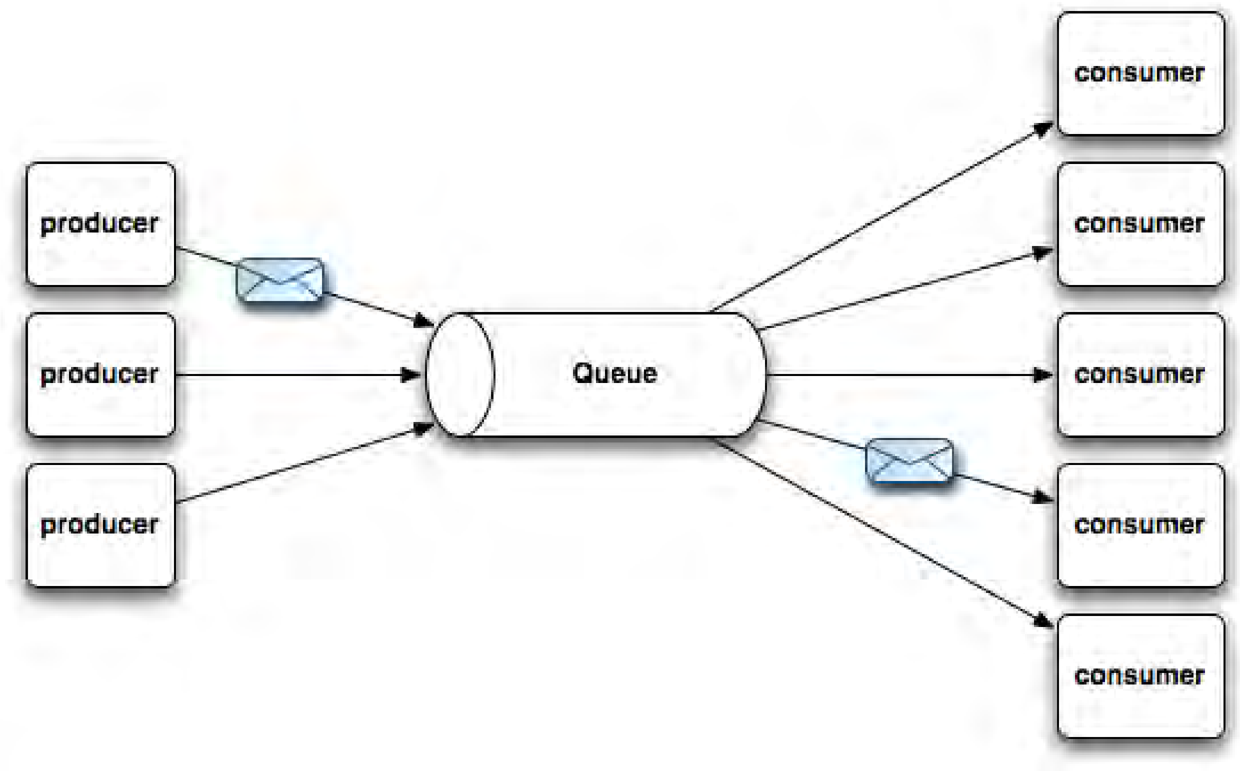


Figure 1.6. Point-to-point messaging uses a one-to-one messaging paradigm

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 1.6, 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 1.6 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

1.4.5.2. 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 1.7 below.

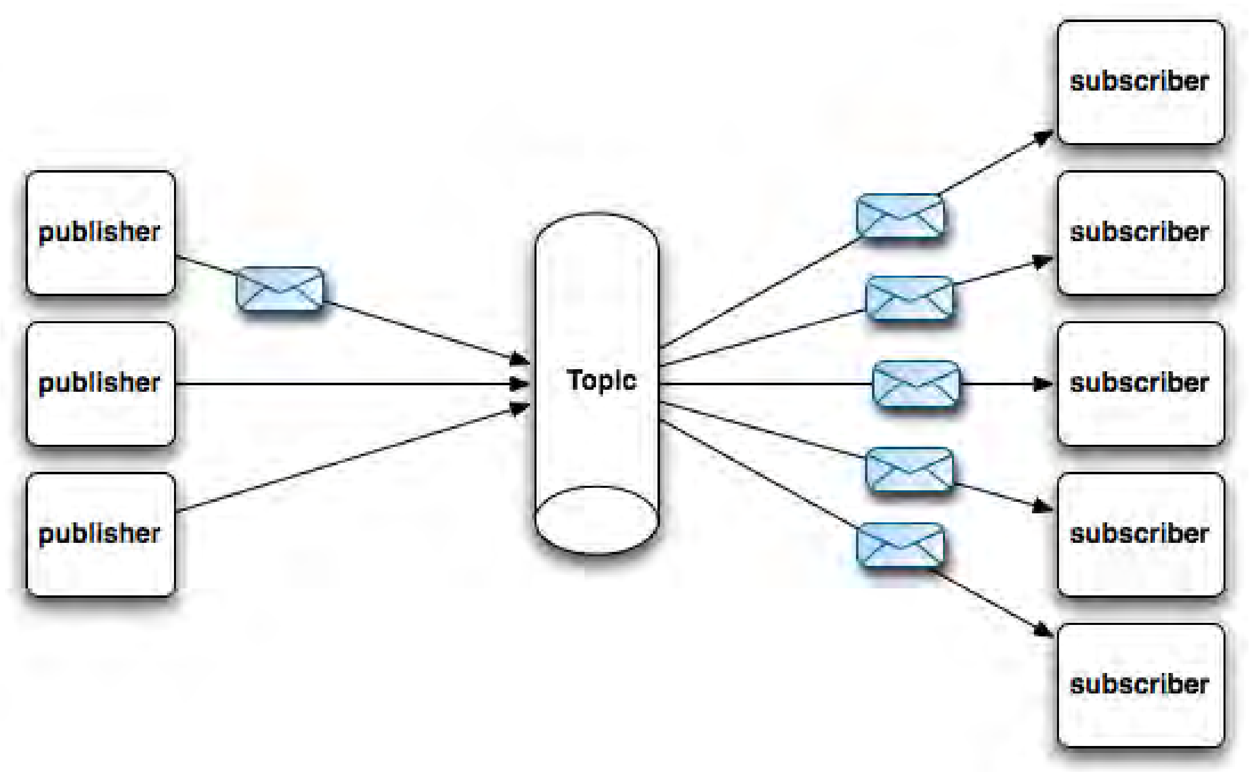


Figure 1.7. Publish/Subscribe uses a one-to-many messaging paradigm

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

1.4.5.2.1. 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. It's 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 1.8 that is, one reply per request.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

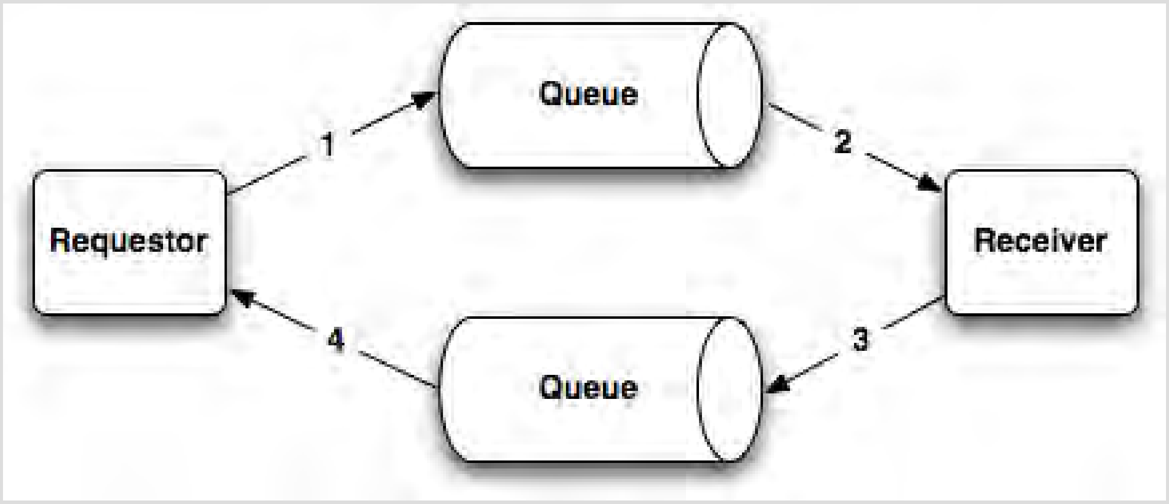


Figure 1.8. The steps involved in basic request/reply messaging

Figure 1.8, “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.

1.4.6. 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 most common when the JMS provider is hosted inside of a Java EE environment. The JMS spec defines two types of administered objects: `ConnectionFactory` and

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

`Destination`.

1.4.6.1. 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.

1.4.6.2. 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.

1.4.7. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 1.4 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 1.5 demonstrates the steps above for creating a JMS producer. Comments have been added to explain what is happening with each call.

Example 1.5. Using the JMS API to send a message

```
...
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
ConnectionFactory connectionFactory;
Connection connection;
Session session;
Destination destination;
MessageProducer producer;
Message message;
boolean useTransaction = false;
try {
  // An initial context is usually created with a JNDI path. This one is
  // for demonstration purposes only.
  Context ctx = new InitialContext();
  // 1) From the initial context, lookup a JMS connection factory using
  // the unique name for the connection factory.
  connectionFactory =
        (ConnectionFactory) ctx.lookup("ConnectionFactoryName");

  // 2) Using the connection factory, create a JMS connection.
  connection = connectionFactory.createConnection();

  // 3) Make sure to call start() on the connection factory to enable
  // messages to start flowing
  connection.start();

  // 4) Using the connection, create a JMS session. In this case we're
  // just using auto-acknowledgement of messages
  session = connection.createSession(useTransaction,
        Session.AUTO_ACKNOWLEDGE);

  // 5) Create a JMS queue using the session
  destination = session.createQueue("TEST.QUEUE");

  // 6a) Create a JMS producer using the session and the destination
  producer = session.createProducer(destination);

  // The JMS delivery mode is persistent by default, but we're just doing
  // it explicitly here
  producer.setDeliveryMode(DeliveryMode.PERSISTENT);

  // 6b) Create a simple text message containing only a payload
  message = session.createTextMessage("this is a test");

  // 8) Using the producer, send the message to the destination
  producer.send(message);

} catch (JMSException jmsEx) {
...
} finally {
  // 9) Close the objects that were used above
  producer.close();
  session.close();
  connection.close();
}
...
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

The example above in Example 1.5 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 1.6 demonstrates the steps for creating a JMS consumer and receiving the message.

Example 1.6. Using the JMS API to receive a message

```
...
ConnectionFactory connectionFactory;
Connection connection;
Session session;
Destination destination;
MessageProducer producer;
Message message;
boolean useTransaction = false;
try {
  // An initial context is usually created with a JNDI path. This one is
  // for demonstration purposes only.
  Context ctx = new InitialContext();

  // 1) From the initial context, lookup a JMS connection factory using
  // the unique name for the connection factory.
  connectionFactory =
        (ConnectionFactory) ctx.lookup("ConnectionFactoryName");

  // 2) Using the connection factory, create a JMS connection.
  connection = connectionFactory.createConnection();

  // 3) Make sure to call start() on the connection factory to enable
  // messages to start flowing
  connection.start();

  // 4) Using the connection, create a JMS session. In this case we're
  // just using auto-acknowledgement of messages
  session = connection.createSession(useTransaction,
        Session.AUTO_ACKNOWLEDGE);

  // 5) Create a JMS queue using the session
  destination = session.createQueue("TEST.QUEUE");
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
  // 7a) Create a JMS consumer using the session and destination
  consumer = session.createConsumer(destination);

  // 8) Using the consumer, receive the message that was sent to
  // destination in the previous example
  message = (TextMessage) consumer.receive(1000)

  System.out.println("Received message: " + message);

} catch (JMSException jmsEx) {
...
} finally {
  // 9) Close the objects that were used above
  producer.close();
  session.close();
  connection.close();
}
...
```

The example above in Example 1.6 is very similar to Example 1.5 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

1.4.8. 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 1.7. 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) {
                textMessage = (TextMessage) message;
                System.out.println("Received message: " + msg.getText());
                processMessage(textMessage);
            } else {
                System.out.println("Incorrect message type: " +
                    message.getClass().getName());
            }
        } catch (JMSException 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
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
    }
}
```

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

1.5. Conclusion

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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Chapter 2. Introduction to Apache ActiveMQ

2.1. Introduction

Your first steps with ActiveMQ are important to your success in using it in 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

2.2. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

redistribution of changes. This is a critical point for many businesses who use ActiveMQ in a strategic manner. As described in chapter one, the job of a MOM is to mediate events and messages amongst distributed systems, guaranteeing that they reach their intended destination(s). 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.

2.2.1. 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 in chapter one, 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

integrate with other containers is also very important. Chapter three 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 chapters four and five.

Integration With Many Java Containers - ActiveMQ provides the ability to easily integrate with many popular Java containers including:

- Apache Geronimo
- Apache Tomcat
- JBoss
- Jetty
- Spring Framework
- POJOs
- Weblogic
- Websphere

Integrating ActiveMQ with Java containers is covered in chapter seven.

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 to 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 chapters six and eight.

Broker Clustering - Many ActiveMQ brokers can work together as a federated

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 nine.

Many Advanced Broker Features and Client Options - ActiveMQ provides many sophisticated features for both the broker and the clients connecting to the broker. These features are discussed in chapters 10 and 11.

Enterprise Integration Patterns With Apache Camel - A very popular feature in ActiveMQ is the ability to utilize enterprise integration patterns using Apache Camel, a very powerful integration framework based on the Spring Framework. Chapter 13 introduces the use of Camel with ActiveMQ.

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 14.

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 new questions about why you might use ActiveMQ.

2.2.2. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

coupled applications can easily deal with unforeseen changes.

Technologies such as those mentioned in chapter one (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 ??? below depicts this concept.

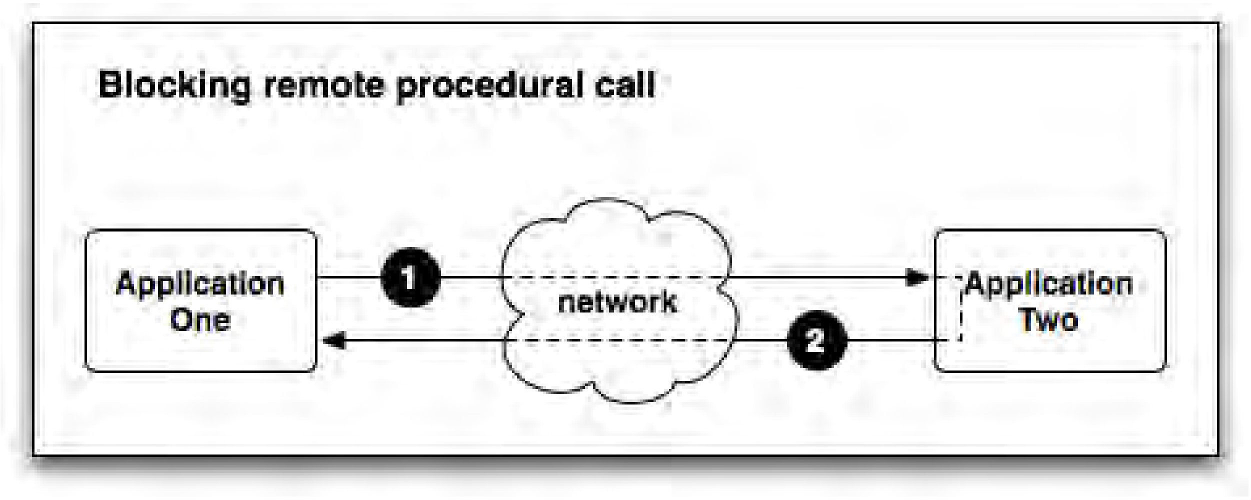


Figure 2.1. Two tightly-coupled RPC applications

The caller (application one) in ??? 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 ???.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

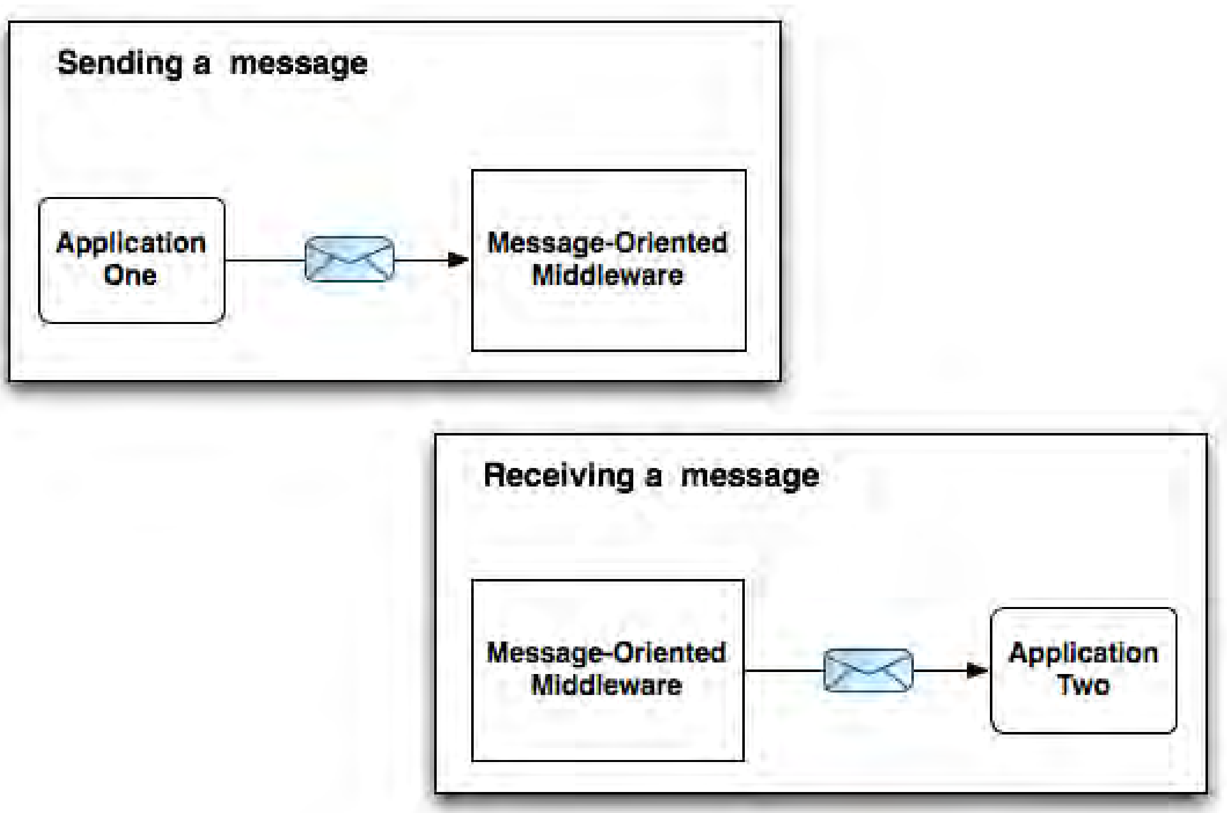


Figure 2.2. Two loosely-coupled JMS applications

Application one in ??? 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. ActiveMQ 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 not

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 to any applications consuming the messages. 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?

2.2.3. 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 solution, 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 in a negative manner, but this is not a requirement, especially when introducing loose coupling. 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. Let's look at some

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

details of each one now.

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.

These two use cases are extremely common and although the pub/sub use case focuses on stock prices, each messaging domain can be applied to many business domains. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 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 where a single ActiveMQ instance is used and 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 each application. 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 for messages. In ActiveMQ parlance, this is known as a network of brokers and is most commonly used to increase the amount of messages ActiveMQ can handle but it is also used to mitigate various network designs where direct remote connection to an ActiveMQ instance is not feasible due to the network design. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

ActiveMQ just about any messaging need can be met. Now let's get started using ActiveMQ and digging a bit deeper into the stock portfolio example and the job queue example.

2.3. 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 download it.

2.3.1. Downloading ActiveMQ

Downloading ActiveMQ can be achieved using a web browser or downloading via a tool like wget. Simply visit the ActiveMQ downloads page (http://activemq.apache.org/download.html) and select the version you'd like to use. As of this writing, the latest release is ActiveMQ 5.1.0 so we'll download that and use it for all the examples in the book. Click on the link named *ActiveMQ 5.1.0 Release* and you'll be taken to the 5.1.0 release page. On this page, you can download the ActiveMQ binary distribution for the platform you're using - Windows or Linux/Unix. All the work in this book will be demonstrated from the command line since that will work with either Windows or Unix. I'm downloading the tarball because I use MacOS X. Below is the command to download ActiveMQ 5.1.0 from the command line:

```
[~]$ wget http://apache.oregonstate.edu/activemq/apache-activemq/5.1.0/apache-activemq-5
```

(Remember that when downloading from the ASF, you are usually downloading from a mirror server instead of from a server hosted by the ASF. The URL above is a mirror and was selected at random - there are many other mirrors from which to choose.) After the ActiveMQ archive is downloaded, place it in a directory where you'd like to expand it and use it.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

2.3.2. Expanding the ActiveMQ Archive

The next step is to expand the archive so that we can start working with ActiveMQ. Below is a command to expand the tarball:

```
[~]$ tar zxvf apache-activemq-5.1.0-bin.tar.gz
```

The command above will create a directory named `apache-activemq-5.1.0`, change into this directory:

```
[~]$ cd apache-activemq-5.1.0
```

Before we dig into the examples, let's examine the apache-actvemq-5.1.0 directory a bit.

2.3.3. Examining the ActiveMQ Directory

While in the apache-activemq-5.1.0 directory, list the contents:

```
[apache-activemq-5.1.0]$ ls -l
total 7536
-rw-r--r--   1 bsnyder  staff    40581 May  1 12:35 LICENSE
-rw-r--r--   1 bsnyder  staff     4188 May  1 12:35 NOTICE
-rw-r--r--   1 bsnyder  staff     2583 May  1 12:35 README.txt
-rw-r--r--   1 bsnyder  staff     2038 May  1 12:35 WebConsole-README.txt
-rw-r--r--   1 bsnyder  staff  3794617 May  1 12:44 activemq-all-5.1.0.jar
drwxr-xr-x   9 bsnyder  staff      306 Jun 20 11:48 bin
drwxr-xr-x   9 bsnyder  staff      306 Jun 20 11:48 conf
drwxr-xr-x   3 bsnyder  staff      102 Jun 20 11:48 data
drwxr-xr-x   3 bsnyder  staff      102 Jun 20 11:48 docs
drwxr-xr-x   8 bsnyder  staff      272 Jun 20 11:48 example
drwxr-xr-x  21 bsnyder  staff      714 Jun 20 11:48 lib
-rw-r--r--   1 bsnyder  staff     2678 May  1 12:35 user-guide.html
drwxr-xr-x   6 bsnyder  staff      204 Jun 20 11:48 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

- **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-5.1.0.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
- **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

Now let's dig into the examples to test ActiveMQ.

2.3.4. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

following command in a LInux/Unix environment:

```
[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 s
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
INFO  TransportConnector               - Connector openwire Started
INFO  TransportServerThreadSupport     - Listening for connections at: ssl://mongoose.local
INFO  TransportConnector               - Connector ssl Started
INFO  TransportServerThreadSupport     - Listening for connections at: stomp://mongoose.loc
INFO  TransportConnector               - Connector stomp Started
INFO  TransportServerThreadSupport     - Listening for connections at: xmpp://mongoose.loca
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.mor
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
INFO  log                              - Started SelectChannelConnector@0.0.0.0:8161
INFO  FailoverTransport                - Successfully connected to tcp://localhost:61616
```

This command starts up the ActiveMQ broker and some of it's connectors to expose it 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, so let's dig into those examples now.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 for 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 defintion 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 utlizes 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 defintion 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:

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
<!--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.

2.3.4.1. Verifying Your ActiveMQ Setup With the Examples

OK, so ActiveMQ is running in one terminal, now we're gonna create a couple more terminals for using the examples to verify that ActiveMQ is working correctly. In the second terminal, change into the example directory and do a listing of the contents:

```
[~]$ cd ./apache-activemq-5.1.0/examples/
[example]$ ls -l
total 24
-rw-r--r--   1 bsnyder  staff  10503 May  1 12:35 build.xml
drwxr-xr-x   6 bsnyder  staff    204 Jun 20 11:48 conf
drwxr-xr-x   3 bsnyder  staff    102 Jun 20 11:48 perfharness
drwxr-xr-x   7 bsnyder  staff    238 Jun 20 11:48 ruby
drwxr-xr-x  10 bsnyder  staff    340 Jun 20 11:48 src
drwxr-xr-x   5 bsnyder  staff    170 Jun 20 11:48 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

- **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

In this chapter, we will focus on the Java examples and using them via the build.xml file. So let's get started.

In 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-5.1.0/example/target/classes
    [mkdir] Created dir: /Users/bsnyder/amq/apache-activemq-5.1.0/example/src/ddl

compile:
    [javac] Compiling 8 source files to /Users/bsnyder/amq/apache-activemq-5.1.0/example

consumer:
     [echo] Running consumer against server at $url = tcp://localhost:61616 for 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 and you can see from the output that this consumer is:

- Connecting to the broker using the TCP protocol (tcp://localhost:61616)
- Watching a queue named TEST.FOO
- Using non-durable subscription

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

- Waiting to receive 2000 messages and then shutting down

Basically, the JMS consumer is connected to ActiveMQ and waiting for messages. So let's send it some messages.

In the third terminal, change into the example directory and start up a JMS producer to send messages:

```
[~]$ cd ./apache-activemq-5.1.0/examples/
[example]$ ant producer
Buildfile: build.xml

init:

compile:

producer:
     [echo] Running producer against server at $url = tcp://localhost:61616 for 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 lastSampleTi
description: Number of messages exchanged }
     [java]     messageRateTime{ count: 0 maxTime: 0 minTime: 0 totalTime: 0 averageTime
averageTimeExMinMax: 0.0 averagePerSecond: 0.0 averagePerSecondExMinMax: 0.0 unit: millis
1213991298654 lastSampleTime: 1213991298654 description: Time taken to process a message
}
     [java]     pendingMessageCount{ count: 0 unit: count startTime: 1213991298654 lastSa
1213991298654 description: Number of pending messages }
     [java]     expiredMessageCount{ count: 0 unit: count startTime: 1213991298654 lastSa
1213991298654 description: Number of expired messages }
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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

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

The command above starts up a simple JMS producer and you can see from the output that it is:

- Connecting to the broker using the TCP protocol (tcp://localhost:61616)
- Publishing messages to a queue named TEST.FOO
- Using non-persistent 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:

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
     [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
```

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 we learn here? Through the use of the Java examples that come with ActiveMQ, we proved 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 your 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. 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

2.4. Understanding the Example Use Cases

There are two examples that will be used throughout the book. One will focus on the point-to-point (PTP) messaging domain and the other will focus on the publish/subscribe (pub/sub) domain. 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 identical, 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.

2.4.1. Prerequisites

Before moving on to work with the examples, you will need a few pieces of software installed. Below are instructions for downloading and installing each.

2.4.1.1. Download and Install the Java SE

The book 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)
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
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.

2.4.1.2. Download and Install Apache Maven

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

http://maven.apache.org/download.html

Maven is provided in tarball and zip format so grab the appropriate version for your operating system. Also, make sure you're using the latest version, which, at the time of this writing, is version 2.0.9. To install Maven, follow the instructions available on the Maven website at the following URL:

http://maven.apache.org/download.html#Installation

Once you've installed Maven, make sure to test it out using the command listed below:

```
[~]$ mvn -v
Maven version: 2.0.9
Java version: 1.5.0_13
OS name: "mac os x" version: "10.5.4" arch: "i386" Family: "unix"
```

The output should state the Maven version, the Java version and some information about the operating system. The important part is that the information is actually output because this tells us that Maven is working. If it is not, then you should walk through the installation steps again to double-check that everything was set up correctly.

Maven Needs An Internet Connection

To make use of Maven for the examples in this book, it will need a broadband connection to the Internet. This is so that it can connect to some remote

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Maven repositories and download the necessary artifacts to the run the examples for the book.

2.4.1.3. Download and Install Apache 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 5.1.0 release and you will find both tarball and zip formats available. Once you have downloaded one of the archives, expand it and you're ready to go. Nothing more needs to be done at this time to start up ActiveMQ and begin using it. Below is an example of starting it up from the command line.

```
[~]$ cd ./apache-activemq-5.1.0/
[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 s
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
INFO  TransportConnector             - Connector openwire Started
INFO  TransportServerThreadSupport   - Listening for connections at: ssl://mongoose.local
INFO  TransportConnector             - Connector ssl Started
INFO  TransportServerThreadSupport   - Listening for connections at: stomp://mongoose.loc
INFO  TransportConnector             - Connector stomp Started
INFO  TransportServerThreadSupport   - Listening for connections at: xmpp://mongoose.loca
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.mor
via org.mortbay.log.Slf4jLog
INFO  log                            - jetty-6.1.9
INFO  WebConsoleStarter              - ActiveMQ WebConsole initialized.
INFO  /admin                         - Initializing Spring FrameworkServlet 'dispatcher'
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
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
INFO  log                             - Started SelectChannelConnector@0.0.0.0:8161
INFO  FailoverTransport               - Successfully connected to tcp://localhost:61616
```

The default ActiveMQ configuration was used in the example above so your output should look the same with the exception of the machine- and platform-specific items.

If for some reason ActiveMQ is not starting up properly, search the archives for the ActiveMQ user mailing list for your problem. If you are unable to find any information about the issue you're experiencing in the archives, send a message to the ActiveMQ user mailing list to ask questions. Information about the ActiveMQ mailing lists is available at the following URL:

http://activemq.apache.org/mailing-lists.html

Once you get to this point, you should have the Java SE, Maven and ActiveMQ all set up and working correctly. On to the examples...

2.4.2. 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.

2.4.2.1. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

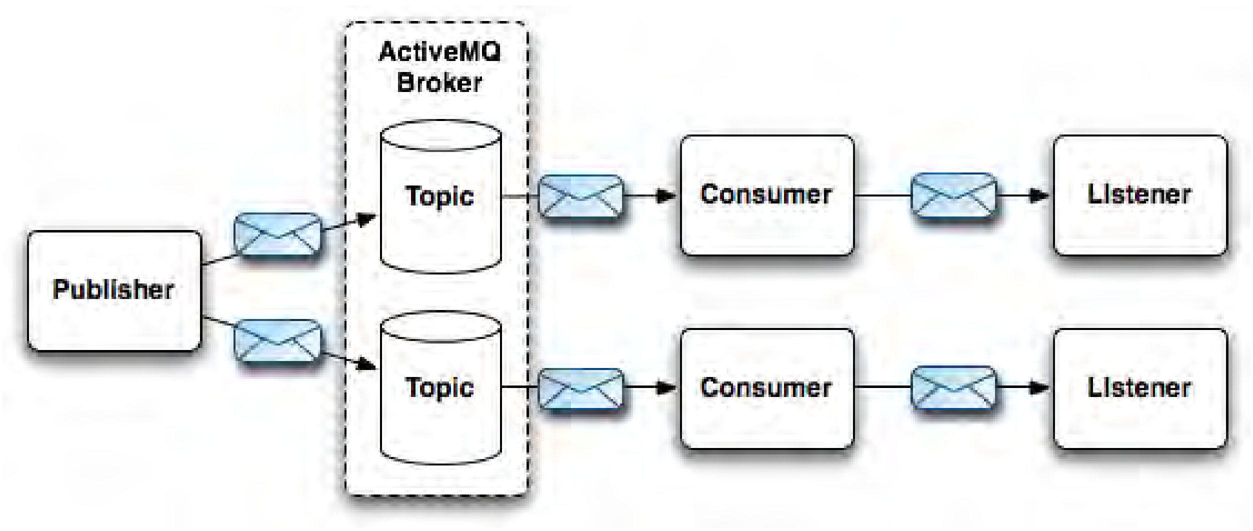


Figure 2.3. The stock portfolio example

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.

2.4.2.2. Running the Stock Portfolio Example

There are three basic steps to running this example including:

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 s
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
INFO  TransportConnector             - Connector openwire Started
INFO  TransportServerThreadSupport   - Listening for connections at: ssl://mongoose.local
INFO  TransportConnector             - Connector ssl Started
INFO  TransportServerThreadSupport   - Listening for connections at: stomp://mongoose.loc
INFO  TransportConnector             - Connector stomp Started
INFO  TransportServerThreadSupport   - Listening for connections at: xmpp://mongoose.loca
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.mor
via org.mortbay.log.Slf4jLog
INFO  log                            - jetty-6.1.9
INFO  WebConsoleStarter              - ActiveMQ WebConsole initialized.
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
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
INFO  log                            - Started SelectChannelConnector@0.0.0.0:8161
INFO  FailoverTransport              - Successfully connected to tcp://localhost:61616
```

The output above is identical to that in Section 2.4.1.3. Nothing has been changed, the default ActiveMQ configuration is used here as well.

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 AXP"
[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-ma
exec-maven-plugin-1.1.pom
3K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/codehaus/mojo/mojo/17
16K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/codehaus/mojo/exec-ma
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/
maven-plugins-8.pom
5K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven-pa
maven-parent-5.pom
14K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/apache/3/apach
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
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/activemq/active
activemq-all-5.1.0.pom
2K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/activemq/active
activemq-parent-5.1.0.pom
43K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/apache/4/apache
4K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/javax/transaction/jta/1.0
515b downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/springframework/spring
spring-2.5.1.pom
12K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/xbean/xbean-sp
xbean-spring-3.3.pom
4K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/xbean/xbean/3.3
19K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/commons-logging/commons-lo
commons-logging-1.0.3.pom
866b downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/junit/junit/3.8.1/junit-3
998b downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/log4j/log4j/1.2.14/log4j-1
2K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/javax/xml/stream/stax-api/
stax-api-1.0-2.pom
167b downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/activemq/active
activemq-all-5.1.0.jar
3705K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/javax/transaction/jta/1.0
8K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/springframework/spring
spring-2.5.1.jar
2766K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/xbean/xbean-sp
xbean-spring-3.3.jar
125K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/log4j/log4j/1.2.14/log4j-1
358K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/javax/xml/stream/stax-api/
stax-api-1.0-2.jar
22K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven-pro
maven-project-2.0.4.pom
1K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven/2.0
maven-2.0.4.pom
11K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven-set
maven-settings-2.0.4.pom
1K downloaded
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven-mod
maven-model-2.0.4.pom
2K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/codehaus/plexus/plexus
plexus-utils-1.1.pom
767b downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/codehaus/plexus/plexus
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
1K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/codehaus/plexus/plexus
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
747b downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven-pro
maven-profile-2.0.4.pom
1K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven-art
2.0.4/maven-artifact-manager-2.0.4.pom
1K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven-rep
2.0.4/maven-repository-metadata-2.0.4.pom
1K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/maven-art
maven-artifact-2.0.4.pom
765b downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/wagon/wag
1.0-alpha-6/wagon-provider-api-1.0-alpha-6.pom
588b downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/apache/maven/wagon/wag
wagon-1.0-alpha-6.pom
6K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/codehaus/plexus/plexus
plexus-utils-1.5.1.pom
2K downloaded
Downloading: http://localhost:8081/nexus/content/groups/public/org/codehaus/plexus/plexus
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 *AXP*. These two topic names were picked at random and can be replaced with any Strings you desire. The important part is that the same

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

arguments be used for both the Consumer and the Publisher (the Publisher is shown next) via the system property named `exec.args`.

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 2.4.1.2, “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 AXP"
[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: {offer=58.31986582023755, price=58.26160421602154, up=true, stock=AXP} on desti
topic://STOCKS.AXP
Sending: {offer=58.46614894941039, price=58.4077412082022, up=true, stock=AXP} on destina
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
topic://STOCKS.AXP
Sending: {offer=58.718323292827435, price=58.65966362919824, up=true, stock=AXP} on desti
topic://STOCKS.AXP
Sending: {offer=58.345627177563735, price=58.287339837726016, up=false, stock=AXP} on des
topic://STOCKS.AXP
Sending: {offer=36.37572048021212, price=36.339381099113005, up=true, stock=CSCO} on dest
topic://STOCKS.CSCO
Sending: {offer=36.57877346104115, price=36.54223122981134, up=true, stock=CSCO} on desti
topic://STOCKS.CSCO
Sending: {offer=58.2126392468028, price=58.15448476204077, up=false, stock=AXP} on destin
topic://STOCKS.AXP
Sending: {offer=36.695153690818174, price=36.65849519562256, up=true, stock=CSCO} on dest
topic://STOCKS.CSCO
Sending: {offer=36.68585049103857, price=36.649201289748824, up=false, stock=CSCO} on des
topic://STOCKS.CSCO
Sending: {offer=58.153551810338584, price=58.09545635398461, up=false, stock=AXP} on dest
topic://STOCKS.AXP
Published '10' of '10' price messages
Sending: {offer=36.4966203873684, price=36.460160227141266, up=false, stock=CSCO} on dest
topic://STOCKS.CSCO
Sending: {offer=58.045779233365835, price=57.98779144192392, up=false, stock=AXP} on dest
topic://STOCKS.AXP
Sending: {offer=36.843532042734964, price=36.80672531741755, up=true, stock=CSCO} on dest
topic://STOCKS.CSCO
Sending: {offer=36.954536453437285, price=36.917618834602685, up=true, stock=CSCO} on des
topic://STOCKS.CSCO
Sending: {offer=58.49426529786688, price=58.43582946839849, up=true, stock=AXP} on destin
topic://STOCKS.AXP
Sending: {offer=58.594484920787735, price=58.53594897181593, up=true, stock=AXP} on desti
topic://STOCKS.AXP
Sending: {offer=59.01185791171931, price=58.952905006712605, up=true, stock=AXP} on desti
topic://STOCKS.AXP
Sending: {offer=37.10996253306843, price=37.072889643425015, up=true, stock=CSCO} on dest
topic://STOCKS.CSCO
Sending: {offer=37.10871702959351, price=37.071645384209305, up=false, stock=CSCO} on des
topic://STOCKS.CSCO
Sending: {offer=59.27734586883228, price=59.218127741091195, up=true, stock=AXP} on desti
topic://STOCKS.AXP
Published '10' of '20' price messages
Sending: {offer=58.79092604485173, price=58.73219385100074, up=false, stock=AXP} on desti
topic://STOCKS.AXP
Sending: {offer=59.25190817806627, price=59.19271546260367, up=true, stock=AXP} on destin
topic://STOCKS.AXP
Sending: {offer=37.26677291595445, price=37.22954337258187, up=true, stock=CSCO} on desti
topic://STOCKS.CSCO
Sending: {offer=58.939385104126835, price=58.88050459952731, up=false, stock=AXP} on dest
topic://STOCKS.AXP
Sending: {offer=37.39307124450011, price=37.355715528971146, up=true, stock=CSCO} on dest
topic://STOCKS.CSCO
Sending: {offer=37.11888897612781, price=37.08180716895886, up=false, stock=CSCO} on dest
topic://STOCKS.CSCO
Sending: {offer=37.34651626026739, price=37.30920705321418, up=true, stock=CSCO} on desti
topic://STOCKS.CSCO
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
Sending: {offer=59.3224657152465, price=59.26320251273378, up=true, stock=AXP} on destina
topic://STOCKS.AXP
Sending: {offer=37.416557967252466, price=37.379178788464, up=true, stock=CSCO} on destin
topic://STOCKS.CSCO
Sending: {offer=59.0915505887881, price=59.032518070717394, up=false, stock=AXP} on desti
topic://STOCKS.AXP
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]
AXP     58.26   58.32   up
AXP     58.41   58.47   up
AXP     58.66   58.72   up
AXP     58.29   58.35   down
CSCO    36.34   36.38   up
CSCO    36.54   36.58   up
AXP     58.15   58.21   down
CSCO    36.66   36.70   up
CSCO    36.65   36.69   down
AXP     58.10   58.15   down
CSCO    36.46   36.50   down
AXP     57.99   58.05   down
CSCO    36.81   36.84   up
CSCO    36.92   36.95   up
AXP     58.44   58.49   up
AXP     58.54   58.59   up
AXP     58.95   59.01   up
CSCO    37.07   37.11   up
CSCO    37.07   37.11   down
AXP     59.22   59.28   up
AXP     58.73   58.79   down
AXP     59.19   59.25   up
CSCO    37.23   37.27   up
AXP     58.88   58.94   down
CSCO    37.36   37.39   up
CSCO    37.08   37.12   down
CSCO    37.31   37.35   up
AXP     59.26   59.32   up
CSCO    37.38   37.42   up
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
AXP     59.03   59.09   down
AXP     59.12   59.18   up
CSCO    37.51   37.55   up
AXP     59.20   59.26   up
AXP     59.40   59.46   up
AXP     59.76   59.82   up
AXP     59.85   59.91   up
AXP     59.26   59.32   down
CSCO    37.54   37.58   up
AXP     58.90   58.96   down
...
```

The output above comes from the Listener class that is registered by the Consumer on the two topics named AXP 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.

2.4.2.3. 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

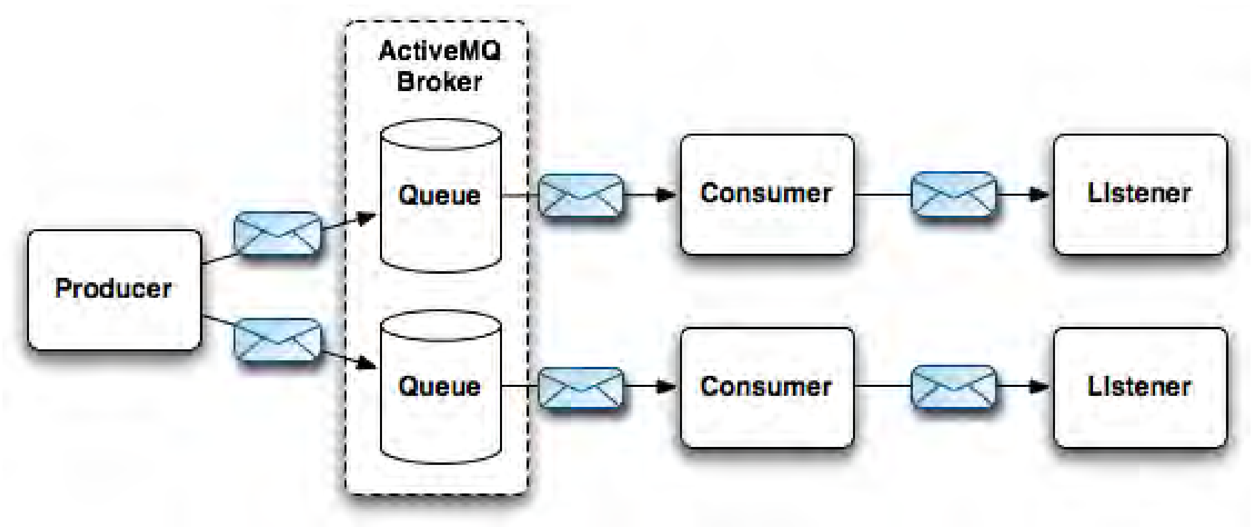


Figure 2.4. The job queue example

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.

2.4.2.4. 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 s
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
INFO  TransportConnector             - Connector openwire Started
INFO  TransportServerThreadSupport   - Listening for connections at: ssl://mongoose.local
INFO  TransportConnector             - Connector ssl Started
INFO  TransportServerThreadSupport   - Listening for connections at: stomp://mongoose.loc
INFO  TransportConnector             - Connector stomp Started
INFO  TransportServerThreadSupport   - Listening for connections at: xmpp://mongoose.loca
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.mo
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
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 2.4.1.3, “Download and Install Apache 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.Producer
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'exec'.
[INFO] ------------------------------------------------------------------------
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
[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:

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
[trunk]$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch2.jobs.Consumer -Dexec
[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
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
...
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

2.5. 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. You should now have ActiveMQ installed and have a familiarity with the two examples that will be used throughout the rest of the chapters in this book. The next step is to begin to understand the ActiveMQ configuration so that you can configure connectors for clients and additional ActiveMQ brokers.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Part II. Configuring ActiveMQ

[Intro goes here]

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Chapter 3. Understanding Connectors

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

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

- *Connector URIs*, which make possible to address brokers
- *Transport connectors*, we will use to expose brokers to clients
- *Network connectors*, used to create networks of brokers
- *Discovery Agents*, that allows us to discover brokers in a cluster
- We will explain in details some of the widely used connectors

Details provided in this chapter, will help you understand protocols used to connect to ActiveMQ brokers and choose the right setup for your needs.

3.1. Understanding Connector URIs

Before we start discussing particular connectors and their role in the overall ActiveMQ architecture, it is important to understand *connector URIs* as they are a standardized way of addressing connectors.

Uniform Resource Identifiers (URIs), as a concept, are not new and you have probably used it over and over again for different purposes. They 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Internet services. For example, web and email addresses we use every day are just some of the examples of URIs in practice.

Without going too deep into discussing URI basics (since I'm sure you're more than familiar with the concept) let's just briefly recapitulate 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 high-level string format:

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

If you, for example, take a look at the following URI:

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

you can see that we used `mailto` schema followed by an email address to uniquely identify both service we are going to use and the particular resource within.

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 address:

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

uses `http` schema 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 broker's through different type of connectors. If we go back to examples discussed in Chapter 2, Introduction to Apache ActiveMQ, you can see we used the following URI:

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

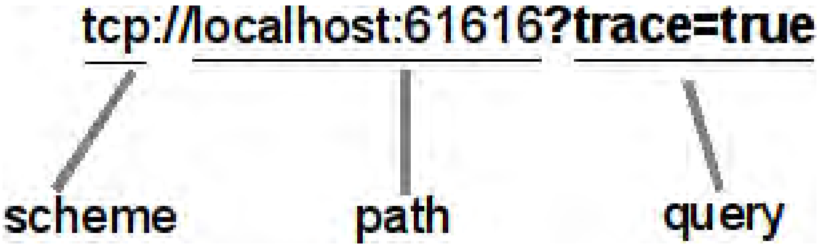
to create a connection to the broker.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

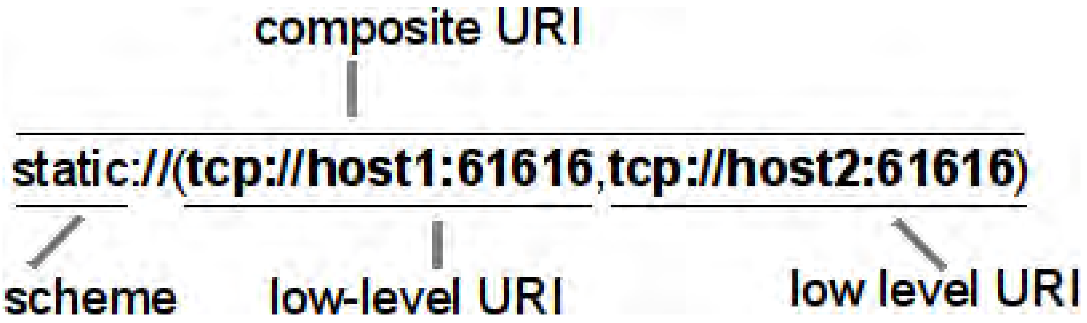
This is a typical hierarchical URI used in ActiveMQ which translates to "create a TCP connection to local host on port 61616".

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 schema part to identify underlying network protocol, path element to identify a network resource (usually host and port) and query element to specify additional configuration parameters for the connector. For example, the following URI:



extends the one previously used and additionally tells the broker to log all commands sent over this connector.

When using ActiveMQ (or messaging in general) you will often find your self in a situation where you want to implement some logic on top of a couple of broker addresses. For example, usually you will want to implement some kind of reconnection logic, so you can connect to another broker in case you current connection goes down. Luckily, ActiveMQ supports practically all such use cases (as we will see in Chapter 9, Broker Topologies) and make them easy to use and configure through *composite URIs* (and connectors). Composite URIs are used to create an additional logic or functionality on top of low-level connectors. In the following snippet you can see an example of a typical composite URI:



As you can see, the schema part now identifies the complex protocol being used (the static protocol will be described later in this chapter) and the schema specific

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

part usually contains one or more low-level URIs that will be used to make a connection. Of course, every low-level URI and the main 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. This, of course, is not allowed since the URI specification (and its standard Java implementation) does not allow it. This is one of the common ActiveMQ configuration mistakes, so be careful not to put white spaces in your URIs.

Now that we are familiar with ActiveMQ URI basics, let's discuss connector types supported by ActiveMQ and their purpose. In the rest of this chapter, we will discuss transport and network connectors; how we can configure them in ActiveMQ configuration file and later use in our client Java applications.

Preventing Automatic Broker Discovery

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 for 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 defintion from the `conf/activemq.xml` configuration file:

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
  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 utlizes 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 defintion 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

3.2. Configuring Transport Connectors

As a JMS broker, ActiveMQ must allow client applications (producers and consumers) to connect to its destinations. As we will see in this and chapters that follows, you have a wide range of protocols you can use to achieve this. 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 listen to and accept connections from clients. If you take a look at the default ActiveMQ configuration file (`conf/activemq.xml`), you can see the configuration snippet similar to this one (depending on the version you are using):

```
    <!-- 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>` tag. You define particular connectors with `appropriate` `<transportConnector>` tags. Naturally, the ActiveMQ broker can simultaneously support many protocols listening on different ports. For a particular connector, you have to define a name and an URI (defining the network protocol and other protocol parameters to be used) with appropriate attributes. The `discoveryUri` is an optional attribute and will be discussed in section 3.4.1, Discovery Agents.

This connection snippet defines four transport connectors and if you run your broker you should see the following log messages on your console:

```
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
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
INFO  TransportConnector            - Connector xmpp Started
```

From the client's perspective, the transport connector URI is used to create a connection to the broker, which will be later used to send and receive messages. This topic will be discussed in more details in Chapter 6, Creating Java Applications With ActiveMQ, but the following code snippet should be enough to demonstrate the usage of 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);
```

Here, we have used one of the transport connector URIs defined in ActiveMQ configuration to create an appropriate client connection to the broker.

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.

Now let's dive into transport connectors and some of the frequently used protocols for client to broker communication.

3.2.1. Using Transport Connectors

In Chapter 2, Introduction to ActiveMQ, we have defined a stock portfolio example 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

can work over any provided protocol.

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

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

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

     Publisher publisher = new Publisher(args[0]);            #A

     String[] topics = new String[args.length - 1];           #B
     System.arraycopy(args, 1, topics, 0, args.length - 1);   #B
        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();
    }

 #A Define connection URI
 #B Extract topics from rest of arguments
```

The additional code (marked bold) is here to ensure we have passed the connector URI as a first argument and to extract topic names from the rest of the arguments passed to the application. Now we can run this stock portfolio publisher with the following command:

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
...

Sending: {price=65.71335660140976, stock=JAVA, offer=65.77906995801116, up=true}
  on destination: topic://STOCKS.JAVA
Sending: {price=66.07160567194602, stock=JAVA, offer=66.13767727761795, up=true}
  on destination: topic://STOCKS.JAVA
Sending: {price=65.92903500162092, stock=JAVA, offer=65.99496403662253, up=false}
  on destination: topic://STOCKS.JAVA

...
```

Note that we have one more argument now, and that is the URL we should use to connect to the appropriate broker.

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

Example 3.2. 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]);               #A

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

 #A Define connection URI
 #B Extract topics from the rest of arguments
```

In order to achieve the same functionality as in our chapter 2 example, you should run the consumer with an extra URI argument. The following example shows how

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

to do it:

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

...

JAVA 65.71 65.78 up
JAVA 66.07 66.14 up
JAVA 65.93 65.99 down
IONA 23.30 23.33 up

...
```

As you can notice the message flow between the producer and the consumer is the same as in our original example. Now that we have our consumer and producer ready, we can discuss and run examples over variety of supported protocols.

3.2.2. Using Network protocols

In a most common usage scenario, you will want to configure and run your broker 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 this kind of client-to-broker communication.

3.2.2.1. 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 on-line communication. It is used as an underlying network protocol for wide range of Internet services, such as email or Web for example.

I'm sure you are familiar with the basics of TCP and you've probably used it in your projects on various occasions (at least indirectly), but let's start our discussion of TCP by quoting its specification - RFC 793 (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. ”

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 JMS implementation. So it should not come as a surprise that *TCP transport connector* is a default and definitely one of the most frequently used ActiveMQ connectors.

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

```
tcp://hostname:port?transportOptions
```

with the required part marked as bold.

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 TCP transport connector, as we have seen in previous sections, you have to add the configuration snippet similar to the following to your ActiveMQ configuration file:

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

Note that we have used the `trace` option to instruct the broker to log all commands sent over this connector.

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 IONA JAVA"
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

As we said before, this is a default transport connector and you will probably use it to communicate with the broker from Java applications over the network. Some of the benefits of the TCP transport connectors are:

- Efficiency - since this protocol uses OpenWire marshaling protocol to convert messages to stream of bytes (and back), it is very efficient in terms of network usage and performances

- Availability - TCP is one of the most widespread network protocols and it is supported in Java from its early days, so it is almost certainly supported on your platform of choice

- Reliability - 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.

3.2.2.2. 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.

This brief introduction to NIO, implies that *NIO transport connector* is practically the same as the standard TCP connector, in terms it uses TCP as an underlying network protocol and OpenWire as a message serialization protocol. The only difference is that NIO connector is implemented using the NIO API, making it more suitable in situations where:

- *You have a large number of clients you want to connect to your broker*. Generally, the number of clients that can connect to the broker is limited by

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

the number of threads supported by your operating system. Since the NIO connector implementation starts less threads per client than the appropriate TCP connector, you should consider using NIO in case TCP could not meet your needs.

- *You have a heavy network traffic to your broker*. Again, the NIO connector generally has a better performance than the TCP one, so you can consider using it when you find that the default connector does not meet your needs.

At this point it is important to say that performance tunning of ActiveMQ is not just related to choosing the right connector. It is also done by choosing the right network of brokers topology (Chapter 9, Broker Topologies) and setting various options for brokers, producers and consumers (Chapter 12, Tuning ActiveMQ For Performance).

The URI syntax for the NIO connector is practically the same as the TCP connector URI syntax. The only difference is that you should use the `nio` schema instead of `tcp`, like it is shown here:

```
nio://hostname:port?transportOptions
```

Now take a look at the following configuration snippet:

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

Here, we defined two transport connectors: a `tcp` connector that listens on port `61616` and a `nio` connector that listens on port `61618`.

Now let's run our stock portfolio example, but this time we will connect our publisher and consumer over different transport connectors. As figure 3.1 shows, we will instruct our publisher to send messages over the NIO transport connector, while the consumer will receive those messages using the TCP transport connector.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

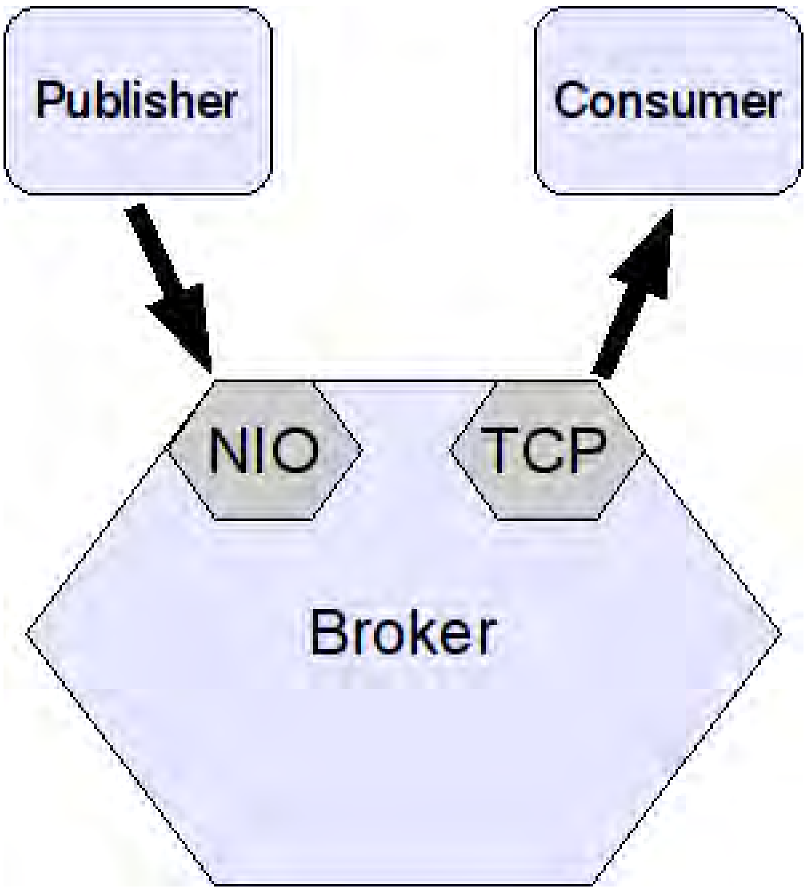


Figure 3.1. Figure 3.1: NIO example

To achieve this we should run our stock portfolio publisher with the following command:

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

Notice that we have used `nio` scheme in the connection URI to specify the appropriate connector.

The consumer should be run over the TCP connector using the command we have already defined earlier:

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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

3.2.2.3. User Datagram Protocol (UDP)

User Datagram Protocol (UDP) along with TCP makes 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:

- *TCP is stream oriented protocol*, which means that order of data packets are guaranteed. In other words there is no chance for data packets to come duplicated or out of order. UDP, on the other hand, does not guarantee ordering so receiver can expect data packets to come duplicated or 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 these guarantees.

As you can conclude from the discussion above, TCP is used in applications that requires reliability (such as email), while UDP usually finds it place in applications that requires fast data transfers and can handle occasional packet loss (such as VoIP or online gaming, for example).

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` schema, like it is shown in the following snippet:

```
udp://hostname:port?transportOptions
```

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

So, when should you use UDP instead of default TCP connector? There are basically two such situations:

- You broker is located behind a firewall you don't control and you can access it only over UDP ports.
- You have 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 this 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 our ActiveMQ to use both TCP and UDP protocol (on different ports, of course):

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

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 IONA JAVA"
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 IONA JAVA"
```

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

3.2.2.4. Secure Sockets Layer Protocol (SSL)

Imagine yourself in a situation where you need to expose your broker over the unsecured network and want to achieve data privacy. The same requirement emerged when the Web outgrew its academic roots and was considered for corporate usage. Sending of plain data over TCP became unacceptable and solution had to be found. The solution for secure data transfers was *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. In this respect, ActiveMQ implements the *SSL transport connector*, which adds an SSL layer over the TCP communication channel, providing an encrypted communication between brokers and clients.

The URI syntax for this protocol is:

```
ssl://hostname:port?transportOptions
```

and since it 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 *JavaTM Secure Socket Extension (JSSE)* to implement SSL functionality. Since its detailed description is out of the scope of this book, please be sure that you are familiar with JSSE (http://java.sun.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html) before proceeding to the rest of this section.

As you can see from a default configuration file, the SSL connector is configured to listen for connections on port `61617`. We can create a setup that will configure

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

only TCP and SSL transport protocols by changing the `<transportConnectors>` tag in the `${ACTIVEMQ_HOME}/conf/activemq.xml` file to the following:

```
<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 files in order to work properly. As you can assume, these files contain certificates for successful SSL communication. Basically, JSSE define two type 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*.

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

In order to successfully use SSL, you have to import broker's certificate into the client's truststore. If you plan to use the default broker certificate you'll need to prepare client keystore and truststore (containing broker's certificate) files. To save you some time, ActiveMQ comes with these two stores predefined for you and you can find them in the `client.ks` and `client.ts` files respectively. Finally, if you want to import broker's default certificate into your existing truststore, you can find it in the `broker-localhost.cert` file. It's important to say that in this default configuration, broker and client certificates are the same and the password to all stores is the "password" keyword.

Now, that we understand all necessary elements needed for successful SSL communication, let's see how we can connect to the secured broker. In the rest of

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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. If you try to connect our stock portfolio consumer in this way:

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

you can expect the following exception to be thrown:

```
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 entry:

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

This means that SSL connection could not be established and this is the error all clients will get when trying to connect to the "untrusted" broker.

In JSSE you provide your SSL parameters through the appropriate system properties. So in order to successfully connect to the broker over SSL we have to provide the keystore (with its appropriate password) and truststore our application should use. This is done with 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

- 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 IONA JAVA"
```

After providing these extra SSL-related parameters, we can see that our publisher connects successfully to the broker and works as intended. Of course, if your client is not located on the same computer as your broker, you'll have to copy these files and adapt paths to them.

Similarly, we can run our consumer with 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 IONA JAVA"
```

In this way we managed to set our applications to communicate with the broker over the encrypted network channels.

Working with default certificates is good for development purposes, but for a production system you'll want to use you're own certificates. In most cases you will buy an appropriate SSL certificate from the trusted certificate authority.

In some cases, however, 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 distributed with Java.

First of all, we have to create a keystore and a certificate for the broker. You can do this with the following command:

```
$ keytool -genkey -alias broker -keyalg RSA -keystore mybroker.ks
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
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>
 (RETURN if same as keystore password):
```

The `keytool` application will prompt 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. It could be 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 created a file named `mybroker_cert`, containing a broker certificate.

Now let's create a client keystore with the appropriate certificate. We can do that with command similar to the one we have used for creating 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
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
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 this command is the `myclient.ks` file with the appropriate certificate. Finally, we have to create the client truststore and import the broker's certificate in it. The keytool allows us to do 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 we created all necessary certificates, so we are ready to run our stock portfolio example using them.

For starters, we have to start a broker with this newly created certificate. One way to do it is to replace default files described above (`broker.ks` for example) with ones we just created. The other scenario includes passing of SSL-related system properties to the command we use to start our broker. We will use this approach in our example, so supposing that we created our certificates in the broker's `conf/` folder, we can start it like this:

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

Now let's see how to reflect these changes to our clients. If you try to run some of the client applications with the old certificate file, you will get the "unknown certificate" exception just as it was when we have tried to access the broker without using any certificate. So, you'll have to update your command to

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

something like this:

```
$ mvn \
 -Djavax.net.ssl.keyStore=/opt/apache-activemq-5.1-SNAPSHOT/conf/myclient.ks \
 -Djavax.net.ssl.keyStorePassword=test123 \
 -Djavax.net.ssl.trustStore=/opt/apache-activemq-5.1-SNAPSHOT/conf/myclient.ts \
 exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Publisher \
 -Dexec.args="ssl://localhost:61617 IONA JAVA"
```

in order to make it work. Here we have instructed our publisher to use newly created client stores. And of course, after these changes our stock portfolio application works again.

3.2.2.5. Hypertext Transfer Protocol (HTTP/HTTPS)

In many environments you can find strict firewalls in place, allowing only basic services (such as web an email) to be used. So how to use messaging and ActiveMQ in those environments?

Hypertext Transfer Protocol (HTTP) is originally designed to transmit hypertext (HTML) pages over the Web. It uses TCP as an underlying network protocol and adds an extra logic for communication between browsers and web servers. After the first boom of the Internet, web infrastructure and the HTTP protocol in particular found the 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 we transmit XML formatted data using the HTTP protocol rather then HTML pages (as it is the case with the "regular" Web).

ActiveMQ implements the *HTTP transport connector* which allows you to exchange XML-formatted messages with the broker over the HTTP protocol and thus bypass strict firewall restrictions.

The URI syntax of this protocol is:

```
http://hostname:port
```

or in case you want to use it over SSL:

```
https://hostname:port
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Let's now create an example configuration and see how to run our examples over the HTTP transport. The transport connectors section in this case looks very similar to those we used in previous sections:

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

Here we have set the standard TCP transport along with HTTP transport which, as you can see, listens to port 8080 (one of the standard HTTP ports).

In order to run our clients over the HTTP transport protocol, there is one more thing we should do and that is set our classpath correctly. The HTTP transport is located in the ActiveMQ optional module, so you'll have to add it to your application's classpath (along with appropriate dependencies). If you use Maven, you should add the following dependency to your project configuration:

```
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>activemq-optional</artifactId>
    <version>5.0.0</version>
</dependency>
```

In other case, be sure that along with `activemq-optional` JAR file you add the following JARs into your classpath:

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

Finally, we are ready to run our stock portfolio publisher over the HTTP transport:

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

As we already said, when you use this protocol all communication between broker

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

and clients is performed by sending XML messages. This certainly makes a great impact on the overall system performance compared to the pure TCP network and openwire data protocol tuned just for messaging purposes. So if performances are your main concern, you should stick to the TCP connector and find some other workaround for your firewall issues.

3.2.3. Using Virtual Machine protocols

Thus far we have covered protocols used to connect brokers and clients over the network. In some applications, however, you'll find that all clients will use the broker from the same Java Virtual Machine (JVM). So is there a real need to connect to the remote broker over the network in this usage scenario?

ActiveMQ allows you to embed a broker in your application and thus communicate with the broker "directly", instead of sending messages over the network. In order to support this kind of "in virtual machine" broker to clients communication, ActiveMQ provides a special protocol (and the appropriate transport connector). This connector will be the subject of the following section.

3.2.3.1. VM Protocol

The *VM transport connector* is used by Java applications to launch and connect to the embedded broker. In this case, no network connections are created between clients (producers and consumers) and the embedded broker. The whole communication is performed through direct method invocations of the broker object, and this, as you can assume significantly improves performances of the whole system. The broker is started at the moment when the first client tries to connect to the broker using the VM transport protocol. All subsequent VM transport connections from the same virtual machine will connect to the same broker (in case that they specify the same broker name, as we will see in the moment).

Of course, the broker created in this way doesn't lack any of the standard ActiveMQ features, so for example it 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 active connections to other applications), the

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

broker will automatically shut down.

The URI syntax for VM transport is as follows:

```
vm://brokerName?transportOptions
```

The broker name plays an important role in the VM transport connector URI, since it uniquely identifies the broker. For example, you can create two different embedded brokers by specifying different broker names.

Transport options are set as a query part of the URI, just as it was the case with other transports we covered. 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 starting with the "`broker.`" prefix are used to tune the broker, so for example the following URI:

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

starts the broker that will not persist received messages (message persistence is explained in the following chapter).

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

```
vm:broker:(transportURI,network:networkURI)/brokerName?brokerOptions
```

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 transports, which broker will use to connect to other Java applications. 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 additionally configured it to create a TCP transport connector that listens for

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

connections on port `6000`. Finally, we instructed this broker not to persist messages, through the appropriate transport option. Figure 3.2 can help us better visualize this configuration.

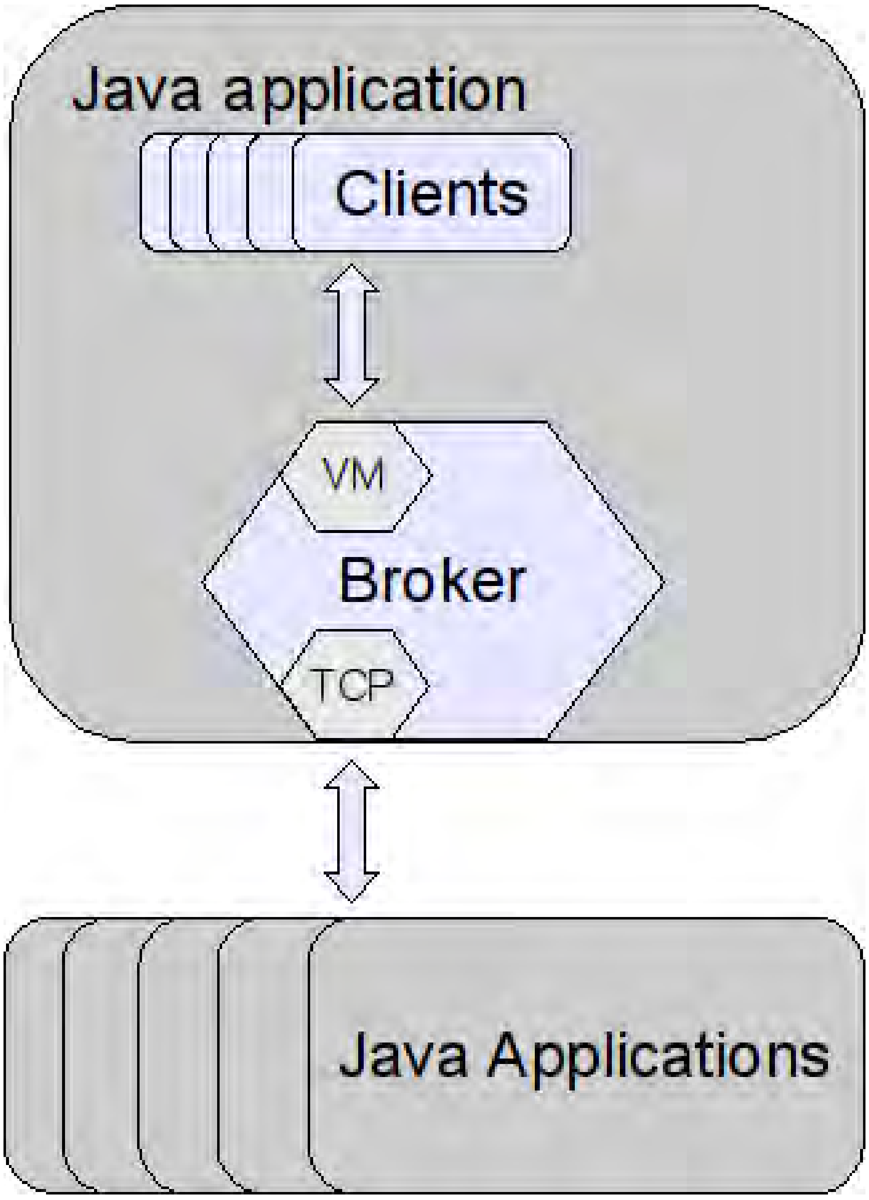


Figure 3.2. Figure 3.2: VM transport

Now, clients that connect from the application that embeds the broker will use the VM transport, while external applications can connect to that embedded broker

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

using the TCP connector (like they would do in case of any standalone broker).

Finally, those who want to configure the embedded broker using an external configuration file can achieve that using the `brokerConfig` transport option. You have to pass an URI of your `activemq.xml` file to this option and the transport will use it to configure the embedded broker. For example, the following URI:

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

will try to locate the `activemq.xml` file in the classpath. With this approach you can configure your embedded broker in the same way as you'd configure any other standalone broker.

Now it's time to start our stock portfolio publisher with an embedded broker. This is easily done with the following command:

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

As you can notice, everything work as expected, without any external broker. We'll let you configure this broker with the additional TCP transport and connect the portfolio consumer as an external Java application as an exercise.

At the end of this section we should discuss pros and cons of embedding a broker in your application. For starters, one obvious advantage of this approach is improved performances when communicating with local clients. Also, you'll have only one Java application to run (one JVM) instead of two, which can ease your deployment process. So, if you plan to use the broker mainly from one application, you should consider using the embedded broker.

On the other hand, if you have too many Java applications that use embedded brokers, you can run into maintenance problems trying to configure and backup data from all these brokers. In such situations it's always easier to create a small cluster of standalone brokers and configure applications to use them.

We'll continue exploring various embedding techniques for ActiveMQ in Chapter 6, Creating Java Applications With ActiveMQ.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

3.3. Configuring Network Connectors

Having one JMS broker to serve all your application needs is fine in case you have modest messaging needs. But some users need advanced features, like load balancing and high-availability, which can't be achieved with a single broker. ActiveMQ brokers can be connected into *networks of brokers* creating flexible clusters that meet these advanced messaging usage scenarios. Various topologies of these networks, their purpose and configuration details are explained in details in Chapter 9, Broker Topologies. In this section we will just briefly explain *network connectors*, used to network brokers, and protocols that support this broker-to-broker communication.

With this discussion, it's clear that we can divide ActiveMQ connectors on transport connectors that are used to connect clients to the broker and network connectors that establish communication channel between broker and other brokers in the corresponding network.

The network connection is a one way channel by default, meaning that the particular broker will only forward messages it receives to brokers on the other side of the connection. This setup is usually called a *forwarding bridge* and is sufficient for various topologies. In some situations, however, you'll want to create a bidirectional communication channel between brokers. ActiveMQ supports this kind of bidirectional connectors, which are usually referred to as *duplex connectors*. The Figure 3.3 shows one example of netwotk of brokers that contains both forwarding bridge and duplex connectors.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

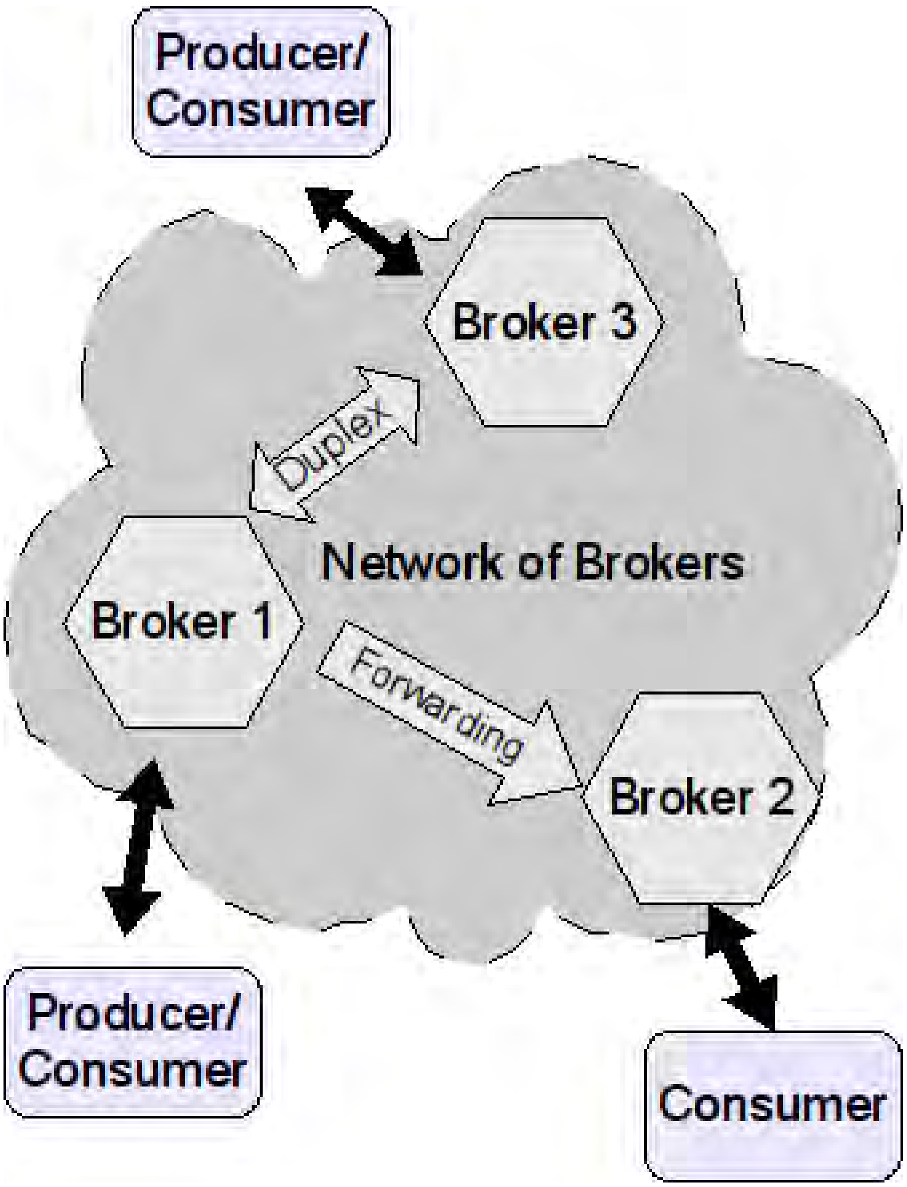


Figure 3.3. Figure 3.3: Network of Brokers

Network connectors are configured through the ActiveMQ XML configuration file in a similar fashion as it was the case with transport connectors. For starters, let's take a look at the default configuration (the actual snippet can vary depending on the ActiveMQ version):

```
<!-- The store and forward broker networks ActiveMQ will listen to -->
<networkConnectors>
  <!-- by default just auto discover the other brokers -->
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
    <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 inside the `<networkConnectors>` tag, which contains configuration for one or more connectors in an appropriate `<networkConnector>` tag. As was the case with transport connectors, mandatory attributes for network connectors are the `name` and `uri`. All other optional attributes are used to additionally tune the connector, as we will see in the moment.

3.3.1. Understanding Discovery Agents

In the rest of this chapter we will discuss various ActiveMQ protocols and techniques used to configure and connect to network of brokers. But before we start discussing particular protocols, there is one more important ActiveMQ concept we should explain. *Discovery*, in general, is a process of detecting remote broker services. Clients usually want to discover all available brokers they can connect to. Brokers, on the other hand, want to find other available brokers so they can create a network of brokers.

There are two approaches we can take in configuring and connecting our clients to the network of brokers. For starters, we can statically "wire" brokers and instruct our clients to connect to those brokers. Protocols used for this kind of ActiveMQ setup are explained in the following section. In the second scenario, brokers and clients can use specialized ActiveMQ concept called *discovery agents* to dynamically detect desired services they want to use. Discovery agents, and protocols they use, are explained in section 3.4.1.2, Dynamic networks.

3.3.1.1. Defining Static networks

As we have explained in the previous section, the first approach you can use to configure and connect to the network of brokers is through statically configured URIs. The only prerequisite for successfully employing static networks is that you

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

know addresses of all brokers you want to use. Now let's take a look at protocols we can use to create static networks of brokers.

3.3.1.1.1. Static Protocol

The *static network connector* is used for static wiring of multiple brokers into the network. This protocol uses a composite URI that consists of multiple broker addresses that are on the other end of the network connection.

The URI syntax for this protocol is:

```
static:(uri1,uri2,uri3,...)?transportOptions
```

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 snippet:

```
    <networkConnectors>
      <networkConnector name="local network"
        uri="static://(tcp://host1:61616,tcp://host2:61616)"/>
    </networkConnectors>
```

Assuming that this configuration is for the broker on the `localhost` and that brokers on hosts `host1` and `host2` are up and running, you will notice the following messages when you start your local broker:

```
INFO  DiscoveryNetworkConnector      - Establishing network connection between
      from vm://localhost to tcp://host1:61616
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://host1:61616 has been established.
INFO  DemandForwardingBridge         - Network connection between vm://localhost#2
      and tcp://host2:61616 has been established.
```

This means that you have successfully configured a *forwarding bridge* between your local broker and two other brokers in the network. In other words, all messages sent to your local broker will be forwarded to brokers running on `host1` and `host2`.

To demonstrate this, let's run a stock portfolio example, but now over the network

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

of brokers configured above. We will have our stock data publisher running and sending messages to the broker running on the local host. As we already said, all these messages are forwarded to brokers running on `host1` and `host2` machines, so we will consume messages from the broker located on `host1`. The following diagram gives you a better perspective of the broker topology used in this example (Figure 3.4).

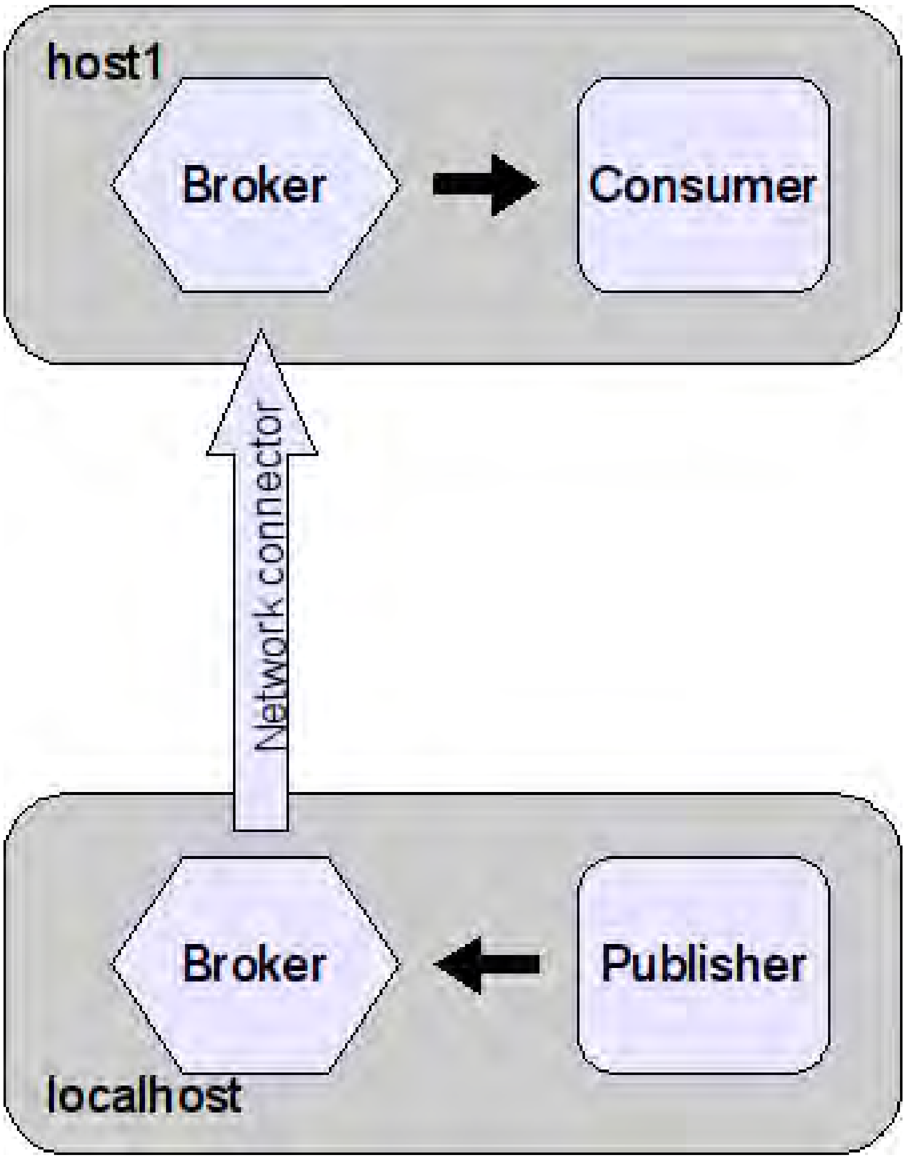


Figure 3.4. Figure 3.4: Static network

To make this example work, first we will start the publisher and instruct it to send

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

messages to the local broker:

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

This is practically the same command we have used for our original TCP connector example. Now let's start the consumer:

```
$ mvn exec:java -Dexec.mainClass=org.apache.activemq.book.ch3.Consumer \
-Dexec.args="tcp://host1:61616 IONA JAVA"
```

As the consumer is running on the `host1`, we could instruct it to use its local broker as well, but just for the example clarity we used `host1` explicitly. In this setup, messages are published to the broker on the localhost of our publisher. These messages are forwarded to the broker on the host1, where they are consumed by our consumer. The overall functionality of our example has not been changed and our publisher and consumer behaves the same as when we used the single broker. The only difference is that the publisher and consumer are now running on different hosts, while they still use only local brokers.

As we already said, use cases for different broker topologies will be discussed more thoroughly in Chapter 9, Broker Topologies. But from this simple example you can conclude that this particular configuration (usually called *store and forward*) can help you in situations when you need your distributed clients to benefit all performance advantages of communicating with the local broker.

3.3.1.1.2. Failover Protocol

In all examples thus far we have configured our clients 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, our stock portfolio example run over protocols described thus far were not immune to network problems and unavailable brokers. That's where protocols like *failover* comes in and implement a reconnection logic on top of low-level transport connectors described earlier. Like it was the case with network connectors, there are two ways you can use to provide a list of suitable brokers the client can connect to. In the first case, you'll provide a

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

static list of available brokers and that is approach used by the *failover transport connector*. The other way assumes dynamic discovery of available brokers and it will be explained in later sections.

Now let's dig into the failover transport connector. The URI syntax for this connector is very similar to the URI of its network connector counterpart, the static network connector:

```
failover:(uri1,...,uriN)?transportOptions
```

or

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

if you want to use the shorter syntax. 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 to connect to. If connection fails (both on startup or in a later stage), the transport will pick another URI and try to reconnect. A default configuration also implements a *reconnection delay logic*, meaning that the transport will start with 10 ms delay for the first reconnection attempt and double this time for any subsequent attempt up to 30000 ms. Also, the reconnection logic will try to reconnect indefinitely. Of course, all reconnection parameters could be reconfigured according to your needs with the appropriate transport options.

Remember the static network of brokers we have defined in the previous section. There, we had all messages sent to the local broker forwarded to the brokers located on `host1` and `host2`. So because all messages are sent to both of these brokers, we can consume from any of them. So, let's run our stock portfolio consumer and configure it to connect to one of these 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) IONA JAVA"
```

The beauty of this solution is that we didn't have to change our application in order

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

to support the reconnection logic. It came with the use of the appropriate transport connector.

Now let's see the failover connector at work. Imagine that the random algorithm has chosen to connect the consumer to the broker on `host1`. If that was the case, you can expect that your consumer prints 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 our publisher to the local broker will be forwarded to the broker on host1 and received by the consumer. Now we can try simulating what happens when this broker becomes unavailable. You can achieve this by shutting down the `host1` broker and see that 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
```

This means that our consumer has successfully reconnected to the other broker and as you can see it resumed its normal operation without any assistance from our side. You can use the failover transport even if you want to connect to a single broker. For example, the following URI:

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

tells us that the client will try to reestablish its connection in case the broker shuts down for any reason. This practically means that you don't have to restart your clients in case you shut down your broker for any reason (maintenance for instance). As soon as the broker becomes available again, clients will reconnect by themselves.

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, Broker Topologies.

3.3.1.2. Defining Dynamic networks

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

3.3.1.2.1. Multicast Protocol

IP multicast is a network protocol used for transmitting data from one source to a group of interested receivers. An application may join a certain group, or better say IP address in the range of `224.0.0.0` to `239.255.255.255`. After joining a group it will receive all data packets sent to that group. An application does not have to be a member of a group to send data to it.

ActiveMQ brokers use multicast protocol to advertise its services (such as transport connectors) and locate services of other brokers (in order to create networks). Clients, on the other hand, use multicast to find brokers they can connect to. In this section we will discuss how brokers use multicast, while client protocols based on multicast will be discussed later.

For starters, let's take a look at a snippet of the default ActiveMQ configuration:

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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

The one of the first things you can notice in this example is the URI syntax of the multicast protocol:

```
multicast://ipaddress:port?transportOptions
```

In this example we have used a *default* group name, instead of the specific IP address, which means that we 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, we used transport connector's `discoveryUri` attribute to join the default group and advertise this transport connector to interested parties. Thus, all clients interested in finding available broker would find this connector (as we will see in the following section).

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

At 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 the limitation of using the IP multicast protocol.

3.3.1.2.2. Discovery Protocol

Now that we know how to configure discovery for our brokers, let's see how we can instruct our client applications to use it. The *discovery transport connector* is "the client side" of the AcitveMQ 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

The syntax of this protocol is:

```
discovery:(discoveryAgentURI)?transportOptions
```

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

Now let's see how we can use the discovery connector in our applications. For starters, let's go back to our network of brokers example and say that out local host broker advertise its TCP transport connector using multicast. If that is the case, we can run our stock portfolio publisher with the following command:

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

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
```

which means that the application successfully used multicast to discover and connect to the local broker. As an additional exercise, we'll let you configure the network of brokers using multicast and use discovery for reconnection purposes instead of the failover protocol.

3.3.1.2.3. Zeroconf Protocol

The zeroconf protocol is an alternative protocol to multicast. The only difference between these two protocols is that zeroconf uses the Bonjour (previous called Rendezvous) protocol (http://developer.apple.com/networking/bonjour/faq.html) instead of IP multicast.

The URI syntax of this protocol is:

```
rendezvous://groupname
```

and you can find its complete reference at the following URL:

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at http://www.manning-sandbox.com/forum.jspa?forumID=496

http://activemq.apache.org/zeroconf-transport-reference.html

Now let's take a look at the following configuration snippet:

```
<transportConnectors>
    <transportConnector
         name="default"
         uri="tcp://localhost:61616"
         discoveryURI="rendezvous://group1"
    />
</transportConnectors>
<networkConnectors>
   <networkConnector
       name="default"
       uri="rendezvous://group1"
   />
</networkConnectors>
```

As you can assume, it has practically the same effect as the previously described multicast protocol.

Alternatively, you can set the zeroconf discovery agent by adding the following configuration snippet to your ActiveMQ configuration file:

```
  <discoveryAgent>
    <zeroconfDiscovery type="group1"/>
  </discoveryAgent>
```

Although it is good to have an alternative, multicast discovery is probably the one you will use (in case you need discovery at all).

3.3.1.2.4. Peer Protocol

As we have seen before, networks of brokers and embedded brokers are very useful ActiveMQ concepts that allows you to fit brokers to your infrastructure needs. Of course, it is theoretically possible to create networks of embedded brokers, but this could be cumbersome to configure. That's why ActiveMQ introduces the *peer transport connector* that allows you to network embedded brokers in an extremely easy way. 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:

```
peer://peergroup/brokerName?brokerOptions
```

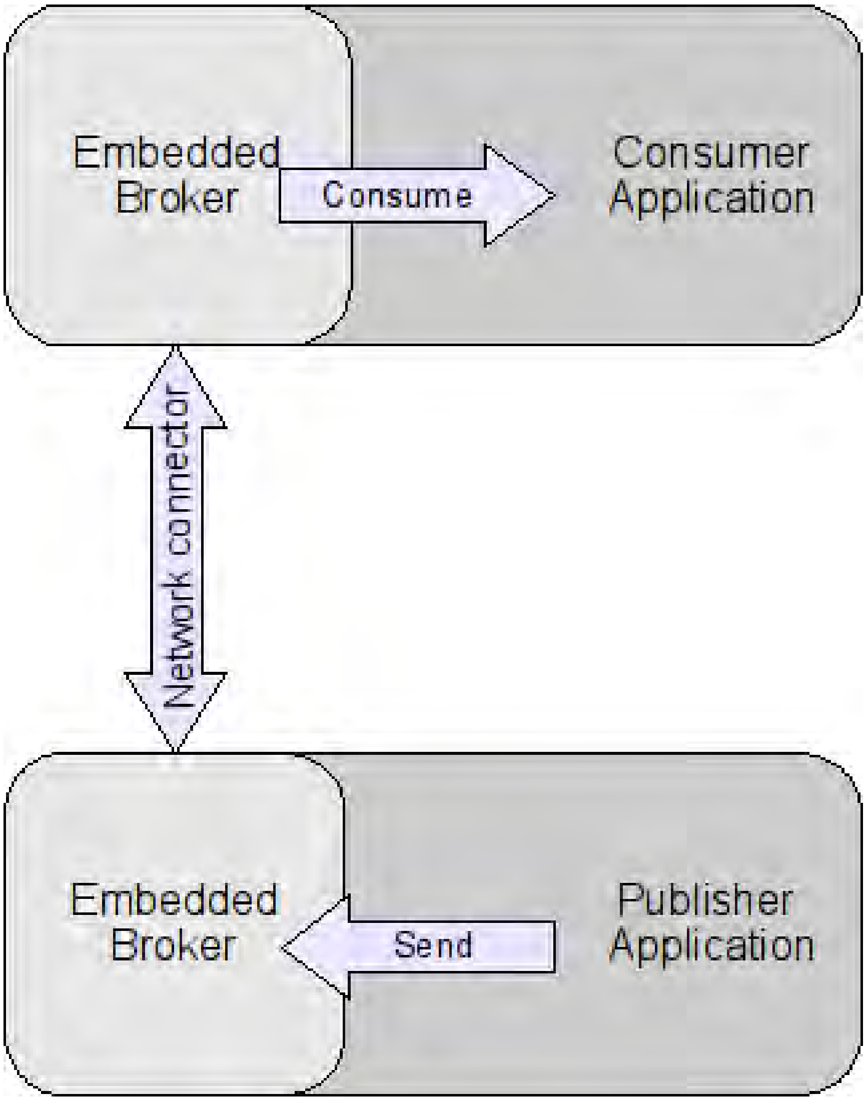
Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

and 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 start an embedded broker (just as was the case with the VM protocol), but it will additionally configure the broker to establish network connections to other brokers in the local network with the same group name.

Now let's see how we can run our stock portfolio example using the peer protocol. In this case, both our publisher and consumer will use their embedded brokers, that will be appropriately networked. The Figure 3.5 can give you a better perspective of this solution.



Download at Boykma.Com

Please post comments or corrections to the Author Online forum at http://www.manning-sandbox.com/forum.jspa?forumID=496

Figure 3.5. Figure 3.5: Peer protocol

So if we start our portfolio publisher application like this:

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

and the appropriate consumer application like this:

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

we basically started two embedded brokers (one in each application) and made a peer-to-peer network named `group1` with these two brokers. All messages sent to one broker will be available in another and possibly all other brokers that join the group. You can notice that the overall system works like these two applications use the same broker. We have already discussed pros and cons of using embedded brokers in section explaining the VM connector. All those advices are valid for the peer protocol as well.

3.3.1.2.5. 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:

```
fanout:(fanoutURI)?transportOptions
```

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

The `fanoutURI`, defined in the syntax, can be either static or multicast protocol URI. In the following example:

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

the client will try to connect to three brokers statically defined using the static protocol (Figure 3.6).

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

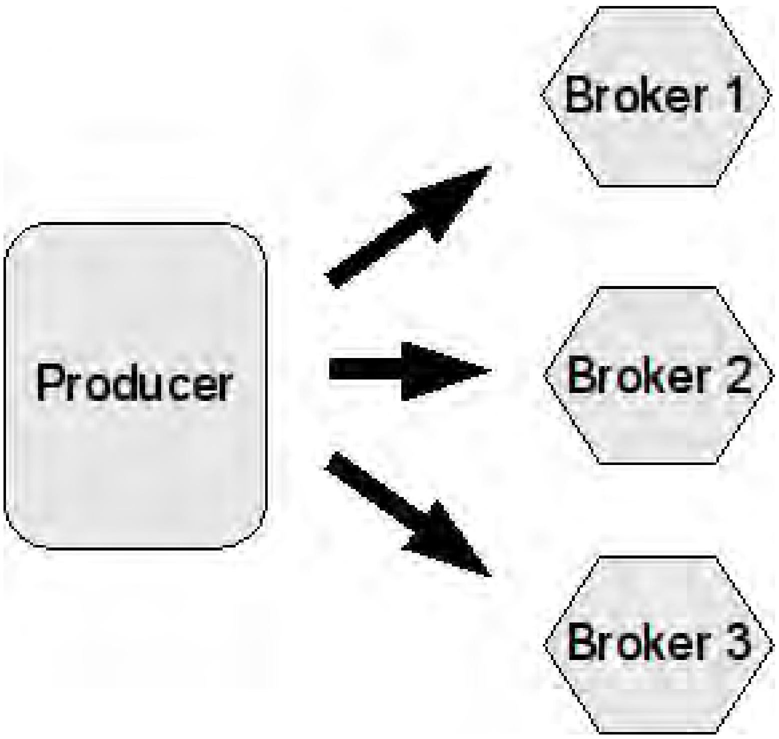


Figure 3.6. Figure 3.6: Fanout

The similar effect could be accomplished with the following URI:

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

assuming that brokers 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. It's 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

3.4. Summary

In this chapter we have learned various connectivity options ActiveMQ provides for client applications. We have seen how ActiveMQ URIs are formed and learned a difference between transport and network connectors. We have also described terms like embedded brokers and network of brokers and seen how we can 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, Broker Topologies. In the following chapter we will discuss message persistence. We will see when and how ActiveMQ persist messages in the store and what configuration options are provided for you.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Chapter 4. Message Persistence

The JMS specification supports two types of message delivery, persistent and non-persistent, but ActiveMQ can be configured to support message recovery - which is 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 your messages to survive even if there has been a system failure. One a message has been consumed and acknowledged by a message consumer, it is typically deleted from the broker's message store.

In this chapter we will explain what message persistence options there are and when to use them. We will cover:

- Why messages are stored differently for Queues and Topics
- How the different message storage options work and when to use them
- How the ActiveMQ can temporally cache messages for retrieval

This chapter will provide a detailed guide to message persistence, what the storage options are for ActiveMQ and when to use them.

4.1. How are messages stored for Queues and Topics?

It is important to get some basic understanding of the storage mechanisms for messages in an ActiveMQ message store - as it will help with configuration and understanding on what is happening in the ActiveMQ broker when it is delivering persistent messages.

Messages for Queues and Topics are stored slightly differently, and match delivery

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

semantics for the point-to-point and publish/subscribe message domains. Storage of Queues are straightforward, they are basically stored in first in, first out order (FIFO) - see Figure 4.1, with only one message being dispatched between one of potentially many consumers. When that message has been consumed, it can be deleted from the store.

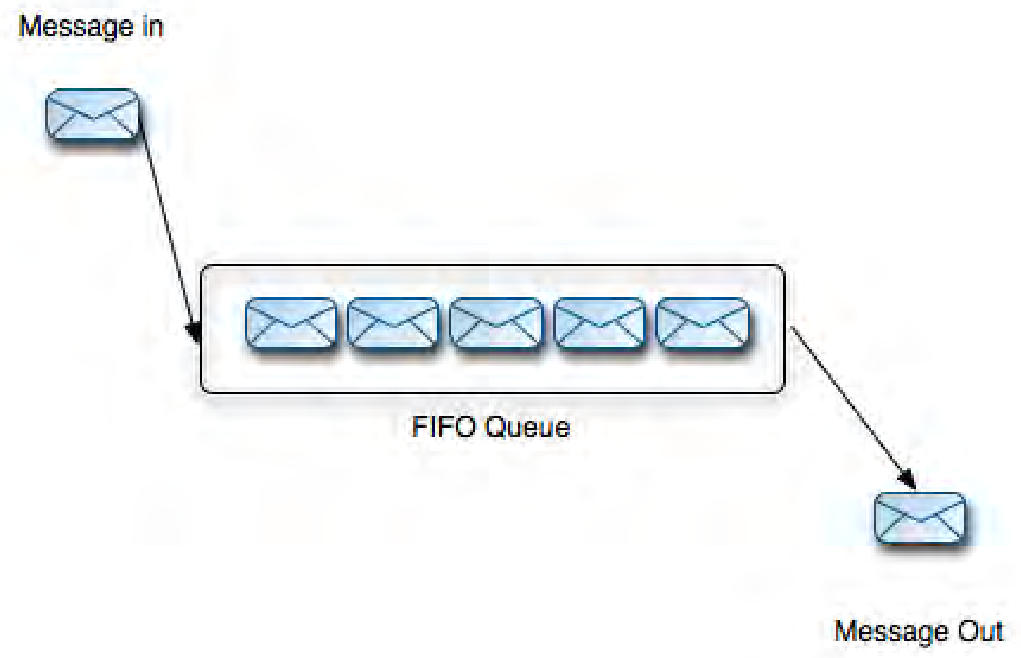


Figure 4.1. Figure 4.1: Messages Stored for Queues

For durable subscribers on a Topic, each consumer gets a copy of the message - but in order to save space (some messages can be very large!) messages are in fact only stored once. 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 4.2. A message store is implemented in this way for durable subscribers because durable subscribers will be consuming messages at different rates; and may not be active at the same time.

In addition, as every message can potentially have many consumers, a message cannot be deleted from the store until it has been delivered to every interested

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

consumer.

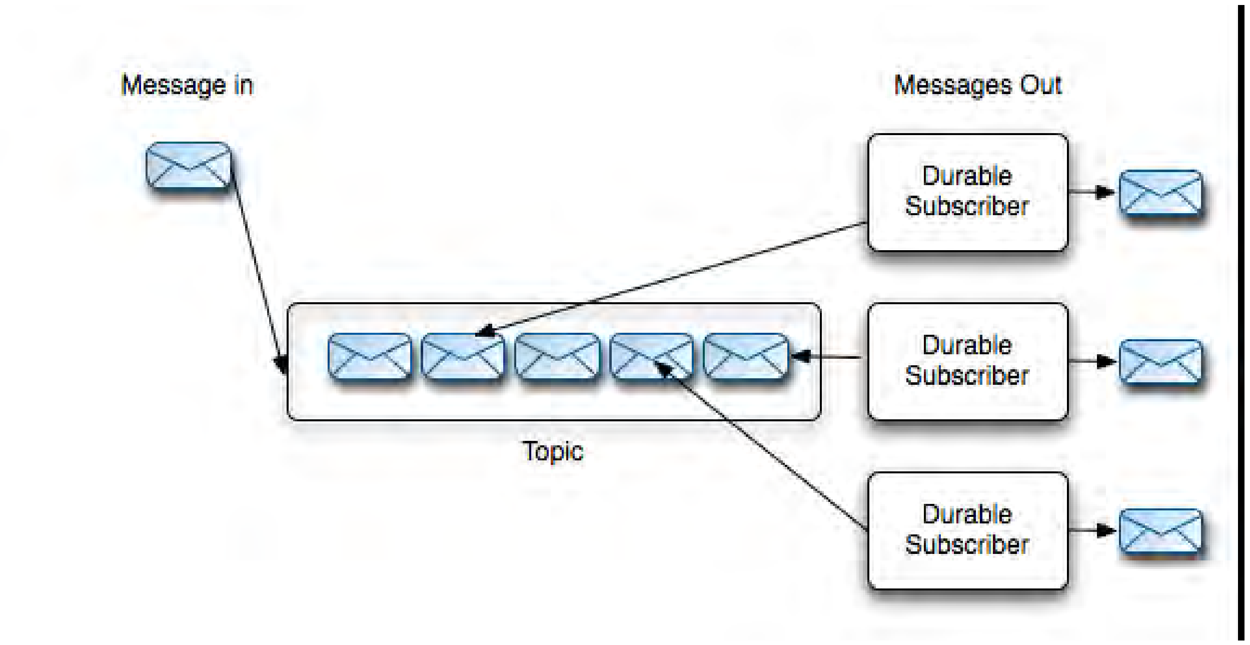


Figure 4.2. Figure 4.2: Messages Stored for Topics

Every ActiveMQ Message store supports persisting messages in this way, though obviously the implementation differs between storage types.

In the rest of this chapter we will be going into more detail about configuring the different ActiveMQ Message Stores and their advantages and disadvantages.

4.2. Configuring ActiveMQ Message Stores

ActiveMQ supports pluggable message stores - either relational databases using a JDBC connector (so embedded relational databases like JavaDB(Apache Derby) or Oracle, MySQL, Postgress etc) or a combination of both by using the high performance journal with a relational database. There is even an option to use an in-memory database too.

If you don't supply any persistence configuration for the message broker, the AMQ Message Store will be used with its default settings. To use an alternative message

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

store, or change the default behavior of the AMQ Message store, you need to configure the persistence adaptor by using the persistenceAdaptor element in the broker's configuration file e.g.

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

 </beans>
```

If you embed an ActiveMQ broker inside your application, you can also programmatically configure and set the message store:

```



import org.apache.activemq.broker.BrokerService;
import org.apache.activemq.store.amq.AMQPersistenceAdapterFactory;

   public class AMQStoreEmbeddedBroker {

      public void createEmbeddedBroker() throws Exception {

          BrokerService broker = new BrokerService();
          //initialize the PersistenceAdaptorFactory
          AMQPersistenceAdapterFactory persistenceFactory =
              newAMQPersistenceAdapterFactory();

          //set some properties on the factory
          persistenceFactory.setMaxFileLength(1024*16);
          persistenceFactory.setPersistentIndex(true);
          persistenceFactory.setCleanupInterval(10000);
          broker.setPersistenceFactory(persistenceFactory);

          //create a transport connector
          broker.addConnector("tcp://localhost:61616");
          //start the broker
          broker.start();
      }
   }
```

Every ActiveMQ message store implementation uses a *PersistenceAdaptor* object for access.*PersistenceAdaptor*s are usually initialized from a *PersistenceAdaptorFactory*. The above code snippet is creating a broker and

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

initializing the default Message Store implementation factory, the AMQPersistenceAdaptorFactory, setting configuration parameters, and initializing the broker with it.

In the next section we will go through in more detail about the default message store for ActiveMQ - the AMQ Message Store.

4.3. Using The AMQ Message Store

The default message store for ActiveMQ is the AMQ Message store - an embedded, 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 make ActiveMQ easy to use and as fast as possible. So the use of a file based message database makes ActiveMQ easy to run, as there is no pre-requisite of configuring a 3rd part database, enabling an ActiveMQ broker to be run as soon as its downloaded and installed.

In addition, the structure of the AMQ Message Store has been streamlined for specialist requirements of message storage.

We will now quickly go through the internals of the AMQ Message Store, which will help understand how to use and configure it.

4.3.1. How the AMQ Message Store works

What makes the AMQ Message Store so fast is the combination of a fast transactional journal together with optimized message id indexes - that reference the message data in the journal, combined with in memory message caching - as can be seen in Figure 4.3

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

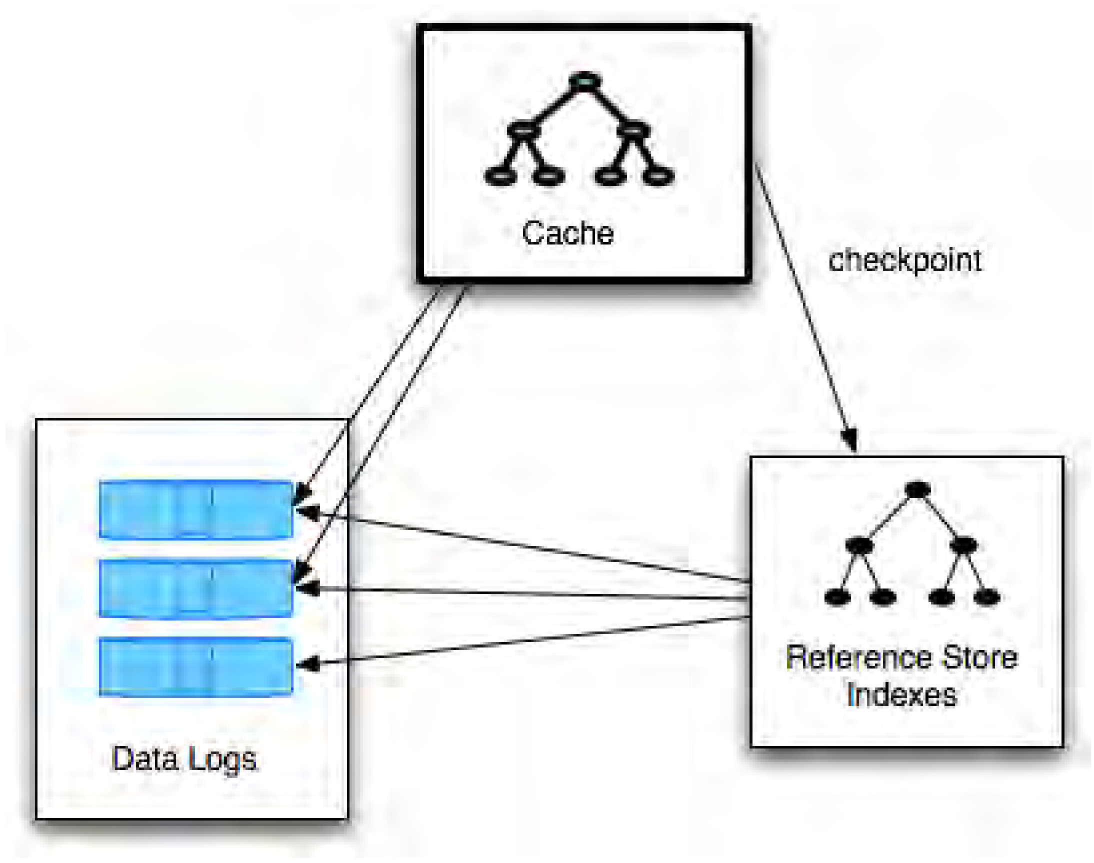


Figure 4.3. Figure 4.3: The AMQ Message Store

There are three distinct parts of the AMQ Message Store - these are:
It is important to understand the file based directory structure used by the ActiveMQ Message Store - this will help in configuration and problem identification when using it.

- The Journal, which consists of a rolling log of messages and commands (such as transactional boundaries and message delete) stored in data files of a maximum length. When the maximum length of the currently used data file is reached a new one is started. All the messages in a data file are reference counted, so once every message in that data file are 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

- The Cache, which 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 is called a checkpoint. Once the Reference Store is updated, messages can be removed from the Cache. The period of time between the Cache updating the Reference Store is configurable and can be set by the checkpoinInterval property. A checkpoint will also happen if the ActiveMQ Message Broker is reaching its memory limit.

- The Reference Store - hold references to the messages in the Journal - which 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. Which type of Index to use is configurable, the default being a disk based Hash Index. It is also possible to use an in-memory HashMap too, but this is only recommended if the total number of messages you expect to hold in the Message Store is less than 1 million at any particular time.

4.3.2. The AMQ Message Store directory structure

When you start the broker with the AMQ Message Store it will automatically create a directory data structure to store the persistent messages in. The AMQ Message Store data directory contains sub-directories for all the brokers that are running on the system - for this reason it is strongly recommended that you give your broker an easily recognizable but unique name. In the default configuration for ActiveMQ the broker name is 'localhost' - and this is represented in Figure 4.4 - the AMQ Store directory structure.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

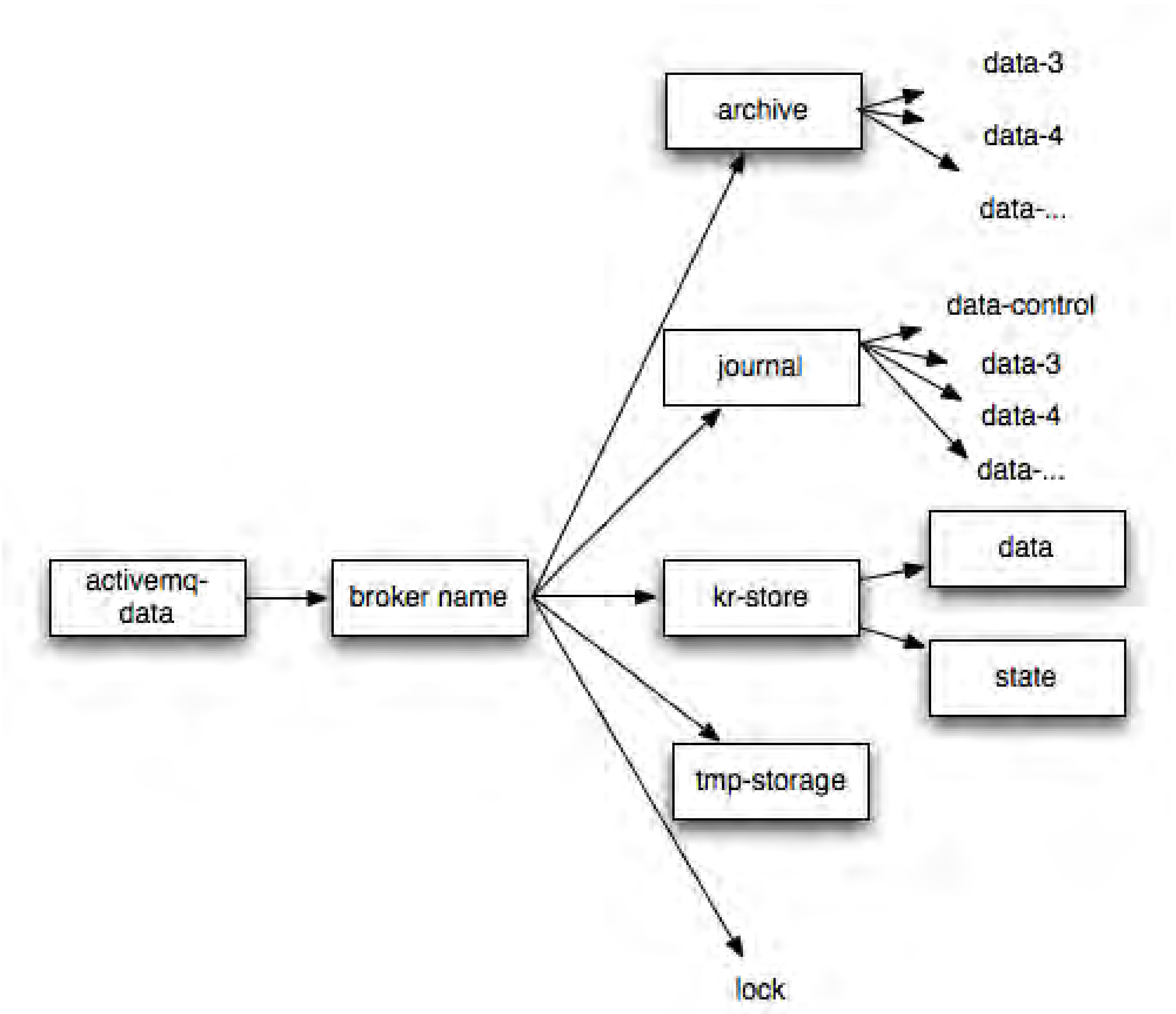


Figure 4.4. Figure 4.4: The AMQ Message Store Directory Structure

Under the broker name directory (localhost) the following can be found:

- the data directory - which 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- the state directory holds information about durable topic

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 a time - this can be used for hot-standby, have more than one broker with the same name on the same system, and only one will get the lock and start, while the other(s) are in stand-by mode.

- a temp-storage directory - this is used for storing non-persistent messages that can no longer be stored in broker memory - typically awaiting delivery to a slow consumer.

- The kr-store - this is the directory structure used by the reference (index) part of the AMQ Message Store. It uses the Kaha database by default (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 - which contains the data files for the journal, and a data-control file - for some meta information. The data files are reference counted, so when all the messages they contain are delivered, the data file can be deleted (or archived).

- The archive directory - 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 with as its location. It will automatically replay the data logs in the journal.

Now we know the basics, we can delve in to the configuration!

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

4.3.3. Configuring the AMQ Message Store

The AMQ Message Store is highly configurable, and allows the user to change its basic behaviour around indexing, checkpoint intervals and the size of the Journal data files.

key properties include:

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	4kb	the maximum memory used for a transaction before writing to disk
cleanupInterval	3000(ms)	time before checking which journal data files are still required
checkpointInterval	20000(ms)	time before moving cached

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

property name	default value	description
		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 not shutdown cleanly - this errs on the side of extreme caution
forceRecoverReferenceStore	false	force a recovery of the reference store

For example, to set some of these properties, your ActiveMQ broker XML configuration will look like:

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

4.3.4. When to use the AMQ Message Store

The AMQ Message store is the default message store of ActiveMQ and balances 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 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 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.

4.4. The JDBC Message Store

The flexibility of the ActiveMQ pluggable Message Store strategy allows for a large choice through the Java Database Connectivity (JDBC) option for messaging persistence.

By default, ActiveMQ is configured for Apache Derby when using the JDBC Message Store, but there are many other databases that ActiveMQ has been tried and tested with.

4.4.1. Databases supported by the JDBC Message Store

This isn't an extensive list, but ActiveMQ has been shown to work with the following database implementations:

- Apache Derby
- MySQL

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

- PostgreSQL
- Oracle
- SQLServer
- Sybase
- Informix
- MaxDB

It can be useful to use SQL to query the database you choose to examine how the messages are stored - and we will cover that now.

4.4.2. How the JDBC Message Store tables are used

The JDBC message store uses two tables for storing messages and a lock table, to ensure only one ActiveMQ broker can access the broker at one time.

One of the benefits of using a relational database is that it is relatively straight forward to use an SQL query find information about which messages are stored. For this reason we are going to outline the message related tables here.

The Message Table by default is named *ACTIVEMQ_MSGS* defined as follows:

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 is used
MSGID_SEQ	INTEGER	the producer sequence number for the message.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

column name	default type	description
		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 inserted in to this 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 called in the *ACTIVEMQ_ACKS* which is defined as follows:

column name	default type	description
CONTAINER	VARCHAR(250)	the destination
SUB_DEST	VARCHAR(250)	the destination of the durable subscriber (could 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

column name	default type	description
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 pointer into the *ACTIVEMQ_MSGS* and enables messages for a particular durable subscriber to be selected from the *ACTIVEMQ_MSGS* table.

We will now go through some examples of configuring JDBC message stores.

4.4.3. Configuring the JDBC Message Store

Its very straight forward to configure the default JDBC Message Store (which uses Apache Derby) in the broker configuration:

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

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

  </broker>

</beans>
```

One of the key properties on the JDBC persistence adaptor (the interface onto the JDBC Message Store) is the dataSource property - which defines a factory to define connections to a physical relational database. Configuring the dataSource object enables the JDBC persistence adaptor to use different physical databases other than the default (Apache Derby).

Below is an example of configuring an ActiveMQ broker to use the JDBC Message Store with a MySLQL database. Note that we use an apache commons DataSource implementation to define the connectivity.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
<?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>
```

This is an example of using Oracle as the dataSource property:

```
<?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"/>
        <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>
```

We have shown some example configurations for the JDBC Message Store and we look at when it is best to use this type of persistence.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

4.4.4. When to use the JDBC Message Store

The JDBC Message Store often chosen when there is already expertise in the administration of a relational database. Being embedded, Apache Derby (the default for this Message Store type) offers superior performance.

However, the use of a shared database is particularly useful for making a redundant master/slave topology of 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 (who becomes the master) will succeed. 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.

4.4.5. Using JDBC Message Store with the ActiveMQ Journal

It is possible to use the same ActiveMQ Journal technology that is part of the ActiveMQ message store with a relational database.

The Journal ensures the consistency of JMS transactions; and 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 - in this case we are using the Apache Derby Data Source.

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

The Journal can be used with any JDBC Data Source but its important to know when it should and shouldn't be used and we cover that now.

4.4.6. When to use the JDBC Message Store and Journal

The Journal offers considerable performance advantages over just using a 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 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.

4.5. The Kaha Message Store

The Kaha Message Store is another file based message embedded database like the AMQ Message Store. It predates the AMQ Message Store, and is faster by about 50% - but the speed comes at a cost of unreliability.

The Kaha Store is very straight forward to use, and this is demonstrated in its limited configuration.

4.5.1. Configuring the Kaha Store

The Kaha Message Store actually offers few options for configuration:

property name	default value	description
directory	activemq-data	the directory path used by the AMQ Message Store
maxFileLength	32mb	the maximum size of the message journal data files

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

property name	default value	description
		before a new one is used
persistentIndex	true	persistent indexes are used. If false - an in memory HashMap is used

Configuration for :

```
<?xml version="1.0" encoding="UTF-8"?>
  <beans>

     <broker brokerName="test-broker"
             xmlns="http://activemq.apache.org/schema/core">
       <persistenceAdapter>
         <kahaPersistenceAdapter directory="target/Broker2-data/activemq-data"
           maxFileLength="10mb" />
      </persistenceAdapter>

    <broker>

  </beans>
```

4.5.2. When to use the Kaha Store

The AMQ Message Store incorporates a transactional journal, so its state can be recovered, whilst the Kaha Message Store does not. However, if your main aim is only to use persistence for delivering messages to consumers at a later point and you are not so concerned about recovery (you may be using a master/slave configuration) - this store could meet your needs.

4.6. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

messages that may exist in this message store at one time.

4.6.1. Configuring the Memory Store

Configuration of the Memory Store is very straight forward, its the implementation used when you set the broker property persistence=false e.g.

```
<?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>
```

Embedding an ActiveMQ broker with the memory store is even easier - the following code 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();
  }
```

4.6.2. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

Now we have gone through the different message storage options for persistent options, we will cover a specialized case for caching of messages in the ActiveMQ broker for non-durable Topic subscribers.

4.7. Caching Messages in the Broker for Consumers

Although one of the most important aspects of message persistence is that the messages will survive long term storage, there are a number of use cases where you require messages to be available for consumers that were disconnected from the broker, but if having the messages persisted in a database is too slow. Real-tme data delivery of price information for a trading platform is a good example of this.

Typically real-time data applications use messages that are only valid for a finite amount of time (often less than a minute) so its pointless having them persisted to survive a system outage.

ActiveMQ supports caching of messages for these types of systems using message caching in the broker, using a configurable policy for deciding which types of messages should be cached, how many and for how long.

4.7.1. How Message Caching for Consumers Works

The ActiveMQ message broker cache 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 done 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

destination when the Topic Consumer is created:

```
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);
        //mark that consumers will be retroactive
        Topic topic = session.createTopic("TEST.TOPIC?consumer.retroactive=true");

        MessageConsumer consumer = session.createConsumer(topic);
    }
```

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*. We will walk through the different *subscriptionRecoveryPolicy*s in the next section.

4.7.2. The ActiveMQ *subscriptionRecoveryPolicys*

There are a number of different policies which allows us to fine tune how long and what types of messages are cached for non-durable Topic consumers. We will discuss them here:

4.7.2.1. The ActiveMQ *FixedSizeSubscriptionRecoveryPolicy*

This limits the number of messages cached by the Topic by the amount of memory they use - and is the default in ActiveMQ. You can choose to have the cache limit applied to all Topics, or on an individual by individual Topic basis. The properties available to configure are:

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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

4.7.2.2. The ActiveMQ *FixedCountSubscriptionRecoveryPolicy*

limits the number of messages cached by the Topic by the number of messages in the Topic's cache. There is only one property to configure:

property name	default value	description
maximumSize	100	the number of messages allowed in the Topics cache

4.7.2.3. The ActiveMQ *QueryBasedSubscriptionRecoveryPolicy*

This limits the number of messages cached by a JMS properties selector applied to the messages. There is only one property that needs to be configured:

property name	default value	description
query	null	caches only messages that match the query

4.7.2.4. The ActiveMQ *TimedSubscriptionRecoveryPolicy*

This limits the number of messages cached by the Topic by an expiration time that is applied to the message. The expiration time is independent from the timeToLive

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

set by the *MessageProducer*

property name	default value	description
recoverDuration	60000	the time in milliseconds to keep messages in the cache

4.7.2.5. The ActiveMQ *LastImageSubscriptionRecoveryPolicy*

This policy holds the last message sent to a Topic only. It can be useful for real-time price information, where if you used a price per Topic, you would only want the last price sent to that Topic.

There are no configuration properties for this policy.

4.7.2.6. The ActiveMQ *NoSubscriptionRecoveryPolicy*

This policy disables caching for Topics in the ActiveMQ Message broker.

There are no properties to configure for this policy.

4.7.3. Configuring *SubscriptionRecoveryPolicy* for the Message Broker

You can configure the *SubscriptionRecoveryPolicy* for either individual Topics, or you can use *wild cards*, in the ActiveMQ broker configuration XML - an example of which 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>
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
        <policyEntries>
          <policyEntry topic="Topic.FixedSizedSubs">
            <subscriptionRecoveryPolicy>
              <fixedSizedSubscriptionRecoveryPolicy maximumSize="2000000"
                useSharedBuffer="false"/>
            </subscriptionRecoveryPolicy>
          </policyEntry>

          <policyEntry topic="Topic.LastImageSubs">
            <subscriptionRecoveryPolicy>
              <lastImageSubscriptionRecoveryPolicy/>
            </subscriptionRecoveryPolicy>
          </policyEntry>

          <policyEntry topic="Topic.NoSubs">
            <subscriptionRecoveryPolicy>
              <noSubscriptionRecoveryPolicy/>
            </subscriptionRecoveryPolicy>
          </policyEntry>

          <policyEntry topic="Topic.TimedSubs">
            <subscriptionRecoveryPolicy>
              <timedSubscriptionRecoveryPolicy recoverDuration="25000"/>
            </subscriptionRecoveryPolicy>
          </policyEntry>

        </policyEntries>
      </policyMap>
    </destinationPolicy>
  </broker>

</beans>
```

4.8. Summary

In this chapter we have seen how messages are stored differently for Queues and Topics and the different message stores.

We have learned about the flexibility of ActiveMQ persistence storage, how to configure different stores and when and where to use them.

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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Chapter 5. Securing ActiveMQ

The first time you step outside of your trusted environment, you'd want to secure access to your JMS broker and specific queues and topics. For that reason ActiveMQ provides a flexible and customizable security model that can be adapted to the security mechanisms used in your environment.

In this chapter, we will go through:

- 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 your brokers and plug them into your existing security infrastructure.

5.1. Introducing Basic Security Concepts

Before we start discussing how to secure ActiveMQ broker, let's briefly recapitulate basic terms related to the security and how they fit the ActiveMQ security model.

Authentication is a process of obtaining a credentials of a certain entity. Since we are talking about Java applications (producers and consumers) in the context of ActiveMQ, it is basically a process of identifying Java applications trying to connect to our broker. As in other realms of information technology, applications usually provide username and password in order to identify themself and in case of wrong credentials their attempt to make connection will be refused.

Once successfully authenticated and connected, the application will try to access

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

certain resources. That's where *authorization* takes place and determines whether an entity is eligible to use a resource in an intended way. In ActiveMQ terminology, this means that in this process we decide which applications can publish, consume or create new destinations (queues and topics) in our broker.

After this brief reminder, let's take a look at some practical examples of AcitveMQ security mechanism configurations.

5.2. Authenticating Clients

All security concepts in ActiveMQ are implemented as plugins, which makes it easy to configure and customize them. This also means that authentication and authorization details are configured inside the special `<plugin>` tag of the ActiveMQ configuration file. There are two plugins built into ActiveMQ that you can use off the shelf to authenticate users:

- Simple authentication plugin - allows you to specify credentials directly in the xml configuration file
- JAAS authentication plugin - implements "Java Authentication and Authorization Service" API and thus provides more powerful and customizable solution

5.2.1. Configuring Simple Authentication Plugin

The easiest way to secure your broker is by authentication credentials directly in the broker's xml configuration file. Let's take a look at the following example:

```
<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                         #A
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
         username="admin"                        #A
         password="password"                     #A
         groups="admins,publishers,consumers"/>  #A
     <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>

 #A Define Authentication User
```

By using this simple configuration snippet, We've added four users that can access ActiveMQ broker. As you can see, for each user you have to define username and password for authentication purposes. Additionally, you'll provide the `groups` attribute which contains comma-separated list of groups user belongs to. This information is used for authorization purposes, as we will see in the moment.

Now, let's see how this configuration affects our stock portfolio application from chapter two. First, we need to start a broker with the configuration file defined above:

```
${ACTIVEMQ_HOME}/bin/activemq \
xbean:src/main/resources/org/apache/activemq/book/ch5/activemq-simple.xml
```

If you now try to run the stock publisher defined in chapter 2, you will get the following exception:

```
Exception in thread "main" javax.jms.JMSException:
  User name or password is invalid.
```

We could expect such behavior, since we have a security plugin activated and didn't define any authentication credentials for the publisher. We can now modify the publisher in the following manner:

```
    private String username = "publisher";
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
    private String password = "password";

    public Publisher() throws JMSException {
        factory = new ActiveMQConnectionFactory(brokerURL);
        connection = factory.createConnection(username, password);
        connection.start();
        session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        producer = session.createProducer(null);
    }
```

As you can see, the only difference is that we have defined username and password to be used and passed them as attributes to the `createConnection()` method. If we run publisher modified in this manner, we can expect it to work properly.

5.2.2. Configuring JAAS Plugin

The simple security plugin is a proper solution for simple use cases and usage in the development environment. However, in a production environment you will probably want to plug ActiveMQ into existing security infrastructure. This is where JAAS plugin, implementing "Java Authentication and Authorization Service" API, can help you achieve your goal. Detailed explanation of JAAS is beyond the scope of this book, so instead we will briefly introduce basic concepts and show how `PropertiesLoginModule` can be used to achieve the same functionality as we had with the simple security plugin (for more information about JAAS please refer to 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 (text files, relational database, LDAP, etc.). All we have to do is to make implementation of `javax.security.auth.spi.LoginModule` interface (http://java.sun.com/javase/6/docs/api/javax/security/auth/spi/LoginModule.html) and configure ActiveMQ to use one. Fortunately, ActiveMQ comes with built-in implementations of modules that can authenticate users from properties files, LDAP and using SSL certificates, which should be enough for most use cases.

Now, let's see how we can configure `PropertiesLoginModule` and instruct ActiveMQ to use it. First of all, we need to create a `login.config` file which is the

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

standardized way of configuring JAAS (http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/tutorials/LoginConfigFile.h

```
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";
};
```

Here, we defined an `activemq-domain` which uses properties login module and configures it to use `users.properties` and `groups.properties` file for its initialization. Now let's see how to define user credentials in these properties files. The `users.properties` file defines each user in a separate line along with its password, like show in the following example:

```
admin=password
publisher=password
consumer=password
guest=password
```

The `groups.properties` file define a comma-separated list of groups for each user:

```
admins=admin
publishers=admin,publisher
consumers=admin,publisher,consumer
guests=guest
```

After we have set all configuration files for JAAS, we can configure the JAAS plugin:

```
  <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>
      <jaasAuthenticationPlugin configuration="activemq-domain" />
    </plugins>
  </broker>
```

As you can see, the JAAS plugin only needs the name of the JAAS configuration

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

entry. The ActiveMQ will try to locate `login.config` file on the classpath or will use `java.security.auth.login.config` system property if it is set. In the following example, you can see how ActiveMQ could be started using the JAAS configuration:

```
${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
```

After this, we have a broker secured exactly like it was in the previous section, only using different mechanisms. You can try all examples again and broker should be behaving exactly the same as in the previous section.

5.3. Authorizing Operations

Knowing how to authenticate applications connecting to our broker, we go further and see how to set a fine-grained security for certain destinations. For starters, take a look at the following configuration snippet:

```
  <plugins>
   <jaasAuthenticationPlugin configuration="activemq-domain" />
   <authorizationPlugin>
    <map>
     <authorizationMap>
      <authorizationEntries>
       <authorizationEntry topic=">"
                           read="admins"
                           write="admins"
                           admin="admins" />
       <authorizationEntry topic="STOCKS.>"                              #A
                           read="consumers"                              #A
                           write="publishers"                            #A
                           admin="publishers" />                         #A
       <authorizationEntry topic="STOCKS.JAVA"
                           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>
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
#A Authorization entry
```

Here, we have defined an authorization plugin and provided it with a map of authorization entries. The first thing we should define for an entry is the destination which we want secure as a `topic` or `queue` attributes. Next, we want to declare which user groups have particular privileges on that destination. There are three authorization levels you can set (matching the appropriate attribute):

- read - allows specified groups to consume messages from the destinations
- write - allows specified groups to publish to the destinations
- admin - allows specified groups to lazily create destinations in the destination subtree

The destination values can be specified using wildcards, so the `STOCKS.>` basically means that entry applies to all destinations in the `STOCKS` path (starting with `STOCKS.`). Also, authorization operations accepts either a single group or comma-separated list of groups as a value.

So, 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 `STOCKS.JAVA` topic

In order to start the broker with the authorization plugin configured, run the following command:

```
${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 that we have JAAS configuration here as well, since we are using the JAAS plugin for authentication purposes.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Now we can try using our examples (we will use stock portfolio consumer in particular) and see how they behave with this new configuration. For starters we need to modify it the similar manner as it was the case with the producer in the section 5.3.1 Simple Authentication Plugin.

```
    private String username = "guest";
    private String password = "password";

    public Consumer() throws JMSException {
     factory = new ActiveMQConnectionFactory(brokerURL);
     connection = factory.createConnection(username, password);
     connection.start();
     session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    }
```

We have added credentials to be used and created a connection with appropriate username and password. You can try starting 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.IONA` topic, you will get the following exception

```
Exception in thread "main" javax.jms.JMSException:
User guest is not authorized to read from: topic://STOCKS.IONA
```

which is exactly what is expected to happen. Consuming from STOCKS.JAVA topic will be performed without any restrictions.

5.4. Implementing Message Level Authorization

In previous sections of this chapters we have learned how to authenticate users and authorize access to particular destinations of our broker. However, as you could see, authorization of access was granted or denied in the process of connection creation. In some situations you might want to authorize users (applications) to access not whole destination but particular messages. In this section, we will see how message level authorization could be achieved in ActiveMQ.

We will implement a simple authorization plugin that allows only applications

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

running on the same host as the broker (localhost) to consume messages. The first thing we need to do is make 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                                        #1
 (ConnectionContext context, Message message)                             #1
 {
      LOG.info(context.getConnection().getRemoteAddress());
      if (context.getConnection().getRemoteAddress()
             .startsWith("/127.0.0.1")) {
        return true;
      } else {
        return false;
      }
 }

}
```

As you can see, the `MessageAuthorizationPolicy` interface is simple and defines only one `isAllowedToConsume()` method #1. This method has an access to the message in question and the context of the connection in which the message has been consumed. In this example, we have used a remote address property of the connection (identifying a remote application trying to consume a message and made a decision whether this access is allowed. Of course, you can use any message property or even a content to make this decision.

Let's see how to install and configure this plugin with our ActiveMQ broker. The first and obvious step is to compile this class and package it in the appropriate JAR archive (usually along with other AcitveMQ extensions). Place such JAR into the `lib/` folder of your ActiveMQ distribution and the policy is ready to be used. The policy is configured by instantiating a bean (an instance of our class) inside the `<messageAuthorizationPolicy>` tag of the xml configuration file:

```
<messageAuthorizationPolicy>
   <bean class="org.apache.activemq.book.ch5.AuthorizationPolicy" xmlns="" />
</messageAuthorizationPolicy>
```

Now we are ready to start a broker, using the appropriate configuration file:

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
${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.

5.5. Building Custom Security Plugin

Thus far we have seen how you can work with built-in security mechanisms which should provide enough functionality for majority of users. However, in case you have security requirements that can not be met by these mechanisms, you can easily customize ActiveMQ to support whatever you need. Basically, you have two choices:

- Implement JAAS login module and configure it with JAAS plugin - there is a good chance that you already have JAAS used in your Java infrastructure, so in this case it's 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 go deeper into this topic.

- Implement custom security plugin - ActiveMQ provides a very flexible plugin mechanism used among other things to implement security plugins described in the previous section. So if you have requirements that cannot be met by implementing a JAAS module, writing your own plugin is a way to go.

In this section we will describe how to write a simple security plugin that allows connecting to the broker only from a certain set of IP addresses. The first thing you need to do to create a plugin is to extend `org.apache.activemq.broker.BrokerFilter` class. Let's start with an example and add explanations later:

```
public class IPAuthenticationBroker extends BrokerFilter {

 List<String> allowedIPAddresses;
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
 Pattern pattern = Pattern.compile("^/([0-9\\.]*):(.*)");

 public IPAuthenticationBroker(Broker next, List<String> allowedIPAddresses) {
  super(next);
  this.allowedIPAddresses = allowedIPAddresses;
 }

 public void addConnection(ConnectionContext context,                         #1
   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);
 }

}
```

`BrokerFiler` class defines methods that allows you to intercept broker operations such as add a connection, remove a subscriber, etc. (for a full list of operations please refer to the JavaDocs). All you have to do is to override methods of your interest and implement a logic of your plugin. In this example we have implemented the `addConnection()` method #1 and added a logic that checks whether a remote address (an address of a connecting client) in a list of allowed IP addresses. In case that connection is accepted, the call will be delegated to the broker or thrown exception in other case. One more thing is worth noting in this example; the `BrokerFilter` class has a constructor that accepts a broker as an argument so be sure to call it from your own.

After the plugin logic has been implemented by the broker filter, we need a mechanism to configure and install it. For that purpose we have to make an implementation of the `org.apache.activemq.broker.BrokerPlugin` interface.

```
public class IPAuthenticationPlugin implements BrokerPlugin {
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
 List<String> allowedIPAddresses;

 public Broker installPlugin(Broker broker) throws Exception {        #1
  return new IPAuthenticationBroker(broker, allowedIPAddresses);      #1
 }                                                                    #1

 public List<String> getAllowedIPAddresses() {
  return allowedIPAddresses;
 }

 public void setAllowedIPAddresses(List<String> allowedIPAddresses) {
  this.allowedIPAddresses = allowedIPAddresses;
 }

}
```

This interface defines the `installPlugin()` method #1 that is used to instantiate the plugin and return new intercepted broker for the next plugin in the chain. This class also contains getter and setter methods used to configure the plugin as we will see in the moment.

The next step could be to provide custom XML configuration for our plugin using XBean, but we will skip this step and configure the plugin as a regular Spring bean:

```
 <broker xmlns="http://activemq.org/config/1.0"
  brokerName="localhost" dataDirectory="${activemq.base}/data"
  plugins="#ipAuthenticationPlugin">                            #1

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

 <bean id="ipAuthenticationPlugin"                             #A
  class="org.apache.activemq.book.ch5.IPAuthenticationPlugin"> #A
  <property name="allowedIPAddresses">                         #A
   <list>                                                      #A
    <value>127.0.0.1</value>                                   #A
   </list>                                                     #A
  </property>                                                  #A
 </bean>                                                       #A

 #A Plugin Bean
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Note that the `<broker>` tag supports the `plugins` attribute #1, which is used to define all plugins that does not have XBean configuration and cannot be configured inside the `<plugins>` tags.

All we have to do now is put our plugin into broker's classpath and run it:

```
${ACTIVEMQ_HOME}/bin/activemq \
xbean:src/main/resources/org/apache/activemq/book/ch5/activemq-custom.xml
```

You can try connecting to a broker from the localhost and everything should be working fine. If you try to connect from another host (or change a configuration) you can expect the exception similar to the one below to be thrown:

```
Exception in thread "main" javax.jms.JMSException:
Connecting from IP address 127.0.0.1 is not allowed
```

5.6. Summary

In this chapter we have seen how ActiveMQ broker can be secured of unauthenticated and unauthorized access and integrated into existing IT security infrastructure. For the most simple purposes, you can use "Simple Authentication Plugin" which allows you to define security credentials directly into the configuration file. The "JAAS plugin" allows you to plug in into the existing Java security modules and allows you to authenticate users from various sources, such as LDAP, properties files and so on.

Also, we have seen how ActiveMQ plugin mechanism could be used to customize security features of the broker. You can build your own JAAS modules and the whole security plugins as well. You can also restrict consumption on the message basis, using message properties and content, but also the connection context of the consumer.

With security topic we have covered basic ActiveMQ broker features and can move on to exploring how we can integrate it into various environments. In the following part of the book, we will see how ActiveMQ can be integrated into Java applications and some of the most popular application servers. We will also, explore various aspects of connecting to the broker from different environments,

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

such as .NET, Web, scripting languages, etc.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Chapter 6. Creating Java Applications With ActiveMQ

6.1. Introduction

Thus far we were mainly focusing on the ActiveMQ broker as a standalone middleware component used to exchange messages between Java applications. But as we have seen in Chapter 3, Understanding Connectors, 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 Spring framework (http://www.springframework.org)

6.2. 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.

6.2.1. Embedding The Broker

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

6.2.1.1. 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 staring point. This class is used to configure broker and manage its life cycle. The best way to 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 6.1. 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,c
    <authenticationUser username="publisher" password="password" groups="publishers,cons
    <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 6.2. Listing 6.2: Broker service example

```
public static void main(String[] args) throws Exception {
 BrokerService broker = new BrokerService();                     #A
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
 broker.setBrokerName("localhost");                                #A
 broker.setDataDirectory("data/");                                 #A

 SimpleAuthenticationPlugin authentication =
    new SimpleAuthenticationPlugin();

 List<AuthenticationUser> users =
    new ArrayList<AuthenticationUser>();
 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

6.2.1.2. 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 6.3. 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(                  #A
   "xbean:src/main/resources/org/apache/activemq/book/ch5/activemq-simple.xml"  #A
  ));                                                                           #A
  broker.start();
 }

}

#A Creating broker from XML
```

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

performed completely vie configuration URI. Take a look at the following URI for example:

```
broker:(tcp://localhost:61616,network:static:tcp://remotehost:61616)?persistent=false&us
```

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

6.2.2. 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.

6.2.2.1. 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 6.4. 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;
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
public class SpringBroker {

 public static void main(String[] args) throws Exception {
     if (args.length == 0) {
      System.err.println("Please define a configuration file!");
      return;
     }
     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
     }                                                                  #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>
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

6.2.2.1.1. 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 6.5. 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" /> #
  <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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

6.2.2.1.2. 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 6.6. 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.xsdA
 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"
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
                              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>

</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.

6.2.2.1.3. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

6.2.2.2. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

to rewrite our portfolio application to use all benefits of the ActiveMQ Spring integration.

6.2.2.2.1. 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 6.7. 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 6.2.2.2.4 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 6.8. Listing 6.5: Pooled Connection Factory

```
<bean id="pooledJmsFactory" class="org.apache.activemq.pool.PooledConnectionFactory" de
 <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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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:

```
<dependency>
 <groupId>commons-pool</groupId>
 <artifactId>commons-pool</artifactId>
 <version>1.4</version>
</dependency>
```

to your project configuration.

6.2.2.2.2. 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 6.9. 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.

6.2.2.2.3. Defining consumers

Now that we have connections and destinations ready, we can start consuming and

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 6.10. 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.DefaultMessageListenerCo
 <property name="connectionFactory" ref="jmsFactory"/>
 <property name="destination" ref="javaDest" />
 <property name="messageListener" ref="portfolioListener" />
</bean>

<bean id="ionaConsumer" class="org.springframework.jms.listener.DefaultMessageListenerCo
 <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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

6.2.2.2.4. 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/JmsTe 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/Messa 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 6.11. 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
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
  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.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 6.12. 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>();
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
 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");
   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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

All that is left to be done is to configure it properly.

Example 6.13. Listing 6.10: Spring Publisher Configuration

```
<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
 <property name="connectionFactory">
  <ref local="pooledJmsFactory"/>
 </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.

6.2.2.2.5. Putting it all together

After implementing all pieces of our example we are ready to put them all together and run the application.

Example 6.14. Listing 6.11: Spring client

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
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();
 }

}
```

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=fa
  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=
  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
  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
  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=
  on destination: topic://STOCKS.IONA
topic://STOCKS.IONA 80.51 80.59 down
Sending: {price=65.48960846536974, stock=topic://STOCKS.JAVA, offer=65.5550980738351, up=
  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.

6.3. Summary

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Part III. Using ActiveMQ

[Intro goes here]

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Chapter 8. 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.

8.1. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

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 =                                      #A
          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
                                                                      #A
        writer.writeStartElement("offer");                            #A
        writer.writeCharacters(String.valueOf(offer));                #A
        writer.writeEndElement();                                     #A
                                                                      #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
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
        message.setText(res.toString());                            #B
        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="ION
```

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
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

8.2. 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
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
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.

Listing 8.2: STOMP session

```
$ telnet localhost 61613
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

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 to is to define a transport connector with `stomp`

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

keyword for an URI schema and you are ready to go.

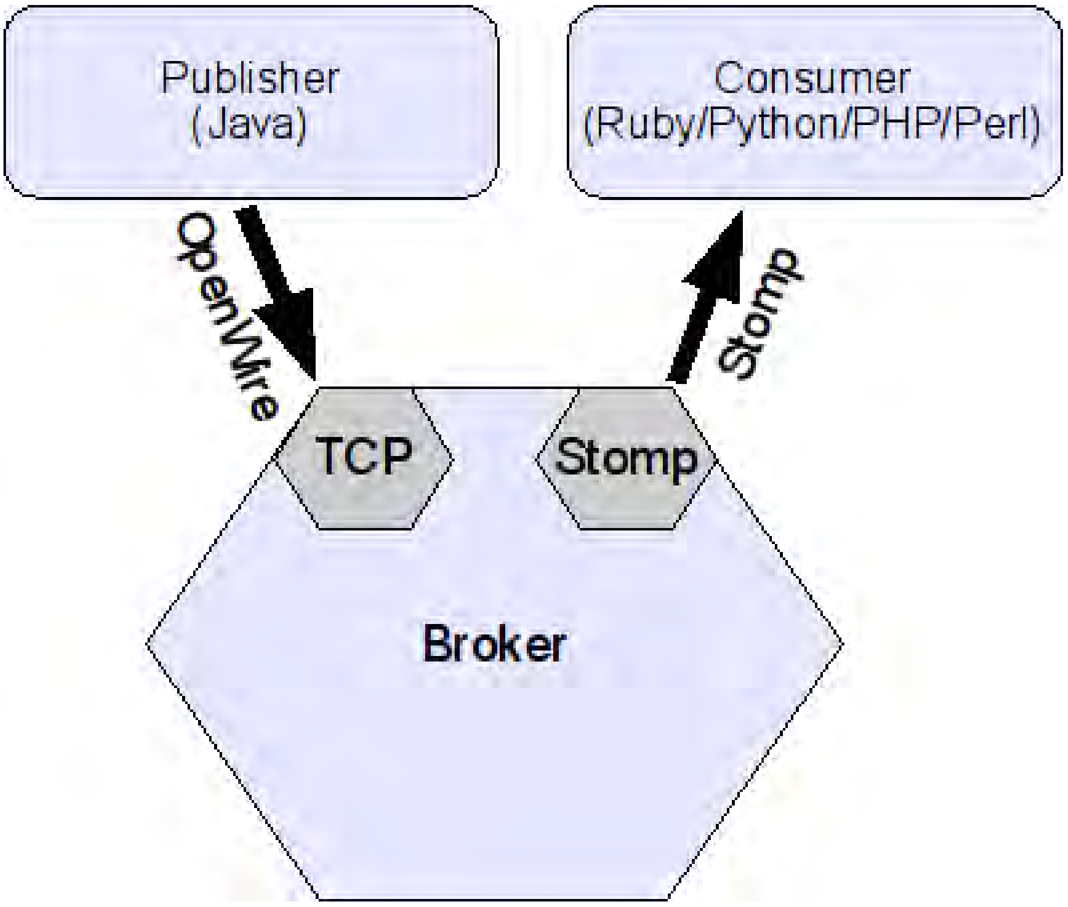


Figure 8.1: Stomp Example

Now let's see how to implement consumers of our stock portfolio data in some of the most of the popular scripting languages.

8.2.1. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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:

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

`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
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
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' see how to implement a similar stock portfolio data consumer in Python.

8.2.2. 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 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/. Now let's take a look at the example in Listing 8.5:

Listing 8.5: Python consumer

```
#!/usr/bin/env python

import time
import sys
from elementtree.ElementTree import ElementTree, XML

import stomp

class MyListener(object):                                   #A
    def on_error(self, headers, message):
        print 'received an error %s' % message

    def on_message(self, headers, message):
        xml = XML(message)

        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"
        )
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
conn = stomp.Connection()                                    #1
conn.add_listener(MyListener())                              #2
conn.start()                                                 #3
conn.connect()

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 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
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
```

As we said before, there are a few other Python clients that could exchange messages with ActiveMQ. If you prefer to use the OpenWire protocol you can consider the *pyactivemq* project (http://code.google.com/p/pyactivemq/), which wraps ActiveMQ C++ library (describer a bit later) and supports both Stomp and

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

OpenWire protocols.

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.

8.2.3. 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. *Stompcli* library (http://stomp.codehaus.org/PHP) provides an easy way to use asynchronous messaging in PHP applications. The example shown in Listing 8.6 and explained afterwards, demonstrates how to create a stock portfolio data consumer in PHP:

Listing 8.6: 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


?>
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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
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.

8.2.4. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

module (http://search.cpan.org/dist/Net-Stomp/). The following example contains an implementation of the stock portfolio consumer in Perl.

Listing 8.7: 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
    }
);
$stomp->subscribe(
    {   destination             => '/topic/STOCKS.IONA',
        'ack'                   => 'client',
        'activemq.prefetchSize' => 1                                          #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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 inducted 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. 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 far 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

8.3. 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 8.8:

Listing 8.8: C# Consumer

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

namespace Apache.NMS.ActiveMQ.Book.Ch8
{
   public class Consumer
   {
      public static void Main(string[] args)
      {
        NMSConnectionFactory NMSFactory =
          new NMSConnectionFactory("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
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
        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>43.05663
<?xml version="1.0" ?><stock name="JAVA"><price>43.39372871092703</price><offer>43.43712
<?xml version="1.0" ?><stock name="JAVA"><price>43.31253506864452</price><offer>43.35584
<?xml version="1.0" ?><stock name="JAVA"><price>43.579419162289334</price><offer>43.6229
<?xml version="1.0" ?><stock name="JAVA"><price>43.268719403943344</price><offer>43.3119
<?xml version="1.0" ?><stock name="JAVA"><price>43.03515076051564</price><offer>43.07818
<?xml version="1.0" ?><stock name="JAVA"><price>42.75679069998244</price><offer>42.79954
```

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++

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

developers have if they want to use messaging with ActiveMQ.

8.4. 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

Listing 8.9: C++ Consumer

```
class SimpleAsyncConsumer : public ExceptionListener,
                            public MessageListener
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

The MessageListener interface defines the `onMessage()` method, which handles received messages. In our example it boils down to printing and acknowledging the message.

Listing 8.10: 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 (CMSException& e) {
            e.printStackTrace();
        }
    }
```

The `ExceptionListener` interface defines the `onException()` method called when connection problems are detected.

Listing 8.11: C++ Exception Listener

```
    virtual void onException( const CMSException& ex AMQCPP_UNUSED) {
        printf("CMS Exception occured.  Shutting down client.\n");
    }
```

As you can see, thus far CMS mimics the JMS API completely, which is great for

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

Listing 8.12: C++ creating Consumer

```
    void runConsumer() {

        try {

            ActiveMQConnectionFactory* connectionFactory =
                new ActiveMQConnectionFactory( brokerURI );                            #A

            connection = connectionFactory->createConnection();                        #B
            delete connectionFactory;
            connection->start();                                                       #C

            connection->setExceptionListener(this);

            if( clientAck ) {                                                          #D
                session = connection->createSession( Session::CLIENT_ACKNOWLEDGE );
            } else {
                session = connection->createSession( Session::AUTO_ACKNOWLEDGE );
            }

            if( useTopic ) {                                                           #E
                destination = session->createTopic( destURI );
            } else {
                destination = session->createQueue( destURI );
            }

            consumer = session->createConsumer( destination );                          #
            consumer->setMessageListener( this );                                       #

        } catch (CMSException& e) {
            e.printStackTrace();
        }
    }

#A Define Connection Factory
#B Create Sonnection
#C Start Sonnection
#D Create Session
#E Create Destination
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
#F Create Consumer
#G Add Message Listener
```

All that is left to be done is to initialize everything and run the application.

Listing 8.13: 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";
```

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
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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</
Message #2 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.638920307293986<
Message #3 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.828934270661314<
Message #4 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.41909588529107</
Message #5 Received: <?xml version="1.0" ?><stock name="JAVA"><price>53.95595590764363</
Message #6 Received: <?xml version="1.0" ?><stock name="JAVA"><price>53.74094054512154</
Message #7 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.24485518988984</
Message #8 Received: <?xml version="1.0" ?><stock name="JAVA"><price>53.72415991559902</
Message #9 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.091504162570935<
Message #10 Received: <?xml version="1.0" ?><stock name="JAVA"><price>53.960072785363025
Message #11 Received: <?xml version="1.0" ?><stock name="JAVA"><price>53.82082586259414<
Message #12 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.032852016137156
Message #13 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.31012714496037<
Message #14 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.76169437095268<
Message #15 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.92747610279041<
Message #16 Received: <?xml version="1.0" ?><stock name="JAVA"><price>55.074451578258646
Message #17 Received: <?xml version="1.0" ?><stock name="JAVA"><price>55.56804101263522<
Message #18 Received: <?xml version="1.0" ?><stock name="JAVA"><price>55.77722621685933<
Message #19 Received: <?xml version="1.0" ?><stock name="JAVA"><price>55.58192302489994<
Message #20 Received: <?xml version="1.0" ?><stock name="JAVA"><price>55.561645147383295
```

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.

8.5. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

8.5.1. 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 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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
```

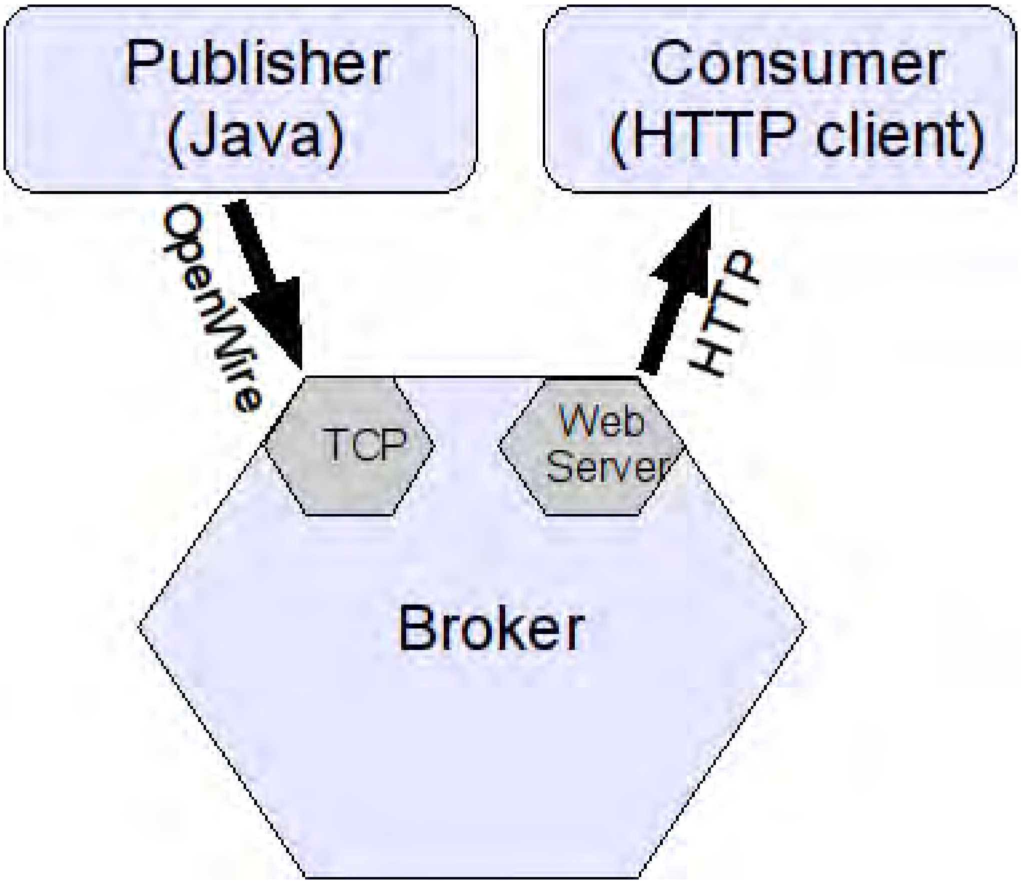


Figure8.2. Figure 8.2: Rest Example

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

Listing 8.14: 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>

    <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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Listing 8.15: 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:

Listing 8.16: 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`

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

8.5.2. 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

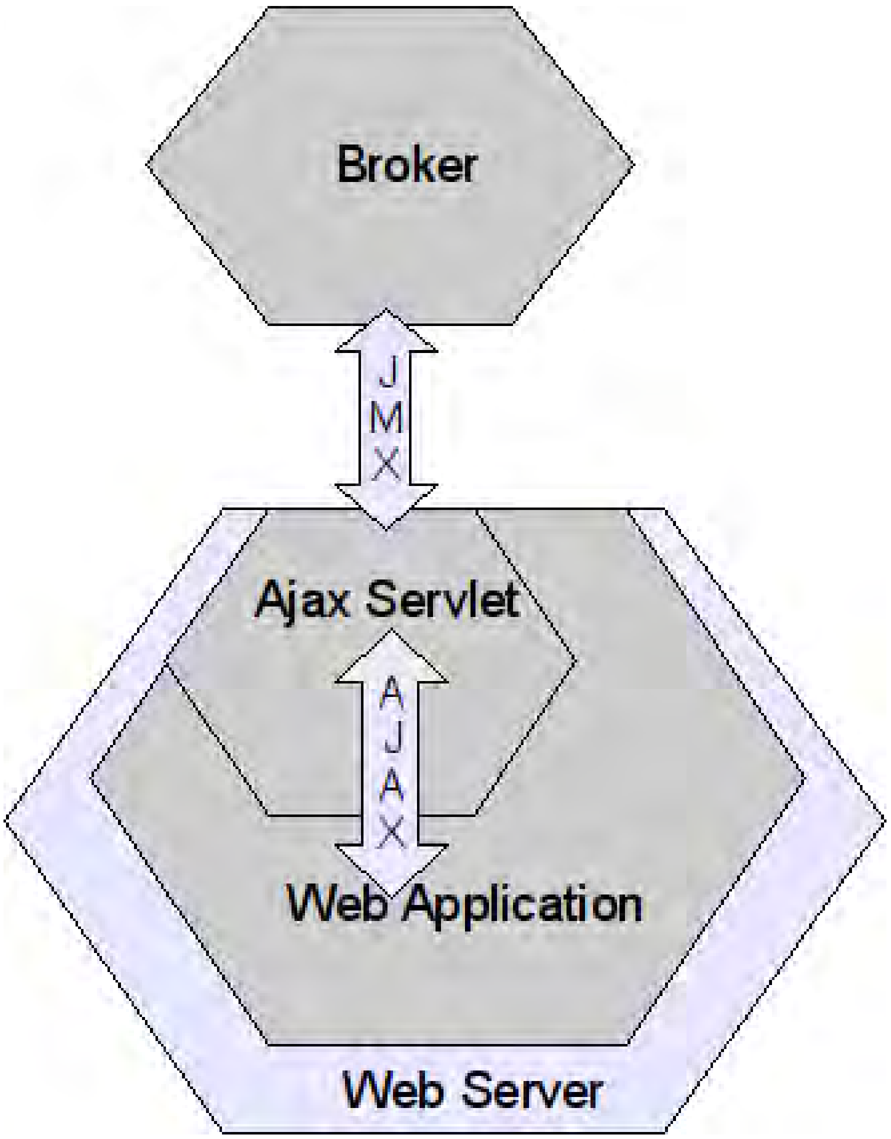


Figure 8.3: Ajax API

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.

Listing 8.17: Ajax server configuration

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
  <servlet>
    <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 8.18 shows how to achieve this.

Listing 8.18: 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:

Listing 8.19: 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 8.20:

Listing 8.20: 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
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
    }
}

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&st
```

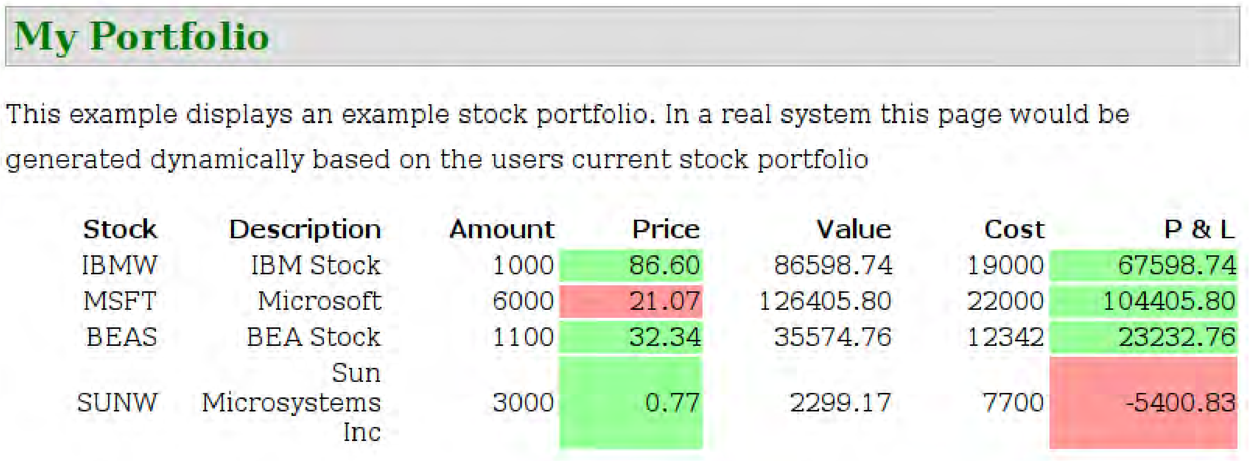
The Ajax consumer example is located at the following address:

```
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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496



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 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.

8.6. 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 tunning, 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Chapter 12. 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.

12.1. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

12.1.1. 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 - see Figure 12.
- Persisting messages to the message store (which typically involves writing to disk) is slow compared to messaging over a network

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

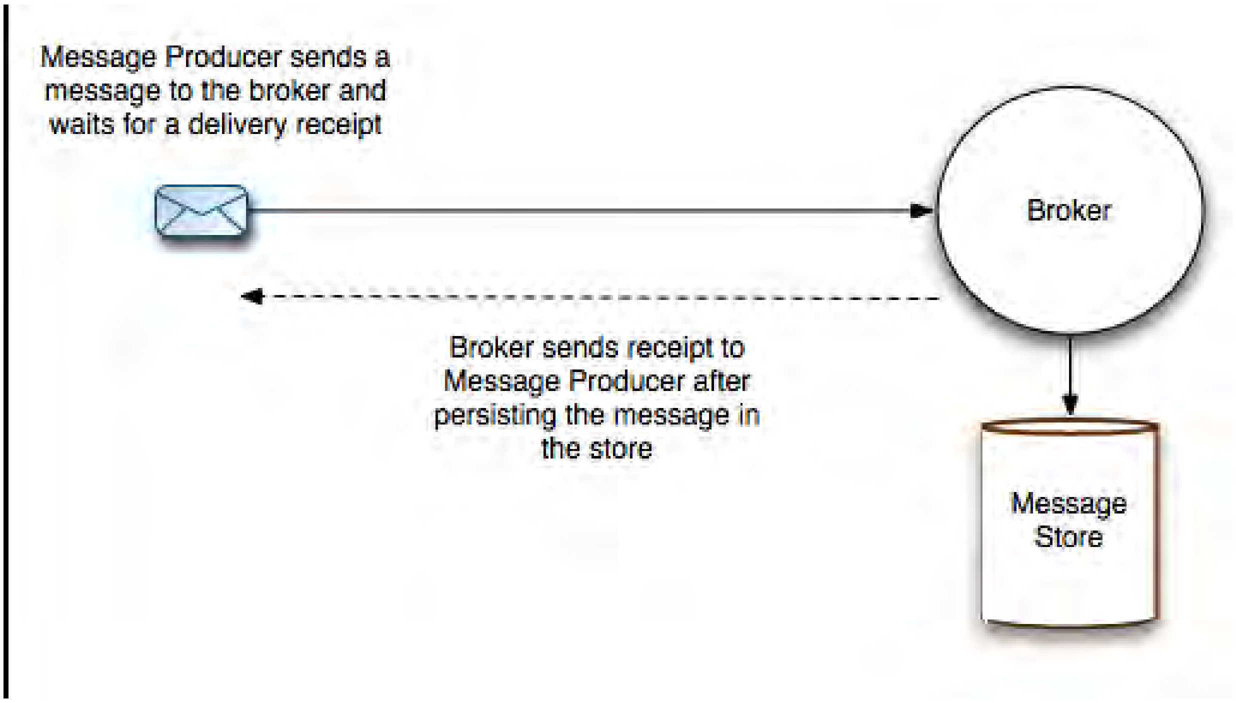

Message Producer sends a message to the broker and waits for a delivery receipt
Broker
Broker sends receipt to Message Producer after persisting the message in the store
Message Store

Figure 12.1: 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 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 see in Listing 12.1:

Listing 12.1: Setting the delivery mode

```
MessageProducer producer = session.createProducer(topic);
producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

12.1.2. 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 Listing 12.2 :

Listing 12.2: Transacted and non-transacted example

```
public void sendTransacted() throws JMSException {

//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){
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
            session.commit();
        }
    }
}

public void sendNonTransacted() throws JMSException {

   //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_ACKNOWELDGE);
   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.

12.1.3. 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 12.2. 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

be transported across the network, making it ideal for applications that have service lots of responses very quickly.

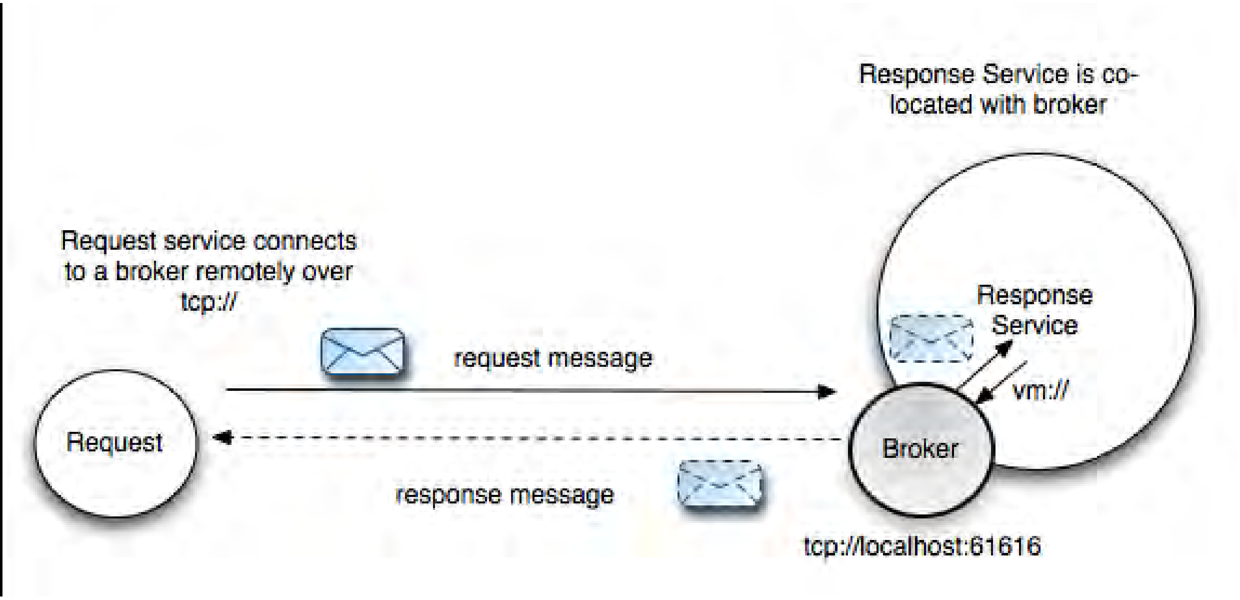

Response Service is co-located with broker
Request service connects to a broker remotely over tcp://
Response Service
request message
vm://
Request
Broker
response message
tcp://localhost:61616

Figure 12.2: 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 Listing 12.3 is an example of setting up a service using an embedded broker to listen for requests on a Queue named servce.queue

Listing 12.3: 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();
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
connection.start();
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 (JMSException 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 Listing 12.4 below:

Listing 12.4: 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());
}
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 Listing 12.5:

Listing 12.5: 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.

12.1.4. 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 12.1 are some OpenWire wire format parameters that are relevant to performance

Table 12.1: OpenWire Tuning Parameters

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.
cache enabled	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 where caching is enabled. However, one cache will be

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

parameter name	default value	description
		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 - this example in Listing 12.6 demonstrates disabling tight encoding - using the tightEncodingEnabled parameter:

Listing 12.6: Setting OpenWire options

```
String uri = "failover://(tcp://localhost:61616?wireFormat.cacheEnabled=false&wireFormat
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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

12.1.5. 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 recieve 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 Listing 12.7 is an example transport URI where tcpNoDelay is enabled:

Listing 12.7: 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.

12.2. Optimizing Message Producers

The rate that producers send messages to an ActiveMQ message broker before they

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

12.2.1. 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 hold messages for re-delivery in its failover transport cache). However, you can also get the same reliability for persistent messages - by setting the alwaysSyncSend property on the message producer's ConnectionFactory - eg:

Listing 12.8: Setting the delivery mode

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
cf.setAlwaysSyncSend(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 you application requires guaranteed delivery, it is recommend 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

happening to your applications.

12.2.2. 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 12.3

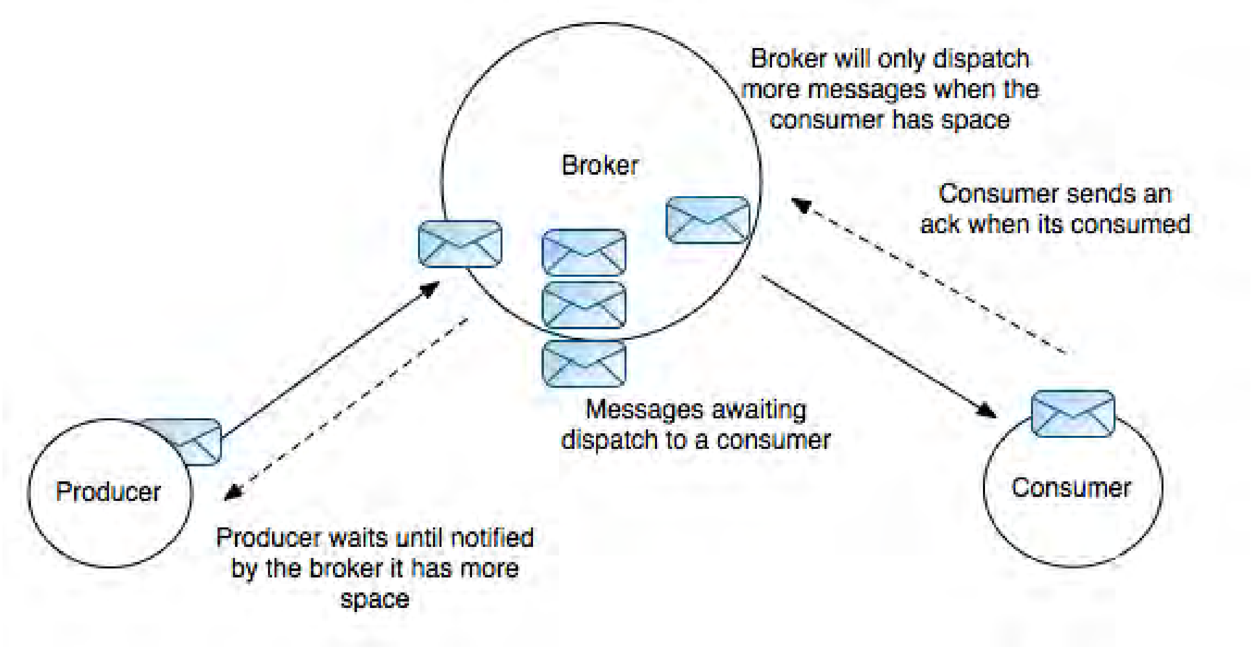

Broker will only dispatch more messages when the consumer has space
Broker
Consumer sends an ack when its consumed
Messages awaiting dispatch to a consumer
Producer
Consumer
Producer waits until notified by the broker it has more space

Figure 12.3: Producer Flow Control enabled

Producer flow control is a necessary 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

connection factory - as in Listing 12.9:

Listing 12.9: 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 so what happens if you disable producer flow control, and you can do that in the Broker configuration on a destination policy like below in Listing 12.10:

Listing 12.10: 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>
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 12.4

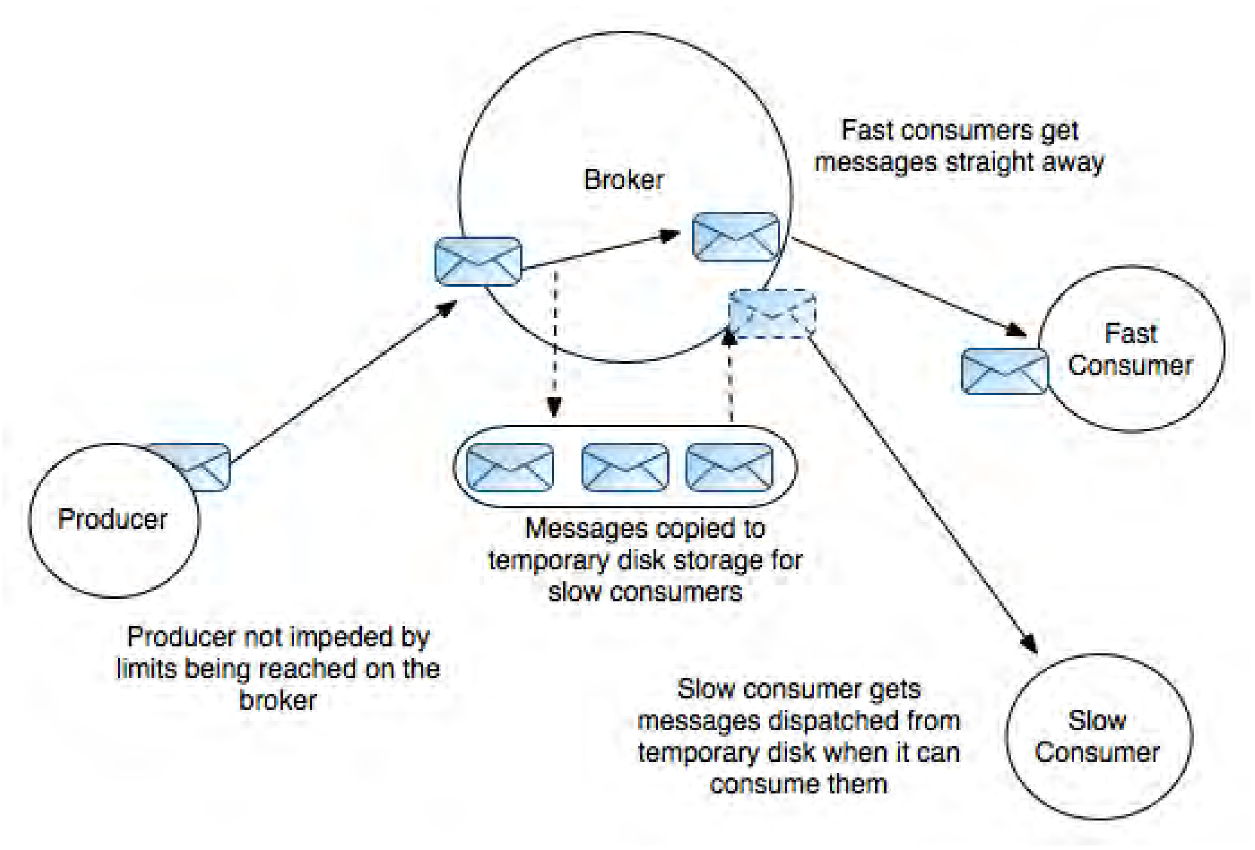

Broker
Fast consumers get messages straight away
Fast Consumer
Producer
Messages copied to temporary disk storage for slow consumers
Producer not impeded by limits being reached on the broker
Slow consumer gets messages dispatched from temporary disk when it can consume them
Slow Consumer

Figure 12.4: 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.

12.3. Optimizing Message Consumers

In order to maximize application performance you have to look at all the

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

12.3.1. 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 - like in figure 12.5 below.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

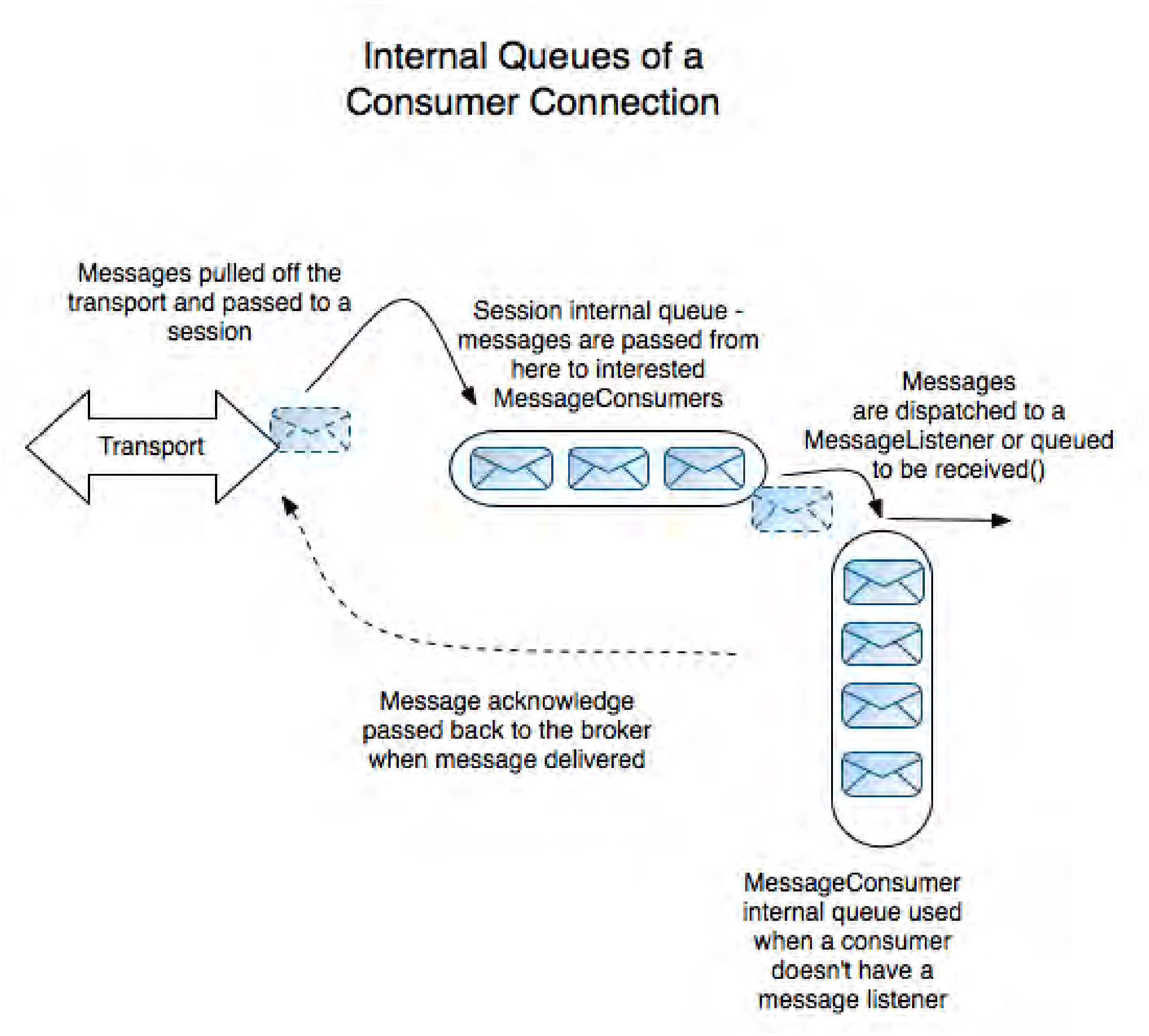

Internal Queues of a Consumer Connection
Messages pulled off the transport and passed to a session
Session internal queue - messages are passed from here to interested MessageConsumers
Messages are dispatched to a MessageListener or queued to be received()
Transport
Message acknowledge passed back to the broker when message delivered
MessageConsumer internal queue used when a consumer doesn't have a message listener

Figure 12.5: 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 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 Listing 12.11:

Listing 12.11: 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 Listing 12.12:

Listing 12.12: Setting prefetch policy on a Destination

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

12.3.2. Delivery and Acknowledgement of messages

Something that should be apparent from figure 12.5 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 Listing 12.13:

Listing 12.13: Setting optimize acknowledge

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
cf.setOptimizeAcknowledge(true);
```

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 to amount of work the MessageConsumer has to do, enabling it to consume messages at a much faster rate.

Table 12.2 below outlines the different options for acknowledging messages and how often they send back a message acknowledgement to the ActiveMQ message broker.

Table 12.2: Different acknowledgment modes

Acknowledgement Mode	**Sends an Acknowledgement**	**description**
Session.SESSION_TRANSACTED	Rolls 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	All 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	Sends 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	Allows the consumer to	An acknowledgement will

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Acknowledgement Mode	**Sends an Acknowledgement**	**description**
	send one acknowledgement back to the ActiveMQ broker for a range of messages consumed.	be sent back when the prefetch limit has reached 50%. The fastest standard way of consuming messages.
ActiveMQSession.INDIVIDUAL_ACKNOWLEDGE	Sends one 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_ACKNOWLEDG 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at http://www.manning-sandbox.com/forum.jspa?forumID=496

12.3.3. Asynchronous dispatch

Every Session maintains an internal queue of messages to be dispatched to interested MessageConsumers (as can be seen from figure 12.5). The usage of an internal queue together with an associated thread to do the dispatching to MessageConsumers can add considerable over head 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 Listing 12.14:

Listing 12.14: Setting always use asynchronous dispatch

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
cf.setAlwaysSessionAsync(false);
```

Disabling asynchronous dispatch allow messages to by pass the internal queueing and dispatching done by the Session - as shown below in Figure 12.6

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

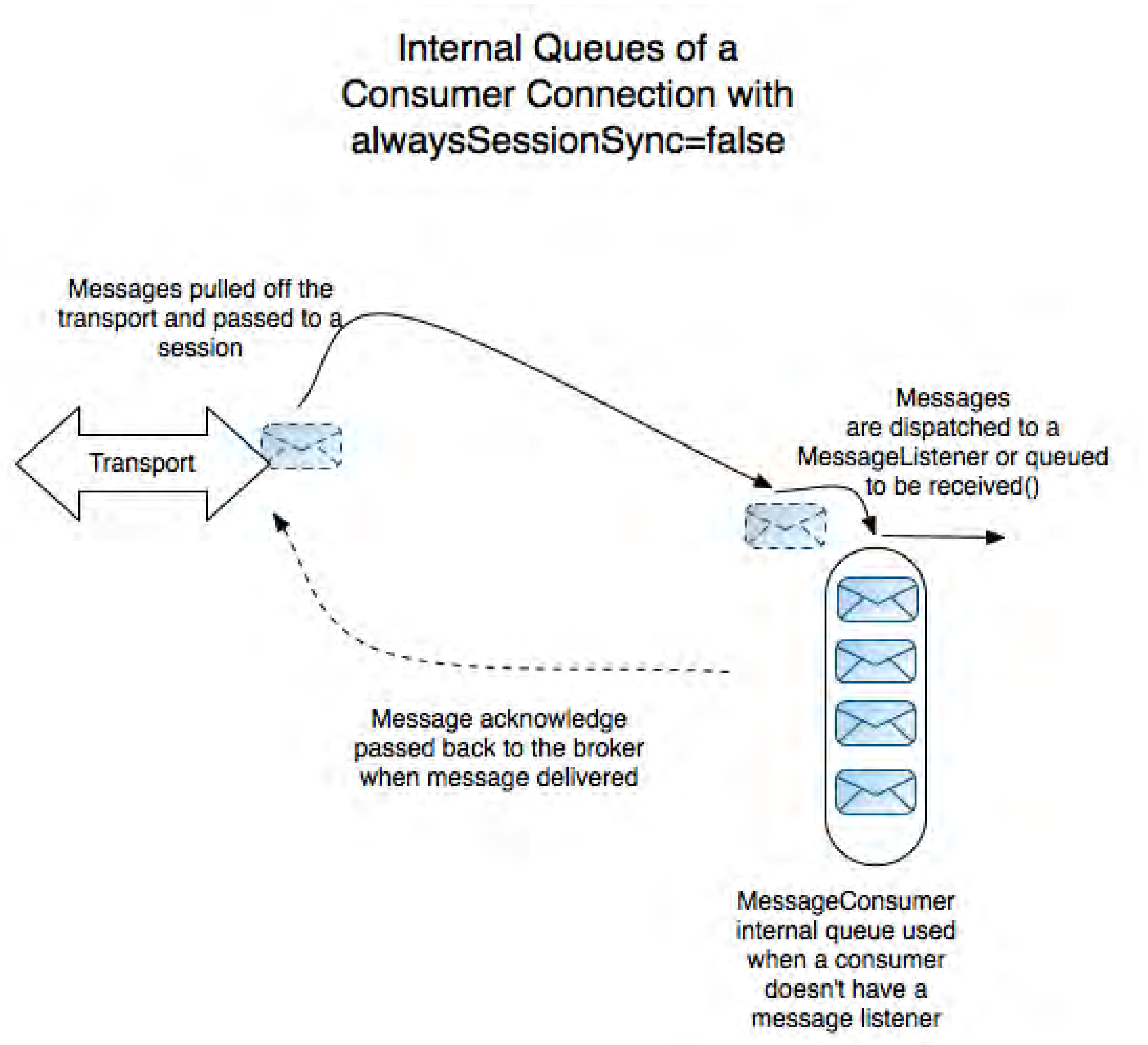

Internal Queues of a Consumer Connection with alwaysSessionSync=false
Messages pulled off the transport and passed to a session
Transport
Messages are dispatched to a MessageListener or queued to be received()
Message acknowledge passed back to the broker when message delivered
MessageConsumer internal queue used when a consumer doesn't have a message listener

Figure 12.6: 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 -

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

a real-time data feed.

12.4. 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 12.7.

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.

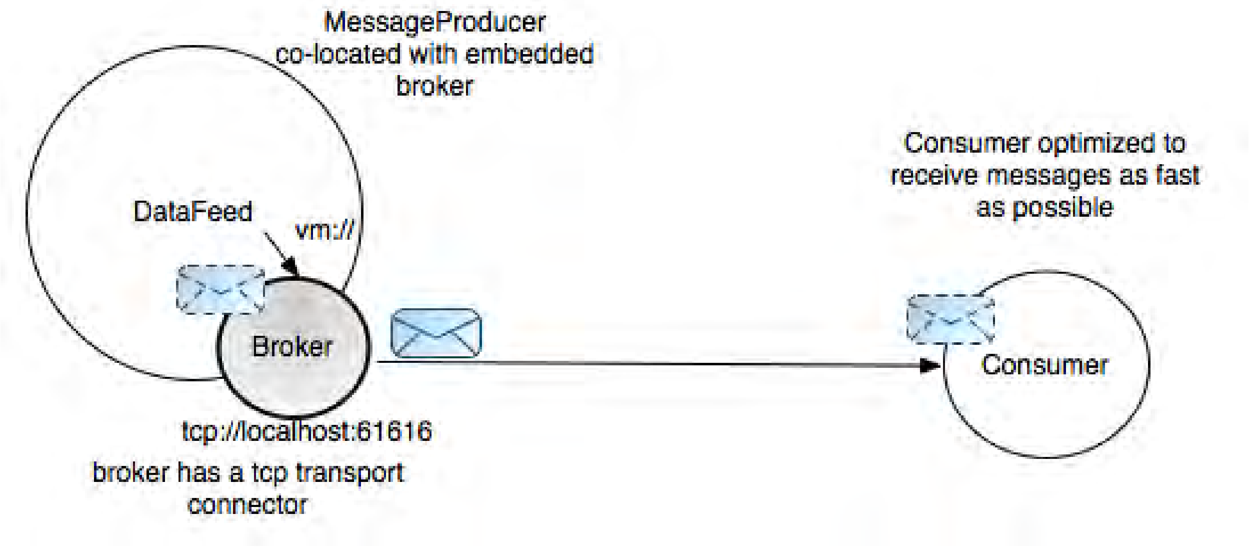

MessageProducer co-located with embedded broker
DataFeed
vm://
Broker
tcp://localhost:61616
broker has a tcp transport connector
Consumer optimized to receive messages as fast as possible
Consumer

Figure 12.7: Data feed application

Firstly we setup the broker to be embedded, with the memory limit set to a

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 Listing 12.15. 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 to 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.

Listing 12.15: Creating the embedded broker

```
import org.apache.activemq.broker.BrokerService;
import org.apache.activemq.broker.region.policy.PolicyEntry;
import org.apache.activemq.broker.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();
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 Listing 12.16 below:

Listing 12.16: 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 Listing 12.17 below:

Listing 12.17: Creating the Consumer

```
//set up the connection factory to connect the the producer's embedded broker
//using tcp://
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory("failover://(tcp://localhos

//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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

12.5. 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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 manged. These interfaces consist of *Management Beans*, usually called *MBeans*, which expose resources of your application to external management applications.

Enabling JMX Support in ActiveMQ

For starters, the JMX support in ActiveMQ must be enabled. So let's take a look at the following configuration we will use for our JMX-related examples.

Listing 14.1: 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>
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
      <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 turns JMX support on or off. The value of this attribute is `true` by default so the broker uses JMX by default. We included it in this example configuration just 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 it 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
```

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.

Using the JMX APIs With ActiveMQ

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Using the JMX API, statistics can be obtained from a broker at runtime. The example shown in Listing 14.2 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.

Listing 14.2: 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");                 #1
  JMXConnector connector = JMXConnectorFactory.connect(url, null);          #1
  connector.connect();                                                      #1
  MBeanServerConnection connection = connector.getMBeanServerConnection();  #1

  ObjectName name = new ObjectName(                                         #2
    "my-broker:BrokerName=localhost,Type=Broker");                          #2
  BrokerViewMBean mbean = (BrokerViewMBean) MBeanServerInvocationHandler    #2
    .newProxyInstance(connection, name, BrokerViewMBean.class, true);       #2

  System.out.println("Statistics for broker " + mbean.getBrokerId()         #3
    + " - " + mbean.getBrokerName());                                       #3
  System.out.println("\n-----------------\n");                              #3
  System.out.println("Total message count: " + mbean.getTotalMessageCount() + "\n");  #3
  System.out.println("Total number of consumers: " + mbean.getTotalConsumerCount());  #3
  System.out.println("Total number of Queues: " + mbean.getQueues().length);          #3

  for (ObjectName queueName : mbean.getQueues()) {                          #D
   QueueViewMBean queueMbean = (QueueViewMBean) MBeanServerInvocationHandler  #D
     .newProxyInstance(connection, queueName,                               #D
       QueueViewMBean.class, true);                                         #D
   System.out.println("\n-----------------\n");                             #D
   System.out.println("Statistics for queue " + queueMbean.getName());      #D
   System.out.println("Size: " + queueMbean.getQueueSize());                #D
   System.out.println("Number of consumers: " + queueMbean.getConsumerCount());  #
  }
 }

}
```

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

the broker's MBean server #1. 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 #2. 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 Listing 14.1 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 Listing 14.2 #3. 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 Listing 14.2. But first the consumer must be slowed down a bit to be sure that some messages exist in the

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 8.3.1: “Prefetch Limit”).

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) {
   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:

```
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
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
-----------------

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 could be very useful in both production and development environments.

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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:

Listing 14.3: Configuring Advisory Support

```
<broker xmlns="http://activemq.org/config/1.0" useJmx="true"
   brokerName="localhost" dataDirectory="${activemq.base}/data"
   advisorySupport="true">                                            #1

  <destinationPolicy>
    <policyMap>
      <policyEntries>
        <policyEntry topic=">" sendAdvisoryIfNoConsumers="true"/>       #2
      </policyEntries>
    </policyMap>
  </destinationPolicy>

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

</broker>
```

There are two important things we can learn from the configuration file above:

1. Advisory support can be enabled using the `advisorySupport` attribute of the `<broker>` element #1. 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.

2. The second and more important item above is the use of a *destination policy* to enable more complex advisories for your destinations #2. In example above, 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 the destination.

To demonstrate this functionality, start the broker using the example configuration

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

from above (named `activemq-advisory.xml`) via the following command:

```
$ ${ACTIVEMQ_HOME}/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 JMSException {
        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 Listing 14.4.

Listing 14.4: Advisory example

```
public class Advisory {

    protected static String brokerURL = "tcp://localhost:61616";
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
    protected static transient ConnectionFactory factory;
    protected transient Connection connection;
    protected transient Session session;

    public Advisory() throws Exception {                                          #1
     factory = new ActiveMQConnectionFactory(brokerURL);
     connection = factory.createConnection();
     connection.start();
     session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    }

    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);                  #2
       System.out.println("Subscribing to advisory " + consumerTopic);
       MessageConsumer consumerAdvisory = session.createConsumer(consumerTopic);
       consumerAdvisory.setMessageListener(new ConsumerAdvisoryListener());

       Destination noConsumerTopic =
         AdvisorySupport.getNoTopicConsumersAdvisoryTopic(destination);          #3
       System.out.println("Subscribing to advisory " + noConsumerTopic);
       MessageConsumer noConsumerAdvisory = session.createConsumer(noConsumerTopic);
       noConsumerAdvisory.setMessageListener(new NoConsumerAdvisoryListener());

    }
 }

 public Session getSession() {
  return session;
 }

}
```

This example provides a demonstration using standard JMS messaging. First, initialize the JMS connection and the JMS session in the class constructor #1. 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.IONA`, a subscription is created to the advisory topics named

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

`topic://ActiveMQ.Advisory.Consumer.Topic.STOCKS.IONA` and `topic://ActiveMQ.Advisory.NoConsumer.Topic.STOCKS.IONA`.

> **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 Listing 14.5.

Listing 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 CommandTypes.CONSUMER_INFO:                                 #1
    ConsumerInfo consumerInfo = (ConsumerInfo) ds;
    System.out.println("Consumer '" + consumerInfo.getConsumerId()
      + "' subscribed to '" + consumerInfo.getDestination()
      + "'");
    break;
   case CommandTypes.REMOVE_INFO:                                   #2
    RemoveInfo removeInfo = (RemoveInfo) ds;
    ConsumerId consumerId = ((ConsumerId) removeInfo.getObjectId());
    System.out.println("Consumer '" + consumerId + "' unsubscribed");
    break;
   default:
    System.out.println("Unkown data structure type");
   }
  } else {
   System.out.println("No data structure provided");
  }
 }
}
```

Every advisory is basically a regular instance of a `ActiveMQMessage` object. In

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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` #1 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` as shown in #2, 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 Listing 14.6.

Listing 14.6: No consumer advisory listener

```
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:

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
$ mvn -e exec:java -Dexec.mainClass=org.apache.activemq.book.ch14.jmx.Advisory \
-Dexec.args="tcp://localhost:61616 IONA JAVA"

...

Subscribing to advisory topic://ActiveMQ.Advisory.Consumer.Topic.STOCKS.tcp://localhost:
Subscribing to advisory topic://ActiveMQ.Advisory.NoConsumer.Topic.STOCKS.tcp://localhos
Subscribing to advisory topic://ActiveMQ.Advisory.Consumer.Topic.STOCKS.IONA
Subscribing to advisory topic://ActiveMQ.Advisory.NoConsumer.Topic.STOCKS.IONA
Subscribing to advisory topic://ActiveMQ.Advisory.Consumer.Topic.STOCKS.JAVA
Subscribing to advisory topic://ActiveMQ.Advisory.NoConsumer.Topic.STOCKS.JAVA

...
```

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 IONA JAVA"
```

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.IONA'
Consumer 'ID:dejan-bosanacs-macbook-pro.local-64609-1233592052313-0:0:1:2'
subscribed to 'topic://STOCKS.JAVA'
```

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 IONA JAVA"
```

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:

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
...
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=JAVA, offer=11.817656439151577, price=11.805850588563015}
not consumed by any consumer
Message {up=false, stock=JAVA, offer=11.706856077241527, price=11.695160916325204}
not consumed by any consumer
Message {up=false, stock=JAVA, offer=11.638181080673165, price=11.62655452614702}
not consumed by any consumer
Message {up=true, stock=IONA, offer=36.51689387339347, price=36.480413459933544}
not consumed by any consumer
Message {up=false, stock=JAVA, offer=11.524555643871604, price=11.513042601270335}
not consumed by any consumer
Message {up=true, stock=IONA, offer=36.583094870955556, price=36.54654832263293}
not consumed by any consumer
Message {up=false, stock=JAVA, offer=11.515997849703322, price=11.504493356346975}
not consumed by any consumer
Message {up=true, stock=JAVA, 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 9.1.1.1: “Enabling JMX Support in ActiveMQ”.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 explore 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

Starting and Stopping the Broker

The standard method for starting ActiveMQ is to use the following command on the command line:

```
${ACTIVEMQ-HOME}/bin/activemq
```

In addition, the following command using the `bin/activemq` script can also be used:

```
${ACTIVEMQ_HOME}/bin/activemq-admin start
```

Using the same script, ActiveMQ can also be used stopped using the following command:

```
${ACTIVEMQ_HOME}/bin/activemq-admin stop
```

The `bin/activemq` script is a nice alternative for stopping the broker. It will

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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:

```
${ACTIVEMQ_HOME}/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 `bin/activemq` script 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 any Red Hat Linux-based operating system:

```
#!/bin/bash

# Customize the following variables for your environment
PROG=activemq
PROG_USER=activemq
DAEMON_HOME=/opt/activemq
DAEMON=$DAEMON_HOME/bin/$PROG
LOCKFILE=/var/lock/subsys/$PROG
PIDFILE=/var/run/$PROG.pid

test -x $DAEMON || exit 0

# Source function library.
. /etc/rc.d/init.d/functions

RETVAL=0

usage () {
    echo "Usage: service $PROG {start|stop|restart|status}"
    RETVAL=1
}

start () {
        echo -n $"Starting $PROG: "
        if [ ! -e $LOCKFILE ]; then
                cd $DAEMON_HOME
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
                sudo -i -u $PROG_USER $DAEMON > >(logger -t $PROG) 2>&1 &
        else
                echo -n "Lockfile exists"
                false
        fi
        RETVAL=$?
        if [ $RETVAL -eq 0 ]; then
                logger -t activemq "starting $PROG."
                echo $! > $PIDFILE
                touch $LOCKFILE
        else
                logger -t activemq "unable to start $PROG."
        fi
        [ $RETVAL -eq 0 ] && success $"$PROG startup" || failure $"$PROG startup"
        echo
}

stop () {
        echo -n "Shutting down $PROG: "
        killproc -p $PIDFILE -d 20
        RETVAL=$?
        echo
        [ $RETVAL = 0 ] && rm -f $LOCKFILE
}

case "$1" in
    start) start ;;
    stop) stop ;;
    restart|reload)
        stop
        start
        ;;
    status)
        status $PROG -p $PIDFILE
        RETVAL=$?
        ;;
    *) usage ;;
esac

exit $RETVAL
```

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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Listing Avaiable 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 Listing 14.7.

Listing 14.7: activemq-admin list

```
${ACTIVEMQ_HOME}/bin/activemq-admin list
ACTIVEMQ_HOME: /workspace/apache-activemq-5.2.0
ACTIVEMQ_BASE: /workspace/apache-activemq-5.2.0
BrokerName = localhost
```

As you can see in the example above, we have only one broker in the given context and its name is `localhost`.

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:

Listing 14.8: activemq-admin query

```
${ACTIVEMQ_HOME}/bin/activemq-admin query -QQueue=*
ACTIVEMQ_HOME: /workspace/apache-activemq-5.2.0
ACTIVEMQ_BASE: /workspace/apache-activemq-5.2.0
DequeueCount = 0
Name = example.A
MinEnqueueTime = 0
CursorMemoryUsage = 0
MaxAuditDepth = 2048
Destination = example.A
AverageEnqueueTime = 0.0
InFlightCount = 0
MemoryLimit = 5242880
Type = Queue
EnqueueCount = 0
MaxEnqueueTime = 0
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
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=Non-Durable,destinationType=Queue,destinationName=example.A,
 clientId=ID_xxx-60272-1234785179087-3_0,consumerId=ID_xxx-60272-1234785179087-2_0_1_1]
QueueSize = 0
MaxPageSize = 200
CursorPercentUsage = 0
MemoryPercentUsage = 0
DispatchCount = 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.

Browsing Destinations

Browsing destinations in the broker is another fundamental administrative task. 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:

Listing 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
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 you 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:

Listing 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://localho
 ...
```

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

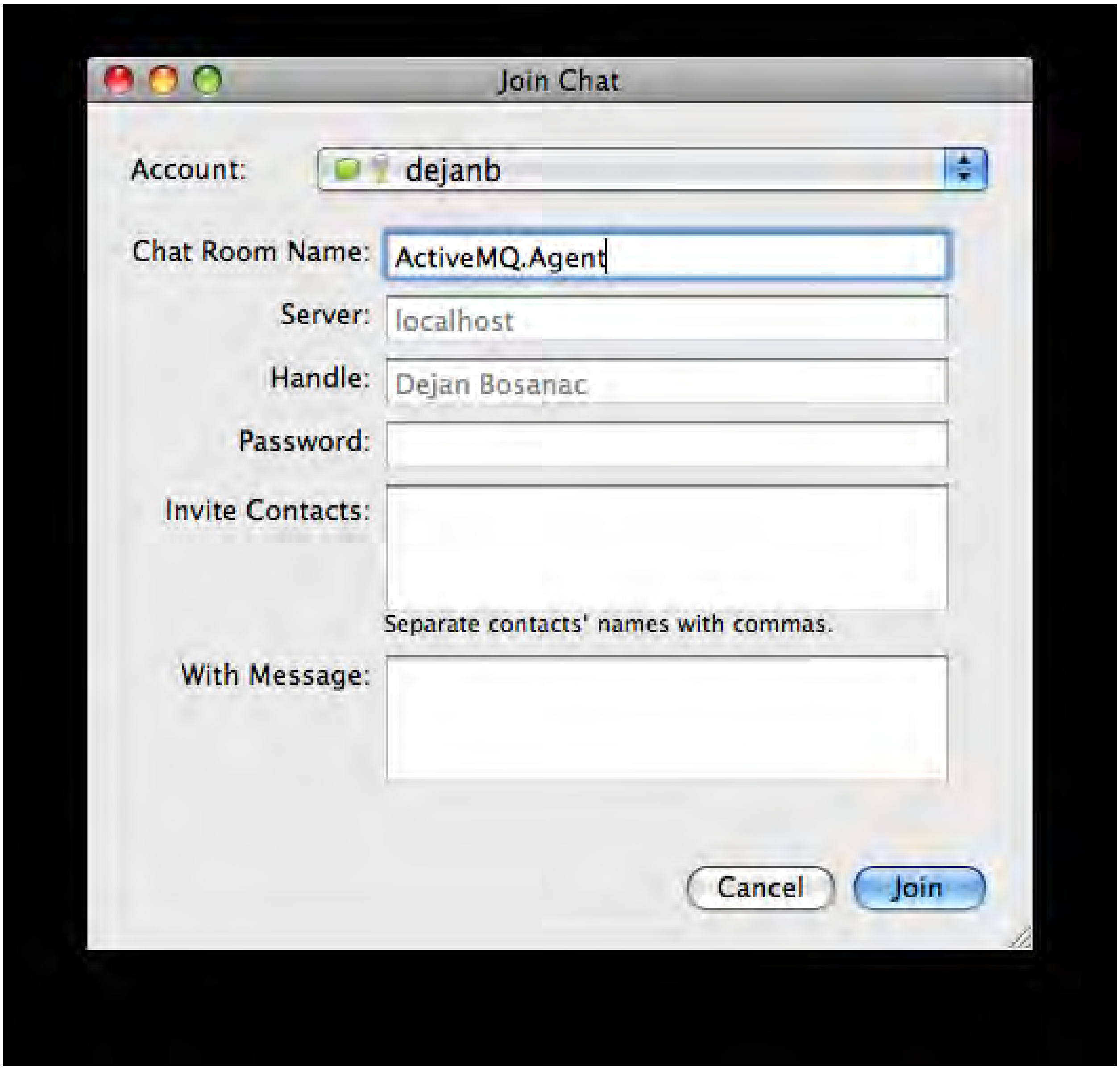
developed by the Jabber project (http://jabber.org/). Since it's open and very widespread, there are a lot of *chat* clients that already support this protocol and you can use these clients to communicate with ActiveMQ.

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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496



Account: dejanb

Chat Room Name: ActiveMQ.Agent

Server: localhost

Handle: Dejan Bosanac

Password:

Invite Contacts:

Separate contacts' names with commas.

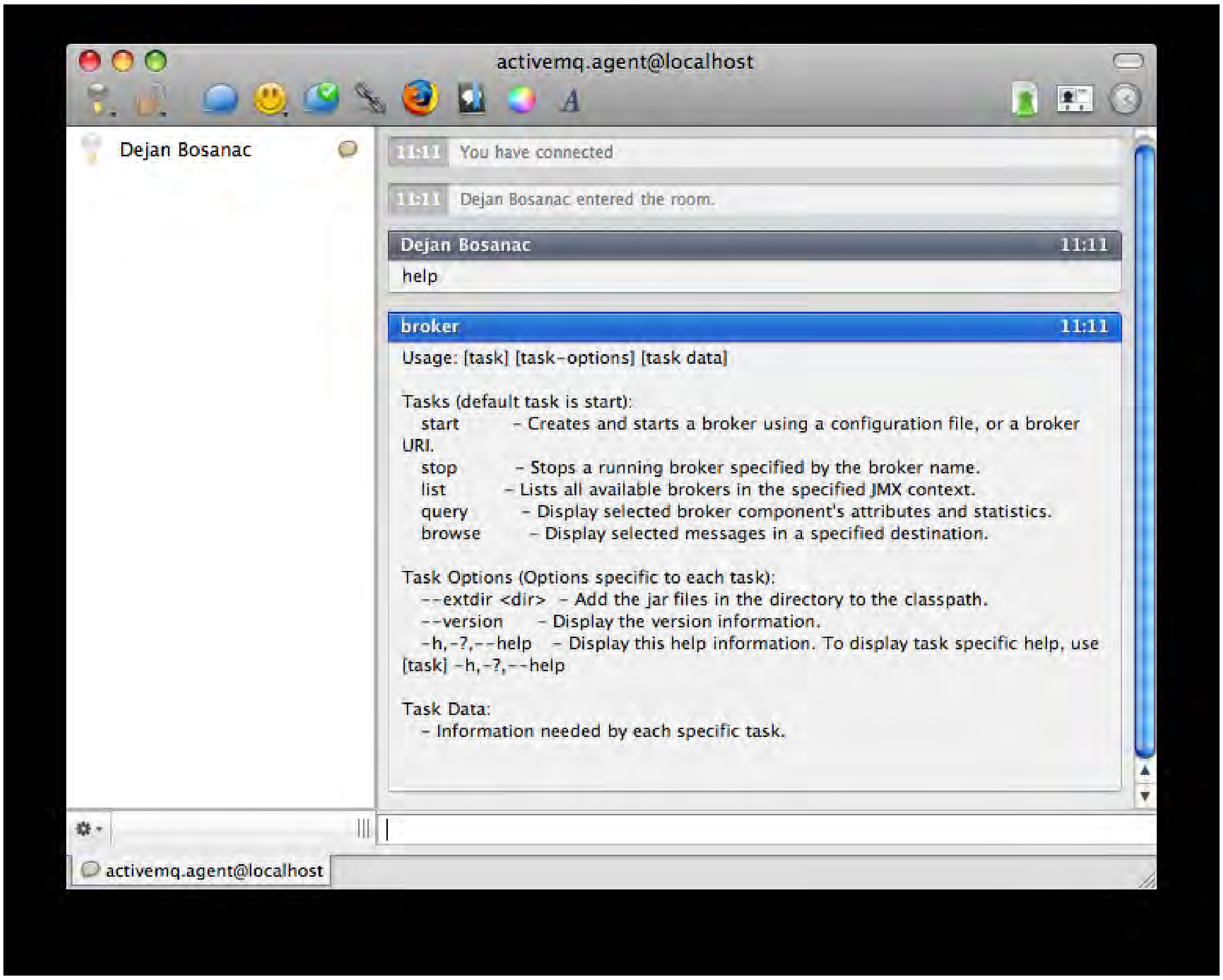
With Message:

Figure 14.1: 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 14.2 below:

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496



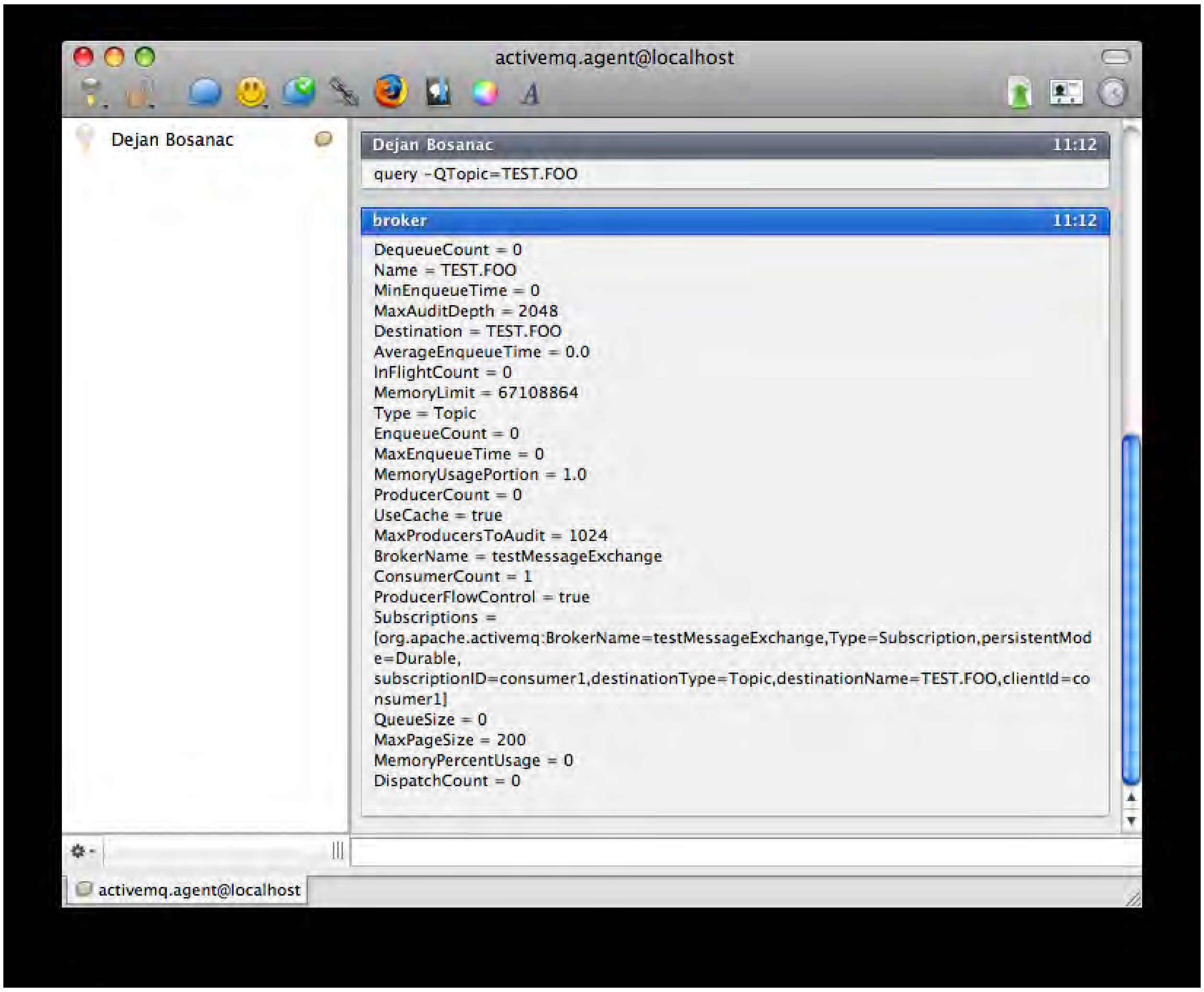
```
11:11  You have connected
11:11  Dejan Bosanac entered the room.

Dejan Bosanac                                            11:11
help

broker                                                   11:11
Usage: [task] [task-options] [task data]

Tasks (default task is start):
   start        - Creates and starts a broker using a configuration file, or a broker
URI.
   stop         - Stops a running broker specified by the broker name.
   list         - Lists all available brokers in the specified JMX context.
   query        - Display selected broker component's attributes and statistics.
   browse       - Display selected messages in a specified destination.

Task Options (Options specific to each task):
   --extdir <dir>  - Add the jar files in the directory to the classpath.
   --version       - Display the version information.
   -h,-?,--help    - Display this help information. To display task specific help, use
[task] -h,-?,--help

Task Data:
   - Information needed by each specific task.
```

Figure 14.2: Command Agent help

Of course, more complex commands are supported as well. Figure 14.3 shows how you can query the topic named `TEST.FOO` using the `query -QTopic=TEST.FOO` command.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496



Dejan Bosanac 11:12

```
query -QTopic=TEST.FOO
```

broker 11:12

```
DequeueCount = 0
Name = TEST.FOO
MinEnqueueTime = 0
MaxAuditDepth = 2048
Destination = TEST.FOO
AverageEnqueueTime = 0.0
InFlightCount = 0
MemoryLimit = 67108864
Type = Topic
EnqueueCount = 0
MaxEnqueueTime = 0
MemoryUsagePortion = 1.0
ProducerCount = 0
UseCache = true
MaxProducersToAudit = 1024
BrokerName = testMessageExchange
ConsumerCount = 1
ProducerFlowControl = true
Subscriptions =
[org.apache.activemq:BrokerName=testMessageExchange,Type=Subscription,persistentMod
e=Durable,
subscriptionID=consumer1,destinationType=Topic,destinationName=TEST.FOO,clientId=co
nsumer1]
QueueSize = 0
MaxPageSize = 200
MemoryPercentUsage = 0
DispatchCount = 0
```

Figure 14.3: 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.

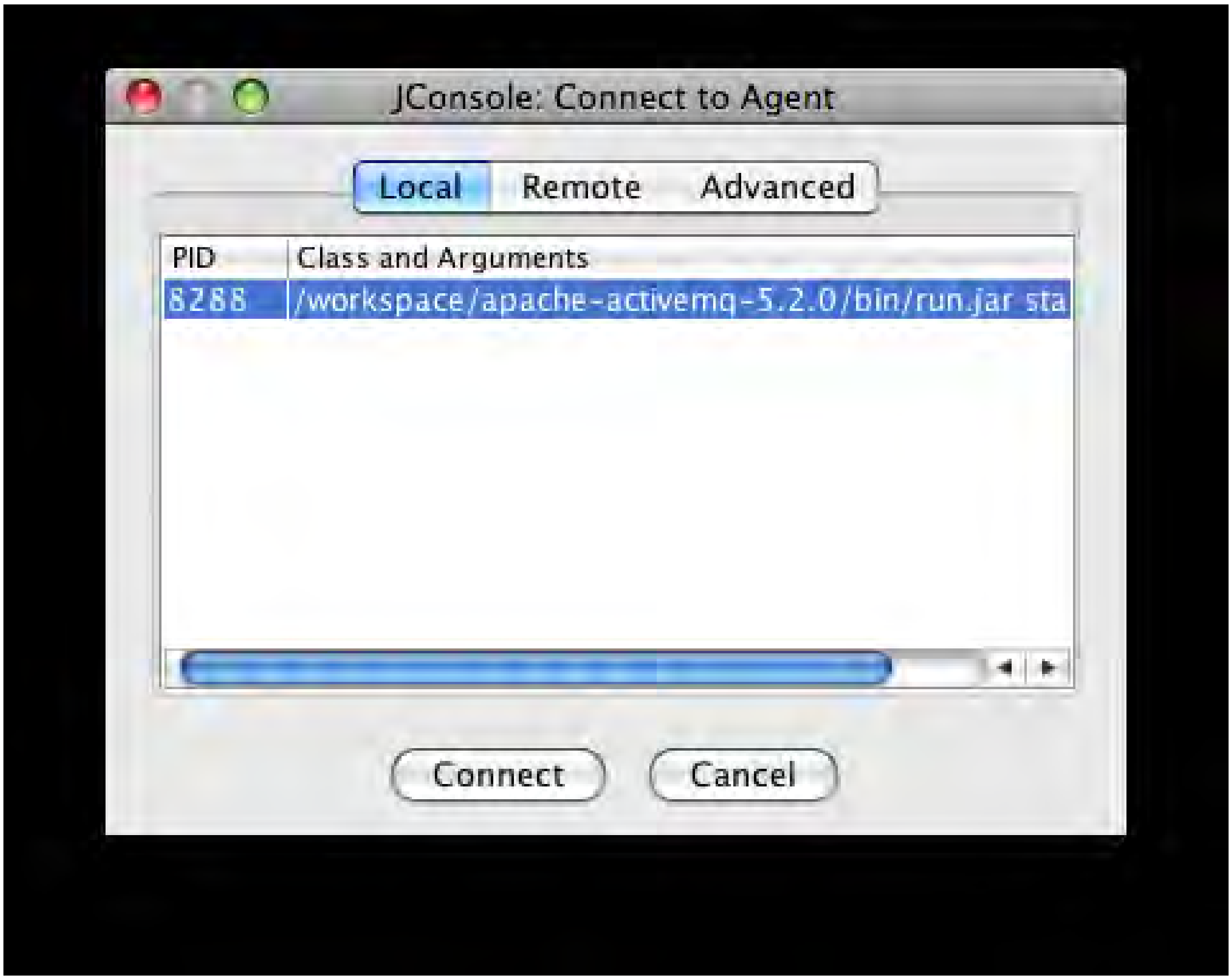
14.2.3. JConsole

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 14.4).



PID	Class and Arguments
8288	/workspace/apache-activemq-5.2.0/bin/run.jar sta

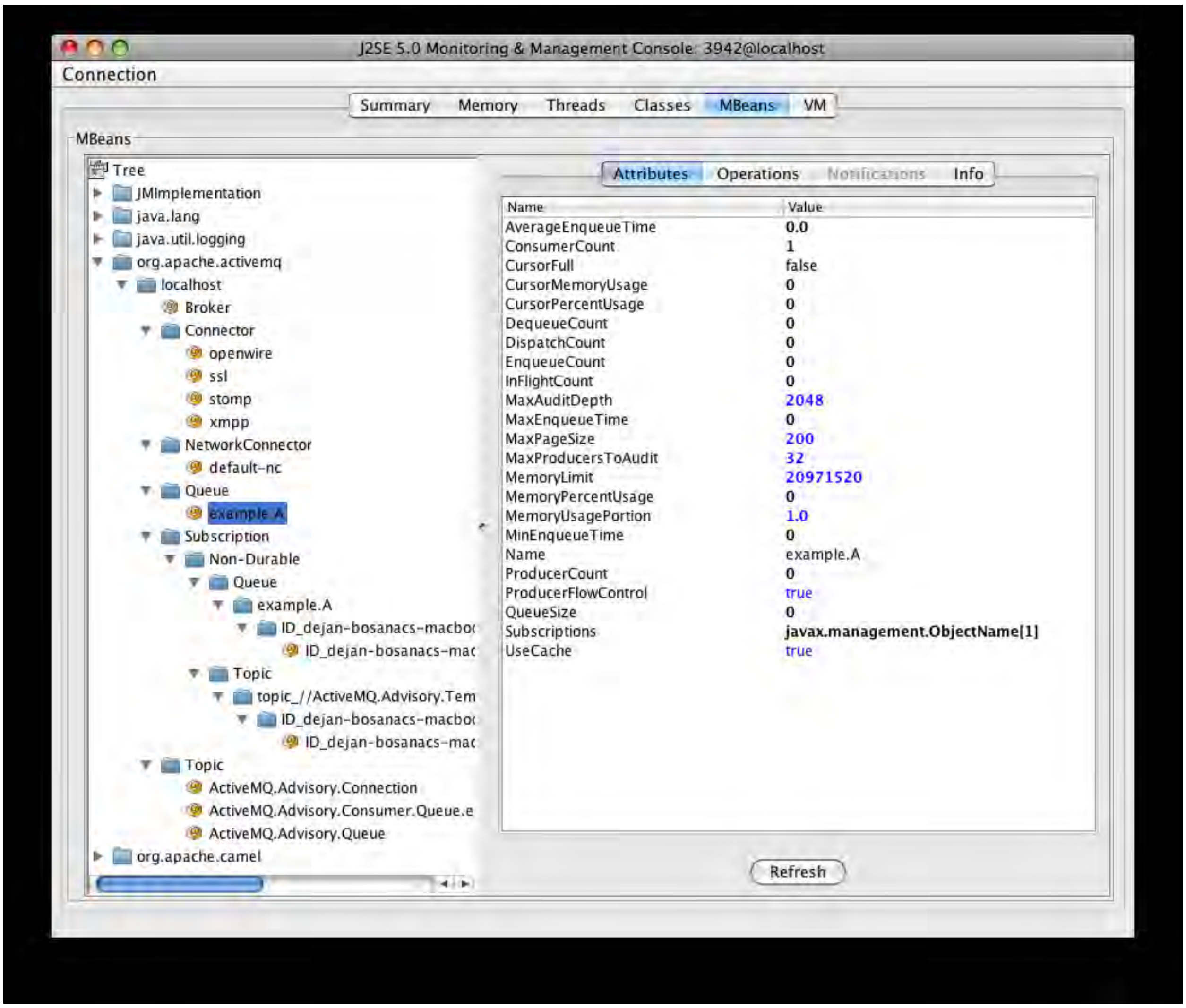
Figure 14.4: JConsole connect

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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/jmxrmi`) in the *Advanced* tab.

Upon successfully connecting to the local ActiveMQ broker, Figure 14.5 demonstrates some of what you are able to see.



```
Tree
▶ JMImplementation
▶ java.lang
▶ java.util.logging
▼ org.apache.activemq
  ▼ localhost
      Broker
    ▼ Connector
        openwire
        ssl
        stomp
        xmpp
    ▼ NetworkConnector
        default-nc
    ▼ Queue
        example.A
    ▼ Subscription
      ▼ Non-Durable
        ▼ Queue
          ▼ example.A
            ▼ ID_dejan-bosanacs-macboo
                ID_dejan-bosanacs-mac
        ▼ Topic
          ▼ topic_//ActiveMQ.Advisory.Tem
            ▼ ID_dejan-bosanacs-macboo
                ID_dejan-bosanacs-mac
    ▼ Topic
        ActiveMQ.Advisory.Connection
        ActiveMQ.Advisory.Consumer.Queue.e
        ActiveMQ.Advisory.Queue
▶ org.apache.camel
```

Name	Value
AverageEnqueueTime	0.0
ConsumerCount	1
CursorFull	false
CursorMemoryUsage	0
CursorPercentUsage	0
DequeueCount	0
DispatchCount	0
EnqueueCount	0
InFlightCount	0
MaxAuditDepth	2048
MaxEnqueueTime	0
MaxPageSize	200
MaxProducersToAudit	32
MemoryLimit	20971520
MemoryPercentUsage	0
MemoryUsagePortion	1.0
MinEnqueueTime	0
Name	example.A
ProducerCount	0
ProducerFlowControl	true
QueueSize	0
Subscriptions	javax.management.ObjectName[1]
UseCache	true

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

Figure 14.5: JConsole Queue View

As you can see in Figure 14.5 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 14.6.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

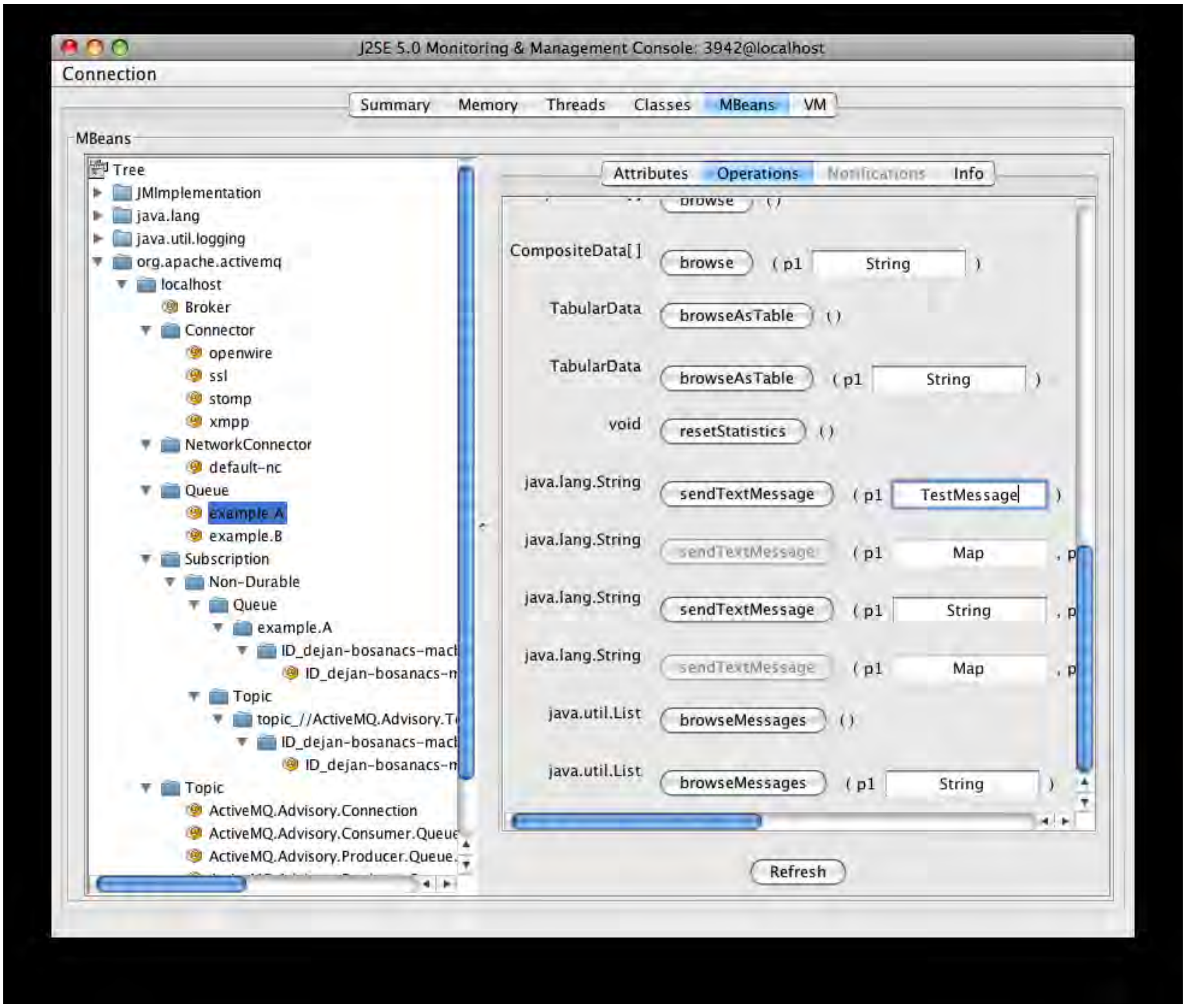

J2SE 5.0 Monitoring & Management Console: 3942@localhost
Connection
Summary
Memory
Threads
Classes
MBeans
VM
MBeans
Attributes
Operations
Notifications
Info
Tree
JMImplementation
java.lang
java.util.logging
org.apache.activemq
localhost
Broker
Connector
openwire
ssl
stomp
xmpp
NetworkConnector
default-nc
Queue
example.A
example.B
Subscription
Non-Durable
Queue
example.A
Topic
ActiveMQ.Advisory.Connection
ActiveMQ.Advisory.Consumer.Queue
ActiveMQ.Advisory.Producer.Queue
CompositeData[ ] browse (p1 String)
TabularData browseAsTable ( )
TabularData browseAsTable (p1 String)
void resetStatistics ( )
java.lang.String sendTextMessage (p1 TestMessage)
java.lang.String sendTextMessage (p1 Map
java.lang.String sendTextMessage (p1 String
java.lang.String sendTextMessage (p1 Map
java.util.List browseMessages ( )
java.util.List browseMessages (p1 String)
Refresh

Figure 14.6: JConsole Queue Operations

As you can see in Figure 14.6, 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

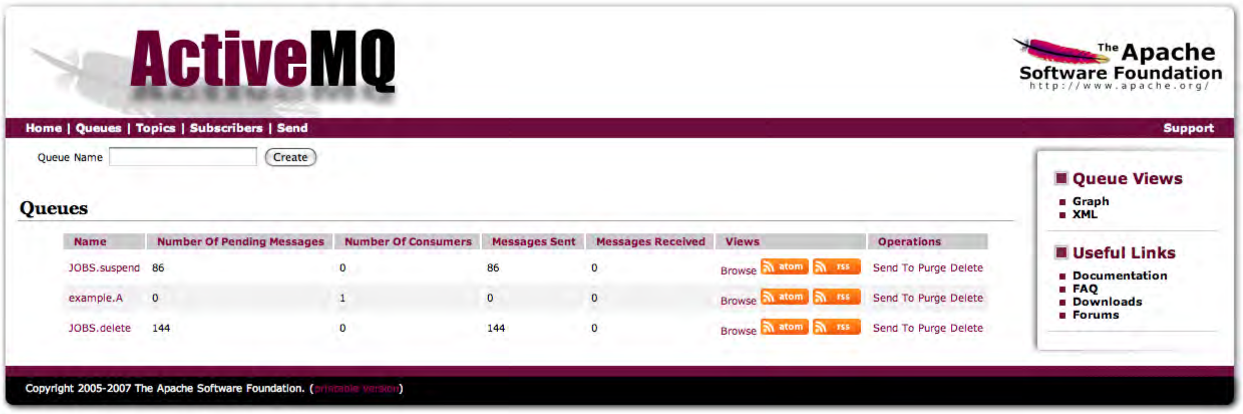
In 7: “*Connecting to ActiveMQ With Other Languages*”, we saw how an internal

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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. Figure 14.7 shows a screenshot of the web cosonle viewing a list of queues with some basic information.



Queues

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

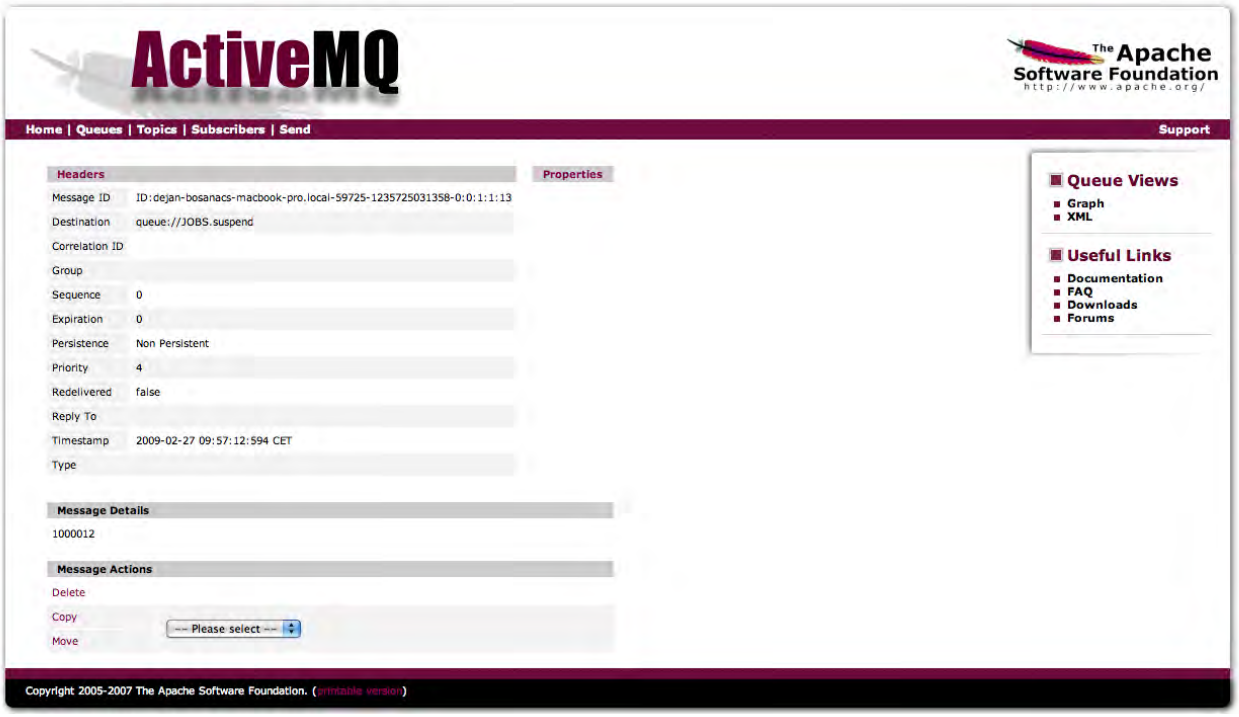
Copyright 2005-2007 The Apache Software Foundation. (printable version)

Figure 14.7: 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 14.8 shows the page that displays basic message properties.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496



Headers	
Message ID	ID:dejan-bosanacs-macbook-pro.local-59725-1235725031358-0:0:1:1:13
Destination	queue://JOBS.suspend
Correlation ID	
Group	
Sequence	0
Expiration	0
Persistence	Non Persistent
Priority	4
Redelivered	false
Reply To	
Timestamp	2009-02-27 09:57:12:594 CET
Type	

Properties

Message Details

1000012

Message Actions

Copyright 2005-2007 The Apache Software Foundation. (printable version)

Figure 14.8: 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 programatically 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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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. Listing 14.11 shows the standard logger configuration:

Listing 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 applicaton 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 st
DEBUG TcpTransport                      - Stopping transport tcp:///127.0.0.1:49383
TRACE TcpTransport                      - TCP consumer thread for tcp:///127.0.0.1:49392 st
DEBUG TcpTransport                      - Stopping transport tcp:///127.0.0.1:49392
```

In addition to starting/stopping ActiveMQ after changing the logging

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 14.9:

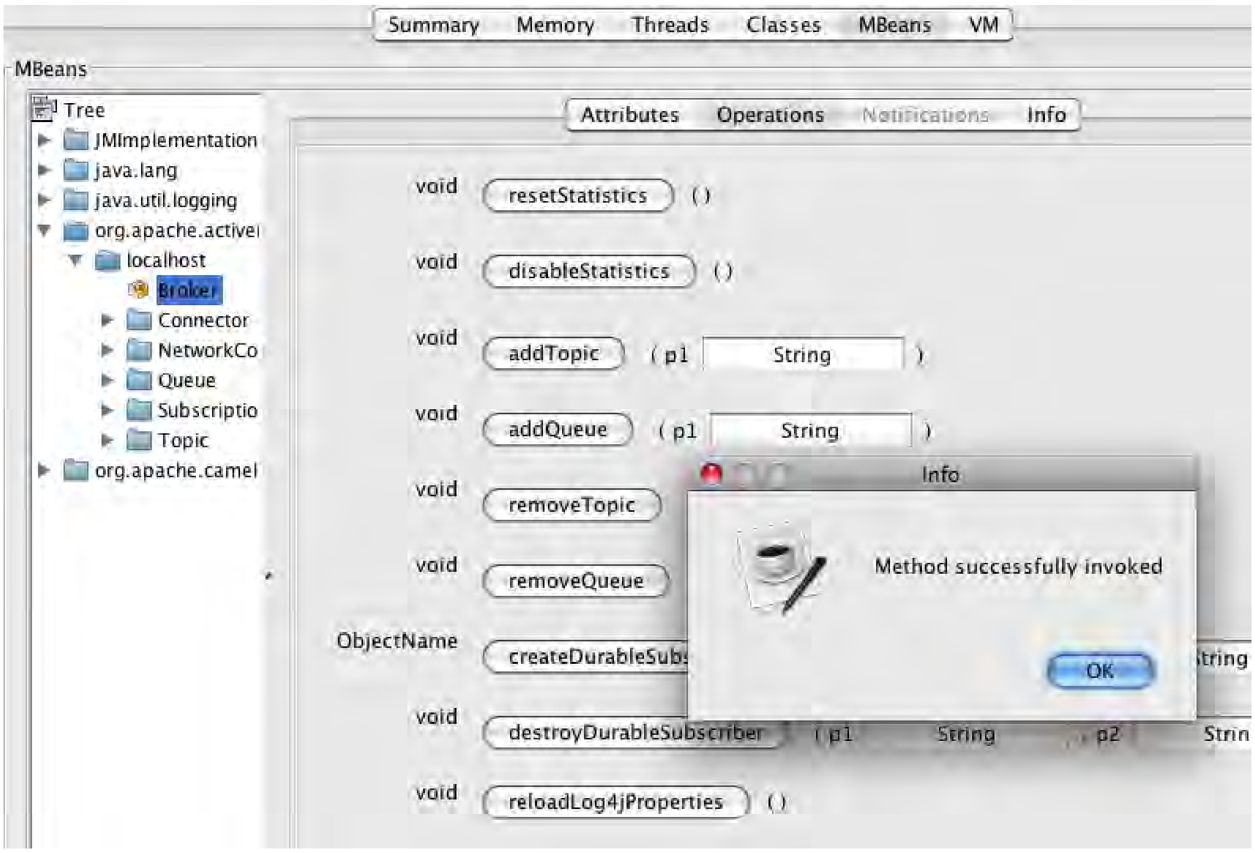


Figure 14.9: Reload Log4J properties

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.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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. Listing 14.12 shows an example Log4J configuration file that will be used in this section.

Listing 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. Addition 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.propert
-Dexec.mainClass=org.apache.activemq.book.ch14.advisory.Publisher \
-Dexec.args="failover:(tcp://localhost:61616?trace=true) IONA JAVA"
```

`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`

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

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 [ublisher.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 [ublisher.main()] DEBUG FailoverTransport
- Started.
2009-03-19 15:47:56,829 [ublisher.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.
```

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

```
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 [ublisher.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 [ublisher.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 [ublisher.main()] DEBUG Connection:11
- SENDING: SessionInfo { ... }
2009-03-19 15:48:07,239 [ublisher.main()] DEBUG Connection:11
- SENDING: ProducerInfo { ... }
Sending: {offer=51.726420585933745, price=51.67474584009366, up=false, stock=IONA}
on destination: topic://STOCKS.IONA
2009-03-19 15:48:07,266 [ublisher.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=JAVA}
on destination: topic://STOCKS.JAVA
```

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

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496

the use of standard Log4J logging. Well, similar functionality is available on the broker side using a *logging interceptor*. ActiveMQ plugins were introduced in 5: "*Securing ActiveMQ*" 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 it's development and new features.

Download at Boykma.Com

Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=496