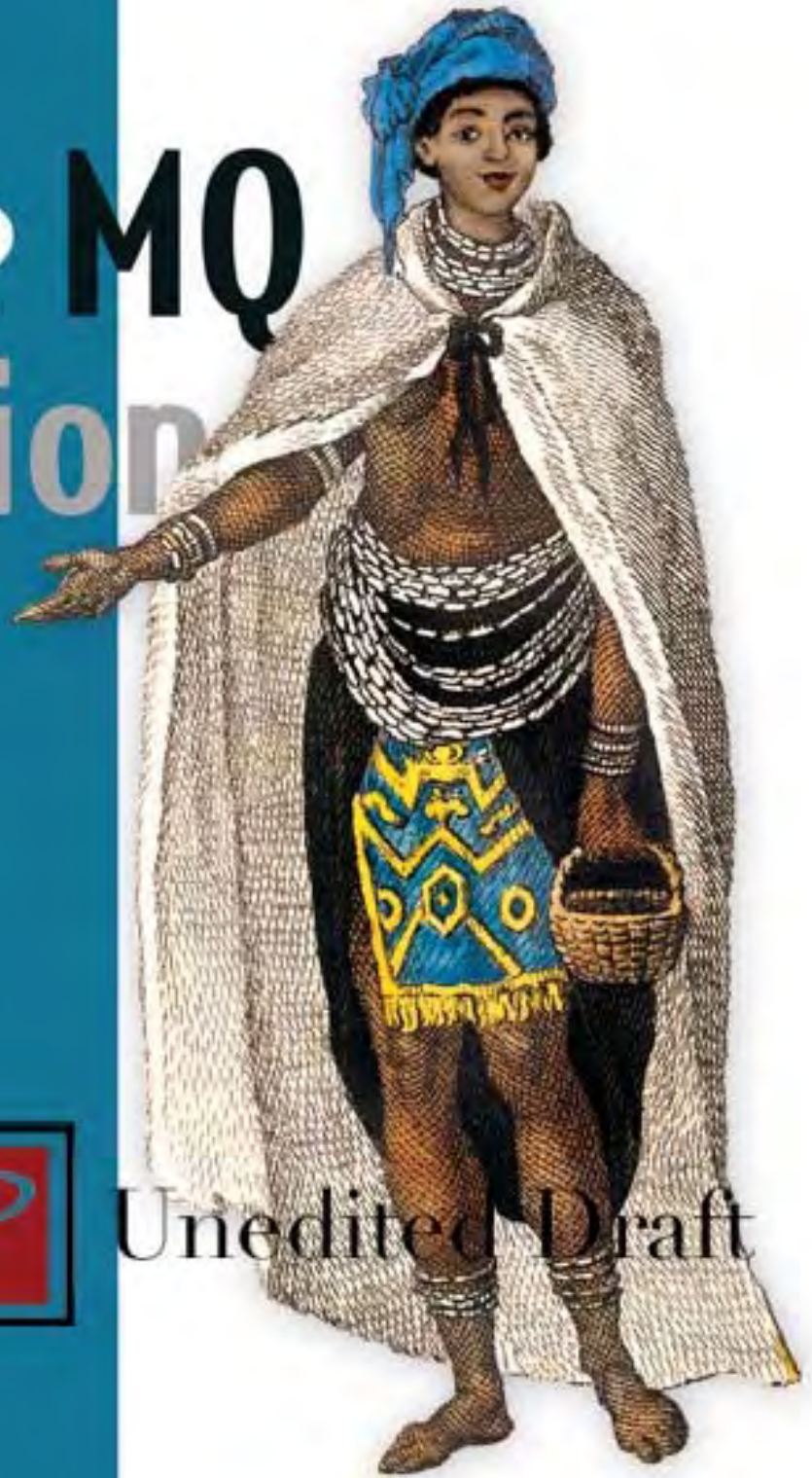


# Active MQ in Action

Bruce Snyder  
Rob Davies  
Dejan Bosanac



 MANNING



Unedited Draft



**MEAP Edition  
Manning Early Access Program**

Copyright 2009 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://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*

Preface

## **Part I Introduction**

Chapter 1 Understanding message-oriented middleware and JMS

Chapter 2 Introduction to Apache ActiveMQ

## **Part II Configuring ActiveMQ**

Chapter 3 Understanding connectors

Chapter 4 Persisting messages

Chapter 5 Securing ActiveMQ

## **Part III Using ActiveMQ**

Chapter 6 Creating Java applications with ActiveMQ

Chapter 7 Embedding ActiveMQ in Java containers

Chapter 8 Connecting to ActiveMQ with other languages

## **Part IV Advanced ActiveMQ**

Chapter 9 Broker topologies

Chapter 10 Advanced Broker features

Chapter 11 Advanced client options

Chapter 12 Tuning ActiveMQ for performance

Chapter 13 Integration patterns with ActiveMQ and Camel

Chapter 14 Administering and managing ActiveMQ

Appendix Options for various ActiveMQ components

---

# Part I. An Introduction to Messaging and ActiveMQ

[Intro goes here]

---

# 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)

- Some examples of using the JMS API

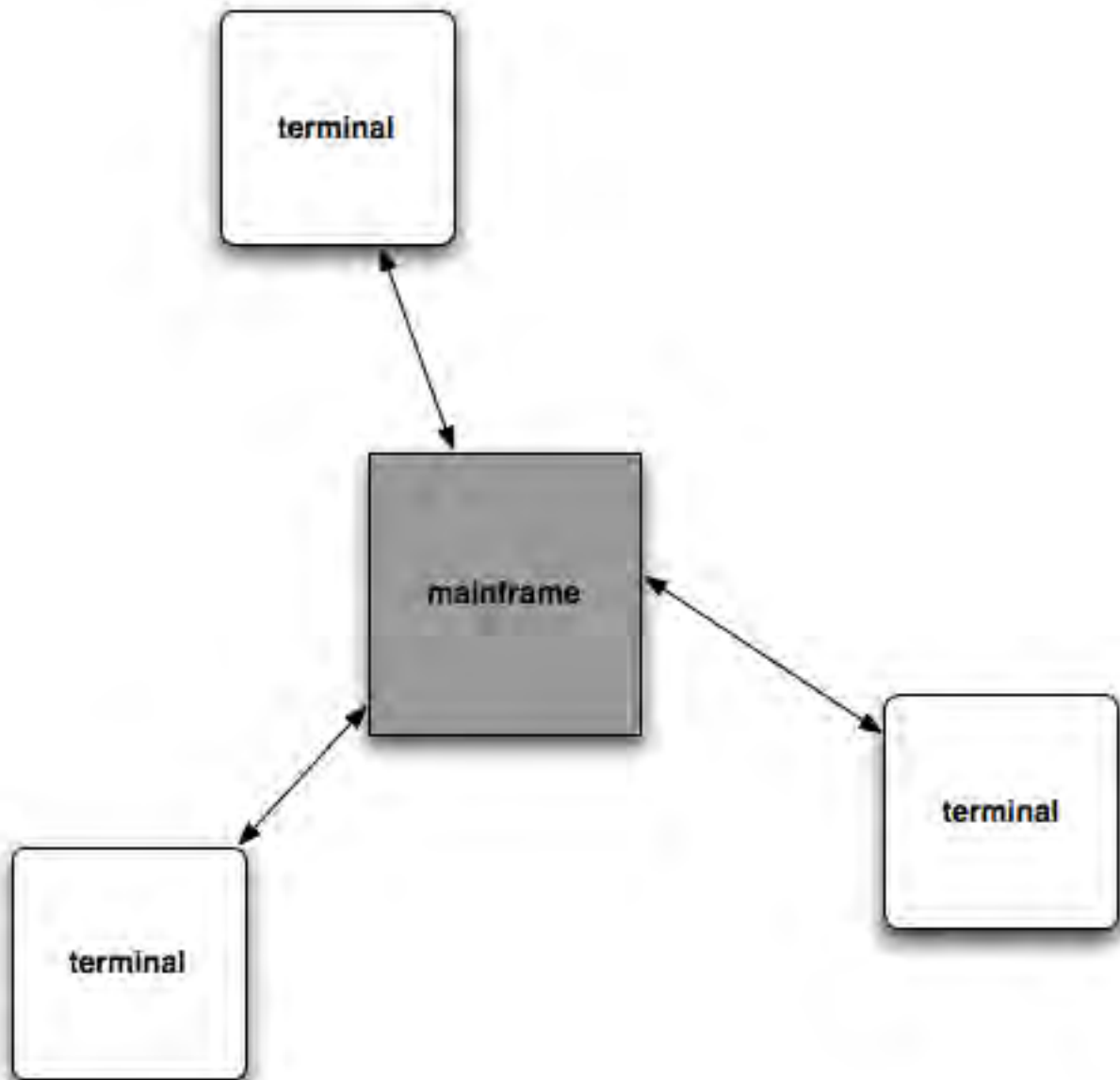
## 1.2. Introduction to Enterprise Messaging

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.

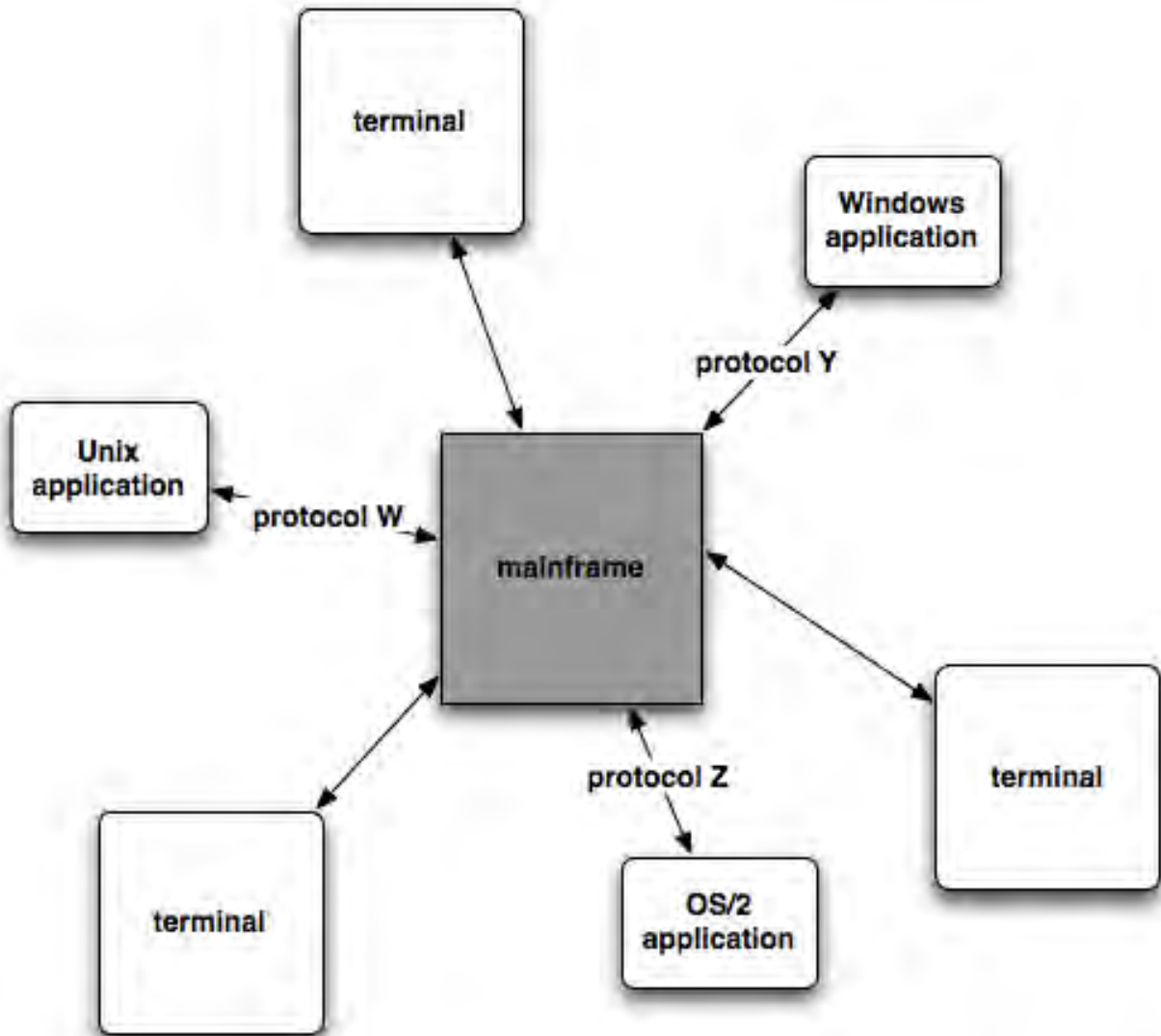
Figure 1.1 shows a diagram demonstrating how terminals connect to a mainframe.



**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

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

















































































































































































































































































































































































```
IONA 35.21 35.25 up
IONA 35.12 35.15 down
JAVA 37.26 37.30 down
```

If you like Rails-like frameworks, you can also check out the ActiveMessaging project (<http://code.google.com/p/activemessaging/>), which brings "simplicity and elegance of rails development to messaging".

Now let's see how to implement a similar stock portfolio data consumer in Python.

### 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"
        )
```

```
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

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

?>
```

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

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

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.

## 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
        }
    }
}
```

```
        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
    {
        IMessage TextMessage = message as IMessage;
        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++

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
```

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 occurred. Shutting down client.\n");
}
```

As you can see, thus far CMS mimics the JMS API completely, which is great for

developers wanting to create cross-platform solutions.

The complete code related to creating and running a consumer is located in the `runConsumer()` method. Here, we have all classic elements of creating a consumer with the appropriate message listener as we have seen in our Java examples. We create a connection, session and destination objects first and then instantiate a consumer and adds this object as a message listener.

### 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 );      #F
        consumer->setMessageListener( this );                   #G

    } catch (CMSException& e) {
        e.printStackTrace();
    }
}

#A Define Connection Factory
#B Create Sonnection
#C Start Sonnection
#D Create Session
#E Create Destination

```

```
#F Create Consumer
#G Add Message Listener
```

All that is left to be done is to initialize everything and run the application.

### 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
```

and run the example with:

```
$ src/examples/simple_async_consumer
=====
Starting the example:
-----
Press 'q' to quit
Message #1 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.33014546680271</price></stock></xml>
Message #2 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.638920307293986</price></stock></xml>
Message #3 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.828934270661314</price></stock></xml>
Message #4 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.41909588529107</price></stock></xml>
Message #5 Received: <?xml version="1.0" ?><stock name="JAVA"><price>53.95595590764363</price></stock></xml>
Message #6 Received: <?xml version="1.0" ?><stock name="JAVA"><price>53.74094054512154</price></stock></xml>
Message #7 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.24485518988984</price></stock></xml>
Message #8 Received: <?xml version="1.0" ?><stock name="JAVA"><price>53.72415991559902</price></stock></xml>
Message #9 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.091504162570935</price></stock></xml>
Message #10 Received: <?xml version="1.0" ?><stock name="JAVA"><price>53.960072785363025</price></stock></xml>
Message #11 Received: <?xml version="1.0" ?><stock name="JAVA"><price>53.82082586259414</price></stock></xml>
Message #12 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.032852016137156</price></stock></xml>
Message #13 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.31012714496037</price></stock></xml>
Message #14 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.76169437095268</price></stock></xml>
Message #15 Received: <?xml version="1.0" ?><stock name="JAVA"><price>54.92747610279041</price></stock></xml>
Message #16 Received: <?xml version="1.0" ?><stock name="JAVA"><price>55.074451578258646</price></stock></xml>
Message #17 Received: <?xml version="1.0" ?><stock name="JAVA"><price>55.56804101263522</price></stock></xml>
Message #18 Received: <?xml version="1.0" ?><stock name="JAVA"><price>55.77722621685933</price></stock></xml>
Message #19 Received: <?xml version="1.0" ?><stock name="JAVA"><price>55.58192302489994</price></stock></xml>
Message #20 Received: <?xml version="1.0" ?><stock name="JAVA"><price>55.561645147383295</price></stock></xml>
```

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

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

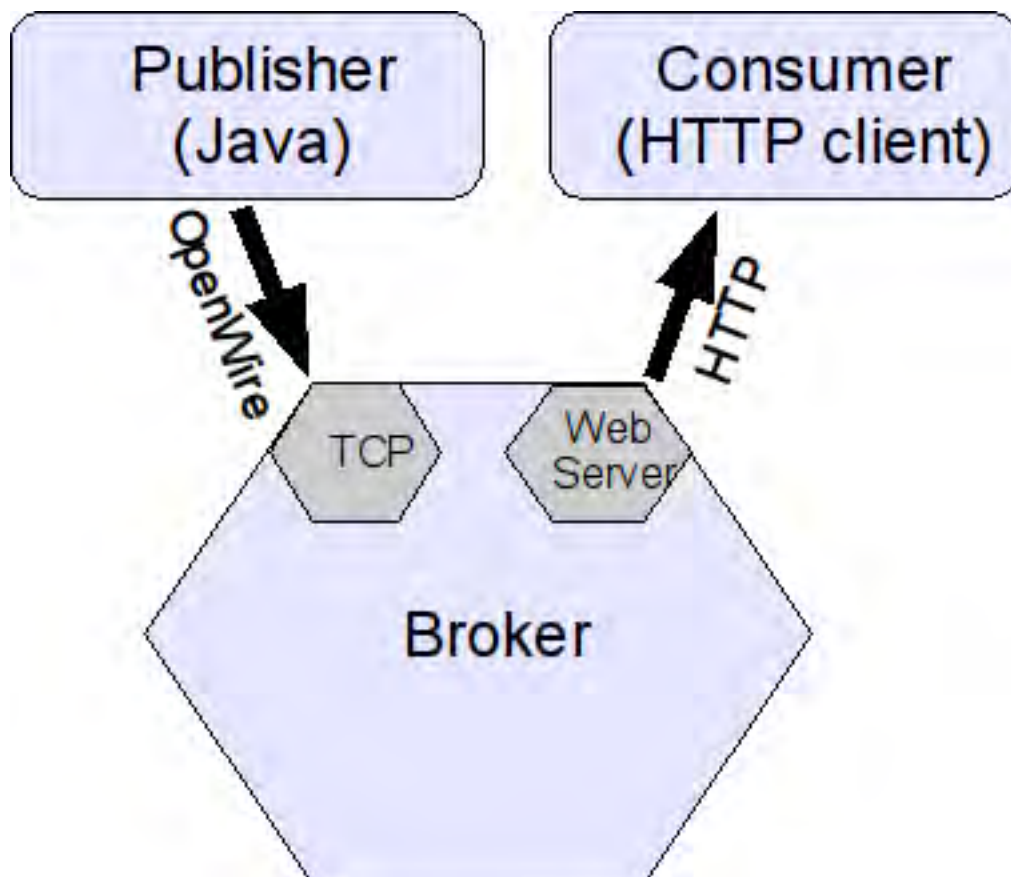
hand, is used to send data to be processed by the server. Apply these principles to your application's resources (destinations and messages in case of a JMS broker) and you have defined a RESTful API. Now let's see how ActiveMQ implements its REST API and how you can use it to send and receive messages from the broker.

ActiveMQ comes with the embedded Web server which starts at the same time your broker starts. This web server is used to provide all necessary Web infrastructure for ActiveMQ broker, including the REST API. By default, the demo application is started at:

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

and it is also configured to expose the REST API at the following URL:

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



**Figure 8.2. Figure 8.2: Rest Example**

The API is implemented by the `org.apache.activemq.web.MessageServlet` servlet and if you wish to configure an arbitrary servlet container to expose the ActiveMQ REST API, you have to define and map this servlet in an appropriate `web.xml` file (of course, all necessary dependencies should be in your classpath). The following listing shows how to configure and map this servlet to the `/message` path as it is done in the demo application.

### 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.

### 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

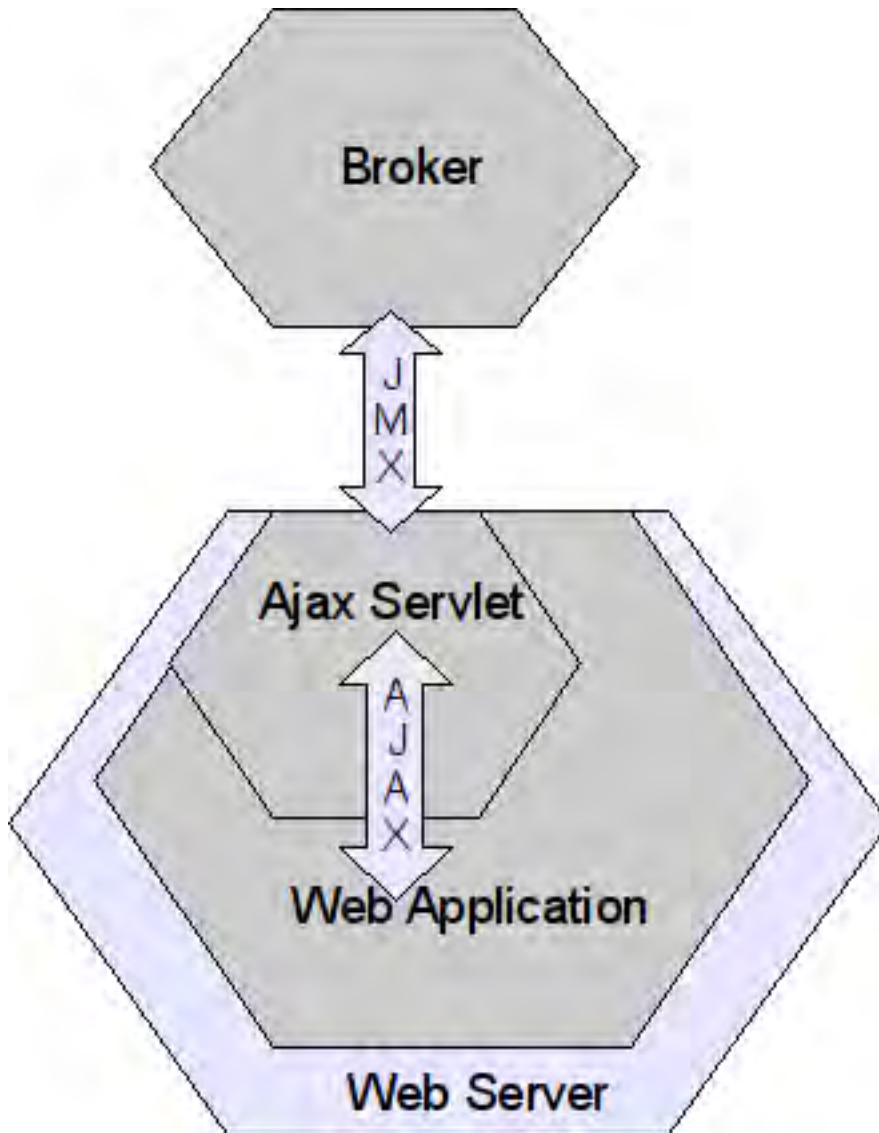
```
$ curl -d "body=message" http://localhost:8161/demo/message/STOCKS/JAVA?type=topic
```

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`

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.



**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**

```
<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

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
  }
}
```

```
    }  
}  
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.

## 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 tuning, monitoring, etc. Now that you know all the basics of ActiveMQ, this final part should teach you how to use your ActiveMQ broker instances to the maximum.

We will start by continuing our discussion started in Chapter 3, Understanding Connectors, regarding network connectors. The following chapter discusses various broker topologies and how they can help you implement functionalities such as load balancing and high availability.

---

# 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

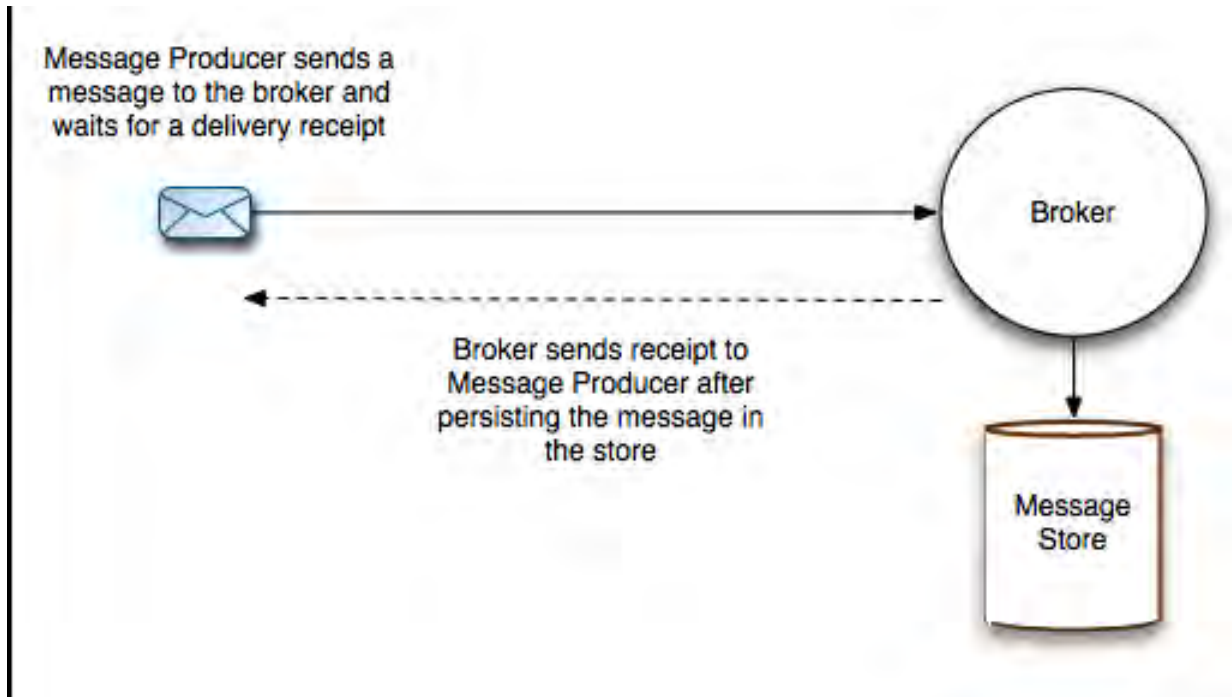
we will explain why non-persistent message delivery and batching messages are faster, and why they could be applicable to use in your application - if you don't need to absolutely guarantee that messages will never, ever be lost.

### 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



**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);
```

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 JMSEException {
    //create a default connection - we'll assume a broker is running
    //with its default configuration
    ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
    Connection connection = cf.createConnection();
    connection.start();

    //create a transacted session

    Session session = connection.createSession(true, Session.SESSION_TRANSACTED);
    Topic topic = session.createTopic("Test.Transactions");
    MessageProducer producer = session.createProducer(topic);
    int count =0;
    for (int i =0; i < 1000; i++) {
        Message message = session.createTextMessage("message " + i);
        producer.send(message);

        //commit every 10 messages

        if (i!=0 && i%10==0){
```

```
        session.commit();
    }
}

public void sendNonTransacted() throws JMSEException {

    //create a default connection - we'll assume a broker is running
    //with its default configuration

    ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory();
    Connection connection = cf.createConnection();
    connection.start();

    //create a default session (no transactions)

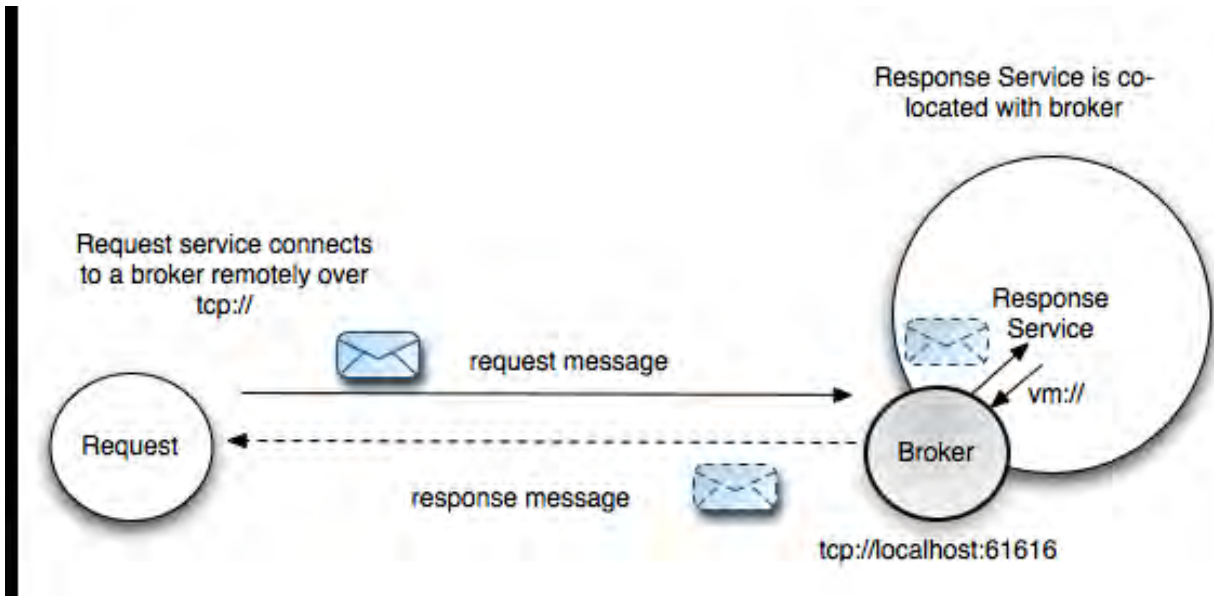
    Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Topic topic = session.createTopic("Test.Transactions");
    MessageProducer producer = session.createProducer(topic);
    int count = 0;
    for (int i = 0; i < 1000; i++) {
        Message message = session.createTextMessage("message " + i);
        producer.send(message);
    }
}
```

So we've covered some of the easy pickings in terms of performance, use non-persistent messaging where you can and now use transaction boundaries for persistent messages if it makes sense for your application. We are now going to start (slowly!) delving in to some ActiveMQ specifics covered under general techniques which can aid performance. The first of which is to use an embedded broker. Embedded brokers cut down on the amount of serialization and network traffic that ActiveMQ uses as messages can be passed around in the same JVM.

### 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

be transported across the network, making it ideal for applications that have service lots of responses very quickly.



**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 `service.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();
```

```
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 (JMSEException e) {
            e.printStackTrace();
        }
    }
});
```

You can test out the above service, with a QueueRequestor that connects to the service's embedded broker by its tcp:// transport connector - as shown in 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());
}
```

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

<b>parameter name</b>	<b>default value</b>	<b>description</b>
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

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(uri);
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 it 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.

## 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 receive data over the TCP transport. Usually the bigger the better (though this is very operating system dependent - so worth testing!). The default value is 65536 - which is the size in bytes.
- `tcpNoDelay` - the default is false. Normally a TCP socket buffers up small sizes of data before being sent. By enabling this option - messages will be sent as soon as possible. Again, its worth testing this out - as it can be operating system dependent if this boosts performance or not.

Below is in 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

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.

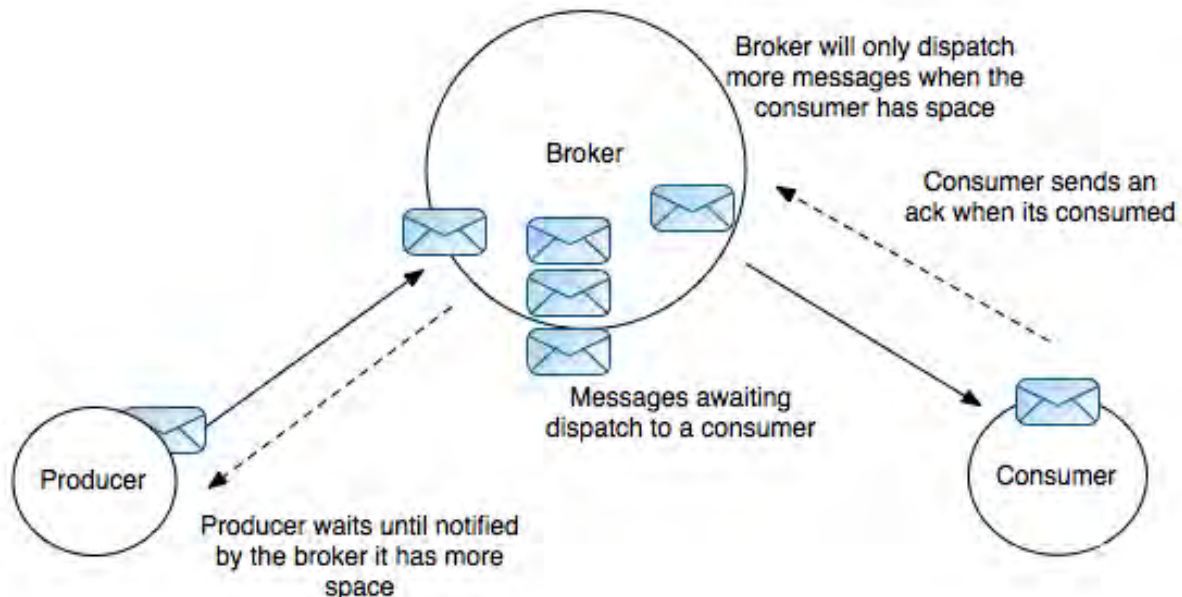
Why using asynchronous message delivery as an option for gaining performance should be well understood, and setting a property on the `ActiveMQ ConnectionFactory` is a really straight forward way of achieving that. What we are going to cover next is one of the most frequently run into gotchas with ActiveMQ - 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

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



**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

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 see what happens if you disable producer flow control, and you can do that in the Broker configuration on a destination policy like below in Listing 12.10:

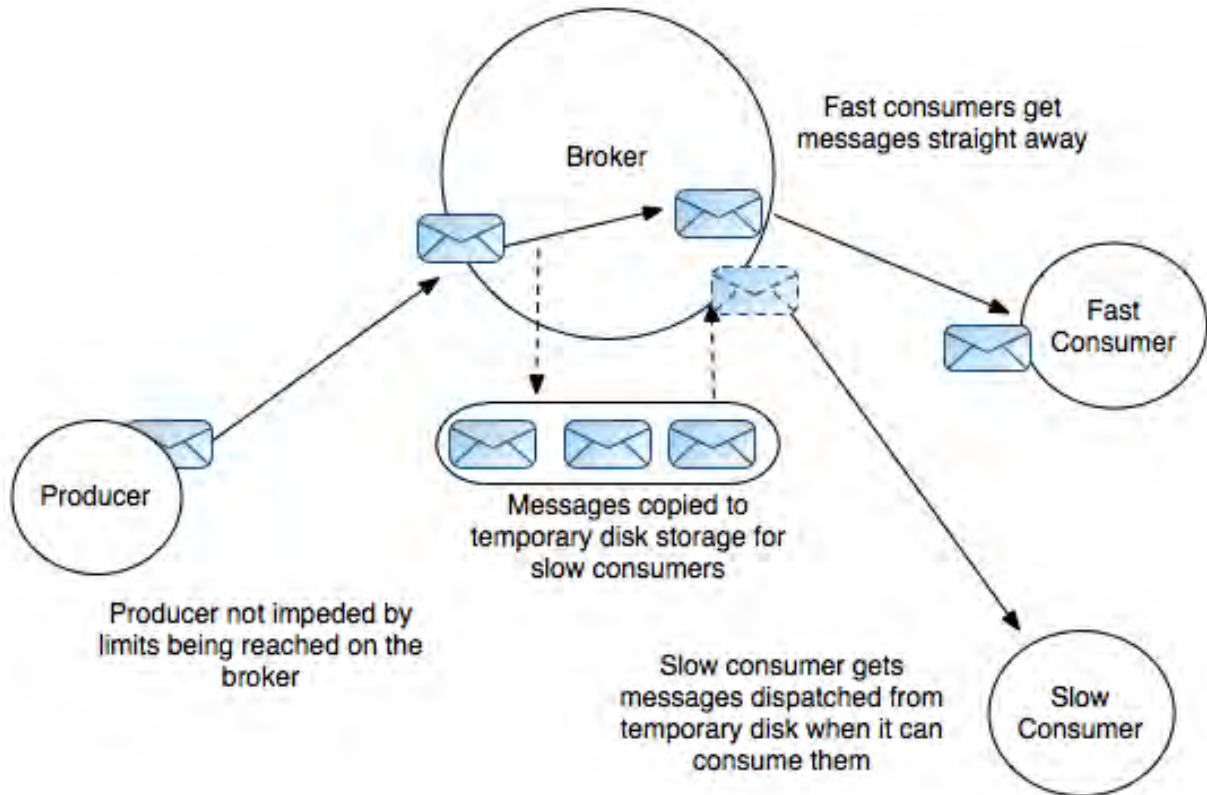
### Listing 12.10: How to disable flow control

```
<destinationPolicy>
  <policyMap>
    <policyEntries>

      <policyEntry topic="FOO.>" producerFlowControl="false" memoryLimit="10mb">
        <dispatchPolicy>
          <strictOrderDispatchPolicy/>
        </dispatchPolicy>
        <subscriptionRecoveryPolicy>
          <lastImageSubscriptionRecoveryPolicy/>
        </subscriptionRecoveryPolicy>
      </policyEntry>

    </policyEntries>
  </policyMap>
</destinationPolicy>
```

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



**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

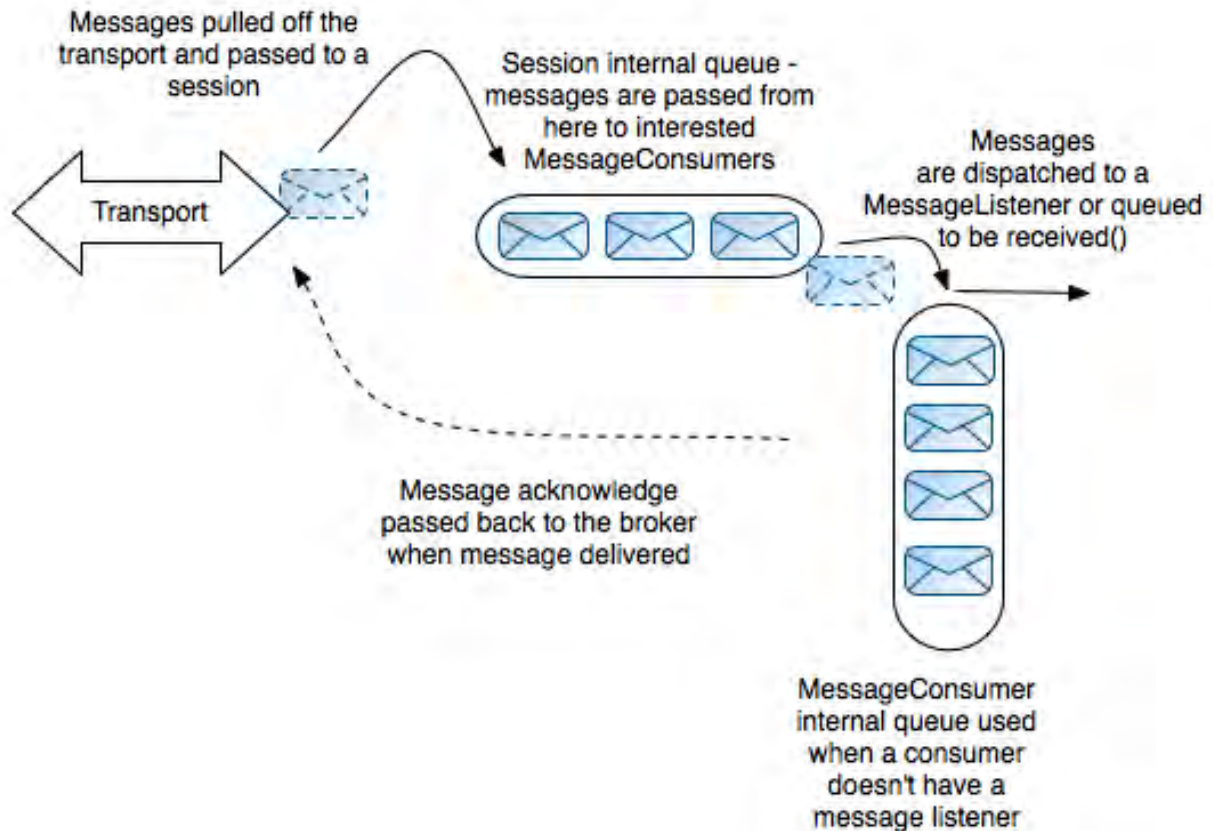
participants - and as we have seen so far, Consumers play a very big part in the overall performance of ActiveMQ. Message Consumers typically have to work twice as hard as a message Producer, because as well as consuming messages, they have to acknowledge the message has been consumed to the broker. 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.

### Internal Queues of a Consumer Connection



**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

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);
```

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();
```

```
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	Reliable acknowledgements with <code>Session.commit()</code>	Reliable way for message consumption - and performs well providing you consume more than one message in a commit.
Session.CLIENT_ACKNOWLEDGE	Messages upto when a message is acknowledged are consumed.	Can perform well, providing the application consumes a lot of messages before calling <code>acknowledge</code> .
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

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 an acknowledgement for every message consumed.	Allows great control by enabling messages to be acknowledged individually - but can be slow.
optimizeAcknowledge	Allows the consumer to send one acknowledgement back to the ActiveMQ broker for a range of messages consumed.	A hint that works in conjunction with Session.AUTO_ACKNOWLEDGE. An acknowledgement will be sent back when the prefetch limit has reached 65%. The fastest way of consuming messages.

The downside to not acknowledging every message individually is that if the MessageConsumer were to lose its connection with the ActiveMQ broker or die - then your messaging application could receive duplicate messages. However for applications that require fast throughput (e.g. real time data feeds) and are less concerned about duplicates - using optimizeAcknowledge is the recommended approach.

The ActiveMQ MessageConsumer incorporates duplicate message detection, which helps minimize the risk of receiving the same message more than once.

### 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 overhead to the consumption of messages.

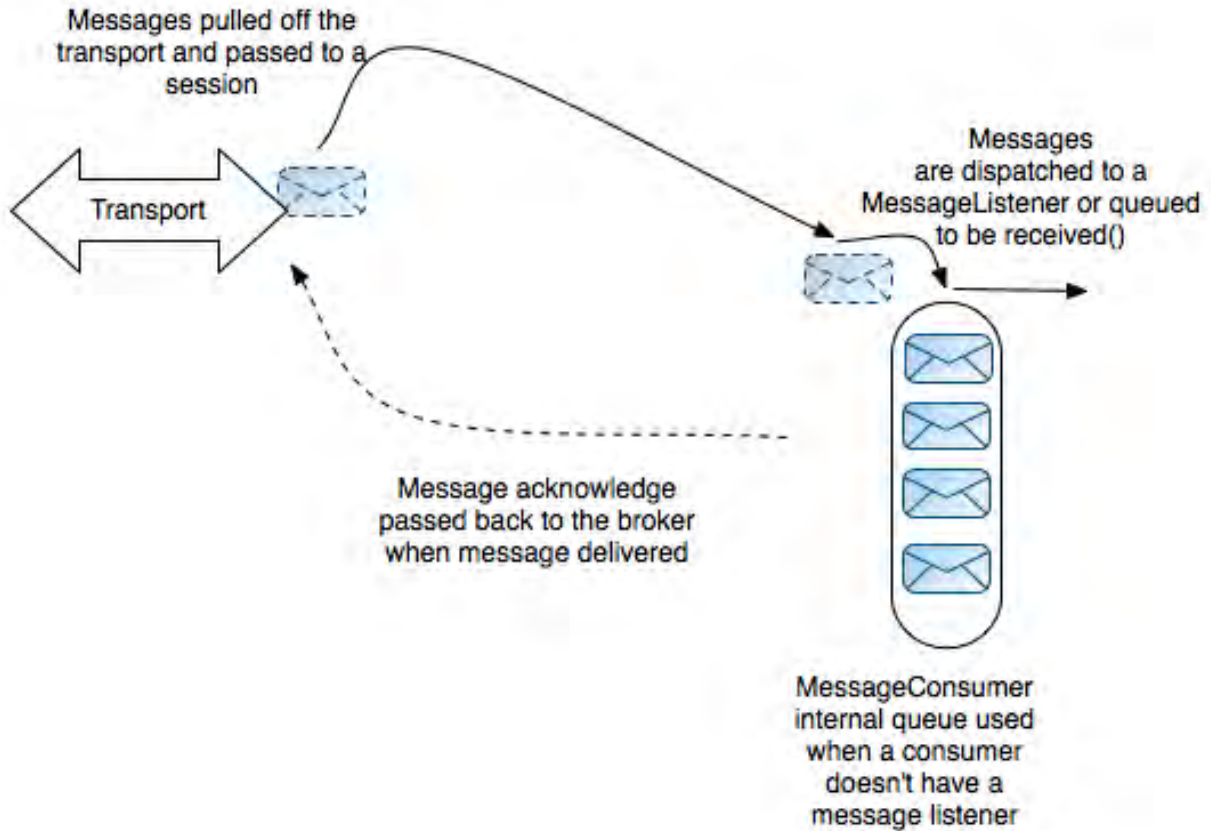
There is a property called `alwaysSessionAsync` you can disable on the ActiveMQ ConnectionFactory to turn this off - allowing messages to be passed directly from the Transport to the MessageConsumer. This property can be disabled as below in Listing 12.14:

#### 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

Internal Queues of a Consumer Connection with `alwaysSessionSync=false`



**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 -

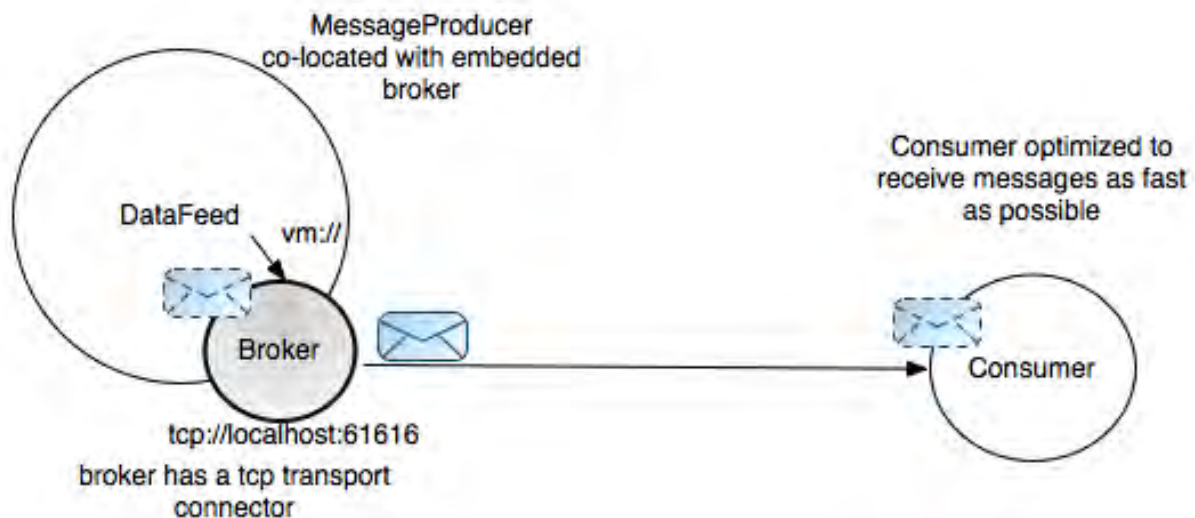
a real-time data feed.

## 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.



**Figure 12.7: Data feed application**

Firstly we setup the broker to be embedded, with the memory limit set to a

reasonable amount (64mb), limits set on to each destination and flow control disabled. The policies for the destinations in the broker are set up using a default PolicyEntry - as seen in the following code snippet in 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();
```

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://
```

```
ActiveMQConnectionFactory cf = new ActiveMQConnectionFactory("failover://(tcp://localhost:61616)");

//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 (JMSEException 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.

## 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.

---

# 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.

In addition to messaging client applications, you may have to the need to create Java applications that will monitor the broker during runtime. Some of those monitoring tasks may include:

- Obtaining broker statistics, such as number of consumers (total or per destination)
- Adding new connectors or removing existing ones
- Changing some of the broker configuration properties

For this purpose, ActiveMQ provides some mechanisms that can be used to manage and monitor your broker during runtime.

### 14.1.1. JMX

Nearly every story on management and monitoring in the Java world begins with *Java Management Extensions (JMX)*. The JMX API allows you to implement *management interfaces* for your Java applications by exposing functionality to be managed. These interfaces consist of *Management Beans*, usually called *MBeans*, which expose resources of your application to external management applications.

#### 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>
```

```
<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

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(  
            "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()); #D  
        } #D  
    }  
}
```

The example above is using the standard JMX API to access and use broker and request information. For starters, we have to create an appropriate connection to

the broker's MBean server #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

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
```

```
-----  
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

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

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 JMSEException {  
    factory = new ActiveMQConnectionFactory(brokerURL);  
    connection = factory.createConnection();  
    connection.start();  
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);  
    producer = session.createProducer(null);  
    producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);  
}
```

The consumer modified in this manner has been placed into package `org.apache.activemq.book.ch14.advisory` and we will use it in the rest of this section.

Now let's take a look at our advisory messages example application shown in Listing 14.4.

### Listing 14.4: Advisory example

```
public class Advisory {  
  
    protected static String brokerURL = "tcp://localhost:61616";  
  
}
```

```
protected static transient ConnectionFactory factory;
protected transient Connection connection;
protected transient Session session;

public Advisory() throws Exception {
    factory = new ActiveMQConnectionFactory(brokerURL);
    connection = factory.createConnection();
    connection.start();
    session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
}

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);
        System.out.println("Subscribing to advisory " + consumerTopic);
        MessageConsumer consumerAdvisory = session.createConsumer(consumerTopic);
        consumerAdvisory.setMessageListener(new ConsumerAdvisoryListener());

        Destination noConsumerTopic =
            AdvisorySupport.getNoTopicConsumersAdvisoryTopic(destination);
        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

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

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:

```
$ 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:

```
...
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”.

## 14.2.1. Command-Line Tools

You already know how to use the `bin/activemq` script to start the broker. In addition to this script, the `bin/activemq-admin` script can be used to monitor the broker state from the command line. The `activemq-admin` script provides the following functionality:

- *Start and stop* the broker
- *List* available brokers
- *Query* the broker for certain state information
- *Browse* broker destinations

In the following sections, this functionality and the command used to expose it will be explored through the use of examples. For the complete reference and explanation of all available command options, you should refer to <http://activemq.apache.org/activemq-command-line-tools-reference.html>

### 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 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

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
```

```
        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.

## Listing Available Brokers

In some situations, there may be multiple brokers running in the same JMX context. Using the `bin/activemq` script you can use the `list` command to list all the available brokers as shown in 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 -Queue=*
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
```

```
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 `-Queue=*`. 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
```

```
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:JMSSObjectClass = java.lang.Integer
JMS_BODY_FIELD:JMSSObjectString = 1000001
JMS_HEADER_FIELD:JMSEExpiration = 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:JMSSObjectClass = java.lang.Integer
JMS_BODY_FIELD:JMSSObjectString = 1000002
JMS_HEADER_FIELD:JMSEExpiration = 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:JMSSObjectClass = java.lang.Integer
JMS_BODY_FIELD:JMSSObjectString = 1000003
JMS_HEADER_FIELD:JMSEExpiration = 0
JMS_HEADER_FIELD:JMSPriority = 4
JMS_HEADER_FIELD:JMSRedelivered = false
JMS_HEADER_FIELD:JMSTimestamp = 1234789436708
```

The `browse` command is different from the previous commands as it does not use `JMX`, but browses queues using the `JMS` API. For that reason, you need to provide it with the broker URL using the `--amqurl` switch. The final parameter provided to this command is the name of the queue to be browsed.

As you can see, there are a fair number of monitoring and administration operations that can be achieved from the command line. This functionality and help you to easily check the broker's state and can be very helpful for diagnosing possible problems. But this is not the end of the administrative tools for ActiveMQ. There are still a few more advanced administrative tools and they are explained in following sections.

## 14.2.2. Command Agent

Sometimes issuing administration commands to the broker from the command line is not easily achievable, mostly in situations when you don't have a shell access to the machines hosting your brokers. In these situations you'll want to administer 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://localhost"
...

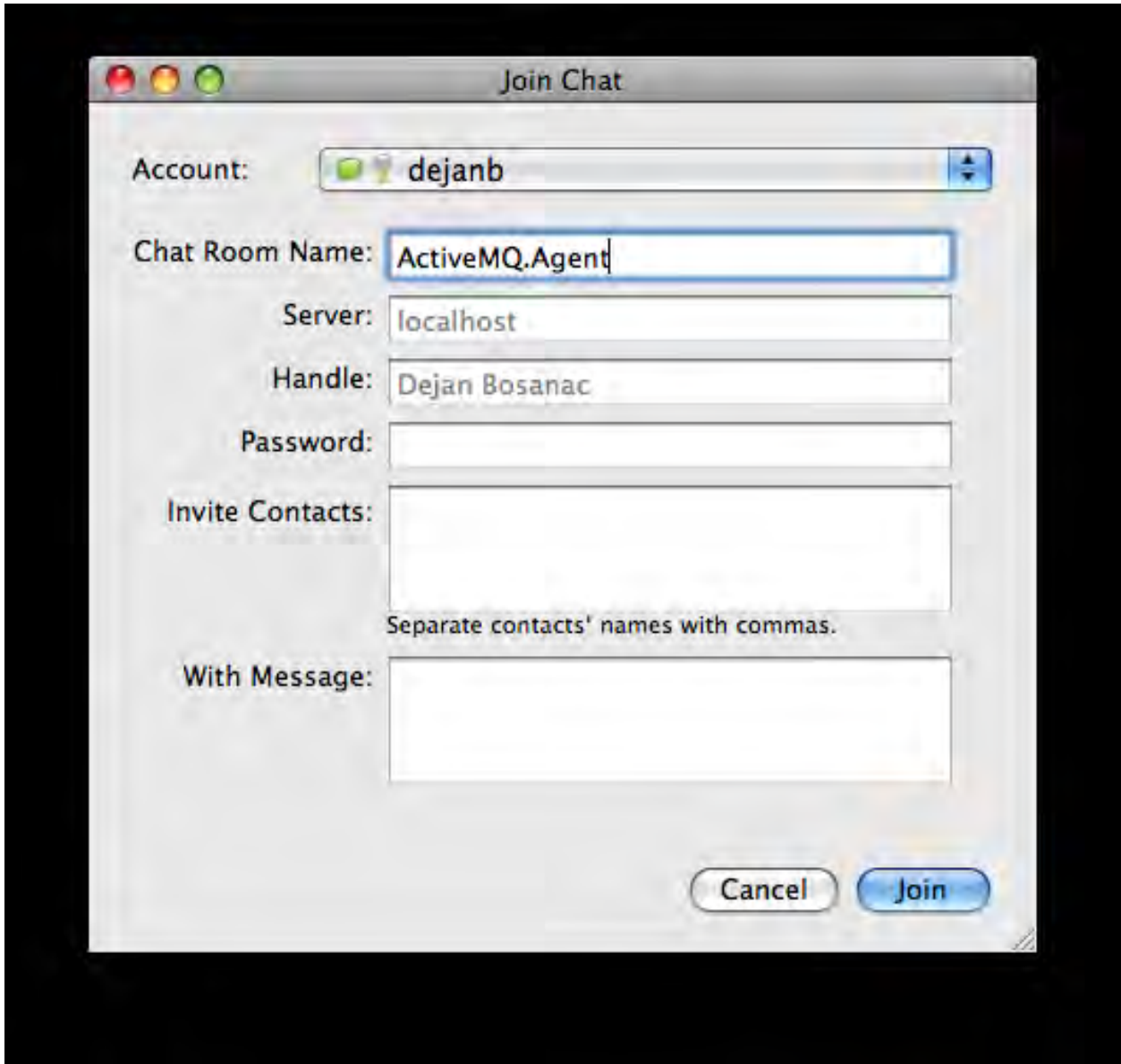
```

There are two important details in this configuration fragment. First we have started the XMPP transport connector on port 61222 to expose the broker to clients via XMPP (the *Extensible Messaging and Presence Protocol*). This was achieved by using the appropriate URI scheme, like we do for all supported protocols. XMPP is an open XML-based protocol mainly used for instant messaging and

developed by the Jabber project (<http://jabber.org/>). Since it's open and very widespread, there are a lot of *chat* clients that already support this protocol and you can use these clients to communicate with ActiveMQ.

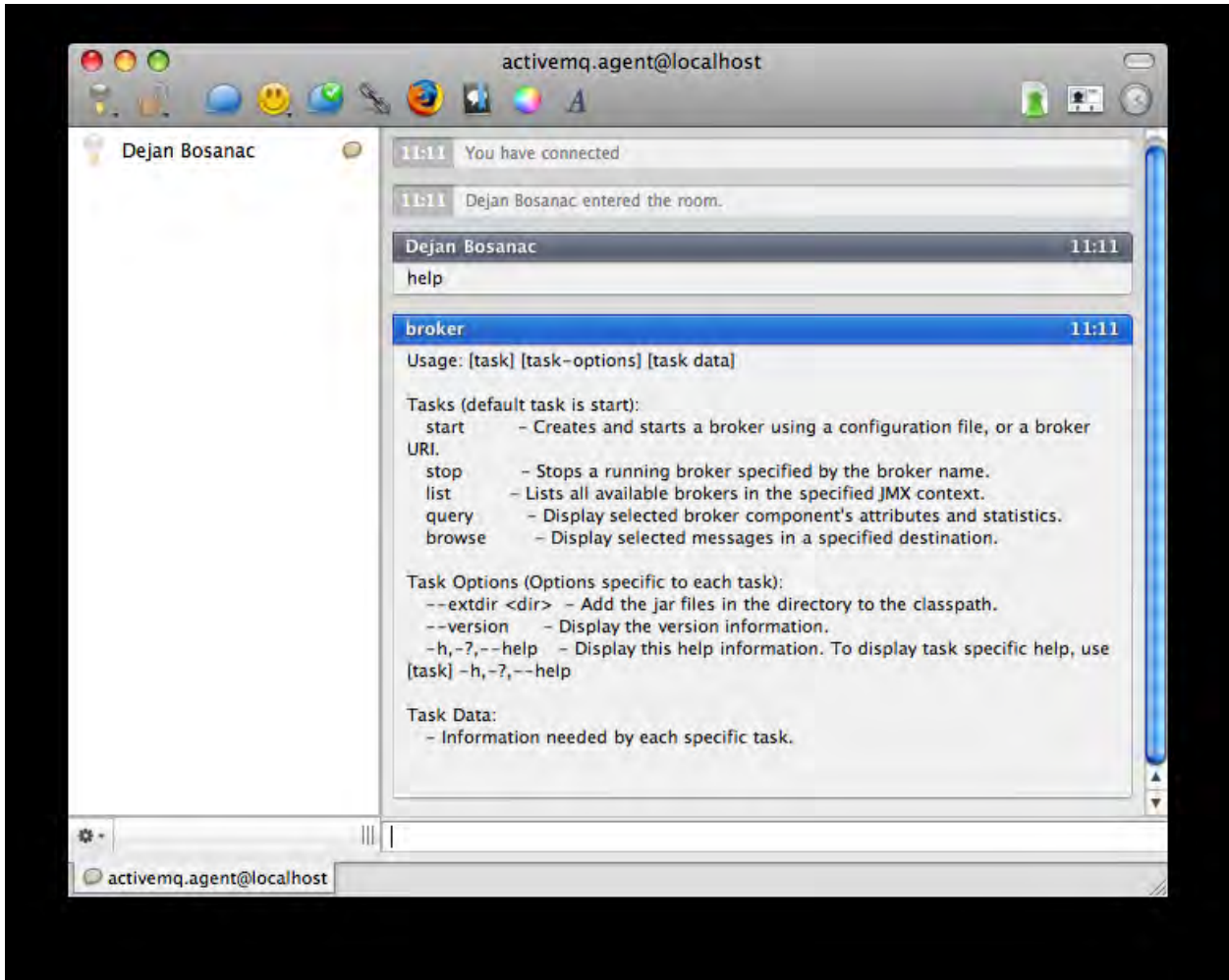
For the purpose of this book, we chose to use the Adium (<http://www.adiumx.com/>), instant messaging client. This client runs on MacOS X and speaks many different protocols including XMPP. Any XMPP client can be used here. The first step is always to provide the details to connect to ActiveMQ to the XMPP client such as server host, port, username and password. Of course, you should connect to your broker on port 61222 since that's where the XMPP transport connector is running and you can use any user and password.

After successfully connecting to the broker you have to join the appropriate chat room which basically means that you'll subscribe to the topic with the same name. In this example we will subscribe to the `ActiveMQ.Agent` topic, so we can use the command agent.



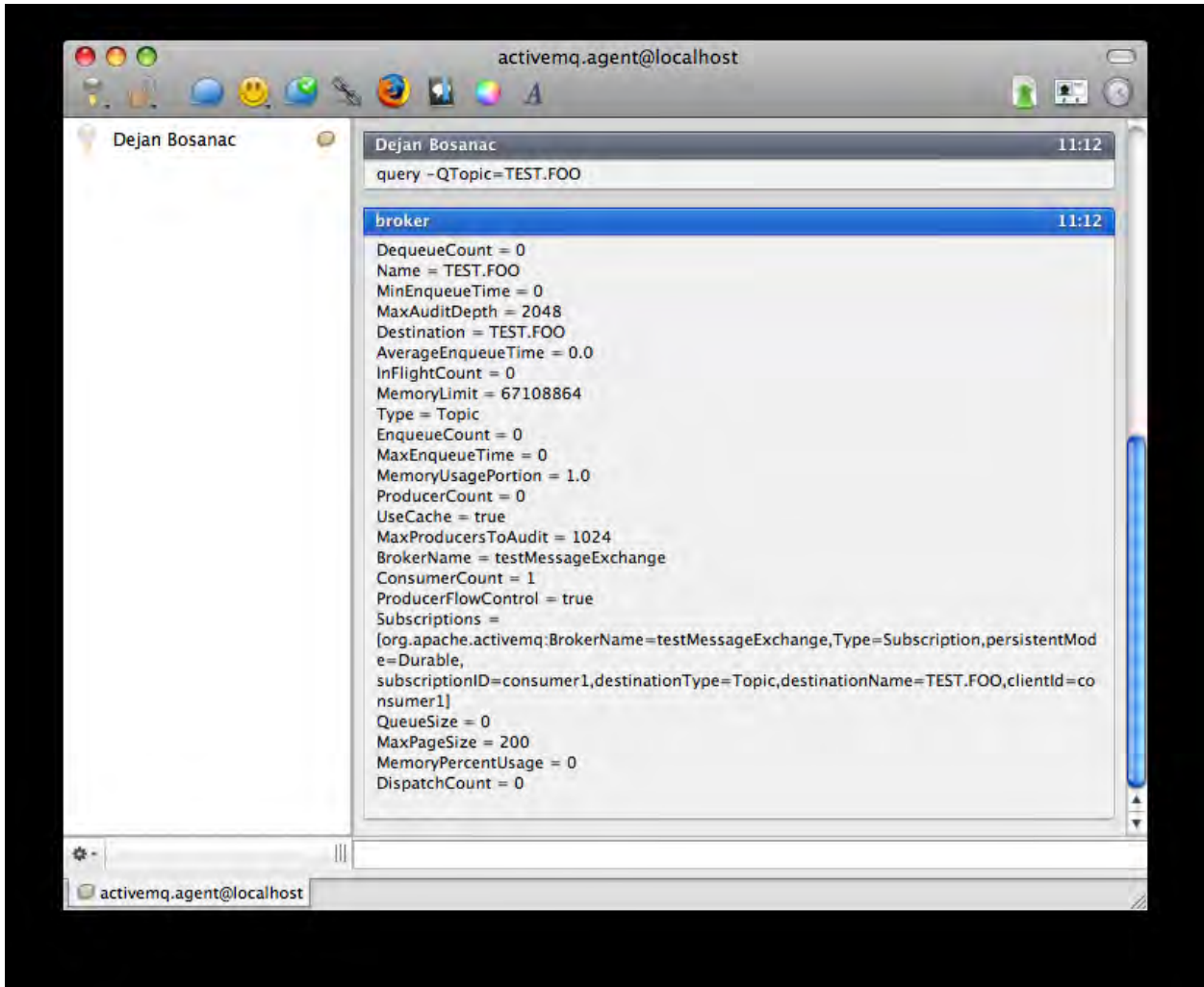
**Figure 14.1: XMPP subscribe to agent topic**

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:



**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.



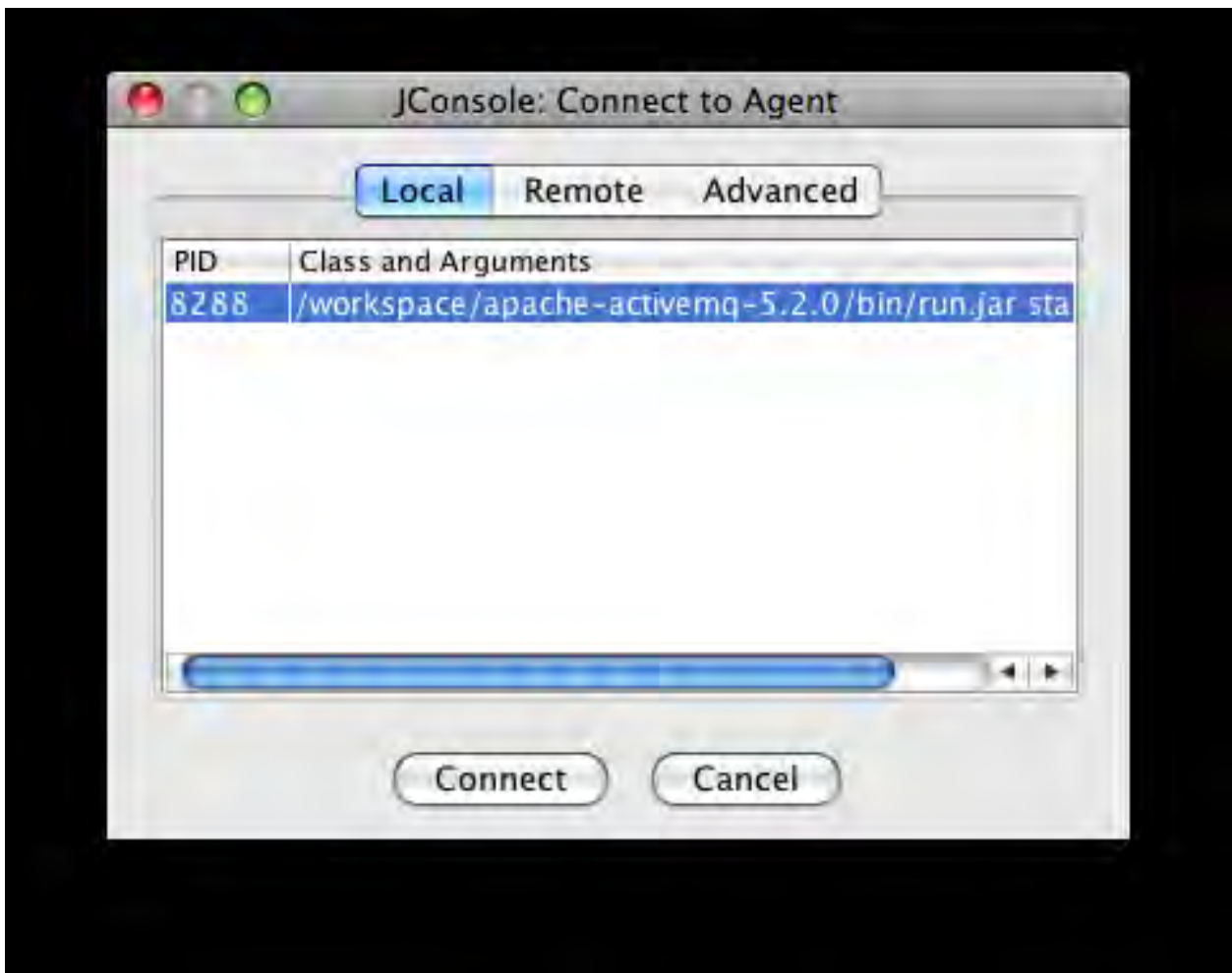
**Figure 14.3: Command Agent query**

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

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).



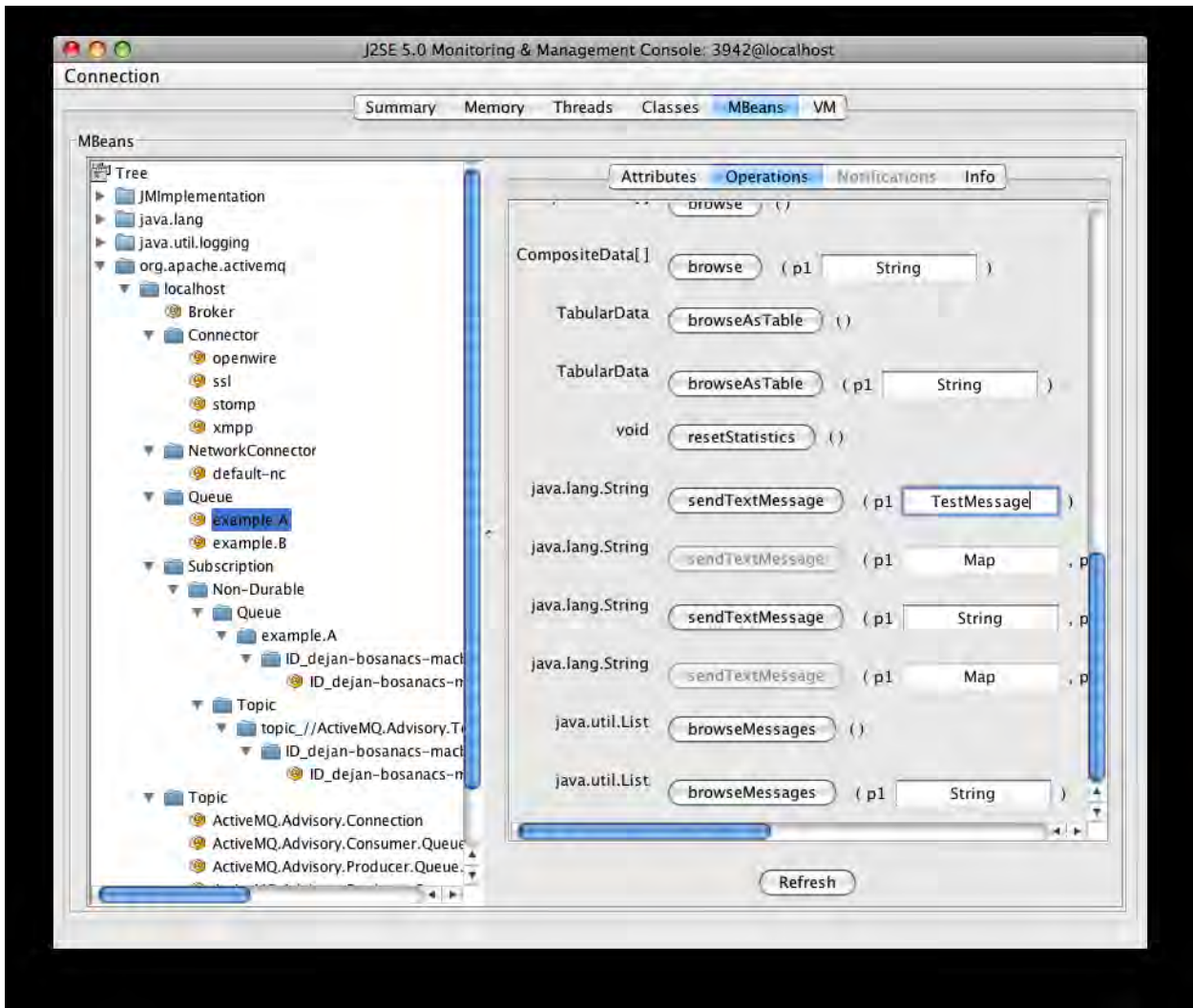
**Figure 14.4: JConsole connect**



### 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.



**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.

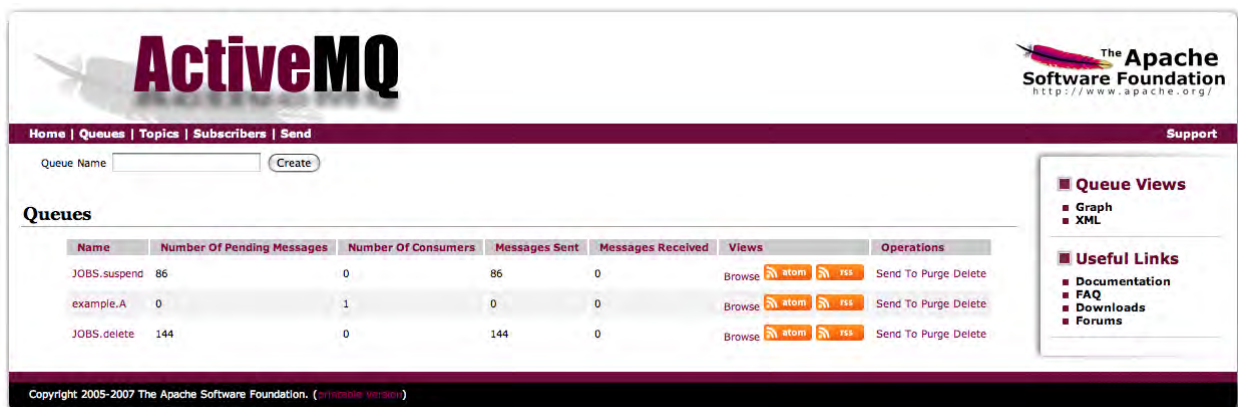
Now let's look at another similar tool that is distributed with ActiveMQ.

## 14.2.4. Web Console

In 7: “Connecting to ActiveMQ With Other Languages”, we saw how an internal

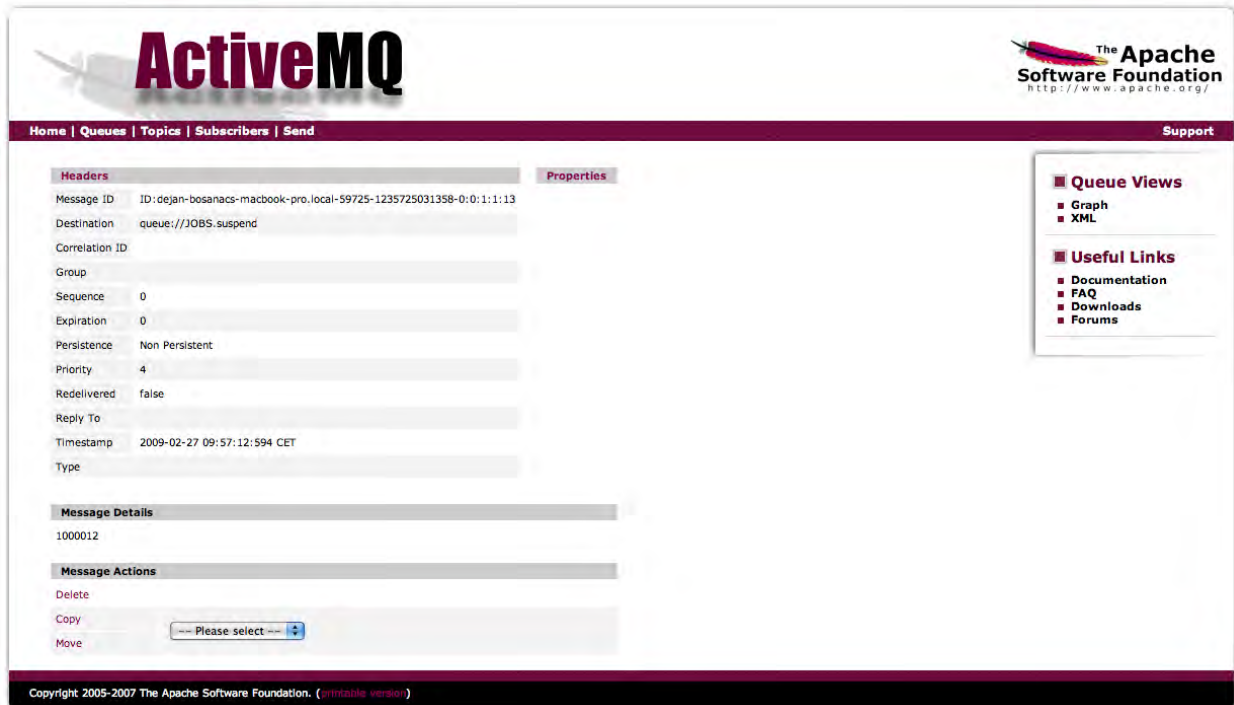
web server is used to expose ActiveMQ resources via REST and Ajax APIs. The same web server is used to host the *web console*, which provides basic management functions via a web browser. Upon starting ActiveMQ using the default configuration, you can visit <http://localhost:8161/admin/> to view the 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 console viewing a list of queues with some basic information.



**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.



**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 programmatically or using tools such as JConsole. But there is, of course, one more way you can use to peek at the broker status and that's through its internal logging mechanism. When you experience problems with broker's behavior the first and most common place to begin looking for a potential cause of the problem is the `data/activemq.log` file. In this section you'll learn how you can adjust the logging to suit your needs and how it can help you in detecting potential broker problems.

ActiveMQ uses *Apache Commons Logging* API

<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 application and want to turn on more detailed debugging, you should change the level for the root logger to `DEBUG`. Just be aware that the `DEBUG` logging level that ActiveMQ will output considerably more logging information, so you'll probably want to narrow debug messages to a particular Java package. To do this, you should leave the root logger at the `INFO` level and add a line that turns on debug logging on the desired class or even package. For example to turn tracing on the TCP transport you should add the following configuration:

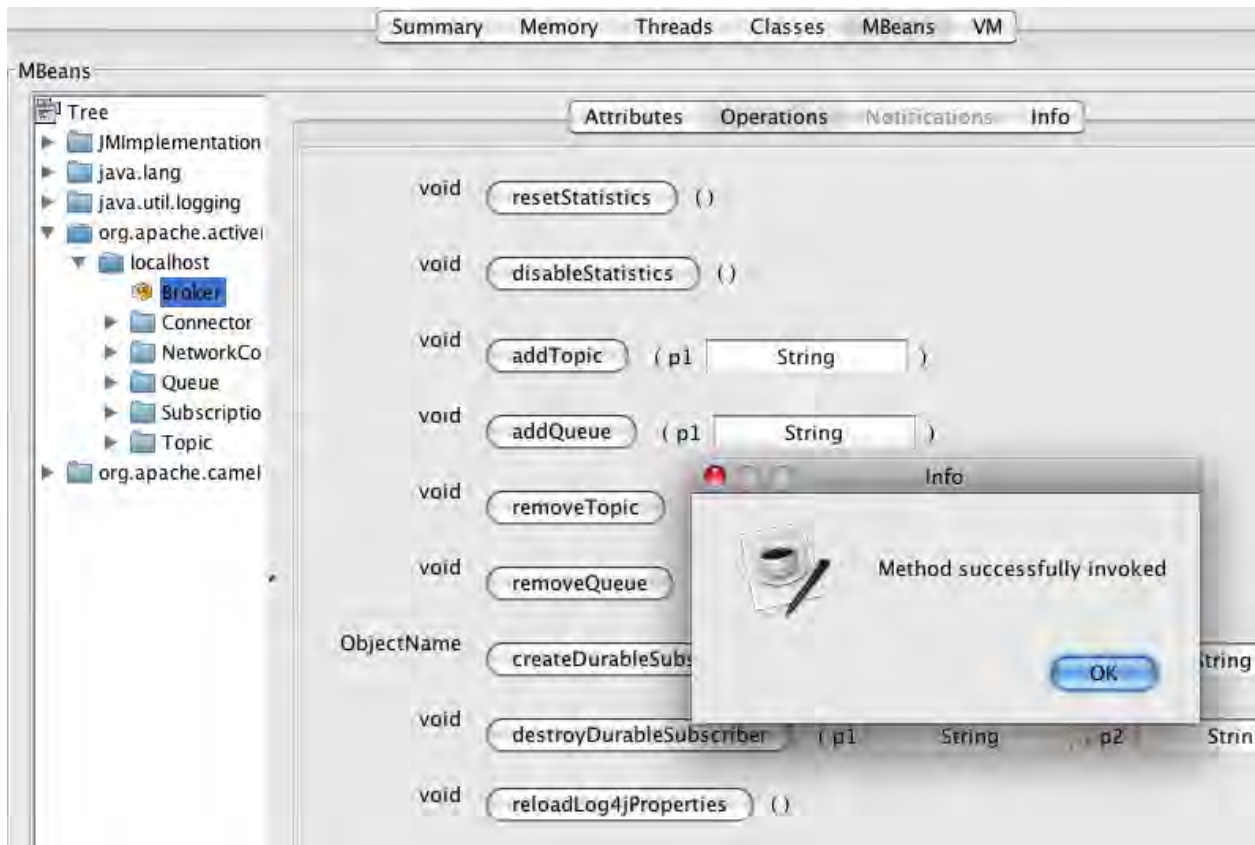
```
log4j.logger.org.apache.activemq.transport.tcp=TRACE
```

After making this change in the `conf/log4j.properties` file and restarting ActiveMQ, you will begin to see the following log output:

```
TRACE TcpTransport          - TCP consumer thread for tcp:///127.0.0.1:49383 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

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.

### 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. Additional configuration has been added (marked in bold) to monitor the failover transport and TCP communication.

Now, let's run our stock portfolio publisher example, but with some additional properties that will allow us to use logging settings previously defined.

```
$ mvn exec:java \
-Dlog4j.configuration=file:src/main/resources/org/apache/activemq/book/ch14/log4j.properties \
-Dexec.mainClass=org.apache.activemq.book.ch14.advisory.Publisher \
-Dexec.args="failover:(tcp://localhost:61616?trace=true) 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`

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.
```

```
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 [ubliher.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 [ubliher.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 [ubliher.main()] DEBUG Connection:11
- SENDING: SessionInfo { ... }
2009-03-19 15:48:07,239 [ubliher.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 [ubliher.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

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 planned 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.