SYBEX Sample Chapter

SOAP Programming with Java™

Bill Brogden

Chapter 8: SOAP Architecture Using Messages

Copyright © 2002 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

ISBN: 0-7821-2928-5

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the USA and other countries.

TRADEMARKS: Sybex has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer. Copyrights and trademarks of all products and services listed or described herein are property of their respective owners and companies. All rules and laws pertaining to said copyrights and trademarks are inferred.

This document may contain images, text, trademarks, logos, and/or other material owned by third parties. All rights reserved. Such material may not be copied, distributed, transmitted, or stored without the express, prior, written consent of the owner.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturers. The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Sybex Inc.
1151 Marina Village Parkway
Alameda, CA 94501
U.S.A.
Phone: 510-523-8233
www.sybex.com

Chapter 8

SOAP Architecture Using Messages

- Point-to-Point vs. Publish-Subscribe messaging
- The principles of Java Message Service
- Working SOAP examples with Java Message Service
- The JavaSpaces architecture
- The advantages of JavaSpaces over Java Message Service
- Getting JavaSpaces running on your system
- Working SOAP examples with JavaSpaces

It is important to stress that SOAP is quite flexible with transport mechanisms, which is what makes it such an attractive technology. Having covered the HTTP-based architecture in the previous chapters, this chapter reviews two message transport mechanisms, Java Messaging and JavaSpaces, with extensive example code for each.

Messaging Systems in General

Messaging systems came into being with the first networks, and a variety of messages are exchanged over computer networks. Naturally, one thinks first of the ubiquitous e-mail; however, many other types of valuable data are moved over networks, making messaging systems very important in enterprise-level computing, even without e-mail.

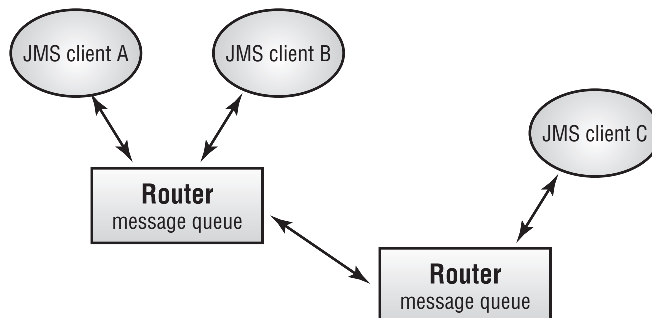
There has been a recent trend to recognize messaging systems as a separate class of application, frequently called Message-Oriented Middleware (MOM). A MOM system takes responsibility for transmitting application messages over a network and provides support for load balancing, fault tolerance, and transactions. Basically, there are two models for messaging: point-to-point and publish/subscribe.

The Point-to-Point Model

In this model, a message is transmitted from a sender to a single recipient. As shown in Figure 8.1, once the sender, client A, dispatches a message to a local router, the message can travel directly to the recipient, client B, or be relayed through additional routers to client C. Once the sender has succeeded in getting the message to the router, the sender's work is done.

The router holds the message in a message *queue*, which may or may not be backed up to permanent storage. Each queue has a unique name. A given queue can be associated with one recipient or multiple recipients, but the router ensures that each message goes only to a single recipient. You might use multiple recipients for load-balancing jobs between multiple worker processes.

FIGURE 8.1: Point-to-point message transmission



The Publish/Subscribe Model

In a point-to-point system, a message is sent to a named queue where it is held until it's retrieved by a specific receiver. In the publish/subscribe model, a message is sent by a message producer, or publisher, to a *topic*, and copies are sent to all of the topic's subscribers. This architecture is depicted in Figure 8.2 with a single router, but the message distribution system could involve many router applications.

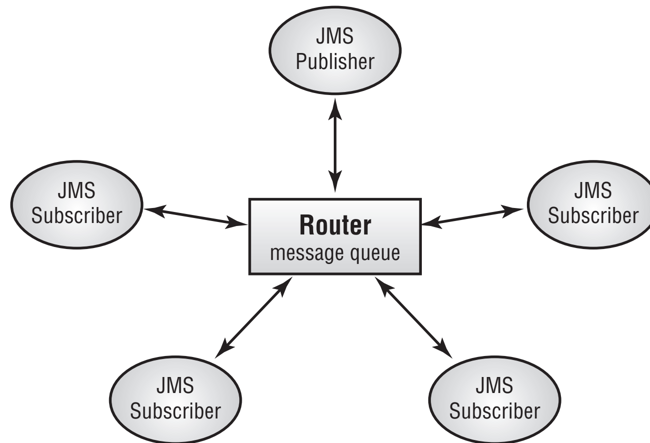


FIGURE 8.2: The publish/subscribe architecture

Advantages

Because of the services a MOM system provides, messaging offers some advantages over the client-server or remote procedure call (RPC) architectures:

Asynchronous With MOM, the sender and receiver of a message do not have to be working, be connected, and have available CPU cycles at the same time.

Fault tolerance MOM software can route around network problems, adjust to network loads, and resend messages in the event of errors.

Ignore network architecture Applications using MOM can totally ignore all of the complexities of the network architecture past the initial entry point.

Security MOM systems can enforce security precautions independently from the applications.

Disadvantages

An obvious disadvantage for MOM systems is speed. Message transmission through an intermediary server can never be as fast as direct client-server or RPC connectivity. Another possible

disadvantage is the added complexity of maintaining yet another application in the form of the message server, with the attendant storage, network traffic, and processing power requirements.

Reliability Requirements

As the number of separate components in enterprise networks grows, messaging becomes more and more attractive, despite the disadvantages. The rush to wireless connectivity and the trend toward the distribution of computing power into smaller and simpler devices emphasizes the problem of maintaining reliability. Although device reliability keeps increasing, the number of connected devices increases even faster. Therefore, it is a good probability that some part of your network will not be working at some point along the way.

SOAP Standards and Messaging

Although SOAP is basically a protocol for sending an XML-encoded message via any transport mechanism, work on standards and example implementations has not been evenly distributed over the universe of possible SOAP uses. In fact, so far the standardization effort has emphasized HTTP transport and the RPC model of request and response. The Apache SOAP implementation reflects this emphasis; it provides minimal support for various messaging architectures.

Fortunately, the suitability of Java for creating messaging services has been apparent from the very start. There are two approaches to messaging with Java APIs: Java Message Service (JMS) API and JavaSpaces. The JMS API has been mature since 1998 and has numerous commercial implementations. The more innovative JavaSpaces API has been developed in connection with Sun's Jini initiative and does not have as many commercial implementations.

Java Message Service

Creation of the JMS API was accomplished by Sun in close cooperation with industry vendors of MOM software. Although the initial intent was simply to create an API for messaging, the increasing industry interest in messaging as an alternative to Remote Method Invocation (RMI) resulted in a much more complete API than originally planned. The current reference version of JMS can be downloaded from the following site:

```
http://java.sun.com/products/jms/
```

Message Characteristics

The most important element in the JMS API is the `javax.jms.Message` interface, which defines all of the basic characteristics of `Message` objects. This interface defines a huge number of methods, but for the purposes of this chapter, the most important variables that a message has in addition to payload are delivery mode, time-to-live, and priority.

Delivery Mode

The `DeliveryMode` interface defines two integer constants:

PERSISTENT Messages with this mode must be saved to a storage medium when received by the JMS provider. After the message expiration time has elapsed, it may be discarded, which is the default value for delivery mode.

NON_PERSISTENT The JMS provider is not required to save the message; it is simply delivered to currently active clients.

Time-to-Live

When a JMS provider receives a `Message`, the time-to-live (TTL) value attached to it is added to the current system time to create an expiration date. The TTL value is specified in milliseconds and can be set by the method that transmits the message. By default, messages have an infinite TTL and never expire.

Priority

The JMS API provides for 10 levels of priority, from 0, the lowest, to 9. The default priority is 4 and is provided as a constant `DEFAULT_PRIORITY` in the `Message` interface.

Types of JMS Messages

Naturally, each message sent by JMS has a corresponding Java class. The JMS API defines a base `Message` interface and five derived interfaces. Each vendor provides a concrete implementation of the interfaces. The actual content of a JMS message is very similar to a SOAP message: there is a header for routing and identifying messages, a set of properties to be used by the receiving application, and a message body. (I'll cover only message body content here, but the message properties can also be useful.)

The `TextMessage` interface is ideal for sending an XML message, and in fact, the API designers had XML in mind specifically. Essentially, it carries a single Java `String` object. Other JMS message classes are as follows:

BytesMessage This class provides methods that let you write a message to a byte-oriented stream similar to the `DataOutputStream` class in the `java.io` package. The resulting content is a byte array.

MapMessage This class lets you write name-value pairs where the value can be a Java primitive or object.

ObjectMessage This is the form to use when sending a serialized Java object.

StreamMessage This class is similar to the `BytesMessage` class but is slightly simpler.

The SwiftMQ Implementation

The SwiftMQ implementation of JMS 1.0.2 that I use here is a creation of IIT GmbH, of Bremen, Germany. This is a pure Java implementation that requires Java SDK 1.2 or higher. The free download distribution package may be obtained from the site at:

```
http://www.swiftmq.com/
```

The rather compact download of about 3MB contains everything you need to get started with JMS, including:

- Support for both point-to-point and publish/subscribe messaging.
- A Java Naming and Directory Interface (JNDI) implementation as a JMS service. If you already have a JNDI installed, perhaps as part of a J2EE package, see the SwiftMQ documentation for instructions on using it.
- JNDI API documentation in Javadoc format.
- Two JMS implementations, one of which is a compact version for small systems such as hand-held devices.
- JMS administration programs.
- JMS API documentation in Javadoc format.
- JMS sample applications.
- Support for a variety of router networking topologies.

SwiftMQ provides add-on functionality in modules called “swiftlets.” For example, the JNDI implementation and a SMTP mailer are both swiftlet modules.

Setting Up SwiftMQ for SOAP Examples

In the following, I assume that you have already installed the SwiftMQ distribution and that you are using a Windows system. Equivalent commands are available on Unix systems.

1. To start the router, execute the `smqr1.bat` batch file from the `swiftmq\scripts\win32` directory in a new command prompt window. This starts router1, and you should see various startup messages.
2. To start the Explorer administrator, execute the `explorer.bat` file in another command prompt window.
3. Choose the Connect option from the Connection menu.
4. When the `router1` entry appears, open it and expand the Queue Manager Swiftlet entry.
5. Select the Queues item, right-click it, and choose the Create a New Entity option from the pop-up menu. Note that to get this menu, the Queues item must show as selected when you right-click it.

6. In the new Queue dialog, name the queue `eventlog` and leave the other parameters at their defaults.
7. Open the Topic Manager Swiftlet entry and expand it.
8. Select the Topics item, right-click it, and choose the Create a New Entity option from the pop-up menu.
9. In the new topic dialog, name the topic `policychange`.
10. Select the router1 item, right-click it, and choose the Save This Router Configuration option from the pop-up menu.

That accomplishes the configuration of the messaging system. You must have `router1` running for all of the following experiments with JMS.

A Point-to-Point SOAP Messaging Example

The following example application is a simple centralized event logging system. For the purposes of this exercise, I assume that there are multiple applications throughout an unspecified network that must log messages with a central facility. To keep this example as simple as possible, the SOAP message contains only a single `String`.

Sending a SOAP Message with JMS

The `SStest` class is used to generate some SOAP messages and send them to the router queue. The messages are created by combining strings with the basic XML framework of a message, plus data generated on the fly. Listing 8.1 shows the initialization data for the `SStest` class.

The `timeout` parameter for the property named `smqpURL` sets the amount of time allowed for JNDI to return an `InitialContext` object. If the named queue is not available, a `NameNotFoundException` is thrown. The `javax.naming` package provides a large number of very specific exceptions for various JNDI problems.

Listing 8.1: Static Initialization of the *SStest* Class That Generates Message Text

```
package com.lanw.clients ;
import java.util.* ;
public class SStest{
 public static Properties prop ;
 static {
   prop = new Properties() ;
   prop.put( "smqpURL", "smqp://localhost:4001/timeout=15000" );
    // the Context Provider URL
   prop.put( "qcfName", "plainsocket@router1" );
    // the QueueConnectionFactory
```

```
    prop.put( "queueName", "eventlog@router1" );
     // the Queue Name
  }
  static String[] msgStart = {
 "<?xml version='1.0' encoding='UTF-8'?>",
  "<SOAP-ENV:Envelope ",
  "xmlns:SOAP-ENV=\"http://schemas.xmlsoap.org/soap/envelope/\"",
  "xmlns:xsi=\"http://www.w3.org/1999/XMLSchema-instance\"",
  "xmlns:xsd=\"http://www.w3.org/1999/XMLSchema\">",
  "<SOAP-ENV:Body><ns1:addEntry xmlns:ns1=\"urn:EventLogger\"" ,
  "SOAP-ENV:encodingStyle=\"" +
  "http://schemas.xmlsoap.org/soap/encoding/\">" ,
  "<event xsi:type=\"xsd:string\">"
  } ;
  static String[] msgEnd = {
    "</event>",
    "</ns1:addEntry>",
    "</SOAP-ENV:Body>",
    "</SOAP-ENV:Envelope>"
  } ;
```

The main method of the `SStest` class is shown in Listing 8.2. When executed, this method creates a new `SoapSender` object initialized to send to the `eventlog` queue. Then it calls the `createConnection` method, creates and sends two example messages, and calls `closeConnection`. Note that I have provided for timing the various parts of the program.

Listing 8.2: The *main* Method of the *SStest* Class

```
  public static void main(String[] args)
  {
    try {
     String qN = (String)prop.getProperty("queueName");
     SoapSender ss = new SoapSender(qN, prop );
     long start = System.currentTimeMillis();
     ss.createConnection();
     long mark = System.currentTimeMillis();
     System.out.println("Time to create connection " +
           (mark - start));
     start = mark ;
     for(int i = 0 ; i < 2 ; i++ ){
       ss.sendMsg( createMsg( i ) );
       mark = System.currentTimeMillis();
       System.out.println("Send time: " + ( mark - start ));
       start = mark ;
     }
     ss.closeConnection();
     System.out.println("Time to close connection " +
        (mark - start));
    }catch(Exception e ){
```

```
      e.printStackTrace( System.out );
    }
  }

  static String createMsg( int n ){
    StringBuffer sb = new StringBuffer( 1000 );
    for( int i = 0 ; i <  msgStart.length ; i++ ){
      sb.append( msgStart[i] ); sb.append("\r\n");
     }
    if( (n & 0x01) == 0 ){ sb.append( "User x logged on");
    }
    else { sb.append("User x logged off");
    }
    for( int i = 0 ; i < msgEnd.length ;i++ ){
      sb.append( msgEnd[i] ); sb.append("\r\n");
    }
    return sb.toString();
  }
}
```

The SoapSender class has been designed to be a complete mechanism for sending text messages to a given queue. As shown in Listing 8.3, the constructor takes a String giving the name of the queue and a Properties object containing various initialization parameters.

The constructor locates the router and creates a connection factory using JNDI lookup facilities. In Listing 8.3, the String named initContextFacImpl names the class that SwiftMQ provides to implement the InitialContext for this implementation of JNDI; if you are using a different version of JNDI, a different class name would be required. The InitialContext and Context classes are in the javax.naming package.

Listing 8.3: Instance Variables and Constructor of the *SoapSender* Class

```
package com.lanw.clients ;

import javax.jms.*;
import javax.naming.*;
import java.util.*;

public class SoapSender
{
  static String initContextFacImpl =
      "com.swiftmq.jndi.InitialContextFactoryImpl";
  //
  Queue queue ; // the queue identity in JMS terms
  QueueConnectionFactory conFac ;
  QueueConnection connection ;
  QueueSession session ;
  QueueSender sender ;
```

```
public SoapSender(String qName, Properties prop )
        throws Exception {
  Hashtable env = new Hashtable();
  // this is javax.naming.Context
  env.put(Context.INITIAL_CONTEXT_FACTORY,initContextFacImpl);
  env.put(Context.PROVIDER_URL,prop.get("smqpURL"));
  System.out.println("try to create InitialContext");
  //
  InitialContext ctx = new InitialContext(env);
  System.out.println("Try to lookup " + prop.get("qcfName") );
  conFac = (QueueConnectionFactory)ctx.lookup(
            (String)prop.get("qcfName"));
  System.out.println("Connection Factory created");
  queue = (Queue)ctx.lookup( qName );
  System.out.println("Queue is: " + queue );
  ctx.close(); // must close because it uses a JMS connection
}
```

A SoapSender object is associated with a router address and a queue. To send one or more messages, a connection to the router and message session must be established. There are two important parameters used when creating a session: a flag that indicates whether or not the message is *transacted* and an integer giving the message acknowledgement type.

Use the transacted style for guaranteed delivery—it provides for typical transaction commitment and rollback functions. If the session does not use the transacted style, you have your choice of message acknowledgment methods, as indicated by constants in the Session class. In Listing 8.4, I am using the AUTO_ACKNOWLEDGE approach, in which the message receiver acknowledges the receipt of a message only when the entire message has been received and processed.

Creating a connection is a time consuming operation. Applications that must send a lot of messages should probably keep an open connection, or use a connection pool similar to that used for database connections with JDBC.

The example uses the setDeliveryMode method to set the delivery mode as PERSISTENT—this means that the router writes the message to disk storage as soon as it is received. Even if the router has to be restarted, the messages will not be lost. Other methods that QueueSender inherits from MessageProducer provide for setting priority and time-to-live parameters that will be applied to all messages sent by this instance of QueueSender.

Listing 8.4: The Methods That Create a Connection and Send a Message

```
public void createConnection()throws Exception {
  connection = conFac.createQueueConnection();
  // flag = transacted, int = ack mode
  session = connection.createQueueSession(false,
     Session.AUTO_ACKNOWLEDGE);
  sender = session.createSender(queue);
```

```
      // see MessageProducer - NON_PERSISTENT means lost message
      // can be tolerated
      // PERSISTENT means store to disk immediately
      sender.setDeliveryMode(DeliveryMode.PERSISTENT);
    }

    // createConnection must be called before this
    public void sendMsg(String msg) throws Exception {
      TextMessage txMsg = session.createTextMessage();
      txMsg.setText( msg );
      System.out.println("Sending "+ msg );
      sender.send( txMsg);
    }

    public void closeConnection() throws Exception {
      sender.close();
      session.close();
      connection.close();
      sender = null ; session = null ; connection = null ;
      System.out.println("\nFinished.");
    }
  }
```

For me, the greatest surprise in developing the message sender was that it takes several seconds to create a connection, whereas actual message transmission takes only a few milliseconds.

Receiving a SOAP Message with JMS

I created the `SoapJMSlistener` class for receiving and dispatching SOAP messages. The `SoapJMSlistener` class implements the `MessageListener` interface in the `javax.jms` package. A `MessageListener` has an `onMessage` method that gets called when a complete message is available for processing. This method is responsible for directing the SOAP message to the `EventLogger` class, which is the target of the message in this example.

A class implementing the `QueueConnection` interface is responsible for actually implementing a socket, listening for messages from the router, and calling the `MessageListener` using a dedicated `Thread`. The `Connection` interface is the parent of both `QueueConnection` and the `TopicConnection` interfaces that are used in the publish/subscribe examples.

The *JMStest* Program

Before describing `SoapJMSlistener`, let's take a look at the test program that sets up the example. Listing 8.5 shows this `JMStest` program. Note that there are two static variables: a `Properties` object and the `ddStr String` that contains XML in the form of a deployment descriptor for the `EventLogger` class and `addEntry` method.

Apache SOAP 2.2 does not provide as much support for message applications as it does for RPC applications. To make use of what it does provide, the `JMStest` program creates a `DeploymentDescriptor` object from the XML contained in the `ddStr String` and puts it in

the Properties object. Once the SoapJMSlistener is created, the JMStest program simply waits for a keystroke and then terminates.

Listing 8.5: The *JMStest* Program

```
package com.lanw.soapsrvr ;

import org.apache.soap.server.* ;
import java.io.* ;
import java.util.* ;

import java.util.* ;

public class JMStest {
 static Properties prop ;
 static {
  prop = new Properties() ;
  prop.put( "smqpURL", "smqp://localhost:4001/timeout=15000" );
   // Context Provider URL - 15 second timeout
  prop.put( "qcfName", "plainsocket@router1" );
   // QueueConnectionFactory
  prop.put( "queueName", "eventlog@router1" );
   // Queue Name
 }
 static String ddStr = "<isd:service xmlns:isd=" +
  "\"http://xml.apache.org/xml-soap/deployment\" " +
  "id=\"urn:EventLogger\">" +
  "<isd:provider type=\"java\" scope=\"Application\" " +
  "methods=\"addEntry\">" +
  "<isd:java class=\"com.lanw.soapsrvr.LogEntries\" " +
  "static=\"false\"/>" +
  "</isd:provider>" +
  "<isd:faultListener>org.apache.soap.server.DOMFaultListener" +
  "</isd:faultListener></isd:service>" ;
 public static void main(String[] args ){
   try {
     StringReader rdr = new StringReader( ddStr );
     DeploymentDescriptor dd = DeploymentDescriptor.fromXML( rdr );
     prop.put("dd", dd );
     System.out.println("DeploymentDescriptor is\r\n" +
                 dd.toString() );
     SoapJMSlistener sjms =
             SoapJMSlistener.JMSlistenerFactory( prop );
     System.out.println("Created new SoapJMSlistener");
     int ch = System.in.read();
     System.exit(0);
   }catch(Exception e){
     e.printStackTrace( System.out );
   }
 }
}
```

The *SoapJMSlistener* Class

Listing 8.6 shows the package and import statements for `SoapJMSlistener` and the static `String` naming the SwiftMQ class that is used to create the JNDI `InitialContext`.

Listing 8.6: The Start of the *SoapJMSlistener* Class

```
package com.lanw.soapsrvr ;

import org.w3c.dom.* ;
import javax.xml.parsers.* ;

import org.apache.soap.* ;
import org.apache.soap.util.* ;
import org.apache.soap.rpc.* ;
import org.apache.soap.server.* ;
import org.apache.soap.transport.* ;

import javax.jms.*;
import javax.naming.* ;
import java.io.* ;
import java.util.*;

public class SoapJMSlistener implements javax.jms.MessageListener {
  // the class name for SwiftMQ implementation of JMS
 static String initContextFacImpl =
    "com.swiftmq.jndi.InitialContextFactoryImpl";
```

Rather than using a public constructor for `SoapJMSlistener`, I decided to use a "factory" design pattern, as shown in Listing 8.7. This approach is not essential, but I felt it has more potential for expansion.

Listing 8.7: The *JMSlistenerFactory* Method

```
public static SoapJMSlistener JMSlistenerFactory(Properties prop)
  throws Exception {
   DeploymentDescriptor dd = (DeploymentDescriptor)prop.get("dd");
   Hashtable env = new Hashtable();
        // this is javax.naming.Context
   env.put(Context.INITIAL_CONTEXT_FACTORY,initContextFacImpl);
   env.put(Context.PROVIDER_URL,prop.get("smqpURL"));
   System.out.println("try to create InitialContext");
      //
   InitialContext ctx = new InitialContext(env);
   System.out.println("Try to lookup " + prop.get("qcfName") );
   QueueConnectionFactory qFac =
     (QueueConnectionFactory)ctx.lookup((String)prop.get("qcfName"));
   System.out.println("Connection Factory created");
      //
   Queue queue = (Queue)ctx.lookup( prop.getProperty("queueName") );
   QueueConnection qCnct = qFac.createQueueConnection();
```

```
    //  false means not transacted, AUTO_ACKNOWLEDGE means that when
    //  the message has been processed, the server is sent an
    //  Acknowledge msg so the next message will be sent
    QueueSession qSession = qCnct.createQueueSession(false,
                 Session.AUTO_ACKNOWLEDGE);
    QueueReceiver qRec = qSession.createReceiver(queue);
    SoapJMSlistener lis = new SoapJMSlistener( queue, dd );
    lis.setConnection( qCnct );
    lis.setReceiver( qRec ); //  qRec.setMessageListener(textListener);
    lis.start();
    ctx.close();
    return lis ;
  }
```

As shown in Listing 8.8, the constructor takes a JMS `Queue` object and an Apache SOAP `DeploymentDescriptor`. The `Queue` object essentially encapsulates the identity of the queue to which this `SoapJMSlistener` listens.

The constructor creates a `DocumentBuilderFactory` that will be used later to create a `DocumentBuilder` to parse the SOAP envelope in a message.

Listing 8.8: The Instance Variables and Constructor for *SoapJMSlistener*

```
   // instance variables
  QueueConnection qConnection ;
  QueueReceiver qReceiver ;
  Queue qID ;

  DocumentBuilderFactory docBF ;
  DeploymentDescriptor dd ;
  Object targetObj ;
  int errCt ;

    // the constructor
  private SoapJMSlistener( Queue q, DeploymentDescriptor d )
         throws Exception {
     qID = q ; dd = d ;
     docBF = DocumentBuilderFactory.newInstance();
     docBF.setIgnoringComments( true );
     docBF.setValidating( false ) ; // no DTD for this xml
     System.out.println("Created SoapJMSlistener for " + qID );
     String providerClass = dd.getProviderClass();
     if( dd.getScope() == DeploymentDescriptor.SCOPE_APPLICATION ){
       Class targetClass = Class.forName( providerClass );
       targetObj = targetClass.newInstance();
     }
     /* uncomment to aid debugging
     System.out.println("provider class is: " + dd.getProviderClass()
       + " scope is " + dd.getScope() + " " + targetObj );
     */
  }
```

Listing 8.9 shows various utility methods, including the methods used to attach `QueueConnection` and `QueueReceiver` objects to a `SoapJMSlistener` object. Note that it is the `setReceiver` method that attaches the object as a `MessageListener` to the `QueueReceiver` object.

The `start` method simply calls the `QueueConnection` `start` method, which starts a `Thread` to receive messages. The `finalize` method ensures that the `QueueConnection` is closed to conserve system resources.

The `onMessage` method receives a completed message from the `QueueReceiver` object. It simply extracts the `String` contents and calls the `processMsg` method. Note that in this simple example, all exceptions simply cause output to the `System.out` print stream. A real messaging application would provide a more detailed error logging facility.

Listing 8.9: Some Instance Methods in the *SoapJMSlistener* Class

```
 // instance methods
void setConnection( QueueConnection qc ){ qConnection = qc ; }
void setReceiver( QueueReceiver qr ) throws JMSException {
   qReceiver = qr ;
   qReceiver.setMessageListener( this );
}

void start() throws JMSException {
  qConnection.start();
}

public void finalize() throws Exception {
  System.out.println("SoapJMSlistener.finalize");
  if( qConnection != null ){
    qConnection.close();
  }
}

 // implementation of the MessageListener
public void onMessage(Message message) {
  TextMessage   txtMsg = (TextMessage) message;
  try {
    String msg = txtMsg.getText();
    processMsg( msg );
  } catch (JMSException je) {
     System.out.println("Messaging Exception in onMessage(): " +
         je.toString());
     errCt++ ;
  } catch (Exception e ){
     e.printStackTrace( System.out );
  }
}
```

The processMsg method shown in Listing 8.10 gets the String containing the SOAP message and uses the Apache SOAP TransportMessage class to unmarshall the String and create an org.apache.soap.Envelope object. This is the same Envelope object used in an RPC application, but further processing is completely different from the way the RpcRouterServlet uses it.

The Body in the Envelope is used to get the Element that represents the SOAP target method call. After you extract the method name, the MessageRouter class is used to determine if this is a valid method in the DeploymentDescriptor. If the call is valid, MessageRouter is used to invoke the method.

Listing 8.10: The *processMsg* Method in *SoapJMSlistener*

```
private void processMsg(String msg ) throws Exception {
  SOAPContext ctx = new SOAPContext(); // request ctx
  SOAPContext respCtx = new SOAPContext(); // in case we need it
  TransportMessage trMsg = new TransportMessage(msg,ctx,null );
  trMsg.setContentType("text/xml");
  javax.xml.parsers.DocumentBuilder xdb =
          docBF.newDocumentBuilder();
  Envelope ev = trMsg.unmarshall( xdb );
  Body body = ev.getBody();
  Vector bodyEntries = body.getBodyEntries();
  Element mainEntry = (Element)bodyEntries.elementAt( 0 );
  /* uncomment for debugging information
   String uri = mainEntry.getNamespaceURI();
   System.out.println("NamespaceURI is " + uri );
   System.out.println("Provider type: " + dd.getProviderType());
   System.out.println("message Name: " + mainEntry.getLocalName());
  */
   String msgName = mainEntry.getLocalName();
   if( MessageRouter.validMessage( dd, msgName )){
      MessageRouter.invoke( dd, ev, targetObj, msgName,
            ctx, respCtx );
   }
   else {
      throw new SOAPException( Constants.FAULT_CODE_CLIENT,
          msgName + " Not a valid message name");
   }
}
```

The MessageRouter convention requires all target methods to have the signature:

```
method( Envelope env, SOAPContext reqCtx, SOAPContext respCtx)
```

This leaves all processing of the SOAP message up to the target method but provides the SOAPContext objects to hold general request and response information. For this simple example using the LogEntries class, I am not using any of these capabilities.

The *LogEntries* Class

For this example, the `LogEntries` class simply keeps a specified number of the most recent log messages in a `Vector`. Listing 8.11 shows the start of the code, including instance variables, the constructor, and the `toString` method.

Listing 8.11: The Start of the *LogEntries* Class Source Code

```
package com.lanw.soapsrvr ;

import java.util.* ;
import org.w3c.dom.* ;
import org.apache.soap.* ;
import org.apache.soap.rpc.SOAPContext ;

public class LogEntries{

  static int maxEntries = 20 ;
  Vector recent ;
  int count ;
  String lineSep = System.getProperty("line.separator");

  LogEntries(){
    recent = new Vector();
  }
  public String toString(){
    StringBuffer sb = new StringBuffer("Recent entries");
    sb.append( lineSep );
    synchronized( recent ){
      Enumeration e = recent.elements();
      while( e.hasMoreElements() ){
        sb.append( e.nextElement().toString() ) ;
        sb.append( lineSep );
      }
    } // end synchronized
    return sb.toString() ;
  }
```

The `addEntry` method shown in Listing 8.12 gets the SOAP message in the form of an `Envelope` object. It simply extracts the text of the `event` tag and saves the resulting String. Note that if the tag is not found or is empty, a `SOAPException` is thrown, which indicates a client fault.

Listing 8.12: The *addEntry* Method

```
  // this method signature is required for
  //  Apache SOAP message service dispatch
  public void addEntry( Envelope env, SOAPContext reqCtx,
     SOAPContext respCtx)  throws SOAPException {
```

```
      Body body = env.getBody();
      Vector bodyEntries = body.getBodyEntries();
      Element root = (Element) bodyEntries.elementAt(0);
      NodeList nlist = root.getElementsByTagName( "event" );
      if( nlist.getLength() == 0 ) {
          throw new SOAPException(Constants.FAULT_CODE_CLIENT,
               "No event supplied");
      }
      Element e = (Element)nlist.item(0);
      Node node = e.getFirstChild();
      if( node == null ){ // only possibility is an empty tag
        throw new SOAPException(Constants.FAULT_CODE_CLIENT,
            "Incorrect event format supplied");
      }
      String s = node.getNodeValue().trim();
      System.out.println("Got event: " + s );
      while( recent.size() > maxEntries ){
          recent.removeElementAt(0);
      }
      recent.addElement( s );
      count++ ;
    }
  }
```

Publish/Subscribe Messaging

For the publish/subscribe example, I assumed that numerous applications throughout an enterprise must be aware of certain business policies. New policies and changed policies are published by a central server and picked up by subscribing applications. The topic to be published is named `policychange`; setting up this topic with the message server was discussed previously in the section "Setting Up SwiftMQ for SOAP Examples."

The SOAP Publisher Example

Listings 8.13 and 8.14 show the `PStest` program used to demonstrate an example of a simple SOAP message publisher. As with the point-to-point example, a static `Properties` object is used to hold the initialization data, and a static `String` array provides the basic SOAP skeleton.

Listing 8.13: Start of the Source Code for an Example Message Publisher

```
package com.lanw.clients ;
import java.util.* ;
public class PStest{

 public static Properties prop ;

 static {
  prop = new Properties() ;
```

```
 prop.put( "smqpURL", "smqp://localhost:4001/timeout=15000" );
  // Context Provider URL
 prop.put( "tcfName", "plainsocket@router1" );
  // TopicConnectionFactory name
 prop.put( "topicName", "policychange" );
  // topic Name is "policychange@router1"
}
static String[] msgStart = {
  "<?xml version='1.0' encoding='UTF-8'?>",
"<SOAP-ENV:Envelope xmlns:SOAP-
  ENV=\"http://schemas.xmlsoap.org/soap/envelope/\"",
"xmlns:xsi=\"http://www.w3.org/1999/XMLSchema-instance\"",
"xmlns:xsd=\"http://www.w3.org/1999/XMLSchema\">",
"<SOAP-ENV:Body><ns1:setPolicy xmlns:ns1=\"urn:EventLogger\"" ,
"SOAP-ENV:encodingStyle=\"" +
       " http://schemas.xmlsoap.org/soap/encoding/\">",
"<policy xsi:type=\"xsd:string\">"
} ;
static String[] msgEnd = {
  "</event>",
  "</ns1:addEntry>",
  "</SOAP-ENV:Body>",
  "</SOAP-ENV:Envelope>"
} ;
```

The test program `main` method, shown in Listing 8.14, creates a `PublishSender` object and uses it to send three messages to the topic. As with the point-to-point example, the SOAP message is created by combining static text that defines the SOAP `Envelope` with a single `String` variable. I have included some timing statements to get an idea of which process is most time consuming.

Listing 8.14: The *main* Method of the Publishing Test Program

```
public static void main(String[] args)
{
  try {
   String tN = (String)prop.getProperty("topicName");
   long start = System.currentTimeMillis();
   PublishSender ps = new PublishSender(tN, prop );
   ps.createSession();
   long mark = System.currentTimeMillis();
   System.out.println("Time to create connection " + (mark - start));
   start = mark ;
   for(int i = 0 ; i < 3 ; i++ ){
     ps.sendMsg( createMsg( i ) );
     mark = System.currentTimeMillis();
     System.out.println("Send time: " + ( mark - start ));
     start = mark ;
   }
```

```
      ps.closeConnection();
      System.out.println("Time to close connection " + (mark - start));
    }catch(Exception e ){
      e.printStackTrace( System.out );
    }
  }

  static String createMsg( int n ){
    StringBuffer sb = new StringBuffer( 1000 );
    for( int i = 0 ; i <  msgStart.length ; i++ ){
      sb.append( msgStart[i] ); sb.append("\r\n");
    }
    sb.append("Some Policy Change Data " + n );
    for( int i = 0 ; i < msgEnd.length ;i++ ){
      sb.append( msgEnd[i] ); sb.append("\r\n");
    }
    return sb.toString();
  }
}
```

The `PublishSender` class is a general purpose class for publication to a particular topic established in the constructor. As shown in Listing 8.15, the constructor creates an `InitialContext` that is used to get a `TopicConnectionFactory` based on the parameter named `tcfName`. That factory, in turn, is used to get a `TopicConnection`. In this case, I used the version of `createTopicConnection` that does not specify user name and password, so the default user identity is assumed.

The `InitialContext` is also used to get an object implementing the `Topic` interface using the name passed to the constructor. A `Topic` object represents the identity of the topic, just as a `Queue` object represents the identity of a point-to-point message queue.

Listing 8.15: The Start of the *PublishSender* Class Source Code

```
package com.lanw.clients ;

import javax.jms.*;
import javax.naming.*;
import java.util.*;

public class PublishSender
{
  static String initContextFacImpl =
    "com.swiftmq.jndi.InitialContextFactoryImpl";

  //
  String topName ;
  Topic topic ; // the topic identity in JMS terms
  TopicConnectionFactory topFac ;
```

```
    TopicConnection connection ;
    TopicSession session ;
    TopicPublisher publisher ;

    public PublishSender(String name, Properties prop )
        throws Exception {
     topName = name ;
     Hashtable env = new Hashtable();
      // the constants are in the javax.naming.Context class
     env.put(Context.INITIAL_CONTEXT_FACTORY,initContextFacImpl);
     env.put(Context.PROVIDER_URL,prop.get("smqpURL"));
     System.out.println("try to create InitialContext");
      //
     InitialContext ctx = new InitialContext(env);
     System.out.println("Try to lookup " + prop.get("tcfName") );
     topFac = (TopicConnectionFactory)ctx.lookup(
            (String)prop.get("tcfName"));
     System.out.println("Create TopicConnection from factory "
            + topFac );
     connection = topFac.createTopicConnection();
      // note that an alternative uses username and password
     System.out.println("Connection created " + connection +
       " try to create topic " + topName );
     topic = (Topic) ctx.lookup( topName );
     System.out.println("Topic created " + topic );
     ctx.close(); // must close because it uses a JMS connection
   }
```

Now that we have a `TopicConnection` and a `Topic`, the next step is to get a `TopicSession` and use that to create a `TopicPublisher`. This operation is shown in the `createSession` method in Listing 8.16. It is the `TopicPublisher` that is actually used for message transmission, as shown in the `sendMsg` method. A new `TopicPublisher` starts a new sequence of messages. In point-to-point messaging, the equivalent to `TopicPublisher` is `QueueSender`.

The length of time that messages are retained by the JMS router is determined by the publisher of the message. The `TopicPublisher` interface extends the `MessageProducer` interface, where you find the method `setTimeToLive`. The `sendMsg` method shown uses the publish method that provides a default delivery mode, time-to-live, and priority. The default value for time-to-live is 0, implying that the message never expires.

Listing 8.16: Methods for Managing a *PublishSender* Object

```
  public void createSession()throws Exception {
      // flag = transacted, int = ack mode
    session = connection.createTopicSession(false,
              Session.AUTO_ACKNOWLEDGE);
    publisher = session.createPublisher( topic );
    System.out.println("publisher time to live: " +
```

```
      publisher.getTimeToLive() );

  }

    // createConnection must be called first
    public void sendMsg(String msg) throws Exception {
      // Send the messages to the queue
      TextMessage txMsg = session.createTextMessage();
      txMsg.setText( msg );
      System.out.println("Publishing "+ msg );
      publisher.publish( txMsg);
    }

    public void closeConnection() throws Exception {
      publisher.close();
      session.close();
      connection.close();
      publisher = null ; session = null ; connection = null ;
      System.out.println("\nFinished.");
    }
  }
```

Just to keep things confusing, when you execute the publication test, you will see a message like the following, which mentions a `keepaliveInterval` parameter. This is not the message time-to-live, but a timeout for the `TopicConnection` socket.

```
Create TopicConnection from factory [ConnectionFactoryImpl,
  socketFactoryClass=com.swiftmq.net.PlainSocketFactory,
  hostname=172.16.1.3, port=4001, keepaliveInterval=60000]
```

As with the point-to-point example, creating the initial connection is by far the most time consuming operation. In one test, creating the connection took 5.5 seconds, whereas sending individual messages took from 60 to 110 milliseconds.

The SOAP Subscriber Example

Listing 8.17 shows the test program on the subscriber side. It simply creates three `SoapJMSsubscriber` objects, each with a unique `clientID` name.

Listing 8.17: The *SubTest* Subscriber Test Program

```
package com.lanw.soapsrvr ;
import java.util.*;

public class SubTest {

  static Properties prop ;
  static {
   prop = new Properties() ;
```

```
    prop.put( "smqpURL", "smqp://localhost:4001/timeout=15000" );
      // Context Provider URL
    prop.put( "tcfName", "plainsocket@router1" );
     // TopicConnectionFactory
    prop.put( "topicName", "policychange" );
   }

  public static void main(String[] args)
  {
    try {
     String tName = (String)prop.getProperty("topicName");
     String id = "unique_id1" ; // no spaces allowed
     prop.put("clientID", id );
     SoapJMSsubscriber sub1 = new SoapJMSsubscriber(
        tName, prop );
     id = "unique_id2" ; // no spaces allowed
     prop.put("clientID", id );
     SoapJMSsubscriber sub2 = new SoapJMSsubscriber(
        tName, prop );
     id = "unique_id3" ; // no spaces allowed
     prop.put("clientID", id );
     SoapJMSsubscriber sub3 = new SoapJMSsubscriber(
        tName, prop );

    }catch(Exception e){
      e.printStackTrace( System.out );
    }
  }
 }
```

The `SoapJMSsubscriber` code is shown in Listing 8.18. Note that in this example, the object implementing the `TopicSubscriber` interface is created as a "durable" topic subscriber. This implies that the topic messages will be received even if the subscriber is not active when the message is published.

Note that this does not imply that the subscriber will get messages published to this topic before the `TopicConnection` is created. After the `TopicConnection` has been created, it may be stopped and started repeatedly to control the load on the subscriber. As long as the `TopicConnection` exists, all durable messages sent while the `TopicConnection` is stopped will be received when it is started again. This behavior is in contrast to the JavaSpaces messaging architecture.

Listing 8.18: The *SoapJMSsubscriber* Source Code

```
package com.lanw.soapsrvr ;

import javax.jms.*;
import javax.naming.*;
```

```
import java.util.*;

public class SoapJMSsubscriber implements MessageListener {

  static String initContextFacImpl =
    "com.swiftmq.jndi.InitialContextFactoryImpl";

   // instance variables
     String topicName ;
     String clientID ;
  Topic topic ; // the topic identity in JMS terms

  TopicSession tSession ;
  TopicPublisher publisher ;

  public SoapJMSsubscriber (String tName, Properties prop )
          throws Exception {
    Hashtable env = new Hashtable();
        // this is javax.naming.Context
    env.put(Context.INITIAL_CONTEXT_FACTORY,initContextFacImpl);
    env.put(Context.PROVIDER_URL,prop.get("smqpURL"));
    System.out.println("try to create InitialContext");
    InitialContext ctx = new InitialContext(env);
    System.out.println("Try to lookup " + prop.get("tcfName") );
    TopicConnectionFactory topFac = (TopicConnectionFactory)
           ctx.lookup((String)prop.get("tcfName"));
    System.out.println("TopicConnection Factory created " +
      "try to create Topic from " + tName );
    topic = (Topic) ctx.lookup( tName );
    System.out.println("Topic created: " + topic );
    ctx.close(); // must close because it uses a JMS connection
    TopicConnection tConnect = topFac.createTopicConnection();
      // we now have a open TopicConnection to  the router
    tSession = tConnect.createTopicSession( false,
           Session.AUTO_ACKNOWLEDGE );
     /* A durable subscriber will get all messages, even those that
      were published while the subscriber was down. To accomplish
      this, the subsName must be unique
     */
    clientID = prop.getProperty("clientID");
    TopicSubscriber tSubs = tSession.createDurableSubscriber(
       topic, subsName  );
    topicName = topic.toString() ;
    System.out.println("Topic is: " + topic );
    tSubs.setMessageListener( this );
    tConnect.start();
  }

  public void onMessage( Message msg ){
    TextMessage tMsg = (TextMessage) msg ;
    try {
```

```
        String s = tMsg.getText();
        System.out.println("client " + clientID + " got Msg, size " + s.length() );
      }catch( JMSException je ){
      }
    }
  }
```

A `TopicConnection` is always created in an inactive or stopped mode; it cannot send or receive messages until specifically started. A subscribing application can stop and restart the `TopicConnection` if necessary.

Timing Results

As with the point-to-point example, the main time consuming operation is the creation of the initial connection to the messaging server. In one test run, creating the connection took about 5.5 seconds, whereas sending a single message took between 100 and 150 milliseconds.

JavaSpaces

The JavaSpaces API is based on an academic research project called Linda at Yale University that got started in about 1982. Researchers from that project have been instrumental in the JavaSpaces project at Sun, which has been closely associated with Sun's Jini distributed computing initiative. Another major "spaces" initiative, also in Java, is IBM's TSpaces project. For more information about TSpaces, see:

```
http://www.almaden.ibm.com/cs/TSpaces/
```

In theory, the basic data structure in a space is called a *tuple*. This is essentially a collection of fields, where each field contains a typed value. Values can be primitive types or complex types such as an array, Java objects, or Java classes. Everything in a space is a tuple. In a JavaSpace, a tuple is represented as a Java object with a collection of public values.

An essential feature of space architecture is the use of associative addressing. This means that entries are located according to their content, rather than any sort of index or a single valued locator such as a topic in JMS. Thus, rather than getting messages by subscribing to a simple topic such as `policychange`, as in my earlier example, an application based on a spaces architecture might request entries by matching multiple values like this:

```
type = policychange
department = marketing
product-line = software
```

The intent of JavaSpaces is to provide a simple unified mechanism for distributed computing. Computing entities can exchange messages, share data, and distribute objects by means of a

shared space in a very loosely coupled architecture. This shared space is managed by a JavaSpaces server. For Sun's JavaSpaces Technology Kit and API documents, visit the following site:

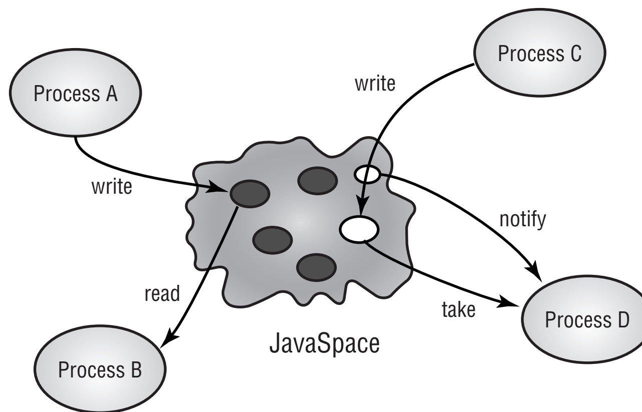
```
http://java.sun.com/products/javaspaces/
```

Operations in a JavaSpaces Space

Figure 8.3 summarizes the operations that can be conducted on a space. It is traditional to represent a space in diagrams like this as a cloud to emphasize the absence of any ordering imposed by the space.

FIGURE 8.3:

A JavaSpace diagram showing processes and operations



The basic operations in a space managed by JavaSpaces are extremely simple:

write A process writes a Java object into the space.

read A process gets a copy of a selected object in the space. Selection is accomplished by providing a template object to the space server.

take A process removes a selected object.

notify A process gets an event when an object matching a particular template is written into the space.

A single JavaSpaces server can manage a number of separate spaces, each identified by a unique name.

Selection by Template

The revolutionary concept embodied in spaces is selection of entries by means of a template. In JavaSpaces, the template for retrieval of objects of type A is simply another object of type A with instance variables set to values that control retrieval. The rules are very simple:

- Only entries with the same type as the template are eligible for selection.

- If the template has a variable set to `null`, any value for that variable will be accepted.
- All template variables that have a reference will be checked versus the matching candidate entry variable using the `equals` method. If all match, the entry will be selected.

It is important to note that selection by template does not provide for any sort of selection by a range of values the way a JDBC-compliant database does.

Persistent versus Nonpersistent Storage

When choosing a messaging system for SOAP messages, you must balance your requirements for persistent and safe storage of messages versus your requirements for speed and simplicity. Both JMS and JavaSpaces servers can be set up for simple and fast nonpersistent storage of messages or for persistent storage.

The JavaSpaces approach offers greater control over long-term storage. One advantage is that because the retrieval mechanism does not depend on preassigned topics, new applications can be written to access entries long after the program that wrote them is no longer connected to the system.

What's Different About Spaces?

It is reasonable to ask "what's the big deal" about JavaSpaces—at first glance, it looks like a slight extension from JMS messaging. Here are the differences:

Truly asynchronous When an entry is written, a process does not have to be a subscriber to read it.

Controlled persistence The lease concept used by JavaSpaces is considerably more flexible than the simple time-to-live used by JMS. For example, a lease can be renewed or cancelled.

Associative lookup The flexibility of template matching goes far beyond the simple topic selection of JMS. Essentially, a space functions like a simple object database.

Object storage JavaSpaces applications are not limited to the simple variable types of JMS. Because a complete Java object is stored and retrieved, applications can execute operations directly on retrieved objects.

Event notification Processes may be notified when an entry matching a particular template has been written.

Applicability to SOAP Messages

It seems to me that JavaSpaces technology is a very good match for SOAP messaging. For example, each of the message processing functions that are indicated in SOAP headers could

be carried out by Java processes watching a space. Given the following header on a message to be processed:

```
<SOAP-ENV:Header>
 <a:Authenticate xmlns:a="AuthenticateURI"
   mustUnderstand="true">username,password
 </a:Authenticate>
</SOAP-ENV:Header>
```

The entry object placed in the space could have the following values:

needsAuthentication A `Boolean` with the value `true`.

authenticated A `Boolean` with the value `null`.

soapMsg A `String` containing the full SOAP message.

The authentication process would simply read entries where `needsAuthentication` has the value of `true`, process the entry, and rewrite it with changed values.

Some SOAP applications naturally seem to match JavaSpaces capabilities; for example, distributed processing of complex calculations. Suppose your SOAP service processes credit applications received over an enterprise-wide network, and the load is more than any single machine can handle. You can solve this with a pool of systems on a network, all connected to a JavaSpaces server.

The initial application in the form of a SOAP message could be placed in a space with `Boolean` variables indicating the type of processing needed set to `true`. Any available machine can take the entry and perform the needed calculations. This automatically balances the load and makes it easy to add new processing power without taking the system down.

As a further improvement, processing could be broken down into smaller steps, each handled by a specialized application. For example, one application might specialize in lookups in a particular database.

Getting Started with JavaSpaces

It takes a lot of work to get a JavaSpaces system running at the present state of development, but when that initial obstacle is overcome, writing programs to use the system is quite simple. The first step is to locate the needed software packages.

The first version of JavaSpaces was available as a separate download, but now the packages are closely associated with Sun's larger Jini technology for distributed computing. You can get Jini as part of a subscription to Sun's Developer Essentials CD subscription program or download the Jini package from:

```
http://www.sun.com/jini/
```

Here is a summary of the steps to go through to get JavaSpaces running. All will be discussed in detail, but this list will give you an idea of where you are going:

1. Create a policy file. Java provides for very fine-grained control over what applications are going to do. However, for the purposes of development, the simplest thing to do is to allow everything. You should be able to find a file named `policy.all` in the Jini distribution.
2. Decide where you want to keep the log files. Several of the utilities that must be installed keep extensive logs and temporary files.
3. Start a HTTP server that has specific jar files in the root directory. JavaSpaces depends on a web server to provide class files over the network.
4. Start an RMI activation daemon, which is a utility that activates and manages the other needed services.
5. Start the Jini lookup service utility.
6. If your JavaSpaces applications use transactions, start a Transaction Manager.
7. Start the JavaSpaces service.
8. To verify that everything is working, run the space browser utility.

The commands to start these services are generally quite complicated. You should set up batch or shell command files for each of them. The examples in this chapter contain paths related to my specific system configuration; naturally, you must substitute your own values.

The *Policy* File

The `policy.all` file contains the following:

```
grant {
    permission java.security.AllPermission "", "";
};
```

This essentially lets any class do anything, so it is not something you would want to leave on a production system. You will be passing a path to this file to the RMI Activation Daemon so it can be placed anywhere on your system. Further discussion of Java security and permission granting is beyond the scope of this book.

The Log Files

Paying attention to the following note is essential to a frustration free experience with JavaSpaces technology.

NOTE The utilities to be started require that the log directory you specify does not exist when the utility starts. Starting with a new directory presumably ensures that there are no files left over from previous runs to confuse things. This is a bit of a pain when frequently stopping and starting the utilities during development. Plan to delete the previous log directory before starting.

The HTTP Server

The server is used to provide Java class files to the many distributed applications in a Jini or JavaSpaces environment. You must provide a URL that points to specific jar files for the various utilities. The Jini installation `lib` directory contains small jar files containing only the class files used by typical JavaSpaces applications. For example, there are two jar files named `reggie` that are used by the Jini lookup service. The `reggie.jar` file contains the full set of classes, whereas the `reggie-dl.jar` file contains a minimum set and is less than a quarter of the size of the full set. Many classes appear in more than one jar file because the format of the command line used to start the utilities only allows one URL. Here are the files from the `lib` directory that you will need to copy to the HTTP server root directory:

outrigger-dl.jar JavaSpaces classes

mahalo-dl.jar Transaction Manager classes

reggie-dl.jar Lookup service support

space-examples-dl.jar JavaSpaces utilities classes used for sample programs

The RMI Activation Daemon

The Remote Method Invocation (RMI) service must be running before any of the other utilities on all computers accessing a JavaSpaces space. It provides for automatically creating remote objects when requested by a client and providing a reference to that object for use in RMI calls. It is also responsible for enforcing a security policy.

The `rmid` program is part of the standard Java SDK. Here is the command to start the service using the default port of 1098 and specifying a policy file and a log directory:

```
rmid -J-Djava.security.policy=C:\tools\policy.all
   -log C:\temp\Logs\logRMID
```

Note that although the `rmid` instructions for earlier versions of Java do not call for setting a policy in the command line, it is necessary from Java 1.2 on.

The Jini Lookup Service

In version 1.0 of JavaSpaces, you had the option of using either the `rmiregistry` program or Jini for lookup services. The current version requires the use of Jini lookup service, known familiarly as “reggie.” Reggie runs from an executable jar file. When the command line shown is executed, a startup Java program parses the command-line parameters, registers the lookup service with the RMI Activation Daemon (RMID), and then exits.

At this point, the service can be activated but not actually be running. The RMID handles creating a JVM to run the service when it is first requested. Because of this approach, when you issue the following command in an MS-DOS command prompt window, there will be a pause and some disk activity, and then the command will return. This is not the result of some

anonymous unreported error; it is normal behavior. In the event of an error, you may get messages in this command window or in the command window in which RMID is running.

The following shows the command split on separate lines for easier reading; the actual command is all on one line.

```
java -jar
   G:\jini1_1\lib\reggie.jar
   http://localhost:8080/reggie-dl.jar
   C:\tools\jmspolicy.txt
   C:\temp\Logs\logREGGIE
   public
```

Following the complete path to the `reggie.jar` file is the URL that gives the HTTP server location for the client class files. Next is the location of the policy file that controls what the lookup service is allowed to do. Following that is the directory for the service to use for logs and temporary files.

If you get the highly annoying error message that the directory already exists, you must kill the RMID program, erase any log directories, and try again.

The Transaction Manager

If you are using transactions in a JavaSpaces server, the next utility to start is the Jini Transaction Manager server. Here is the command line that registers the service with RMID so that it can be activated and started when needed. As with the `reggie` startup sequence, the command returns when registration is finished.

```
java -jar
  -Djava.security.policy=C:\tools\jmspolicy.txt
  -Dcom.sun.jini.mahalo.managerName=TransactionManager
  G:\jini1_1\lib\mahalo.jar
  http://localhost:8080/mahalo-dl.jar
  C:\tools\jmspolicy.txt C:\temp\Logs\logTM
```

The JavaSpaces Server

Now that the HTTP server, RMID, lookup, and Transaction Manager services are running, it's time to start the JavaSpaces server. Here is an example command to start a persistent JavaSpaces server:

```
java -jar
  -Djava.security.policy=C:\tools\jmspolicy.txt
  -Dcom.sun.jini.outrigger.spaceName=JavaSpaces
  G:\jini1_1\lib\outrigger.jar
  http://localhost:8080/outrigger-dl.jar
  C:\tools\jmspolicy.txt
  C:\temp\Logs\logJS
  public
```

The `java.security.policy` line sets the permissions for the temporary program that registers the service. The second mention of the policy file sets the permissions for the service itself.

The line that sets `spaceName=JavaSpaces` defines the name of the space with which to work. If you needed additional named spaces, they would be defined here as a list of comma-separated names.

The `outrigger.jar` file contains the classes to create a persistent space; you would use `transient-outrigger.jar` to create a transient space. The difference is that entries in a transient space do not survive shutting down and restarting the JavaSpaces service. The URL gives a network location of a HTTP server that can supply class jar files to clients.

The final parameter, `public`, specifies the name of the Jini group of which this service will be a part.

Example SOAP Message in a Space

Now that you have a JavaSpaces server running, the hard part is over. This section shows a minimum set of functions to write and read space entries containing a SOAP message.

The *JSpaceSoapMsg* Class

First, let's look at the object to be managed in a space. All that is required for a custom object is that it implement the `net.jini.core.entry.Entry` interface. This is simply a marker interface that contains no methods.

The `Entry` class implements the `java.io.Serializable` interface, which is also just a marker interface that indicates objects of this class can be serialized for transmission between Java applications. All members of the class must also be serializable, or an exception will be thrown the first time you try to write the object into the space.

Listing 8.19 shows the `JSpaceSoapMsg` class to be used in the demonstration programs. The intent here is that the SOAP message will be stored as one `String` and a topic name will be stored as another. A real application could use a much more complex object with more public variables and methods.

Listing 8.19: The *JSpaceSoapMsg* Class

```
package com.lanw.clients ;

import net.jini.core.entry.Entry; // note, Entry extends Serializable

public class JSpaceSoapMsg implements Entry {
  public String soapmsg ;
  public String topic ;
```

```
  // no args constructor is required
  public JSpaceSoapMsg() {
  }

  public JSpaceSoapMsg( String msg, String tp ){
    soapmsg = msg ; topic = tp ;
  }
}
```

Writing into a Space

The program to write a single entry into a space is shown in Listing 8.20. The call to `SpaceUtil.getSpace` gets a `JavaSpace` reference for the default space named `JavaSpace`.

Listing 8.20: The *JSpaceTest* Program That Writes an Entry

```
package com.lanw.clients ;

import java.io.* ;
import java.util.* ;
import net.jini.space.JavaSpace;

public class JSpaceTest {
public static void main(String[] args ){
   try {
     long start = System.currentTimeMillis();
     JavaSpace jsp = SpaceUtil.getSpace();
     long end = System.currentTimeMillis();
     System.out.println("Got Jsp= " + jsp + " in " +
         (end - start) + " milliseconds" );
     start = end ;
     JSpaceSoapMsg msg = new JSpaceSoapMsg( createMsg(0),
         "policychange" );
     jsp.write( msg, null, 60000 );
     end = System.currentTimeMillis();
     System.out.println("Msg written to space in " +
         (end - start) + " milliseconds");
     int ch = System.in.read();
     System.exit(0);
   }catch(Exception e){
     e.printStackTrace( System.out );
   }
 }

  static String[] msgStart = {
   "<?xml version='1.0' encoding='UTF-8'?>",
 "<SOAP-ENV:Envelope ",
 "xmlns:SOAP-ENV=\"http://schemas.xmlsoap.org/soap/envelope/\"",
 "xmlns:xsi=\"http://www.w3.org/1999/XMLSchema-instance\"",
 "xmlns:xsd=\"http://www.w3.org/1999/XMLSchema\">",
```

```
    "<SOAP-ENV:Body><ns1:addEntry xmlns:ns1=\"urn:EventLogger\"" ,
    "SOAP-ENV:encodingStyle=\"" +
    "http://schemas.xmlsoap.org/soap/encoding/\">" ,
    "<policychange xsi:type=\"xsd:string\">"
    } ;
    static String[] msgEnd = {
      "</policychange>",
      "</ns1:addEntry>",
      "</SOAP-ENV:Body>",
      "</SOAP-ENV:Envelope>"
    } ;

    static String createMsg( int n ){
      StringBuffer sb = new StringBuffer( 1000 );
      for( int i = 0 ; i <  msgStart.length ; i++ ){
        sb.append( msgStart[i] ); sb.append("\r\n");
       }
      sb.append("disallow credit userID=" + n );
      for( int i = 0 ; i < msgEnd.length ;i++ ){
        sb.append( msgEnd[i] ); sb.append("\r\n");
      }
      return sb.toString();
    }
  }
```

Utility methods to acquire a JavaSpace reference are shown in Listing 8.21. The `getSpace` method depends on the `Locator` and `Finder` classes. These classes were created as utilities for the book *JavaSpaces Principles, Patterns and Practice* by Eric Freeman, Susanne Hupfer, and Ken Arnold (Addison-Wesley, 1999). They have been deprecated in the Jini 1.1 release but are still much more convenient than the alternatives and are included in the release, so I used them in this example. By the time this is published, there may be more convenient methods to obtain a JavaSpace reference.

Listing 8.21: The *SpaceUtil* Class

```
package com.lanw.clients ;

import java.rmi.*;
import net.jini.space.JavaSpace;
import net.jini.core.discovery.LookupLocator ;

import com.sun.jini.mahout.binder.RefHolder;
import com.sun.jini.mahout.Locator;
import com.sun.jini.outrigger.Finder;

public class SpaceUtil {

  public static JavaSpace getSpace(String name) {
```

```
    try {
     if (System.getProperty("com.sun.jini.use.registry") == null)
     {
       System.err.println("Use DiscoveryLocator to locate " + name);
       Locator locator =
             new com.sun.jini.outrigger.DiscoveryLocator();
       Finder finder =
             new com.sun.jini.outrigger.LookupFinder();
       return (JavaSpace)finder.find(locator, name);
     } else {
       RefHolder rh = (RefHolder)Naming.lookup(name);
       return (JavaSpace)rh.proxy();
     }
    } catch (Exception e) {
        System.err.println(e.getMessage());
        e.printStackTrace( System.err );
    }
         return null;
     }

     public static JavaSpace getSpace() {
         return getSpace("JavaSpaces");
     }
}
```

Running the JSpaceTest program requires a rather long command line to define a number of variables. I prefer to run the program from a batch file. The following shows the complete command, with line breaks inserted to fit on the page:

```
java -Djava.security.manager
 -Djava.security.policy=C:\tools\jmspolicy.txt
 -Doutrigger.spacename=JavaSpaces
 -Dcom.sun.jini.lookup.groups=public
 -Djava.rmi.server.codebase=http://localhost:8080/
space-examples-dl.jar
 -cp .; G:\jini1_1\lib\space-examples.jar
 com.lanw.clients.JSpaceTest
```

Timing Results

The time to get a JavaSpace reference averaged about 1700 milliseconds (or 1.7 seconds) and the time to write the entry about 110 milliseconds. This compares favorably with the JMS publishing times.

Reading from a Space

Listing 8.22 shows an example of reading a JSpaceSoapMsg object from a space. Selecting a message is done by means of the template object, which is simply a JSpaceSoapMsg object that has a value set for the topic String but has null for the soapmsg variable. The JavaSpace

returns a copy of any entry that has the same class and `topic` variable contents as the template object.

Note that the call to read includes a 60,000 millisecond "lease" time. If no entry satisfying the template requirement appears in that time, the `read` method returns `null`. A lease time of zero will cause an immediate return, with or without an object.

Listing 8.22: The *JSpaceProcess* Program to Read a *JSpaceSoapMsg*

```
package com.lanw.soapsrvr ;

import com.lanw.clients.* ;
import java.io.* ;
import java.util.* ;
import java.rmi.* ;
import net.jini.space.JavaSpace;

public class JSpaceProcess {

 public static void main(String[] args){
   try {
     JavaSpace jsp = SpaceUtil.getSpace();
     System.out.println("Got Jsp= " + jsp );
     JSpaceSoapMsg msgTemplate = new JSpaceSoapMsg();
     msgTemplate.topic = "policychange" ;
     JSpaceSoapMsg msg = (JSpaceSoapMsg) jsp.read( msgTemplate,
          null, 60000 );
     System.out.println("JSpace read got " + msg );
     int ch = System.in.read();
     System.exit(0);
   }catch(Exception e){
     e.printStackTrace( System.out );
   }
  }
 }
```

Here is the command line for executing the `JSpaceProcess` program:

```
java -Djava.security.manager
 -Djava.security.policy=C:\tools\jmspolicy.txt
 -Doutrigger.spacename=JavaSpaces
 -Dcom.sun.jini.lookup.groups=public
 -Djava.rmi.server.codebase=http://localhost:8080/space-examples-dl.jar
 -cp .;G:\jini1_1\lib\space-examples.jar
 com.lanw.soapsrvr.JSpaceProcess
```

Further processing of the SOAP message contents would be similar to that shown in the JMS examples.

An Alternative Based on Events

The example shown in Listing 8.22 is based on a style of programming based on the JavaSpace `read` method waiting for a message to appear. JavaSpaces also supports an event-oriented style in which an application registers as a listener for a particular template.

Listing 8.23 shows an example of a listener being notified. Note that a separate reading step would be required if you wanted to get the message contents.

Listing 8.23: The *JSpaceListener* Class

```
package com.lanw.soapsrvr ;

import com.lanw.clients.* ;
import java.io.* ;
import java.util.* ;
import java.rmi.* ;
import net.jini.space.JavaSpace;
import net.jini.core.event.* ;

public class JSpaceListener implements RemoteEventListener, Serializable {

 public static void main(String[] args){
   try {

     JavaSpace jsp = SpaceUtil.getSpace();
     System.out.println("Got Jsp= " + jsp );
     JSpaceSoapMsg msgTemplate = new JSpaceSoapMsg();
     msgTemplate.topic = "policychange" ;
     JSpaceListener jsListen = new JSpaceListener();
     jsListen.setNotification( jsp, msgTemplate );
     System.out.println("Template written to space");
     int ch = System.in.read();
     System.exit(0);
   }catch(Exception e){
     e.printStackTrace( System.out );
   }

 }

 // instance variables

  public JSpaceListener(){
  }

  public void setNotification( JavaSpace sp, JSpaceSoapMsg msg ){
    try {
     sp.notify( msg, // template
               null, // transaction if any
               this, // listener
```

```
                60000, // lease in msec
                null   // handback
                 );
      }catch( Exception te ){ // may get TransactionException
        te.printStackTrace( System.out );
      }
    }

    public void notify( RemoteEvent evt ){
      long id = evt.getID();
      long seq = evt.getSequenceNumber();
      java.rmi.MarshalledObject obj = evt.getRegistrationObject();
      System.out.println( id + " " + seq + " " + obj );
    }
  }
```

Alternate SOAP Message Transport

It seems to me that the SOAP community has been over-emphasizing RPC applications with HTTP as the transport mechanism. With the excellent Java tools available for other transport methods, you should not feel constrained to use HTTP and servlets. There are plenty of uses for SOAP messages in private networks, away from the Internet using JMS or JavaSpaces.