Sun Java™ System

Message Queue 3 Developer's Guide for Java Clients

2005Q1

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 819-0068-10

Contents

List of Figures 7

List of Tables 9

List of Code Examples 11

Preface 13

Who Should Use This Book 13
Before You Read This Book 14
How This Book Is Organized 14
Conventions Used in this Book 15
Text Conventions 15
Directory Variable Conventions 16
Related Documentation 18
The Message Queue Documentation Set 18
JavaDoc 19
Example Client Applications 19
The Java Message Service (JMS) Specification 20
The SOAP with Attachments API for Java (SAAJ) Specification 20
Books on JMS Programming 20
Related Third-Party Web Site References 21
Sun Welcomes Your Comments 21

Chapter 1 Overview 23
Message Queue for the Java Developer 23
Message Queue Java Client Files 23
JMS Client Setup Operations 24
To Set Up a Client to Produce Messages 24
To Set Up a Client to Consume Messages 24
Quick-Start Tutorial 25
Setting Up Your Environment 26

Starting and Testing the Message Server 28
To Start a Broker 28
To Test a Broker 28
Developing a Simple Client Application 29
Compiling and Running a Client Application 32
To Compile and Run the HelloWorldMessage Application 33
Example Application Code 34
Client Application Deployment Considerations 35

Chapter 2 Using Administered Objects 37
Overview 37
JNDI Lookup of Administered Objects 38
Looking Up ConnectionFactory Objects 39
To Perform a JNDI Lookup of a ConnectionFactory Object 39
Looking Up Destination Objects 40
To Perform a JNDI Lookup of a Destination Object 40
Instantiating Administered Objects 41
Instantiating ConnectionFactory Objects 41
To Directly Instantiate and Configure a ConnectionFactory Object 42
Instantiating Destination Objects 43
To Directly Instantiate and Configure a Destination Object 43
Overriding Administered Object Settings 44

Chapter 3 Message Queue Clients: Design and Features 45
Client Design Considerations 45
Developing Portable Clients 46
Choosing Programming Domains 46
Connections and Sessions 48
Producers and Consumers 49
Assigning Client Identifiers 49
Message Order and Priority 50
Using Selectors Efficiently 50
Balancing Reliability and Performance 52
Managing Client Threads 52
JMS Threading Restrictions 52
Thread Allocation for Connections 53
Managing Memory and Resources 54
Managing Memory 54
Managing Message Size 55
Message Compression 55
Advantages and Limitations of Compression 56
Compression Examples 56

Managing the Dead Message Queue . . . 57
Managing Physical Destination Limits . . . 61
Programming Issues for Message Consumers . . . 61
Using the Client Runtime Ping Feature . . . 61
Preventing Message Loss for Synchronous Consumers . . . 62
Synchronous Consumption in Distributed Applications . . . 62
Factors Affecting Performance . . . 63
Delivery Mode (Persistent/Non-persistent) . . . 64
Use of Transactions . . . 65
Acknowledgement Mode . . . 66
Durable vs. Non-Durable Subscriptions . . . 67
Use of Selectors (Message Filtering) . . . 68
Message Size . . . 68
Message Body Type . . . 69
Client Connection Failover (Auto-Reconnect) . . . 70
Enabling Auto-Reconnect . . . 70
Auto-Reconnect Behaviors . . . 72
Auto-Reconnect Limitations . . . 73
Auto-Reconnect Configuration Examples . . . 74
Single-Broker Auto-Reconnect . . . 74
Parallel Broker Auto-Reconnect . . . 74
Clustered-Broker Auto-Reconnect . . . 75
Custom Client Acknowledgment . . . 75
Using Client Acknowledge Mode . . . 76
Using No Acknowledge Mode . . . 78
Communicating with C Clients . . . 80

Chapter 4 Using the Metrics Monitoring API . . . 81
Monitoring Overview . . . 82
Administrative Tasks . . . 83
Implementation Summary . . . 83
Creating a Metrics-Monitoring Client . . . 84
Format of Metrics Messages . . . 85
Broker Metrics . . . 85
JVM Metrics . . . 87
Destination-List Metrics . . . 87
Destination Metrics . . . 88
Metrics Monitoring Client Code Examples . . . 90
A Broker Metrics Example . . . 90
A Destination List Metrics Example . . . 92
A Destination Metrics Example . . . 95

Chapter 5 Working with SOAP Messages **99**
What is SOAP? 100
SOAP with Attachments API for Java 100
The Transport Layer 101
The SOAP Layer 101
The Language Implementation Layer 102
The Profiles Layer 102
Interoperability 102
The SOAP Message 103
SOAP Packaging Models 104
SOAP Messaging in JAVA 107
The SOAP Message Object 107
Inherited Methods 109
Namespaces 110
Destination, Message Factory, and Connection Objects 113
Endpoint 114
Message Factory 115
Connection 115
Using SOAP Administered Objects 115
SOAP Messaging Models and Examples 117
SOAP Messaging Programming Models 118
Point-to-Point Connections 118
Working with Attachments 119
To Create and Add an Attachment 119
Exception and Fault Handling 120
Writing a SOAP Client 120
Writing a SOAP Service 123
Disassembling Messages 125
Handling Attachments 126
Replying to Messages 126
Handling SOAP Faults 126
Integrating SOAP and Message Queue 130
Example 1: Deferring SOAP Processing 131
To Transform the SOAP Message into a JMS Message and Send the JMS Message 132
To Receive the JMS Message, Transform it into a SOAP Message, and Process It 133
Example 2: Publishing SOAP Messages 134
Code Samples 135

Appendix A Warning Messages and Client Error Codes **141**

Index **155**

List of Figures

Figure 3-1 Performance Impact of Delivery Modes 65
Figure 3-2 Performance Impact of Subscription Types 68
Figure 3-3 Performance Impact of a Message Size 69
Figure 5-1 SOAP Messaging Layers 101
Figure 5-2 SOAP Interoperability 103
Figure 5-3 SOAP Message Without Attachments 105
Figure 5-4 SOAP Message with Attachments 106
Figure 5-5 SOAP Message Object 108
Figure 5-6 Request-Reply Messaging 118
Figure 5-7 SOAP Message Parts 121
Figure 5-8 SOAP Fault Element 127
Figure 5-9 Deferring SOAP Processing 132
Figure 5-10 Publishing a SOAP Message 135

List of Tables

Table 1 Book Contents 14
Table 2 Document Conventions 15
Table 3 Message Queue Directory Variables 16
Table 4 Message Queue Documentation Set 18
Table 1-1 jar File Locations 26
Table 1-2 jar Files Needed in `CLASSPATH` 27
Table 1-3 Location of Message Queue Executables 28
Table 1-4 Example Programs 34
Table 1-5 Checklist for the Message Queue Administrator 35
Table 3-1 JMS Programming Objects 47
Table 3-2 Message Properties Relating to Dead Message Queue 58
Table 3-3 Dead Message Properties 59
Table 3-4 Comparison of High Reliability and High Performance Scenarios 63
Table 4-1 Metrics Topic Destinations 82
Table 4-2 Data in the Body of a Broker Metrics Message 86
Table 4-3 Data in the Body of a JVM Metrics Message 87
Table 4-4 Data in the Body of a Destination-List Metrics Message 88
Table 4-5 Data in the Body of a Destination Metrics Message 88
Table 5-1 Inherited Methods 109
Table 5-2 SOAP Administered Object Information 116
Table 5-3 `JAXMServlet` Methods 124
Table 5-4 SOAP Faultcode Values 128
Table A-1 Message Queue Warning Message Codes 142
Table A-2 Message Queue Client Error Codes 143

List of Code Examples

Code Example 2-1 Looking Up a ConnectionFactory Object 40
Code Example 2-2 Instantiating a ConnectionFactory Object 42
Code Example 2-3 Instantiating a Destination Object 43
Code Example 3-1 Sending a Compressed Message 56
Code Example 3-2 Comparing Size of Compressed and Uncompressed Messages 57
Code Example 3-3 Example of Command to Configure a Single Broker 74
Code Example 3-4 Example of Command to Configure Parallel Brokers 74
Code Example 3-5 Example of Command to Configure a Broker Cluster 75
Code Example 3-6 Syntax for Acknowledge Methods 76
Code Example 3-7 Example of Custom Client Acknowledgement Code 77
Code Example 4-1 Example of Subscribing to a Broker Metrics Topic 90
Code Example 4-2 Example of Processing a Broker Metrics Message 91
Code Example 4-3 Example of Subscribing to the Destination List Metrics Topic 92
Code Example 4-4 Example of Processing a Destination List Metrics Message 93
Code Example 4-5 Example of Extracting Destination Information From a Hashtable 94
Code Example 4-6 Example of Subscribing to a Destination Metrics Topic 95
Code Example 4-7 Example of Processing a Destination Metrics Message 96
Code Example 5-1 Explicit Namespace Declarations 111
Code Example 5-3 Looking up an Endpoint Administered Object 117
Code Example 5-2 Adding an Endpoint Administered Object 117
Code Example 5-5 A Simple Ping Message Service 124
Code Example 5-4 Skeleton Message Consumer 124
Code Example 5-6 Processing a SOAP Message 126
Code Example 5-7 Sending a JMS Message with a SOAP Payload 136
Code Example 5-8 Receiving a JMS Message with a SOAP Payload 138

Preface

This book provides information about concepts and procedures for developing Java™ messaging applications (Java clients) that work with Sun Java™ System Message Queue (formerly Sun™ ONE Message Queue).

This preface contains the following sections:

- "Who Should Use This Book" on page 13
- "Before You Read This Book" on page 14
- "How This Book Is Organized" on page 14
- "Conventions Used in this Book" on page 15
- "Related Documentation" on page 18
- "Related Third-Party Web Site References" on page 21
- "Sun Welcomes Your Comments" on page 21

Who Should Use This Book

This guide is meant principally for developers of Java applications that use Sun Java System Message Queue.

These applications use the Java Message Service (JMS) Application Programming Interface (API), and possibly the SOAP with Attachments API for Java (SAAJ), to create, send, receive, and read messages. As such, these applications are JMS clients and/or SOAP client applications, respectively. The JMS and SAAJ specifications are open standards.

This *Message Queue Developer's Guide for Java Clients* assumes that you are familiar with the JMS APIs and with JMS programming guidelines. Its purpose is to help you optimize your JMS client applications by making best use of the features and flexibility of a Message Queue messaging system.

This book assumes no familiarity, however, with SAAJ. This material is described in Chapter 5, "Working with SOAP Messages," and only assumes basic knowledge of XML.

Before You Read This Book

You must read the *Message Queue Technical Overview* to become familiar with Message Queue's implementation of the Java Message Specification, with the components of the Message Queue service, and with the basic process of developing, deploying, and administering a Message Queue application.

How This Book Is Organized

This guide is designed to be read from beginning to end. The following table briefly describes the contents of each chapter:

Table 1 Book Contents

Chapter	Description
Chapter 1, "Overview"	A high level overview of. It includes a tutorial that acquaints you with the Message Queue development environment using a simple example JMS client application.
Chapter 2, "Using Administered Objects"	Describes how to use Message Queue administered objects in both a provider- independent and provider-specific way.
Chapter 3, "Message Queue Clients: Design and Features"	Describes architectural and configuration issues that depend upon Message Queue's implementation of the Java Message Specification
Chapter 4, "Using the Metrics Monitoring API"	Describes message-based monitoring, a customized solution to metrics gathering that allows metrics data to be accessed programmatically and then to be processed in whatever way suits the consuming client.
Chapter 5, "Working with SOAP Messages"	Explains how you send and receive SOAP messages with and without Message Queue support.

Table 1 Book Contents *(Continued)*

Chapter	Description
Appendix A, "Warning Messages and Client Error Codes"	Provides reference information for warning messages and error codes returned by the Message Queue client runtime when it raises a JMS exception.

Conventions Used in this Book

This section provides information about the conventions used in this document.

Text Conventions

Table 2 Document Conventions

Format	Description
italics	Italicized text represents a placeholder. Substitute an appropriate clause or value where you see italic text. Italicized text is also used to designate a document title, for emphasis, or for a word or phrase being introduced.
`monospace`	Monospace text represents example code, commands that you enter on the command line, directory, file, or path names, error message text, class names, method names (including all elements in the signature), package names, reserved words, and URLs.
`[]`	Square brackets to indicate optional values in a command line syntax statement.
ALL CAPS	Text in all capitals represents file system types (GIF, TXT, HTML and so forth), environment variables (IMQ_HOME), or abbreviations (JSP).
Key+Key	Simultaneous keystrokes are joined with a plus sign: Ctrl+A means press both keys simultaneously.
Key-Key	Consecutive keystrokes are joined with a hyphen: Esc-S means press the Esc key, release it, then press the S key.

Directory Variable Conventions

Message Queue makes use of three directory variables; how they are set varies from platform to platform. Table 3 describes these variables and summarizes how they are used on the Solaris™, Windows, and Linux platforms.

Table 3 Message Queue Directory Variables

Variable	Description
IMQ_HOME	This is generally used in Message Queue documentation to refer to the Message Queue base directory (root installation directory): • On Solaris, there is no root Message Queue installation directory. Therefore, IMQ_HOME is not used in Message Queue documentation to refer to file locations on Solaris. • On Windows, the root Message Queue installation directory is set by the Message Queue installer (by default, as C:\Program Files\Sun\MessageQueue3). • On Linux, there is no root Message Queue installation directory. Therefore, IMQ_HOME is not used in Message Queue documentation to refer to file locations on Linux. • For Sun Java System Application Server, on Windows, Solaris, and Linux, the root Message Queue installation directory is /imq, under the Application Server base directory.
IMQ_VARHOME	This is the /var directory in which Message Queue temporary or dynamically-created configuration and data files are stored. It can be set as an environment variable to point to any directory. • On Solaris, IMQ_VARHOME defaults to the /var/imq directory. • On Windows IMQ_VARHOME defaults to the IMQ_HOME\var directory. • On Linux, IMQ_VARHOME defaults to the /var/opt/sun/mq directory. • For Sun Java System Application Server, on Solaris and Linux, IMQ_VARHOME defaults to the domain/domain1/imq directory under the App Server base directory. • For Sun Java System Application Server, on Windows, IMQ_VARHOME defaults to the domain\domain1\imq directory under the App Server base directory.

Table 3 Message Queue Directory Variables *(Continued)*

Variable	Description
IMQ_JAVAHOME	This is an environment variable that points to the location of the Java runtime (JRE) required by Message Queue executables: • On Solaris, IMQ_JAVAHOME looks for the java runtime in the following order, but a user can optionally set the value to wherever the required JRE resides. Solaris 8 or 9: /usr/jdk/entsys-j2se /usr/jdk/jdk1.5.* /usr/jdk/j2sdk1.5.* /usr/j2se Solaris 10: /usr/jdk/entsys-j2se /usr/java /usr/j2se • On Linux, Message Queue first looks for the java runtime in the following order, but a user can optionally set the value of IMQ_JAVAHOME to wherever the required JRE resides. /usr/jdk/entsys-j2se /usr/java/jre1.5.* /usr/java/jdk1.5.* /usr/java/jre1.4.2* /usr/java/j2sdk1.4.2* • On Windows, IMQ_JAVAHOME defaults to IMQ_HOME\jre, but a user can optionally set the value to wherever the required JRE resides.

In this guide, IMQ_HOME, IMQ_VARHOME, and IMQ_JAVAHOME are shown *without* platform-specific environment variable notation or syntax (for example, $IMQ_HOME on UNIX). Path names generally use UNIX directory separator notation (/).

Related Documentation

In addition to this guide, Message Queue provides additional documentation resources.

The Message Queue Documentation Set

The documents that comprise the Message Queue documentation set are listed in Table 4 in the order in which you would normally use them.

Table 4 Message Queue Documentation Set

Document	Audience	Description
Message Queue Installation Guide	Developers and administrators	Explains how to install Message Queue software on Solaris, Linux, and Windows platforms.
Message Queue Release Notes	Developers and administrators	Includes descriptions of new features, limitations, and known bugs, as well as technical notes.
Message Queue Technical Overview	Developers and administrators	Explains basic messaging concepts and processes.
Message Queue Developer's Guide for Java Clients	Developers	Provides a quick-start tutorial and programming information for developers of Java client programs using JMS and SAAJ and Message Queue software.
Message Queue Developer's Guide for C Clients	Developers	Provides programming and reference documentation for developers of C client programs that use the Message Queue software.
Message Queue Administration Guide	Administrators, also recommended for developers	Provides background and information needed to perform administration tasks using Message Queue administration tools.

JavaDoc

JMS and Message Queue API documentation in JavaDoc format is provided at the following location:

Platform	Location
Solaris	`/usr/share/javadoc/imq/index.html`
Linux	`/opt/sun/mq/javadoc/index.html/`
Windows	`IMQ_HOME/javadoc/index.html`

This documentation can be viewed in any HTML browser such as Netscape or Internet Explorer. It includes standard JMS API documentation as well as Message Queue-specific APIs for Message Queue administered objects (see Chapter 2, "Using Administered Objects").

Example Client Applications

Example applications that provide sample Java client application code are included in the following directories:

Platform	Location
Solaris	`/usr/demo/imq/`
Linux	`/opt/sun/mq/examples/`
Windows	`IMQ_HOME\demo\`

See the `README` file located in that directory and in each of its subdirectories.

The Java Message Service (JMS) Specification

The JMS specification can be found at the following location:

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

The specification includes sample client code.

The SOAP with Attachments API for Java (SAAJ) Specification

The SAAJ specification can be found at the following location:

`http://java.sun.com/xml/downloads/saaj.html`

The specification includes sample client code.

Books on JMS Programming

For background on using the JMS API, you can consult the following publicly-available books:

- *Java Message Service* by Richard Monson-Haefel and David A. Chappell, O'Reilly and Associates, Inc., Sebastopol, CA
- *Professional JMS* by Scott Grant, Michael P. Kovacs, Meeraj Kunnumpurath, Silvano Maffeis, K. Scott Morrison, Gopalan Suresh Raj, Paul Giotta, and James McGovern, Wrox Press Inc., ISBN: 1861004931
- *Practical Java Message Service* by Tarak Modi, Manning Publications, ISBN: 1930110138

Related Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

NOTE Sun is not responsible for the availability of third-party Web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with the use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions.

To share your comments, go to `http://docs.sun.com` and click Send Comments. In the online form, provide the document title and part number. The part number is a seven-digit or nine-digit number that can be found on the title page of the book or at the top of the document.

Chapter 1

Overview

This chapter provides an overall introduction to Sun Java™ System Message Queue and a quick-start tutorial. You should be familiar with the concepts presented in the *Message Queue Technical Overview* before you read this chapter.

The chapter covers the following topics:

- "Message Queue for the Java Developer" on page 23
- "Quick-Start Tutorial" on page 25
- "Client Application Deployment Considerations" on page 35

Message Queue for the Java Developer

This section lists the component files needed for Message Queue client development and provides a summary of the basic steps needed to create applications that produce and consume messages.

Message Queue Java Client Files

The Message Queue files that need to be used in conjunction with the Message Queue Java clients can be found in the `lib` directory in the installed location for Message Queue on your platform.

Message Queue Java clients need to be able to use several `.jar` files found in the `lib` directory when these clients are compiled and run.

JMS Client Setup Operations

The general procedures for producing and consuming messages are introduced below. These procedures have a number of common steps which need not be duplicated if a client is both producing and consuming messages.

For a more detailed presentation of creating, compiling, and running a simple client application, see "Quick-Start Tutorial" on page 25.

➤ **To Set Up a Client to Produce Messages**

1. Look up or instantiate a `ConnectionFactory` object.
2. Use the `ConnectionFactory` object to create a `Connection` object.
3. Use the `Connection` object to create one or more `Session` objects.
4. Look up or instantiate one or more `Destination` objects.
5. Use a `Session` object and a `Destination` object to create any needed `MessageProducer` objects.

   (You can create a `MessageProducer` object without specifying a `Destination` object, but then you have to specify a `Destination` object for each message that you produce.)

   At this point the client has the basic setup needed to produce messages.
6. Use a `Session` object to create a message.
7. Use the `Message` object to set its contents. Use the `Message` object or the `MessageProducer` object to set the message properties.
8. Use the `MessageProducer` object to send the message.

➤ **To Set Up a Client to Consume Messages**

1. Look up or instantiate a `ConnectionFactory` object.
2. Use the `ConnectionFactory` object to create a `Connection` object.
3. Use the `Connection` object to create one or more `Session` objects.
4. Look up or instantiate one or more `Destination` objects.

5. Use a `Session` object and a `Destination` object to create any needed `MessageConsumer` objects.
6. If you want to receive messages asynchronously, instantiate a `MessageListener` object and register it with a `MessageConsumer` object.
7. Tell the `Connection` object to start delivery of messages. This allows messages to be delivered to the client for consumption.
8. If you are using a synchronous receiver, you can now make the call to retrieve messages.

At this point the client has the basic setup needed to consume messages. Note that the consumer can use synchronous or asynchronous consumption, but not both.

Quick-Start Tutorial

This section provides a quick introduction to JMS client programming in a Sun Java System Message Queue environment. It describes the procedures used to create, compile, and run a simple `HelloWorldMessage` example application.

It covers the following topics:

- "Setting Up Your Environment" on page 26
- "Starting and Testing the Message Server" on page 28
- "Developing a Simple Client Application" on page 29
- "Compiling and Running a Client Application" on page 32
- "Example Application Code" on page 34

For the purpose of this tutorial it is sufficient to run the Message Queue server in a default configuration. For instructions on configuring a Message Queue server, please refer to the *Message Queue Administration Guide*.

The minimum JDK level required to compile and run Message Queue clients is 1.2.

Setting Up Your Environment

You need to set the CLASSPATH environment variable when compiling and running a JMS client. (The IMQ_HOME variable, where used, refers to the directory where Message Queue is installed on Windows platforms and some Sun Java System Application Server platforms.)

The value of CLASSPATH depends on the following factors:

- the platform on which you compile or run
- whether you are compiling or running a JMS application
- whether your application uses SOAP
- whether your application uses the SOAP/JMS transformer utilities
- the JDK version you are using (which affects JNDI support).

Table 1-1 specifies the directories where jar files are to be found on the different platforms:

Table 1-1 jar File Locations

Platform	Directory
Solaris™	/usr/share/lib/
Solaris, using the standalone version of Sun Java System Application Server	IMQ_HOME/lib/
Linux	/opt/mq/lib/
Windows	IMQ_HOME\lib\

Table 1-2 lists the jar files you need to compile and run different kinds of code.

Table 1-2 jar Files Needed in CLASSPATH

Code	To Compile	To Run	Discussion

JMS client	`jms.jar`
	`imq.jar`
	`jndi.jar`
		`jms.jar`
		`imq.jar`
		Directory containing compiled Java app or '.'
			See discussion of JNDI jar files, following this table.

SOAP Client	`saaj-api.jar` `activation.jar`	`saaj-api.jar` Directory containing compiled Java app or '.'	See Chapter 5, "Working with SOAP Messages"

SOAP Servlet	`jaxm-api.jar`
	`saaj-api.jar`
	`activation.jar`
			Sun Java System App Server already includes these jar files for SOAP servlet support.

code using SOAP/JMS transformer utilities	`imqxm.jar` (and jars for JMS and SOAP clients)	`imqxm.jar`	Also add the appropriate jar files mentioned in this table for the kind of code you are writing.

A client application must be able to access JNDI jar files (`jndi.jar`) even if the application does not use JNDI directly to look up Message Queue administered objects. This is because JNDI is referenced by `Destination` and `ConnectionFactory` classes.

JNDI jar files are bundled with JDK 1.4. Thus, if you are using this JDK, you do not have to add `jndi.jar` to your `CLASSPATH` setting. However, if you are using an earlier version of the JDK, you must include `jndi.jar` in your classpath.

If you are using JNDI to look up Message Queue administered objects, you must also include the following files in your `CLASSPATH` setting:

- if you are using the File-System service provider for JNDI (with any JDK version), you must include the `fscontext.jar` file.
- if you are using the LDAP context
  - with JDK 1.2 or 1.3, include the `ldap.jar`, `ldabbp.jar`, and `fscontext.jar` files.
  - with JDK 1.4, all files are already bundled with this JDK.

Starting and Testing the Message Server

This tutorial assumes that you do not have a Message Queue server currently running. A message server consists of one or more brokers—the software component that routes and delivers messages.

(If you run the broker as a UNIX startup process or Windows service, then it is already running and you can skip to "To Test a Broker" below.)

➤ To Start a Broker

1. In a terminal window, change to the directory containing Message Queue executables:

Table 1-3 Location of Message Queue Executables

Platform	Location
Solaris	`/usr/bin/`
Linux	`/opt/sun/mq/bin/`
Windows	`IMQ_HOME/bin/`

2. Run the broker startup command (`imqbrokerd`) as shown below.

   ```
   imqbrokerd -tty
   ```

   The `-tty` option causes all logged messages to be displayed to the terminal console (in addition to the log file).

   The broker will start and display a few messages before displaying the message, "imqbroker@host:7676 ready." It is now ready and available for clients to use.

➤ To Test a Broker

One simple way to check the broker startup is by using the Message Queue Command (`imqcmd`) utility to display information about the broker.

1. In a separate terminal window, change to the directory containing `Message Queue` executables (see Table 1-3).

2. Run `imqcmd` with the arguments shown below. (Supply the default password of `admin` whenprompted to do so.)

   ```
   imqcmd query bkr -u admin
   ```

The output displayed should be similar to what is shown below.

```
% imqcmd query bkr -u admin

Querying the broker specified by:

-------------------------
Host         Primary Port
-------------------------
localhost    7676

Version                                          3.6
Instance Name                                    imqbroker
Primary Port                                     7676

Current Number of Messages in System             0
Current Total Message Bytes in System            0

Max Number of Messages in System                 unlimited (-1)
Max Total Message Bytes in System                unlimited (-1)
Max Message Size                                 70m


Auto Create Queues                               true
Auto Create Topics                               true
Auto Created Queue Max Number of Active Consumers  1
Auto Created Queue Max Number of Backup Consumers  0

Cluster Broker List (active)
Cluster Broker List (configured)
Cluster Master Broker
Cluster URL

Log Level                                        INFO
Log Rollover Interval (seconds)                  604800
Log Rollover Size (bytes)                        unlimited (-1)

Successfully queried the broker.

Current Number of Messages in System        0
```

Developing a Simple Client Application

This section leads you through the steps used to create a simple "Hello World" client application that sends a message to a queue destination and then retrieves the same message from the queue. You can find this example, named `HelloWorldMessage` in the `IMQ_HOME/demo/helloworld/helloworldmessage` directory.

The following steps describe the HelloWorldMessage example

1. Import the interfaces and Message Queue implementation classes for the JMS API. (The `javax.jms` package defines all the JMS interfaces necessary to develop a JMS client.)

   ```
   import javax.jms.QueueConnectionFactory;
   import javax.jms.QueueConnection;
   import javax.jms.QueueSession
   import javax.jms.QueueSender;
   import javax.jms.QueueReceiver;
   import javax.jms.Queue;
   import javax.jms.Session;
   import javax.jms.Message;
   import javax.jms.TextMessage;
   ```

   Note: To prevent compilation errors, especially when using JDK 1.5 and above, do not use the wildcard character to specify the files to be imported. For example,

   ```
   import javax.jms.*;\\ DO NOT USE

   import java.util.*; \\ DO NOT USE
   ```

2. Instantiate a Message Queue `QueueConnectionFactory` administered object.

   A `QueueConnectionFactory` object encapsulates all the Message Queue-specific configuration properties for creating `QueueConnection` connections to a Message Queue server.

   ```
   QueueConnectionFactory myQConnFactory =
       new com.sun.messaging.QueueConnectionFactory();
   ```

   `ConnectionFactory` administered objects can also be accessed through a JNDI lookup (see "Looking Up ConnectionFactory Objects" on page 39). This approach makes the client code JMS-provider independent and also allows for a centrally administered messaging system.

3. Create a connection to the message server.

   A `QueueConnection` object is the active connection to the message server in the Point-To-Point programming domain.

   ```
   QueueConnection myQConn =
       myQConnFactory.createQueueConnection();
   ```

4. Create a session within the connection.

   A `QueueSession` object is a single-threaded context for producing and consuming messages. It enables clients to create producers and consumers of messages for a queue destination.

   ```
   QueueSession myQSess = myQConn.createQueueSession(false,
       Session.AUTO_ACKNOWLEDGE);
   ```

   The `myQSess` object created above is non-transacted and automatically acknowledges messages upon consumption by a consumer.

5. Instantiate a Message Queue `queue` administered object that corresponds to a queue destination in the message server.

   Destination administered objects encapsulate provider-specific destination naming syntax and behavior. The code below instantiates a `queue` administered object for a physical queue destination named "world".

   ```
   Queue myQueue = new com.sun.messaging.Queue("world");
   ```

   Destination administered objects can also be accessed through a JNDI lookup (see "Looking Up Destination Objects" on page 40). This approach makes the client code JMS-provider independent and also allows for a centrally administered messaging system.

6. Create a `QueueSender` message producer.

   This message producer, associated with `myQueue`, is used to send messages to the queue destination named "world".

   ```
   QueueSender myQueueSender = myQSess.createSender(myQueue);
   ```

7. Create and send a message to the queue.

   You create a `TextMessage` object using the `QueueSession` object and populate it with a string representing the data of the message. Then you use the `QueueSender` object to send the message to the "world" queue destination.

   ```
   TextMessage myTextMsg = myQSess.createTextMessage();
   myTextMsg.setText("Hello World");
   System.out.println("Sending Message: " + myTextMsg.getText());
   myQueueSender.send(myTextMsg);
   ```

8. Create a `QueueReceiver` message consumer.

   This message consumer, associated with `myQueue`, is used to receive messages from the queue destination named "world".

   ```
   QueueReceiver myQueueReceiver =
       myQSess.createReceiver(myQueue);
   ```

9. Start the `QueueConnection` you created in Step 3.

   Messages for consumption by a client can only be delivered over a connection that has been started (while messages produced by a client can be delivered to a destination without starting a connection, as in Step 7.

   ```
   myQConn.start();
   ```

10. Receive a message from the queue.

    You receive a message from the "world" queue destination using the `QueueReceiver` object. The code, below, is an example of a synchronous consumption of messages.

    ```
    Message msg = myQueueReceiver.receive();
    ```

11. Retrieve the contents of the message.

    Once the message is received successfully, its contents can be retrieved.

    ```
    if (msg instanceof TextMessage) {
        TextMessage txtMsg = (TextMessage) msg;
        System.out.println("Read Message: " + txtMsg.getText());
    }
    ```

12. Close the session and connection resources.

    ```
    myQSess.close();
    myQConn.close();
    ```

Compiling and Running a Client Application

To compile and run Java clients in a Message Queue environment, it is recommended that you use the Java2 SDK Standard Edition v1.4 or later, though version 1.2 is also supported. The recommended SDK can be downloaded from the following location:

`http://java.sun.com/j2se/1.4`

Be sure that you have set the `CLASSPATH` environment variable correctly, as described in "Setting Up Your Environment" on page 26, before attempting to compile or run a client application.

Note: Users of JDK 1.5 may get compiler errors if they use the unqualified JMS `Queue` class along with the following import statement

```
import java.util.*
```

This is because both the `java.util` package and the `javax.jms` package contain a `Queue` class. To avoid the compilation error in this situation, you need either to fully qualify references to the JMS Queue class as `javax.jms.Queue` or to correct your import statements to refer to explicit classes in order to eliminate the ambiguity for the compiler.

The following instructions are based on the HelloWorldMessage application, as created in "Developing a Simple Client Application" on page 29, and located in the Message Queue 3.6 JMS example applications directory (see Table 1-4). Please note that these instructions are furnished as an example. You do not actually need to compile the example; it is shipped precompiled. Of course, if you modify the source for the example, you will need to recompile.

➤ To Compile and Run the HelloWorldMessage Application

1. Make the directory containing the application your current directory.

   The Message Queue 3.6 example applications directory on Solaris is not writable by users, so copy the HelloWorldMessage application to a writable directory and make that directory your current directory.

2. Compile the HelloWorldMessage application as shown below.

   ```
   javac HelloWorldMessage.java
   ```

   This step results in the `HelloWorldMessage.class` file being created in the current directory.

3. Run the HelloWorldMessage application:

   ```
   java HelloWorldMessage
   ```

   The following output is displayed when you run HelloWorldMessage.

   ```
   Sending Message: Hello World
   Read Message: Hello World
   ```

Example Application Code

The example applications provided by Message Queue 3.6 consist of both JMS messaging applications as well as JAXM messaging examples (see "Working with SOAP Messages" on page 99 for more information).

Directories containing example application code are set as follows:

- Solaris: `/usr/demo/imq`
- Linux: `/opt/sun/mq/examples`
- Windows: `IMQ_HOME\demo\`

Each directory (except for the JMS directory) contains a README file that describes the source files included in that directory. Table 1-4 lists and describes the contents of the directories of interest to Message Queue Java clients.

Table 1-4 Example Programs

Directory	Contents
`helloworld`	Simple programs that show how a JMS client is created and deployed in Message Queue. These examples include the steps required to create administered objects in Message Queue, and show to use JNDI in the client to look up and use those objects.
`jms`	Sample programs that demonstrate the use of the JMS API with Message Queue.
`jaxm`	Sample programs that demonstrate how to use SOAP messages in conjunction with JMS in Message Queue.
`applications`	Four directories: • One contains source for a GUI application that uses the Message Queue JMS monitoring API to get the list of queues from a Message Queue broker and browse their contents using a JMS queue browser. • One contains source for a GUI application that uses the JMS API to implement a simple chat application. • One contains the MQ Ping demo program. • One contains the MQ Applet program.
`monitoring`	Sample programs that demonstrate how to use the JMS API for monitoring the broker.
`jdbc`	Examples for plugging in a PointBase and an Oracle database.
`imqobjmgr`	Examples of `imqobjmgr` command files.

Client Application Deployment Considerations

When you are ready to deploy your client application, you should make sure the administrator knows your application's needs. The checklist in Table 1-5 shows the basic information required. Consult with your administrator to determine the exact information needed. In some cases, it might be useful to provide a range of values rather than a specific value. For details on configuration and on attribute names and default values for administered objects, refer to the *Message Queue Administration Guide*.

Table 1-5 Checklist for the Message Queue Administrator

Configuring administered objects:
Connection factories to be created
Type:
JNDI lookup name:
Other attributes:
Destination objects to be created
Type (queue or topic):
JNDI lookup name:
Physical destination name (if your administrator wants it):
Configuring a broker or broker cluster:
Name:
Port
Properties
Configuring physical destinations:
Type:
Name:
Attributes:
Maximum number of messages expected:
Maximum size of messages expected:
Maximum message bytes expected:
Configuring Dead Message Queue
Place dead messages on Dead Message Queue
Log the placement of messages on the Dead Message Queue
Discard the body of messages placed on the Dead Message Queue

Chapter 2

Using Administered Objects

Administered objects encapsulate provider-specific implementation and configuration information in objects that are used by Message Queue clients. This chapter describes the types of administered objects used by a JMS application and explains how they can be created and used. The chapter covers the following topics:

- "Overview" on page 37
- "JNDI Lookup of Administered Objects" on page 38
- "Instantiating Administered Objects" on page 41
- "Overriding Administered Object Settings" on page 44

Overview

Message Queue provides two types of JMS administered objects—connection factory and destination—as well as a JAXM administered object. While all encapsulate provider-specific information, they have very different uses. (For information on using JAXM administered objects, see Chapter 5, "Working with SOAP Messages" on page 99).

`ConnectionFactory` and `XAConnectionFactory` (distributed transaction) objects are used to create connections to the Message Queue server. Destination objects (which represent physical destinations) are used to create JMS message consumers and producers (see "Developing a Simple Client Application" on page 29). The JAXM endpoint administered object is used to send SOAP messages (see Chapter 5, "Working with SOAP Messages").

There are two approaches to the use of administered objects:

- They can be created and configured by an administrator, stored in an object store, accessed by clients through standard JNDI lookup code, and then used in a provider-independent manner.

  NOTE In the case where Message Queue clients are J2EE components, JNDI resources are provided by the J2EE container, and JNDI lookup code might differ from that shown in this chapter. Please consult your J2EE provider documentation for such details.

- They can be instantiated and configured by a developer when writing application code. In this case, they are used in a provider-specific manner.

The approach you take in using administered objects depends on the environment in which your application will be run and how much control you want your client to have over Message Queue-specific configuration details. This chapter describes these two approaches and explains how to code your client for each.

JNDI Lookup of Administered Objects

If you wish an application to be run under controlled conditions in a centrally administered messaging environment, then Message Queue administered objects should be created and configured by an administrator. This makes it possible for the administrator to do the following:

- Control the behavior of connections by requiring clients to access pre-configured `ConnectionFactory` objects through a JNDI lookup.
- Control the proliferation of physical destinations by requiring clients to access only `Destination` objects that correspond to existing physical destinations.

This approach gives the administrator control over message server and client runtime configuration details, and at the same time allows clients to be JMS provider-independent: they do not have to know about provider-specific syntax and object naming conventions or provider-specific configuration properties.

An administrator creates administered objects in an object store using Message Queue administration tools, as described in the *Message Queue Administration Guide*. When creating an administered object, the administrator can specify that it be read only—that is, clients cannot change Message Queue-specific configuration

values specified when the object was created. In other words, application code cannot set attribute values on read-only administered objects, nor can they be overridden using client startup options, as described in "Overriding Administered Object Settings" on page 44.

While it is possible for clients to instantiate `ConnectionFactory` and `Destination` administered objects on their own, this practice undermines the basic purpose of an administered object—to allow an administrator to control the broker resources required by an application and to tune application performance. Instantiating administered objects also makes a client provider specific.

Looking Up ConnectionFactory Objects

➤ **To Perform a JNDI Lookup of a ConnectionFactory Object**

1. Create an initial context for the JNDI lookup.

   The details of how you create this context depend on whether you are using a file-system object store or an LDAP server for your Message Queue administered objects. The code below assumes a file-system store. For information about the corresponding LDAP object store attributes, see the *Message Queue Administration Guide*.

   ```
   Hashtable env = new Hashtable();
   env.put (Context.INITIAL_CONTEXT_FACTORY,
       "com.sun.jndi.fscontext.RefFSContextFactory");
   env.put (Context.PROVIDER_URL,
       "file:///c:/imq_admin_objects");
   Context ctx = new InitialContext(env);
   ```

   NOTE You need to create the directory represented by `c:/`*imq_admin_objects* before referencing it in your code. (that is, `c:/`*imq_admin_objects* must be an existing directory).

   You can also set an environment by specifying system properties on the command line, rather than programmatically, as shown above. For instructions, see the `README` file in the JMS example applications directory.

   If you use system properties to set the environment, then you initialize the context without providing the `env` parameter:

   ```
   Context ctx = new InitialContext();
   ```

2. Perform a JNDI lookup on the "lookup" name under which the `ConnectionFactory` or `XAConnectionFactory` object was stored in the JNDI object store.

   ```
   QueueConnectionFactory myQConnFactory = (QueueConnectionFactory)
       ctx.lookup("MyQueueConnectionFactory");
   ```

   It is recommended that you use this connection factory as originally configured. For a discussion of `ConnectionFactory` and `XAConnectionFactory` object attributes, see the *Message Queue Administration Guide*.

3. Use the ConnectionFactory to create a connection object.

   ```
   QueueConnection myQConn =
       myQConnFactory.createQueueConnection();
   ```

The code in the previous steps is shown in Code Example 2-1. (The directory represented by `c:/`*imq_admin_objects* must be an existing directory.)

Code Example 2-1 Looking Up a ConnectionFactory Object

```
Hashtable env = new Hashtable();
env.put (Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.jndi.fscontext.RefFSContextFactory");
env.put (Context.PROVIDER_URL,
    "file:///c:/imq_admin_objects");
Context ctx = new InitialContext(env);
QueueConnectionFactory myQConnFactory = (QueueConnectionFactory)
    ctx.lookup("MyQueueConnectionFactory");
QueueConnection myQConn =
    myQConnFactory.createQueueConnection();
```

Looking Up Destination Objects

➤ **To Perform a JNDI Lookup of a Destination Object**

1. Using the same initial context used in performing the `ConnectionFactory` lookup, Perform a JNDI lookup on the "lookup" name under which the `Destination` object was stored in the JNDI object store.

   ```
   Queue myQ =
   (Queue) ctx.lookup("MyQueueDestination");
   ```

Instantiating Administered Objects

If you do not wish an application to be run under controlled conditions in a centrally administered environment, then you can instantiate and configure administered objects in application code.

While this approach gives you, the developer, control over message server and client runtime configuration details, it also means that your clients are not portable to other JMS providers. Typically, you might instantiate administered objects in application code in the following situations:

- You are in the early stages of development in which there is no real need to create, configure, and store administered objects. You just want to develop and debug your application without involving JNDI lookups.
- You are not concerned about your clients being ported to other JMS providers or being reconfigurable.

Instantiating administered objects in application code means you are hard-coding configuration values into your application. You give up the flexibility of having an administrator reconfigure the administered objects to achieve higher performance or throughput after an application has been deployed.

Instantiating ConnectionFactory Objects

There are two object constructors for instantiating Message Queue `ConnectionFactory` administered objects, one for each programming domain:

- **Publish/subscribe (Topic) domain**

  ```
  new com.sun.messaging.TopicConnectionFactory();
  ```

  Instantiates a `TopicConnectionFactory` with a default configuration (creates Topic TCP-based connections to a broker running on "`localhost`" at port number `7676`).

- **Point to point (Queue) domain**

  ```
  new com.sun.messaging.QueueConnectionFactory();
  ```

  Instantiates a `QueueConnectionFactory` with a default configuration (creates Queue TCP-based connections to a broker running on "`localhost`" at port number `7676`).

- **Unified domain (Topics and Queues)**

  ```
  new com.sun.messaging.ConnectionFactory();
  ```

  Instantiates a `ConnectionFactory` with a default configuration (creates TCP-based connections to a broker running on "`localhost`" at port number `7676`). Supports both publish/subscribe and point-to-point messaging.

➤ To Directly Instantiate and Configure a ConnectionFactory Object

1. Instantiate a `ConnectionFactory` object using the appropriate constructor.

   ```
   com.sun.messaging.ConnectionFactory myConnFactory =
       new com.sun.messaging.ConnectionFactory();
   ```

2. Configure the `ConnectionFactory` object.

   ```
   myConnFactory.setProperty("imqAdressList", "mq");
   ```

   For a discussion of `ConnectionFactory` configuration properties, see the *Message Queue Administration Guide*

3. Use the ConnectionFactory to create a `Connection` object.

   ```
   Connection myConn =
       myConnFactory.createConnection();
   ```

The code in the previous steps is shown in Code Example 2-2.

Code Example 2-2 Instantiating a ConnectionFactory Object

```
com.sun.messaging.ConnectionFactory myQonnFactory =
    new com.sun.messaging.ConnectionFactory();
try {
    myConnFactory.setProperty("imqAddressList", "mq");
} catch (JMSException je) {
}
Connection myConn =
    myConnFactory.createConnection();
```

Instantiating Destination Objects

There are two object constructors for instantiating Message Queue `Destination` administered objects, one for each programming domain:

- **Publish/subscribe (Topic) domain**

  ```
  new com.sun.messaging.Topic();
  ```

  Instantiates a `Topic` with the default destination name of "Untitled_Destination_Object".

- **Point to point (Queue) domain**

  ```
  new com.sun.messaging.Queue();
  ```

  Instantiates a `Queue` with the default destination name of "Untitled_Destination_Object".

➤ **To Directly Instantiate and Configure a Destination Object**

1. Instantiate a Topic or Queue `Destination` object using the appropriate constructor.

   ```
   com.sun.messaging.Queue myQueue = new com.sun.messaging.Queue();
   ```

2. Configure the Destination object.

   ```
   myQueue.setProperty("imqDestinationName", "new_queue_name");
   ```

3. After creating a session, you use the `Destination` object to create a MessageProducer or MessageConsumer object.

   ```
   QueueSender qs = qSession.createSender((javax.jms.Queue)myQueue);
   ```

The code is shown in Code Example 2-3.

Code Example 2-3 Instantiating a Destination Object

```
com.sun.messaging.Queue myQueue = new com.sun.messaging.Queue();
try {
    myQueue.setProperty("imqDestinationName", "new_queue_name");
} catch (JMSException je) {
}
...
QueueSender qs = qSession.createSender((javax.jms.Queue)myQueue);
...
```

Overriding Administered Object Settings

As with any Java application, you can start Message Queue clients using the command-line to specify system properties. This mechanism can also be used to override attribute values of Message Queue administered objects used in application code. You can override the configuration of Message Queue administered objects that are accessed through a JNDI lookup and Message Queue administered objects that are instantiated and configured using `setProperty()` methods in application code.

To override administered object settings, use the following command line syntax:

`java [[-D`*attribute*`=`*value* `]...]` *clientAppName*

where `attribute` corresponds to any of the `ConnectionFactory` administered object attributes documented in the *Message Queue Administration Guide*.

For example, if you want a client to connect to a different broker than that specified for a `ConnectionFactory` administered object accessed in the application code, you can start up the client using command line overrides that change the values of the brokers specified for the `imqAdminList` attribute.

It is also possible to set system properties within application code using the `System.setProperty()` method. This method will override attribute values of Message Queue administered objects in the same way that command line options do.

If an administered object has been set as read-only, however, the values of its attributes cannot be changed using either command-line overrides or the `System.setProperty()` method. Any such overrides will simply be ignored.

Chapter 3

Message Queue Clients: Design and Features

This chapter addresses architectural and configuration issues that depend upon Message Queue's implementation of the Java Message Specification. It covers the following topics:

- "Client Design Considerations" on page 45
- "Managing Client Threads" on page 52
- "Managing Memory and Resources" on page 54
- "Programming Issues for Message Consumers" on page 61
- "Factors Affecting Performance" on page 63
- "Client Connection Failover (Auto-Reconnect)" on page 70
- "Custom Client Acknowledgment" on page 75
- "Communicating with C Clients" on page 80

Client Design Considerations

The choices you make in designing a JMS client relate to portability, to allocating work between connections and sessions, to reliability and performance, to resource use, and to ease of administration. This section discusses basic issues that you need to address in client design. It covers the following topics:

- "Developing Portable Clients" on page 46
- "Choosing Programming Domains" on page 46
- "Connections and Sessions" on page 48

- "Producers and Consumers" on page 49
- "Balancing Reliability and Performance" on page 52

Developing Portable Clients

The Java Messaging Specification was developed to abstract access to message-oriented middleware systems (MOMs). A client that writes JMS code should be portable to any provider that implements this specification. If code portability is important to you, be sure that you do the following in developing clients:

- Make sure your code does not depend on extensions or features that are specific to Message Queue.
- Look up, using JNDI, (rather than instantiate) administered objects for connection factories and destinations.

  Administered objects encapsulate provider-specific implementation and configuration information. Besides allowing for portability, administered objects also make it much easier to share connection factories between applications and to tune a JMS application for performance and resource use. So, even if portability is not important to you, you might still want to leave the work of creating and configuring these objects to an administrator. See Chapter 2 on page 37 for more information.

Choosing Programming Domains

As described in the *Message Queue Technical Overview*, JMS supports two distinct message delivery models: point-to-point and publish/subscribe. These two message delivery models can be handled using different API objects—with slightly different semantics—representing different programming domains, as shown in Table 3-1, or they can be handled by base (unified domain) types.

Table 3-1 JMS Programming Objects

Unified Domain	Point-to-Point Domain	Publish/Subscribe Domain
Destination (Queue or Topic)[1]	Queue	Topic
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber

1. Depending on programming approach, you might specify a particular destination type.

Using the point-to-point or publish/subscribe domains offers the advantage of a clean API that prevents certain types of programming errors; for example, creating a durable subscriber for a queue destination. However, the non-unified domains have the disadvantage that you cannot combine point-to-point and publish/subscribe operations in the same transaction or in the same session. If you need to do that, you should choose the unified domain API.

The JMS 1.1 specification continues to support the separate JMS 1.02 programming domains. (The example applications included with the Message Queue product as well as the code examples provided in this book all use the separate JMS 1.02 programming domains.) You can choose the API that best suits your needs. The only exception are those developers needing to write clients for the Sun Java System Application Server 7 environment, as explained in the following note.

NOTE Developers of applications that run in the Sun Java System Application Server 7 environment are limited to using the JMS 1.0.2 API. This is because Sun Java System Application Server 7 complies with the J2EE 1.3 specification, which supports only JMS 1.0.2. Any JMS messaging performed in servlets and EJBs—including message-driven beans must be based on the domain-specific JMS APIs and cannot use the JMS 1.1 unified domain APIs. Developers of J2EE applications that will run in J2EE 1.4-compliant servers can, however, use the simpler JMS 1.1 APIs.

Connections and Sessions

A connection is a relatively heavy-weight object because of the authentication and communication setup that must be done when a connection is created. For this reason, it's a good idea to use as few connections as possible. The real allocation of work occurs in sessions, which are light-weight, single-threaded contexts for producing and consuming messages. When you are thinking about structuring your client, it is best to think of the work that is done at the session level.

A session

- Is a factory for its message producers and consumers.
- Supplies provider-optimized message factories.
- Supports a single series of transactions that combine work spanning its producers and consumers into atomic units.
- Defines a serial order for the messages it consumes and the messages it produces.
- Retains messages until they have been acknowledged.
- Serializes execution of message listeners registered with its message consumers.

The requirement that sessions be operated on by a single thread at a time places some restrictions on the combination of producers and consumers that can use the same session. In particular, if a session has an asynchronous consumer, it may not have any other synchronous consumers. For a discussion of the connection and session's use of threads, see "Managing Client Threads" on page 52. With the exception of these restrictions, let the needs of your application determine the number of sessions, producers, and consumers.

Producers and Consumers

Aside from the reliability your client requires, the design decisions that relate to producers and consumers include the following:

- Do you want to use a point-to-point or a publish/subscribe domain?

  There are some interesting permutations here. There are times when you would want to use publish/subscribe even when you have only one subscriber. On the other hand, performance considerations might make the point-to-point model more efficient than the publish/subscribe model, when the work of sorting messages between subscribers is too costly. Sometimes You cannot make these decisions cannot in the abstract, but must actually develop and test different prototypes.

- Are you using an asynchronous message consumer that does not receive nessages often or a producer that is seldom used?

  Let the administrator know how to set the ping interval, so that your client gets an exception if the connection should fail. For more information see "Using the Client Runtime Ping Feature" on page 61.

- Are you using a synchronous consumer in a distributed application?

  You might need to allow a small time interval between connecting and calling the `receiveNoWait()` method in order not to miss a pending message. For more information, see "Synchronous Consumption in Distributed Applications" on page 62.

- Do you need message compression?

  Benefits vary with the size and format of messages, the number of consumers, network bandwidth, and CPU performance; and benefits are not guaranteed. For a more detailed discussion, see "Message Compression" on page 55.

Assigning Client Identifiers

A connection can have a *client identifier.* This identifier is used to associate a JMS client's connection to a message service, with state information maintained by the message service for that client. The JMS provider must ensure that a client identifier is unique, and applies to only one connection at a time. Currently, client identifiers are used to maintain state for durable subscribers. In defining a client identifier, you can use a special variable substitution syntax that allows multiple connections to be created from a single `ConnectionFactory` object using different user name parameters to generate unique client identifiers. These connections can be used by multiple durable subscribers without naming conflicts or lack of security.

Message Queue allows client identifiers to be set in one of two ways:

- Programmatically: You use the `setClientID` method of the `Connection` object. If you use this method, you must set the client id before you use the connection. Once the connection is used, the client identifier cannot be set or reset.
- Administratively: The administrator specifies the client ID when creating the connection factory administrative object.

For more information about client identifiers and how these work with client authentication, see the *Message Queue Administration Guide*.

Message Order and Priority

In general, all messages sent to a destination by a single session are guaranteed to be delivered to a consumer in the order they were sent. However, if they are assigned different priorities, a messaging system will attempt to deliver higher priority messages first.

Beyond this, the ordering of messages consumed by a client can have only a rough relationship to the order in which they were produced. This is because the delivery of messages to a number of destinations and the delivery from those destinations can depend on a number of issues that affect timing, such as the order in which the messages are sent, the sessions from which they are sent, whether the messages are persistent, the lifetime of the messages, the priority of the messages, the message delivery policy of queue destinations (see the *Message Queue Administration Guide*), and message service availability.

Using Selectors Efficiently

The use of selectors can have a significant impact on the performance of your application. It's difficult to put an exact cost on the expense of using selectors since it varies with the complexity of the selector expression, but the more you can do to eliminate or simplify selectors the better.

One way to eliminate (or simplify) selectors is to use multiple destinations to sort messages. This has the additional benefit of spreading the message load over more than one producer, which can improve the scalability of your application. For those cases when it is not possible to do that, here are some techniques that you can use to improve the performance of your application when using selectors:

- Have consumers share selectors. As of version 3.5 of Message Queue, message consumers with identical selectors “share” that selector in `imqbrokerd` which can significantly improve performance. So if there is a way to structure your application to have some selector sharing, consider doing so.

- Use `IN` instead of multiple string comparisons. For example, the following expression:

  ```
  color IN ('red', 'green', 'white')
  ```

  is much more efficient than this expression

  ```
  color = 'red' OR color = 'green' OR color = 'white'
  ```

  especially if the above expression usually evaluates to false.

- Use `BETWEEN` instead of multiple integer comparisons. For example:

  ```
  size BETWEEN 6 AND 10
  ```

  is generally more efficient than

  ```
  size >= 6 AND size  <= 10
  ```

  especially if the above expression usually evaluates to true.

- Order the selector expression so that Message Queue can determine its evaluation as soon as possible. (Evaluation proceeds from left to right.) This can easily double or triple performance when using selectors, depending on the complexity of the expression.

  - If you have two expressions joined by an `OR`, put the expression that is most likely to evaluate to `TRUE` first.
  - If you have two expressions joined by an `AND`, put the expression that is most likely to evaluate to `FALSE` first.

  For example, if `size` is usually greater than 6, but color is rarely `red` you'd want the order of an `OR` expression to be:

  ```
  size > 6 OR color = 'red'
  ```

  If you are using `AND`:

  ```
  color = 'red' AND size > 6
  ```

Balancing Reliability and Performance

Reliable messaging is implemented in a variety of ways: through the use of persistent messages, acknowledgements or transactions, durable subscriptions, and connection failover.

In general, the more reliable the delivery of messages, the more overhead and bandwidth are required to achieve it. The trade-off between reliability and performance is a significant design consideration. You can maximize *performance* and throughput by choosing to produce and consume non-persistent messages. On the other hand, you can maximize *reliability* by producing and consuming persistent messages in a transaction using a transacted session. For a detailed discussion of design options and their impact on performance, see "Factors Affecting Performance" on page 63.

Managing Client Threads

Using client threads effectively requires that you balance performance, throughput, and resource needs. To do this, you need to understand JMS restrictions on thread usage, what threads Message Queue allocates for itself, and the architecture of your applications. This section addresses these issues and offers some guidelines for managing client threads.

JMS Threading Restrictions

The Java Messaging Specification mandates that a session not be operated on by more than one thread at a time. This leads to the following restrictions:

- A session may not have an asynchronous consumer and a synchronous consumer.
- A session that has an asynchronous consumer can only produce messages from within the `onMessage()` method (the message listener). The only call that you can make outside the message listener is to close the session.
- A session may include any number of synchronous consumers, any number of producers, and any combination of the two. That is, the single-thread requirement cannot be violated by these combinations. However, performance may suffer.

The system does not enforce the requirement that a session be single threaded. If your client application violates this requirement, you will get a `JMSIllegalState` exception or unexpected results..

Thread Allocation for Connections

When the Message Queue client runtime creates a connection, it creates two threads: one for consuming messages from the socket, and one to manage the flow of messages for the connection. In addition, the client runtime creates a thread for each client session. Thus, at a minimum, for a connection using one session, three threads are created. For a connection using three sessions, five threads are created, and so on.

Managing threads in a JMS application often involves trade-offs between performance and throughput. Weigh the following considerations when dealing with threading issues.

- When you create several asynchronous message consumers in the same session, messages are delivered serially by the session thread to these consumers. Sharing a session among several message consumers might starve some consumers of messages while inundating other consumers. If the message rate across these consumers is high enough to cause an imbalance, you might want to separate the consumers into different sessions. To determine whether message flow is unbalanced, you can monitor destinations to see the rate of messages coming in. See Chapter 4, “Using the Metrics Monitoring API” on page 81.
- You can reduce the number of threads allocated to the client application by using fewer connections and fewer sessions. However, doing this might slow your application’s throughput.
- You might be able to use certain JVM runtime options to improve thread memory usage and performance. For example, if you are running on the Solaris platform, you may be able to run with the same number (or more) threads by using the following `vm` options with the client: Refer to the JDK documentation for details.
    - Use the `Xss128K` option to decrease the memory size of the heap.
    - Use the `xconcurrentIO` option to improve thread performance in the 1. 3 VM.

Managing Memory and Resources

This section describes memory and performance issues that you can manage by increasing JVM heap space and by managing the size of your messages. It covers the following topics:

- "Managing Memory" on page 54
- "Managing Message Size" on page 55
- "Managing the Dead Message Queue" on page 57
- "Managing Physical Destination Limits" on page 61

You can also improve performance by having the administrator set connection factory attributes to meter the message flow over the client-broker connection and to limit the message flow for a consumer. For a detailed explanation, please see the *Message Queue Administration Guide*.

Managing Memory

A client application running in a JVM needs enough memory to accommodate messages that flow in from the network as well as messages the client creates. If your client gets `OutOfMemoryError` errors, chances are that not enough memory was provided to handle the size or the number of messages being consumed or produced.

Your client might need more than the default JVM heap space. On most systems, the default is 64 MB but you will need to check the default values for your system.

Consider the following guidelines:

- Evaluate the normal and peak system memory footprints when sizing heap space.
- You can start by doubling the heap size using a command like the following:

  ```
  java -Xmx128m MyClass
  ```

- The best size for the heap space depends on both the operating system and the JDK release. Check the JDK documentation for restrictions.
- The size of the VM's memory allocation pool must be less than or equal to the amount of virtual memory that is available on the system.

Managing Message Size

In general, for better manageability, you can break large messages into smaller parts, and use sequencing to ensure that the partial messages sent are concatenated properly. You can also use a Message Queue JMS feature to compress the body of a message. This section describes the programming interface that allows you to compress messages and to compare the size of compressed and uncompressed messages.

Message compression and decompression is handled entirely by the client runtime, without involving the broker. Therefore, applications can use this feature with a pervious version of the broker, but they must use version 3.6 or later of the Message Queue client runtime library.

Message Compression

You can use the `Message.setBooleanProperty()` method to specify that the body of a message be compressed. If the `JMS_SUN_COMPRESS` property is set to `true`, the client runtime, will compress the body of the message being sent. This happens after the producer's send method is called and before the send method returns to the caller. The compressed message is automatically decompressed by the client runtime before the message is delivered to the message consumer.

For example, the following call specifies that a message be compressed:

```
MyMessage.setBooleanProperty("JMS_SUN_COMPRESS",true);
```

Compression only affects the message body; the message header and properties are not compressed.

Two read-only JMS message properties are set by the client runtime after a message is sent.

Applications can test the properties (`JMS_SUN_UNCOMPRESSED_SIZE` and `JMS_SUN_COMPRESSED_SIZE`) after a send returns to determine whether compression is advantageous. That is, applications wanting to use this feature, do not have to explicitly receive a compressed and uncompressed version of the message to determine whether compression is desired.

If the consumer of a compressed message wants to resend the message in an uncompressed form, it should call the `Message.clearProperties()` to clear the `JMS_SUN_COMPRESS` property. Otherwise, the message will be compressed before it is sent to its next destination.

Advantages and Limitations of Compression

Although message compression has been added to improve performance, such benefit is not guaranteed. Benefits vary with the size and format of messages, the number of consumers, network bandwidth, and CPU performance. For example, the cost of compression and decompression might be higher than the time saved in sending and receiving a compressed message. This is especially true when sending small messages in a high-speed network. On the other hand, applications that publish large messages to many consumers or who publish in a slow network environment, might improve system performance by compressing messages.

Depending on the message body type, compression may also provide minimal or no benefit. An application client can use the `JMS_SUN_UNCOMPRESSED_SIZE` and `JMS_SUN_COMPRESSED_SIZE` properties to determine the benefit of compression for different message types.

Message consumers deployed with client runtime libraries that precede version 3.6 cannot handle compressed messages. Clients wishing to send compressed messages must make sure that consumers are compatible. C clients cannot currently consume compressed messages.

Compression Examples

Code Example 3-1 shows how you set and send a compressed message:

Code Example 3-1 Sending a Compressed Message

```
//topicSession and myTopic are assumed to have been created
topicPublisher publisher = topicSession.createPublisher(myTopic);
BytesMessage bytesMessage=topicSession.createBytesMessage();

//byteArray is assumed to have been created
bytesMessage.writeBytes(byteArray);

//instruct the client runtime to compress this message
bytesMessage.setBooleanProperty("JMS_SUN_COMPRESS", true);

//publish message to the myTopic destination
publisher.publish(bytesMessage);
```

Code Example 3-2 shows how you examine compressed and uncompressed message body size. The `bytesMessage` was created as in Code Example 3-1:

Code Example 3-2 Comparing Size of Compressed and Uncompressed Messages

```
//get uncompressed body size
int uncompressed=bytesMessage.getIntProperty("JMS_SUN_UNCOMPRESSED_SIZE");

//get compressed body size
int compressed=bytesMessage.getIntProperty("JMS_SUN_COMPRESSED_SIZE");
```

Managing the Dead Message Queue

When a message is deemed undeliverable, it is automatically placed on a special queue called the dead message queue. A message placed on this queue retains all of its original headers (including its original destination) and information is added to the message's properties to explain why it became a dead message. An administrator or a developer can access this queue, remove a message, and determine why it was placed on the queue.

- For an introduction to dead messages and the dead message queue, see the *Message Queue Technical Overview*.
- For a description of the destination properties and of the broker properties that control the system's use of the dead message queue, see the *Message Queue Administration Guide*.

This section describes the message properties that you can set or examine programmatically to determine the following:

- Whether a dead message can be sent to the dead message queue.
- Whether the broker should log information when a message is destroyed or moved to the dead message queue.
- Whether the body of the message should also be stored when the message is placed on the dead message queue.
- Why the message was placed on the dead message queue and any ancillary information.

Message Queue 3.6 clients can set properties related to the dead message queue on messages and send those messages to clients compiled against earlier versions. However clients receiving such messages cannot examine these properties without recompiling against 3.6 libraries.

The dead message queue is automatically created by the broker and called `mq.sys.dmq`. You can use the message monitoring API, described in Chapter 4 on page 81, to determine whether that queue is growing, to examine messages on that queue, and so on.

You can set the properties described in Table 3-2 for any message to control how the broker should handle that message if it deems it to be undeliverable. Note that these message properties are needed only to override destination, or broker-based behavior.

Table 3-2 Message Properties Relating to Dead Message Queue

Property	Type	Description
`JMS_SUN_PRESERVE_UNDELIVERED`	Boolean	For a dead message, the default value of unset, specifies that the message should be handled as specified by the `useDMQ` property of the destination to which the message was sent. A value of `true` overrides the setting of the `useDMQ` property and sends the dead message to the dead message queue,. A value of `false` overrides the setting of the `useDMQ` property and prevents the dead message from being placed in the dead message queue.
`JMS_SUN_LOG_DEAD_MESSAGES`	Boolean	The default value of unset, will behave as specified by the broker configuration property `imq.destination.logDeadMsgs`. A value of `true` overrides the setting of the `imq.destination.logDeadMsgs` broker property and specifies that the broker should log the action of removing a message or moving it to the dead message queue. A value of `false` overrides the setting of the `imq.destination.logDeadMsgs` broker property and specifies that the broker should not log these actions.

Table 3-2 Message Properties Relating to Dead Message Queue *(Continued)*

Property	Type	Description
JMS_SUN_TRUNCATE_MSG_BODY	Boolean	The default value of unset, will behave as specified by the broker property imq.destination.DMQ.truncateBody. A value of true overrides the setting of the imq.destination.DMQ.truncateBody property and specifies that the body of the message should be discarded when the message is placed in the dead message queue. A value of false overrides the setting of the imq.destination.DMQ.truncateBody property and specifies that the body of the message should be stored along with the message header and properties when the message is placed in the dead message queue.

The properties described in Table 3-3 are set by the broker for a message placed in the dead message queue. You can examine the properties for the message to retrieve information about why the message was placed on the queue and to gather other information about the message and about the context within which this action was taken.

Table 3-3 Dead Message Properties

Property	Type	Description
JMSXDeliveryCount	Integer	Specifies the most number of times the message was delivered to a given consumer. This value is set only for ERROR or UNDELIVERABLE messages.
JMS_SUN_DMQ_UNDELIVERED_TIMESTAMP	Long	Specifies the time (in milliseconds) when the message was placed on the dead message queue.

Table 3-3 Dead Message Properties *(Continued)*

Property	Type	Description
JMS_SUN_DMQ_UNDELIVERED_REASON	String	Specifies one of the following values to indicate the reason why the message was placed on the dead message queue: OLDEST LOW_PRIORITY EXPIRED UNDELIVERABLE ERROR If the message was marked dead for multiple reasons, for example it was undeliverable and expired, only one reason will be specified by this property. The ERROR reason indicates that an internal error made it impossible to process the message. This is an extremely unusual condition, and the sender should just resend the message.
JMS_SUN_DMQ_PRODUCING_BROKER	String	For message traffic in broker clusters: specifies the broker name and port number of the broker that placed the message on the dead message queue. A null value indicates that it was the local broker.
JMS_SUN_DMQ_UNDELIVERED_EXCEPTION	String	Specifies the name of the exception (if the message was dead because of an exception) on either the client or the broker.
JMS_SUN_DMQ_UNDELIVERED_COMMENT	String	An optional comment provided when the message is marked dead.
JMS_SUN_DMQ_BODY_TRUNCATED	Boolean	A value of true indicates that the message body was not stored. A value of false indicates that the message body was stored.

Managing Physical Destination Limits

When creating a topic or queue destination, the administrator can specify how the broker should behave when certain memory limits are reached. Specifically, when the number of unconsumed messages reaching a physical destination exceeds the number specified with the `maxNumMsgs` property or when the total amount of memory allowed for unconsumed messages exceeds the number specified with the `maxTotalMsgBytes` property, the broker takes one of the following actions, depending on the setting of the `limitBehavior` property:

- slows message producers (`FLOW_CONTROL`)
- throws out the oldest message in memory (`REMOVE_OLDEST`)
- throws out the lowest priority message in memory (`REMOVE_LOW_PRIORITY`)
- rejects the newest messages (`REJECT_NEWEST`)

If the default value `REJECT_NEWEST` is specified for the `limitBehavior` property, the broker throws out the newest messages received when memory limits are exceeded. If the message discarded is a persistent message, the producing client gets an exception which should be handled by resending the message later.

If any of the other values is selected for the `limitBehavior` property or if the message is not persistent, the application client is not notified if a message is discarded. Application clients should let the administrator know how they prefer this property to be set for best performance and reliability.

Programming Issues for Message Consumers

This section describes two problems that consumers might need to manage: the undetected loss of a connection, or the loss of a message for distributed synchronous consumers.

Using the Client Runtime Ping Feature

Message Queue defines a connection factory attribute for a *ping interval*. This attribute specifies the interval at which the client runtime should check the client's connection to the broker. The ping feature is especially useful to Message Queue clients that exclusively receive messages and might therefore not be aware that the absence of messages is due to a connection failure. This feature could also be useful to producers who don't send messages frequently and who would want notification that a connection they're planning to use is not available.

The connection factory attribute used to specify this interval is called `imqPingInterval`. Its default value is 30 seconds. A value of -1 or 0, specifies that the client runtime should not check the client connection.

Developers should set (or have the administrator set) ping intervals that are slightly more frequent than they need to send or receive messages, to allow time to recover the connection in case the ping discovers a connection failure. Note also that the ping may not occur at the exact time specified by the value you supply for `interval`; the underlying operating system's use of i/o buffers may affect the amount of time needed to detect a connection failure and trigger an exception.

A failed ping operation results in a `JMSException` on the subsequent method call that uses the connection. If an exception listener is registered on the connection, it will be called when a ping operation fails.

Preventing Message Loss for Synchronous Consumers

It is always possible that a message can be lost for synchronous consumers in a session using `AUTO_ACKNOWLEDGE` mode if the provider fails. To prevent this possibility, you should either use a transacted session or a session in `CLIENT_ACKNOWLEDGE` mode.

Synchronous Consumption in Distributed Applications

Because distributed applications involve greater processing time, such an application might not behave as expected if it were run locally. For example, calling the `receiveNoWait` method for a synchronous consumer might return `null` even when there is a message available to be retrieved.

If a client connects to the broker and immediately calls the `receiveNoWait` method, it is possible that the message queued for the consuming client is in the process of being transmitted from the broker to the client. The client runtime has no knowledge of what is on the broker, so when it sees that there is no message available on the client's internal queue, it returns with a `null`, indicating no message.

You can avoid this problem by having your client do either of the following:

- Use one of the synchronous receive methods that specifies a time-out interval
- Use a queue browser to check the queue before calling the `receiveNoWait` method

Factors Affecting Performance

Application design decisions can have a significant effect on overall messaging performance. The most important factors affecting performance are those that impact the reliability of message delivery; among these are the following:

- Delivery Mode (Persistent/Non-persistent)
- Use of Transactions
- Acknowledgement Mode
- Durable vs. Non-Durable Subscriptions

Other application design factors impacting performance include the following:

- Use of Selectors (Message Filtering)
- Message Size
- Message Body Type

The sections that follow describe the impact of each of these factors on messaging performance. As a general rule, there is a trade-off between performance and reliability: factors that increase reliability tend to decrease performance.

Table 3-4 shows how application design factors affect messaging performance. The table shows two scenarios—a high reliability, low performance scenario and a high performance, low reliability scenario—and the choice of application design factors that characterizes each. Between these extremes, there are many choices and trade-offs that affect both reliability and performance.

Table 3-4 Comparison of High Reliability and High Performance Scenarios

Application Design Factor	High Reliability Low Performance Scenario	High Performance Low Reliability Scenario
Delivery mode	Persistent messages	Non-persistent messages
Use of transactions	Transacted sessions	No transactions

Table 3-4 Comparison of High Reliability and High Performance Scenarios *(Continued)*

Application Design Factor	High Reliability Low Performance Scenario	High Performance Low Reliability Scenario
Acknowledgement mode	`AUTO_ACKNOWLEDGE` or `CLIENT_ACKNOWLEDGE`	`DUPS_OK_ACKNOWLEDGE` `NO_ACKNOWLEDGE`
Durable/non-durable subscriptions	Durable subscriptions	Non-durable subscriptions
Use of selectors	Message filtering	No message filtering
Message size	Small messages	Large messages
Message body type	Complex body types	Simple body types

NOTE In the graphs that follow, performance data was generated on a two-CPU, 1002 Mhz, Solaris 8 system, using file-based persistence. The performance test first warmed up the Message Queue broker, allowing the Just-In-Time compiler to optimize the system and the persistent database to be primed.

Once the broker was warmed up, a single producer and a single consumer were created, and messages were produced for 30 seconds. The time required for the consumer to receive all produced messages was recorded, and a throughput rate (messages per second) was calculated. This scenario was repeated for different combinations of the application design factors shown in Table 3-4.

Delivery Mode (Persistent/Non-persistent)

Persistent messages guarantee message delivery in case of message server failure. The broker stores these message in a persistent store until all intended consumers acknowledge they have consumed the message.

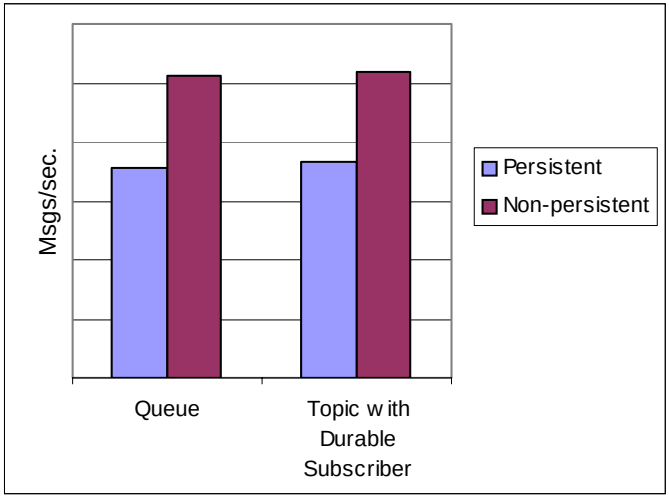
Broker processing of persistent messages is slower than for non-persistent messages for the following reasons:

- A broker must reliably store a persistent message so that it will not be lost should the broker fail.

- The broker must confirm receipt of each persistent message it receives. Delivery to the broker is guaranteed once the method producing the message returns without an exception.
- Depending on the client acknowledgment mode, the broker might need to confirm a consuming client's acknowledgement of a persistent message.

The differences in performance for persistent and non-persistent modes can be significant--about 25% faster for non-persistent messages. Figure 3-1 compares throughput for persistent and non-persistent messages in two reliable delivery cases: 10k-sized messages delivered both to a queue and to a topic with durable subscriptions. Both cases use the `AUTO_ACKNOWLEDGE` acknowledgement mode.

Figure 3-1 Performance Impact of Delivery Modes



Use of Transactions

A transaction guarantees that all messages produced in a transacted session and all messages consumed in a transacted session will be either processed or not processed (rolled back) as a unit. Message Queue supports both local and distributed transactions.

A message produced or acknowledged in a transacted session is slower than in a non-transacted session for the following reasons:

- Additional information must be stored with each produced message.

- In some situations, messages in a transaction are stored when normally they would not be. For example, a persistent message delivered to a topic destination with no subscriptions would normally be deleted, however, at the time the transaction is begun, information about subscriptions is not available.
- Information on the consumption and acknowledgement of messages within a transaction must be stored and processed when the transaction is committed.

Acknowledgement Mode

Other than using transactions, you can ensure reliable delivery by having the client acknowledge receiving a message. If a session is closed without the client acknowledging the message or if the message server fails before the acknowledgment is processed, the broker redelivers that message, setting a `JMSRedelivered` flag.

For a non-transacted session, the client can choose one of three acknowledgement modes, each of which has its own performance characteristics:

- `AUTO_ACKNOWLEDGE`. The system automatically acknowledges a message once the consumer has processed it. This mode guarantees at most one redelivered message after a provider failure.
- `CLIENT_ACKNOWLEDGE`. The application controls the point at which messages are acknowledged. All messages processed in that session since the previous acknowledgement are acknowledged. If the message server fails while processing a set of acknowledgments, one or more messages in that group might be redelivered.

  (Using `CLIENT_ACKNOWLEDGE` mode is similar to using transactions, except there is no guarantee that all acknowledgments will be processed together if a provider fails during processing.)
- `DUPS_OK_ACKNOWLEDGE`. This mode instructs the system to acknowledge messages in a lazy manner. Multiple messages can be redelivered after a provider failure.
- `NO_ACKNOWLEDGE` In this mode, the broker considers a message acknowledged as soon as it has been written to the client. The broker does not wait for an acknowledgement from the receiving client. This mode is best used by typic subscribers who are not worried about reliability.

Performance is impacted by acknowledgement mode for the following reasons:

- Extra control messages between broker and client are required in `AUTO_ACKNOWLEDGE` and `CLIENT_ACKNOWLEDGE` modes. The additional control messages add processing overhead and can interfere with JMS payload messages, causing processing delays.
- In `AUTO_ACKNOWLEDGE` and `CLIENT_ACKNOWLEDGE` modes, the client must wait until the broker confirms that it has processed the client's acknowledgment before the client can consume more messages. (This broker confirmation guarantees that the broker will not inadvertently redeliver these messages.)
- The Message Queue persistent store must be updated with the acknowledgement information for all persistent messages received by consumers, thereby decreasing performance.

Durable vs. Non-Durable Subscriptions

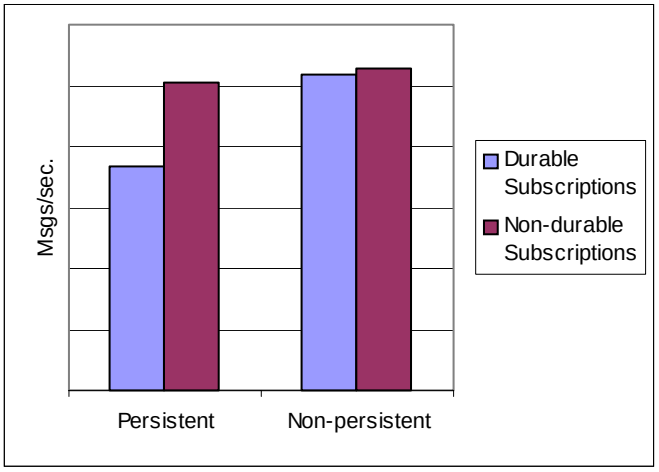
Subscribers to a topic destination have either durable and non-durable subscriptions. Durable subscriptions provide increased reliability at the cost of slower throughput for the following reasons:

- The Message Queue message server must persistently store the list of messages assigned to each durable subscription so that should a message server fail, the list is available after recovery.
- Persistent messages for durable subscriptions are stored persistently, so that should a message server fail, the messages can still be delivered after recovery, when the corresponding consumer becomes active. By contrast, persistent messages for non-durable subscriptions are not stored persistently (should a message server fail, the corresponding consumer connection is lost and the message would never be delivered).

Figure 3-2 compares throughput for topic destinations with durable and non-durable subscriptions in two cases: persistent and non-persistent 10k-sized messages. Both cases use `AUTO_ACKNOWLEDGE` acknowledgement mode.

You can see from Figure 3-2 that using durable subscriptions affects performance only in the case of persistent messages; this is because persistent messages are only stored persistently for durable subscriptions, as explained above.

Figure 3-2 Performance Impact of Subscription Types



Use of Selectors (Message Filtering)

Application developers can have the messaging provider sort messages according to criteria specified in the message selector associated with a consumer and deliver to that consumer only those messages whose property value matches the message selector. For example, if an application creates a subscriber to the topic `WidgetOrders` and specifies the expression `NumberOfOrders >1000` for the message selector, messages with a `NumberOfOrders` property value of `1001` or more are delivered to that subscriber.

Creating consumers with selectors lowers performance (as compared to using multiple destinations) because additional processing is required to handle each message. When a selector is used, it must be parsed so that it can be matched against future messages. Additionally, the message properties of each message must be retrieved and compared against the selector as each message is routed. However, using selectors provides more flexibility in a messaging application and may lower resource requirements at the expense of speed.

Message Size

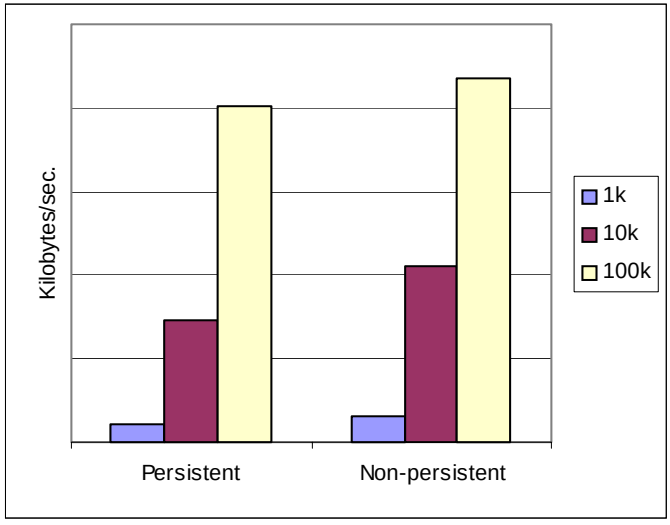
Message size affects performance because more data must be passed from producing client to broker and from broker to consuming client, and because for persistent messages a larger message must be stored.

However, by batching smaller messages into a single message, the routing and processing of individual messages can be minimized, providing an overall performance gain. In this case, information about the state of individual messages is lost.

Figure 3-3 compares throughput in kilobytes per second for 1k, 10k, and 100k-sized messages for persistent and non-persistent messages. All messages are sent to a queue destination and use `AUTO_ACKNOWLEDGE` acknowledgement mode.

Figure 3-3 shows that in both cases there is less overhead in delivering larger messages compared to smaller messages. You can also see that the almost 50% performance gain of non-persistent messages over persistent messages shown for 1k and 10k-sized messages is not maintained for 100k-sized messages, probably because network bandwidth has become the bottleneck in message throughput for that case.

Figure 3-3 Performance Impact of a Message Size



Message Body Type

JMS supports five message body types, shown below roughly in the order of complexity:

- BytesMessage: Contains a set of bytes in a format determined by the application
- TextMessage: Is a simple java.lang.String

- StreamMessage: Contains a stream of Java primitive values
- MapMessage: Contains a set of name-and-value pairs
- ObjectMessage: Contains a Java serialized object

While, in general, the message type is dictated by the needs of an application, the more complicated types (MapMessage and ObjectMessage) carry a performance cost—the expense of serializing and deserializing the data. The performance cost depends on how simple or how complicated the data is.

Client Connection Failover (Auto-Reconnect)

Message Queue supports client connection failover. A failed connection can be automatically restored not only to the original broker, but to a different broker in a broker cluster. There are circumstances under which the client-side state cannot be restored on any broker during an automatic reconnection attempt; for example, when the client uses transacted sessions or temporary destinations. At such times the auto-reconnect will not take place, and the connection exception handler is called instead. In this case the application code has to catch the exception, reconnect, and restore state.

This section explains how automatic reconnection is enabled, how the broker behaves during a reconnect, how automatic reconnection impacts producers and consumers. Reconnection limitations are also discussed and some examples are provided. For additional information about this feature, please see the *Message Queue Administration Guide*.

Enabling Auto-Reconnect

The developer or the administrator can enable automatic reconnection by setting the connection factory `imqReconnectEnabled` attribute to `true`. The connection factory administered object must also be configured to specify the following:

- **A list of message-service addresses** (using the `imqAddressList` attribute). When the client runtime needs to establish or re-establish a connection to a message service, it attempts to connect to the brokers in the list until it finds (or fails to find) an available broker. If you specify only a single broker instance on the `imqAddressList` attribute, the configuration won't support recovery from hardware failure.

  When you specify more than one broker, you can decide whether to use parallel brokers or a broker cluster. In a parallel configuration, there is no communication between brokers, while in a broker cluster, the brokers interact to distribute message delivery loads. (Refer to the *Message Queue Administration Guide* for more information on broker clusters.)

  - To enable parallel-broker reconnection, set the `imqReconnectListBehavior` attribute to `PRIORITY`. Typically, you would specify no more than a pair of brokers for this type of reconnection. This way, the messages are published to one broker, and all clients fail over together from the first broker to the second.
  - To enable clustered-broker reconnection, set the `imqReconnectListBehavior` attribute to `RANDOM`. This way, the client runtime randomizes connection attempts across the list, and client connections are distributed evenly across the broker cluster.

    Each broker in a cluster uses its own separate persistent store (which means that undelivered persistent messages are unavailable until a failed broker is back online). If one broker crashes, its client connections are re-established on other brokers.

- **The number of iterations to be made over the list of brokers** when attempting to create a connection or to reconnect, using the `imqAddressListIterations` attribute.
- **The number of attempts to reconnect to a broker** if the first connection fails, using the `imqReconnectAttempts` attribute.
- **The interval, in milliseconds, between reconnect attempts**, using the `imqReconnectInterval` attribute.

Auto-Reconnect Behaviors

A broker treats an automatic reconnection as it would a new connection. When an original connection is lost, all the resources associated with that connection are released. For example, in a broker cluster, as soon as one broker fails, the other brokers assume that the client connections associated with the failed broker are gone. After auto-reconnect takes place, the client connections are re-created from scratch.

Sometimes the client-side state cannot be fully restored by auto-reconnect. Perhaps a resource that the client needs cannot be re-created. In this case, the client runtime calls the client's connection exception handler and the client must explicitly reconnect and restore state.

If the client is automatically-reconnected to a different broker instance, persistent messages and other state information held by the failed or disconnected broker can be lost. The messages held by the original broker, once it is restored, might be delivered out of order. This is because broker instances in a cluster do not use a shared, highly available persistent store.

A transacted session is the most reliable method of ensuring that a message isn't lost if you are careful in coding the transaction. If auto-reconnect happens in the middle of a transaction, then the broker loses the information, the client runtime throws an exception when the transaction is committed, and the transaction is rolled back. At that point, you must make sure that the client restarts the whole transaction. (This is especially important when you use a broker cluster.)

When auto-reconnect happens in a `CLIENT_ACKNOWLEDGE` session, the client runtime throws a `JMSException` and the acknowledgement of any set of messages must be rolled back. Therefore, if you get a `JMSException` message in such a session, call `session.recover`.

Automatic reconnection affects producers and consumers differently:

- During reconnection, producers cannot send messages. The production of messages (of any operation that involves communication with the message server) is blocked until the connection is re-established.
- For consumers, automatic reconnection is supported for all client acknowledgement modes. After a connection is re-established, the broker will redeliver all unacknowledged messages it had previously delivered, marking them with a `Redeliver` flag. The client can examine this flag to determine whether any message has already been consumed (but not yet acknowledged).

In the case of non-durable subscribers, some messages might be lost because the message server does not hold their messages once their connections have been closed. Any messages produced for non-durable subscribers while the connection is down cannot be delivered when the connections is re-established.

Auto-Reconnect Limitations

Notice the following points when using the auto-reconnect feature:

- Messages might be redelivered to a consumer after auto-reconnect takes place. In an `AUTO_ACKNOWLEDGE` session, you will get no more than one redelivered message. In other session types, all unacknowledged persistent messages are redlivered.
- While the client runtime is trying to reconnect, any messages sent by the broker to non-durable topic consumers are lost.
- Any messages that are in queue destinations and that are unacknowledged when a connection fails are redelivered after auto-reconnect. However, in the case of queues delivering to multiple consumers, these messages cannot be guaranteed to be redelivered to the original consumers. That is, as soon as a connection fails, an unacknowledged queue message might be rerouted to other connected consumers.
- In the case of a broker cluster, the failure of the master broker has more implications than the failure of other brokers in the cluster. While the master broker is down, the following operations on any other broker do not succeed:
  - Creating or destroying a new durable subscription.
  - Creating or destroying a new physical destination using the `imqcmd create dst` command.
  - Starting a new broker process. (However, the brokers that are already running continue to function normally even if the master broker goes down.)

  You can configure the master broker to restart automatically using Message Queue broker support for `rc` scripts or the Windows service manager.
- Auto-reconnect doesn't work if the client uses a `ConnectionConsumer` to consume messages. In that case, the client runtime throws an exception.

Auto-Reconnect Configuration Examples

The following examples illustrate how to enable each type of auto-reconnect support.

Single-Broker Auto-Reconnect

Configure your connection-factory object as follows:

Code Example 3-3 Example of Command to Configure a Single Broker

```
imqobjmgr add -t cf -l "cn=myConnectionFactory" \
    -o “imqAddressList=mq://jpgserv/jms” \
    -o “imqReconnect=true” \
    -o “imqReconnectAttempts=10”
```

This command creates a connection-factory object with a single address in the broker address list. If connection fails, the client runtime will try to reconnect with the broker 10 times. If an attempt to reconnect fails, the client runtime will sleep for three seconds (the default value for the `imqReconnectInterval` attribute) before trying again. After 10 unsuccessful attempts, the application will receive a `JMSException`.

You can ensure that the broker starts automatically with at system start-up time. See the *Message Queue Installation Guide* for information on how to configure automatic broker start-up. For example, on the Solaris platform, you can use `/etc/rc.d` scripts.

Parallel Broker Auto-Reconnect

Configure your connection-factory objects as follows:

Code Example 3-4 Example of Command to Configure Parallel Brokers

```
imqobjmgr add -t cf -l "cn=myCF" \
    -o "imqAddressList=myhost1, mqtcp://myhost2:12345/jms" \
    -o "imqReconnect=true" \
    -o "imqReconnectRetries=5"
```

This command creates a connection factory object with two addresses in the broker list. The first address describes a broker instance running on the host `myhost1` with a standard port number (`7676`). The second address describes a `jms` connection service running at a statically configured port number (`12345`).

Clustered-Broker Auto-Reconnect

Configure your connection-factory objects as follows:

Code Example 3-5 Example of Command to Configure a Broker Cluster

```
imqobjmgr add -t cf -l "cn=myConnectionFactory" \
    -o "imqAddressList=mq://myhost1/ssljms, \
        mq://myhost2/ssljms, \
        mq://myhost3/ssljms, \
        mq://myhost4/ssljms” \
    -o "imqReconnect=true" \
    -o "imqReconnectRetries=5" \
    -o "imqAddressListBehavior=RANDOM"
```

This command creates a connection factory object with four addresses in the `imqAddressList`. All the addresses point to `jms` services running on SSL transport on different hosts. Since the `imqAddressListBehavior` attribute is set to `RANDOM`, the client connections that are established using this connection factory object will be distributed randomly among the four brokers in the address list.

This is a clustered broker configuration, so you must configure one of the brokers in the cluster as the master broker. In the connection-factory address list, you can also specify a subset of all the brokers in the cluster.

Custom Client Acknowledgment

Message Queue supports the standard JMS acknowledgement modes (`AUTO_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE, and DUPS_OK_ACKNOWLEDGE`). When you create a session for a consumer, you can specify one of these modes. Your choice will affect whether acknowledgement is done explicitly (by the client application) or implicitly (by the session) and will also affect performance and reliability. This section describes additional options you can use to customize acknowledgment behavior:

- You can customize the JMS `CLIENT_ACKNOWLEDGE` mode to acknowledge one message at a time.
- If performance is key and reliability is not a concern, you can use the proprietary `NO_ACKNOWLEDGE` mode to have the broker consider a message acknowledged as soon as it has been sent to the consuming client.

The following sections explain how you program these options.

Using Client Acknowledge Mode

For more flexibility, Message Queue lets you customize the JMS `CLIENT_ACKNOWLEDGE` mode. In `CLIENT_ACKNOWLEDGE` mode, the client explicitly acknowledges message consumption by invoking the `acknowledge()` method of a message object. The standard behavior of this method is to cause the session to acknowledge all messages that have been consumed by any consumer in the session since the last time the method was invoked. (That is, the session acknowledges the current message and all previously unacknowledged messages, regardless of who consumed them.)

In addition to the standard behavior specified by JMS, Message Queue lets you use the `CLIENT_ACKNOWLEDGE` mode to acknowledge one message at a time.

Observe the following rules when implementing custom client acknowledgement:

- To acknowledge an individual message, call the `acknowledgeThisMessage()` method. To acknowledge all messages consumed so far, call the `acknowledgeUpThroughThisMessage()` method. Both are shown in Code Example 3-6.

Code Example 3-6 Syntax for Acknowledge Methods

```
public interface com.sun.messaging.jms.Message {
     void acknowledgeThisMessage() throws JMSException;
     void acknowledgeUpThroughThisMessage() throws JMSException;
}
```

- When you compile the resulting code, include both `imq.jar` and `jms.jar` in the classpath.

- Don't call `acknowledge()`, `acknowledgeThisMessage()`, or `acknowledgeUpThroughThisMessage()` in any session except one that uses the `CLIENT_ACKNOWLEDGE` mode. Otherwise, the method call is ignored.
- Don't use custom-acknowledgement in transacted sessions. A transacted session defines a specific way to have messages acknowledged.

If a broker fails, any message that was not acknowledged successfully (that is, any message whose acknowledgement ended in a `JMSException`) is held by the broker for delivery to subsequent clients.

Code Example 3-7 demonstrates both types of custom client acknowledgement.

Code Example 3-7 Example of Custom Client Acknowledgement Code

```
...

import javax.jms.*;

... [Look up a connection factory and create a connection.]

     Session session = connection.createSession(false,
         Session.CLIENT_ACKNOWLEDGE);

... [Create a consumer and receive messages.]

    Message message1 = consumer.receive();
    Message message2 = consumer.receive();
    Message message3 = consumer.receive();

... [Process messages.]

... [Acknowledge one individual message.
    Notice that the following acknowledges only message 2.]

    ((com.sun.messaging.jms.Message)message2).acknowledgeThisMessage();

... [Continue. Receive and process more messages.]

    Message message4 = consumer.receive();
    Message message5 = consumer.receive();
    Message message6 = consumer.receive();

... [Acknowledge all messages up through message 4. Notice that this
    acknowledges messages 1, 3, and 4, because message 2 was acknowledged
    earlier.]

    ((com.sun.messaging.jms.Message)message4).
        acknowledgeUpThroughThisMessage();
```

Code Example 3-7 Example of Custom Client Acknowledgement Code *(Continued)*

```
... [Continue. Finally, acknowledge all messages consumed in the session.
    Notice that this acknowledges all remaining consumed messages, that is,
    messages 5 and 6, because this is the standard behavior of the JMS API.]

    message5.acknowledge();
```

Using No Acknowledge Mode

The NO_ACKNOWLEDGE acknowledgement mode is a non-standard extension to the Java Messaging Specification API(JMS). Normally, the broker waits for a client acknowledgement before considering that a message has been acknowledged and discarding it. That acknowledgement must be made programmatically if the client has specified CLIENT_ACKNOWLEDGE or it can be made automatically, by the session, if the client has specified AUTO_ACKNOWLEDGE or DUPS_OK. If a consuming client specifies the NO_ACKNOWLEDGE mode, the broker discards the message as soon as it has sent it to the consuming client. This feature is intended for use by non-durable subscribers consuming non-persistent messages, but it can be used by any consumer.

Using this feature improves performance by reducing protocol traffic and broker work involved in acknowledging a message. This feature can also improve performance for brokers dealing with misbehaving clients who do not acknowledge messages and therefore tie down broker memory resources unnecessarily. Using this mode has no effect on producers.

You use this feature by specifying NO_ACKNOWLEDGE for the acknowledgeMode parameter to the createSession, createQueueSession, or createTopicSession method. The NO_ACKNOWLEDGE mode must be used only with the connection methods defined in the com.sun.messaging.jms package. Note however that the connection itself must be created using the javax.jms package.

The following are sample variable declarations for `connection`, `queueConnection` and `topicConnection`:

```
javax.jms.connection Connection;

javax.jms.queueConnection queueConnection

javax.jms.topicConnection topicConnection
```

The following are sample statements to create different kinds of `NO_ACKNOWLEDGE` sessions:

```
//to create a no ack session

Session noAckSession  =
    ((com.sun.messaging.jms.Connection)connection)
    .createSession(com.sun.messaging.jms.Session.NO_ACKNOWLEDGE);

// to create a no ack topic session

TopicSession noAckTopicSession  =
    ((com.sun.messaging.jms.TopicConnection) topicConnection)
    .createTopicSession(com.sun.messaging.jms.Session.NO_ACKNOWLEDGE);

//to create a no ack queue session

QueueSession noAckQueueSession  =
    ((com.sun.messaging.jms.QueueConnection) queueConnection)
    .createQueueSession(com.sun.messaging.jms.Session.NO_ACKNOWLEDGE);
```

Specifying `NO_ACKNOWLEDGE` for a session results in the following behaviors:

- The client runtime will throw a JMSException if `Session.recover()` is called on a `NO_ACKNOWLEDGE` session.
- The client runtime will ignore a call to the `Message.acknowledge()` method from a consumer belonging to a session with `NO_ACKNOWLEDGE` mode set.
- Messages can be lost. As opposed to the `DUPS_OK` mode, which can result in duplicate messages being sent, this mode bypasses checks and balances built into the system and may result in message loss.

Communicating with C Clients

Message Queue supports C clients as message producers and consumers.

A Java client consuming messages sent by a C client faces only one restriction: a C client cannot be part of a distributed transaction, and therefore a Java client receiving a message from a C client cannot participate in a distributed transaction either.

A Java client producing messages for a consuming C client must be aware of the following differences in the Java and C interfaces because these differences will affect the C client's ability to consume messages: C clients

- Can only consume messages of type `text` and `bytes`.
- Cannot consume messages whose body has been compressed.
- Cannot participate in distributed transactions.
- Cannot receive SOAP messages.

Chapter 4

Using the Metrics Monitoring API

Message Queue provides several ways of obtaining metrics data as a means of monitoring and tuning performance. One of these methods, *message-based monitoring*, allows metrics data to be accessed programmatically and then to be processed in whatever way suits the consuming client. Using this method, a client subscribes to one or more metrics destinations and then consumes and processes messages produced by the broker to those destinations. Message-based monitoring is the most customized solution to metrics gathering, but it does require the effort of writing a consuming client that retrieves and processes metrics messages.

The three methods of obtaining metrics data are described in the *Message Queue Administration Guide*, which also discusses the relative merits of each method and the set of data that is captured by each. Before you decide to used message-based monitoring, you should consult this guide to make sure that you will be able to obtain the information you need using this method.

Message-based monitoring is enabled by the combined efforts of administrators and programmers. The administrator is responsible for configuring the broker so that it produces the messages of interest at a specified interval and that it persists these messages for a set time. The programmer is responsible for selecting the data to be produced and for creating the client that will consume and process the data.

This chapter focuses on the work the programmer must do to implement a message-based monitoring client. It includes the following sections:

- "Monitoring Overview" on page 82.
- "Creating a Metrics-Monitoring Client" on page 84.
- "Format of Metrics Messages" on page 85.
- "Metrics Monitoring Client Code Examples" on page 90.

Monitoring Overview

Message Queue includes an internal client that is enabled by default to produce different types of metrics messages. Production is actually enabled when a client subscribes to a topic destination whose name matches one of the metrics message types. For example, if a client subscribes to the topic `mq.metrics.jvm`, the client receives information about JMV memory usage.

The metrics topic destinations (metric message types) are described in Table 4-1.

Table 4-1 Metrics Topic Destinations

Topic Destination Name	Type of Metrics Messages
`mq.metrics.broker`	Broker metrics: information on connections, message flow, and volume of messages in the broker
`mq.metrics.jvm`	Java Virtual Machine metrics: information on memory usage in the JVM
`mq.metrics.destination_list`	A list of all destinations on the broker, and their types
`mq.metrics.destination.queue.`*destination_name*	Destination metrics for a queue of the specified name. Metrics data includes number of consumers, message flow or volume, disk usage, and more.
`mq.metrics.destination.topic.`*destination_name*	Destination metrics for a topic of the specified name. Metrics data includes number of consumers, message flow or volume, disk usage, and more.

A *metrics message* that is produced to one of the destinations listed in Table 4-1 is a normal JMS message; its header and body are defined to hold the following information:

- The message header has several properties, one that specifies the metrics message type, one that records the time the message was produced (timestamp), and a collection of properties identifying the broker that sent the metric message (broker host, port, and address/URL).
- The message body contains name-value pairs that vary with the message type.

  The section "Format of Metrics Messages" on page 85 provides complete information about the types of metrics messages and their content (name-value pairs).

To receive metrics messages, the consuming client must be subscribed to the destination of interest. Otherwise, consuming a metrics message is exactly the same as consuming any JMS message. The message can be consumed synchronously or asynchronously, and then processed as needed by the client.

Message-based monitoring is concerned solely with obtaining metrics information. It does not include methods that you can call to work with physical destinations, configure or update the broker, or shutdown and restart the broker.

Administrative Tasks

By default the Message Queue metrics-message producing client is enabled to produce non-persistent messages every sixty seconds. The messages are allowed to remain in their respective destinations for 5 minutes before being automatically deleted. To persist metrics messages, to change the interval at which they are produced, or to change their time-to-live interval, the administrator must set the following properties in the `config.properties` file: `imq.metrics.topic.persist`, `imq.metrics.topic.interval`, `imq.metrics.topic.timetolive`.

In addition, the administrator might want to set access controls on the metrics destinations. This restricts access to sensitive metrics data and helps limit the impact of metrics subscriptions on overall performance. For more information about administrative tasks in enabling message-based monitoring and access control, see *Message Queue Administration Guide*.

Implementation Summary

The following task list summarizes the steps required to implement message based monitoring:

1. The developer designs and writes a client that subscribes to one or more metrics destinations.
2. The administrator sets those metrics-related broker properties whose default values are not satisfactory.
3. The administrator sets entries in the `access.control.properties` file to restrict access to metrics information. (Optional)
4. The developer or the administrator starts the metrics monitoring client.

When consumers subscribe to a metrics topic, the topic's physical destination is automatically created. After the metrics topic has been created, the broker's metrics message producer begins to send metrics messages to the appropriate destination.

Creating a Metrics-Monitoring Client

You create a metrics monitoring client in the same way that you would write any JMS client, except that the client must subscribe to one or more special metrics message topic and must be ready to receive and process messages of a specific type and format.

No hierarchical naming scheme is implied in the metrics-message names. You can't use a wildcard character (*) to identify multiple destination names.

A client that monitors broker metrics must perform the following basic tasks:

1. Create a `TopicConnectionFactory` object
2. Create a `TopicConnection` to the Message Queue service
3. Create a `TopicSession`
4. Create a metrics `Topic` destination object
5. Create a `TopicSubscriber`
6. Register as an asynchronous listener to the topic, or invoke the synchronous `receive()` method to wait for incoming metrics messages
7. Process metrics messages that are received

In general, you would use JNDI lookups of administered objects to make your client code provider-independent. However, the metrics-message production is specific to Message Queue, there is no compelling reason to use JNDI lookups. You can simply instantiate these administered objects directly in your client code. This is especially true for a metrics destination for which an administrator would not normally create an administered object.

Notice that the code examples in this chapter instantiate all the relevant administered objects directly.

You can use the following code to extract the type (`String`) or timestamp (`long`) properties in the message header from the message:

```
MapMessage mapMsg;
/*
* mapMsg is the metrics message received
*/
String type = mapMsg.getStringProperty("type");
long timestamp = mapMsg.getLongProperty("timestamp");
```

You use the appropriate get method in the class `javax.jms.MapMessage` to extract the name-value pairs. The get method you use depends on the value type. Three examples follow:

```
long value1 = mapMsg.getLong("numMsgsIn");
long value2 = mapMsg.getLong("numMsgsOut");
int value3 = mapMsg.getInt("diskUtilizationRatio");
```

Format of Metrics Messages

In order to consume and process a metrics messages, you must know its type and format. This section describes the general format of metrics messages and provides detailed information on the format of each type of metrics message.

Metrics messages are of type `MapMessage`. (A type of message whose body contains a set of name-value pairs. The order of entries is not defined.)

- The message header has properties that are useful to applications. The `type` property identifies the type of metric message (and therefore its contents). It is useful if the same subscriber processes more than one type of metrics message for example, messages from the topics `mq.metrics.broker` and `mq.metrics.jvm`. The `timestamp` property indicates when the metric sample was taken and is useful for calculating rates or drawing graphs. The `brokerHost`, `brokerPort`, and `brokerAddress` properties identify the broker that sent the metric message and are useful when a single application needs to process metric messages from different brokers.
- The body of the message contains name-value pairs, and the data values depend on the type of metrics message. The following subsections describe the format of each metrics message type.

Note that the names of name-value pairs (used in code to extract data) are case-sensitive and must be coded exactly as shown. For example, `NumMsgsOut` is incorrect; `numMsgsOut` is correct.

Broker Metrics

The messages you receive when you subscribe to the topic `mq.metrics.broker` have the type property set to `mq.metrics.broker` in the message header and have the data listed in Table 4-2 in the message body.

Table 4-2 Data in the Body of a Broker Metrics Message

Metric Name	Value Type	Description
numConnections	long	Current number of connections to the broker
numMsgsIn	long	Number of JMS messages that have flowed into the broker since it was last started
numMsgsOut	long	Number of JMS messages that have flowed out of the broker since it was last started
numMsgs	long	Current number of JMS messages stored in broker memory and persistent store
msgBytesIn	long	Number of JMS message bytes that have flowed into the broker since it was last started
msgBytesOut	long	Number of JMS message bytes that have flowed out of the broker since it was last started
totalMsgBytes	long	Current number of JMS message bytes stored in broker memory and persistent store
numPktsIn	long	Number of packets that have flowed into the broker since it was last started; this includes both JMS messages and control messages
numPktsOut	long	Number of packets that have flowed out of the broker since it was last started; this includes both JMS messages and control messages
pktBytesIn	long	Number of packet bytes that have flowed into the broker since it was last started; this includes both JMS messages and control messages
pktBytesOut	long	Number of packet bytes that have flowed out of the broker since it was last started; this includes both JMS messages and control messages
numDestinations	long	Current number of destinations in the broker

JVM Metrics

The messages you receive when you subscribe to the topic `mq.metrics.jvm` have the type property set to `mq.metrics.jvm` in the message header and have the data listed in Table 4-3 in the message body.

Table 4-3 Data in the Body of a JVM Metrics Message

Metric Name	Value Type	Description
`freeMemory`	long	Amount of free memory available for use in the JVM heap
`maxMemory`	long	Maximum size to which the JVM heap can grow
`totalMemory`	long	Total memory in the JVM heap

Destination-List Metrics

The messages you receive when you subscribe to a topic named `mq.metrics.destination_list` have the `type` property set to `mq.metrics.destination_list` in the message header.

Each destination in the broker has a corresponding, unique map name (a name-value pair) in the message body. The name depends on whether the destination is a queue or a topic. The type of the name-value pair is `hashtable`.

Each hashtable in the message contains information about a specific destination on the broker. The sub-table within Table 4-4 describes the key-value pairs that can be used to extract this information.

By enumerating through the map names and extracting the hashtable described in Table 4-4, you can form a complete list of destination names and some of their characteristics.

The destination list does not include the following:

- Destinations that are used by Message Queue administration tools
- Destinations that the Message Queue broker creates for internal use

The message body contains name-value pairs as follows:

Table 4-4 Data in the Body of a Destination-List Metrics Message

Metric Name	Value Type	Value or Description		
One of the following: • `mq.metrics.destination.queue.` *monitored_destination_name* • `mq.metrics.destination.topic.` *monitored_destination_name*	hashtable	The corresponding value for the map name is an object of type `java.util.Hashtable`. This hashtable contains the following key-value pairs.		
		Key (String)	**Value Type**	**Value or Description**
		`name`	String	Destination name.
		`type`	String	Destination type. The value is either `queue` or `topic`.
		`isTemporary`	Boolean	Whether the destination is temporary (`true`) or not (`false`).

Notice that the destination name and type could be extracted directly from the metrics topic destination name, but the hashtable includes it for your convenience.

Destination Metrics

The messages you receive when you subscribe to the topic `mq.metrics.destination.queue.`*monitored_destination_name* have the type property `mq.metrics.destination.queue.`*monitored_destination_name* set in the message header. The messages you receive when you subscribe to the topic `mq.metrics.destination.topic.`*monitored_destination_name* have the type property `mq.metrics.destination.topic.`*monitored_destination_name* set in the message header. Either of these messages has the data listed in Table 4-5 in the message body.

Table 4-5 Data in the Body of a Destination Metrics Message

Metric Name	Value Type	Description
`numActiveConsumers`	long	Current number of active consumers
`avgNumActiveConsumers`	long	Average number of active consumers since the broker was last started
`peakNumActiveConsumers`	long	Peak number of active consumers since the broker was last started
`numBackupConsumers`	long	Current number of backup consumers (applies only to queues)

Table 4-5 Data in the Body of a Destination Metrics Message *(Continued)*

Metric Name	Value Type	Description
avgNumBackupConsumers	long	Average number of backup consumers since the broker was last started (applies only to queues)
peakNumBackupConsumers	long	Peak number of backup consumers since the broker was last started (applies only to queues)
numMsgsIn	long	Number of JMS messages that have flowed into this destination since the broker was last started
numMsgsOut	long	Number of JMS messages that have flowed out of this destination since the broker was last started
numMsgs	long	Number of JMS messages currently stored in destination memory and persistent store
avgNumMsgs	long	Average number of JMS messages stored in destination memory and persistent store since the broker was last started
peakNumMsgs	long	Peak number of JMS messages stored in destination memory and persistent store since the broker was last started
msgBytesIn	long	Number of JMS message bytes that have flowed into this destination since the broker was last started
msgBytesOut	long	Number of JMS message bytes that have flowed out of this destination since the broker was last started
totalMsgBytes	long	Current number of JMS message bytes stored in destination memory and persistent store
avgTotalMsgBytes	long	Average number of JMS message bytes stored in destination memory and persistent store since the broker was last started
peakTotalMsgBytes	long	Peak number of JMS message bytes stored in destination memory and persistent store since the broker was last started
peakMsgBytes	long	Peak number of JMS message bytes in a single message since the broker was last started
diskReserved	long	Disk space (in bytes) used by all message records (active and free) in the destination file-based store
diskUsed	long	Disk space (in bytes) used by active message records in destination file-based store
diskUtilizationRatio	int	Quotient of used disk space over reserved disk space. The higher the ratio, the more the disk space is being used to hold active messages

Metrics Monitoring Client Code Examples

Several complete monitoring example applications (including source code and full documentation) are provided when you install Message Queue. You'll find the examples in your IMQ home directory under `/demo/monitoring`. Before you can run these clients, you must set up your environment (for example, the `CLASSPATH` environment variable). For details, see "Quick-Start Tutorial" on page 25.

Next are brief descriptions of three examples—Broker Metrics, Destination List Metrics, and Destination Metrics—with annotated code examples from each.

These examples use the utility classes `MetricsPrinter` and `MultiColumnPrinter` to print formatted and aligned columns of text output. However, rather than explaining how those utility classes are used, the following code examples focus on how to subscribe to the metrics topic and how to extract information from the metrics messages received.

Notice that in the source files, the code for subscribing to metrics topics and processing messages is actually spread across various methods. However, for the sake of clarity, the examples are shown here as though they were contiguous blocks of code.

A Broker Metrics Example

The source file for this code example is `BrokerMetrics.java`. This metrics monitoring client subscribes to the topic `mq.metrics.broker` and prints broker-related metrics to the standard output.

Code Example 4-1 shows how to subscribe to `mq.metrics.broker`.

Code Example 4-1 Example of Subscribing to a Broker Metrics Topic

```
com.sun.messaging.TopicConnectionFactory     metricConnectionFactory;
    TopicConnection             metricConnection;
    TopicSession                metricSession;
    TopicSubscriber             metricSubscriber;
    Topic                       metricTopic;

    metricConnectionFactory = new
com.sun.messaging.TopicConnectionFactory();

    metricConnection = metricConnectionFactory.createTopicConnection();
    metricConnection.start();

    metricSession = metricConnection.createTopicSession(false,
                        Session.AUTO_ACKNOWLEDGE);

    metricTopic = metricSession.createTopic("mq.metrics.broker");
```

Code Example 4-1 Example of Subscribing to a Broker Metrics Topic *(Continued)*

```
    metricSubscriber = metricSession.createSubscriber(metricTopic);
    metricSubscriber.setMessageListener(this);
```

The incoming message is processed in the `onMessage()` and `doTotals()` methods, as shown in Code Example 4-2.

Code Example 4-2 Example of Processing a Broker Metrics Message

```
public void onMessage(Message m)  {
    try  {
        MapMessage mapMsg = (MapMessage)m;
        String type = mapMsg.getStringProperty("type");

        if (type.equals("mq.metrics.broker"))  {
            if (showTotals)  {
                doTotals(mapMsg);
        ...
        }
}

private void doTotals(MapMessage mapMsg)  {
    try  {
        String oneRow[] = new String[ 8 ];
        int i = 0;

        /*
        * Extract broker metrics
        */
        oneRow[i++] = Long.toString(mapMsg.getLong("numMsgsIn"));
        oneRow[i++] = Long.toString(mapMsg.getLong("numMsgsOut"));
        oneRow[i++] = Long.toString(mapMsg.getLong("msgBytesIn"));
        oneRow[i++] = Long.toString(mapMsg.getLong("msgBytesOut"));
        oneRow[i++] = Long.toString(mapMsg.getLong("numPktsIn"));
        oneRow[i++] = Long.toString(mapMsg.getLong("numPktsOut"));
        oneRow[i++] = Long.toString(mapMsg.getLong("pktBytesIn"));
        oneRow[i++] = Long.toString(mapMsg.getLong("pktBytesOut"));
        ...
    } catch (Exception e)  {
        System.err.println("onMessage: Exception caught: " + e);
    }
}
```

Notice how the metrics type is extracted, using the `getStringProperty()` method, and is checked. If you use the `onMessage()` method to process metrics messages of different types, you can use the `type` property to distinguish between different incoming metrics messages.

Also notice how various pieces of information on the broker are extracted, using the `getLong()` method of `mapMsg`.

Run this example monitoring client with the following command:

```
java BrokerMetrics
```

The output looks like the following:

```
-------------------------------------------------------------
Msgs                Msg Bytes           Pkts                Pkt Bytes
In        Out       In        Out       In       Out        In       Out
-------------------------------------------------------------
0         0         0         0         6        5          888      802
0         1         0         633       7        8          1004     1669
```

A Destination List Metrics Example

The source file for this code example is `DestListMetrics.java`. This client application monitors the list of destinations on a broker by subscribing to the topic `mq.metrics.destination_list`. The messages that arrive contain information describing the destinations that currently exist on the broker, such as destination name, destination type, and whether the destination is temporary.

Code Example 4-3 shows how to subscribe to `mq.metrics.destination_list`.

Code Example 4-3 Example of Subscribing to the Destination List Metrics Topic

```
com.sun.messaging.TopicConnectionFactory
metricConnectionFactory;
TopicConnection                 metricConnection;
TopicSession                    metricSession;
TopicSubscriber                 metricSubscriber;
Topic                           metricTopic;
String                          metricTopicName = null;

metricConnectionFactory = new com.sun.messaging.TopicConnectionFactory();
metricConnection = metricConnectionFactory.createTopicConnection();
metricConnection.start();
```

Code Example 4-3 Example of Subscribing to the Destination List Metrics Topic *(Continued)*

```
metricSession = metricConnection.createTopicSession(false,
                   Session.AUTO_ACKNOWLEDGE);

metricTopicName = "mq.metrics.destination_list";
metricTopic = metricSession.createTopic(metricTopicName);

metricSubscriber = metricSession.createSubscriber(metricTopic);
metricSubscriber.setMessageListener(this);
```

The incoming message is processed in the `onMessage()` method, as shown in Code Example 4-4:

Code Example 4-4 Example of Processing a Destination List Metrics Message

```
public void onMessage(Message m)  {
    try  {
        MapMessage mapMsg = (MapMessage)m;
        String type = mapMsg.getStringProperty("type");

        if (type.equals(metricTopicName))  {
            String oneRow[] = new String[ 3 ];

            /*
            * Extract metrics
            */
            for (Enumeration e = mapMsg.getMapNames();
                e.hasMoreElements();) {

                String metricDestName = (String)e.nextElement();
                Hashtable destValues =
                    (Hashtable)mapMsg.getObject(metricDestName);
                int i = 0;

                oneRow[i++] = (destValues.get("name")).toString();
                oneRow[i++] = (destValues.get("type")).toString();
                oneRow[i++] = (destValues.get("isTemporary")).toString();

                mp.add(oneRow);
            }

            mp.print();
            System.out.println("");

            mp.clear();
        } else  {
            System.err.println("Msg received:
                not destination list metric type");
```

Code Example 4-4 Example of Processing a Destination List Metrics Message

```
            }
        } catch (Exception e)  {
            System.err.println("onMessage: Exception caught: " + e);
        }
    }
```

Notice how the metrics type is extracted and checked, and how the list of destinations is extracted. By iterating through the map names in `mapMsg` and extracting the corresponding value (a hashtable), you can construct a list of all the destinations and their related information.

As discussed in "Format of Metrics Messages" on page 85, these map names are metrics topic names having one of two forms:

`mq.metrics.destination.queue.`*monitored_destination_name*

`mq.metrics.destination.topic.`*monitored_destination_name*

(The map names can also be used to monitor a destination, but that is not done in this particular example.)

Notice that from each extracted hashtable, the information on each destination is extracted using the keys `name`, `type`, and `isTemporary`. The extraction code from the previous code example is reiterated here for your convenience.

Code Example 4-5 Example of Extracting Destination Information From a Hashtable

```
            String metricDestName = (String)e.nextElement();
            Hashtable destValues = (Hashtable)mapMsg.getObject(metricDestName);
            int i = 0;

            oneRow[i++] = (destValues.get("name")).toString();
            oneRow[i++] = (destValues.get("type")).toString();
            oneRow[i++] = (destValues.get("isTemporary")).toString();
```

Run this example monitoring client with the following command:

```
java DestListMetrics
```

The output looks like the following:

```
-------------------------------------------------
Destination Name            Type                        IsTemporary
-------------------------------------------------
SimpleQueue                 queue                       false
fooQueue                    queue                       false
topic1                      topic                       false
```

A Destination Metrics Example

The source file for this code example is `DestMetrics.java`. This client application monitors a specific destination on a broker. It accepts the destination type and name as parameters, and it constructs a metrics topic name of the form `mq.metrics.destination.queue.`*monitored_destination_name* or `mq.metrics.destination.topic.`*monitored_destination_name*.

Code Example 4-6 shows how to subscribe to the metrics topic for monitoring a specified destination.

Code Example 4-6 Example of Subscribing to a Destination Metrics Topic

```
com.sun.messaging.TopicConnectionFactory        metricConnectionFactory;
TopicConnection             metricConnection;
TopicSession                metricSession;
TopicSubscriber             metricSubscriber;
Topic                       metricTopic;
String                      metricTopicName = null;
String                      destName = null,
                            destType = null;

for (int i = 0; i < args.length; ++i)  {
    ...
    } else if (args[i].equals("-n"))  {
        destName = args[i+1];
    } else if (args[i].equals("-t"))  {
        destType = args[i+1];
    }
}

metricConnectionFactory = new com.sun.messaging.TopicConnectionFactory();

metricConnection = metricConnectionFactory.createTopicConnection();
```

Code Example 4-6 Example of Subscribing to a Destination Metrics Topic *(Continued)*

```
metricConnection.start();

metricSession = metricConnection.createTopicSession(false,
                    Session.AUTO_ACKNOWLEDGE);

if (destType.equals("q"))  {
    metricTopicName = "mq.metrics.destination.queue." + destName;
} else  {
    metricTopicName = "mq.metrics.destination.topic." + destName;
}

metricTopic = metricSession.createTopic(metricTopicName);

metricSubscriber = metricSession.createSubscriber(metricTopic);
metricSubscriber.setMessageListener(this);
```

The incoming message is processed in the `onMessage()` method, as shown in Code Example 4-7:

Code Example 4-7 Example of Processing a Destination Metrics Message

```
public void onMessage(Message m)  {
    try  {
        MapMessage mapMsg = (MapMessage)m;
        String type = mapMsg.getStringProperty("type");

        if (type.equals(metricTopicName))  {
            String oneRow[] = new String[ 11 ];
            int i = 0;

            /*
            * Extract destination metrics
            */
            oneRow[i++] = Long.toString(mapMsg.getLong("numMsgsIn"));
            oneRow[i++] = Long.toString(mapMsg.getLong("numMsgsOut"));
            oneRow[i++] = Long.toString(mapMsg.getLong("msgBytesIn"));
            oneRow[i++] = Long.toString(mapMsg.getLong("msgBytesOut"));

            oneRow[i++] = Long.toString(mapMsg.getLong("numMsgs"));
            oneRow[i++] = Long.toString(mapMsg.getLong("peakNumMsgs"));
            oneRow[i++] = Long.toString(mapMsg.getLong("avgNumMsgs"));

            oneRow[i++] =
Long.toString(mapMsg.getLong("totalMsgBytes")/1024);
            oneRow[i++] =

Long.toString(mapMsg.getLong("peakTotalMsgBytes")/1024);
            oneRow[i++] =
```

Code Example 4-7 Example of Processing a Destination Metrics Message *(Continued)*

```
Long.toString(mapMsg.getLong("avgTotalMsgBytes")/1024);

            oneRow[i++] =
Long.toString(mapMsg.getLong("peakMsgBytes")/1024);

            mp.add(oneRow);
            ...
        }
    } catch (Exception e)  {
        System.err.println("onMessage: Exception caught: " + e);
    }
}
```

Notice how the metrics type is extracted, using the `getStringProperty()` method as in the previous examples, and is checked. Also notice how various destination data are extracted, using the `getLong()` method of `mapMsg`.

Run this example monitoring client with one of the following commands:

`java DestMetrics -t t -n` *topic_name*

`java DestMetrics -t q -n` *queue_name*

Using a queue named `SimpleQueue` as an example, the command would be:

```
java DestMetrics -t q -n SimpleQueue
```

The output looks like the following:

```
---------------------------------------------------------------------------------------
Msgs           Msg Bytes       Msg Count             Tot Msg Bytes (k)      Largest Msg
In    Out      In       Out    Curr     Peak   Avg    Curr   Peak   Avg      (k)
---------------------------------------------------------------------------------------
500   0        318000   0      500      500    250    310    310    155      0
```

Chapter 5

Working with SOAP Messages

SOAP is a protocol that allows for the exchange of data whose structure is defined by an XML scheme. Using Message Queue, you can send JMS messages that contain a SOAP payload. This allows you to transport SOAP messages reliably and to publish SOAP messages to JMS subscribers. This chapter covers the following topics:

- "What is SOAP?" on page 100
- "SOAP Messaging in JAVA" on page 107
- "Using SOAP Administered Objects" on page 115
- "SOAP Messaging Models and Examples" on page 117
- "Integrating SOAP and Message Queue" on page 130

If you are familiar with the SOAP specification, you can skip the introductory section and start by reading "SOAP Messaging in JAVA" on page 107.

What is SOAP?

SOAP, the Simple Object Access Protocol, is a protocol that allows the exchange of structured data between peers in a decentralized, distributed environment. The structure of the data being exchanged is specified by an XML scheme.

The fact that SOAP messages are encoded in XML makes SOAP messages portable, because XML is a portable, system-independent way of representing data. By representing data using XML, you can access data from legacy systems as well as share your data with other enterprises. The data integration offered by XML also makes this technology a natural for web-based computing such as web services. Firewalls can recognize SOAP packets based on their content type (`text/xml-SOAP`) and can filter messages based on information exposed in the SOAP message header.

The SOAP specification describes a set of conventions for exchanging XML messages. As such, it forms a natural foundation for web services that also need to exchange information encoded in XML. Although any two partners could define their own protocol for carrying on this exchange, having a standard such as SOAP allows developers to build the generic pieces that support this exchange. These pieces might be software that adds functionality to the basic SOAP exchange, or might be tools that administer SOAP messaging, or might even comprise parts of an operating system that supports SOAP processing. Once this support is put in place, other developers can focus on creating the web services themselves.

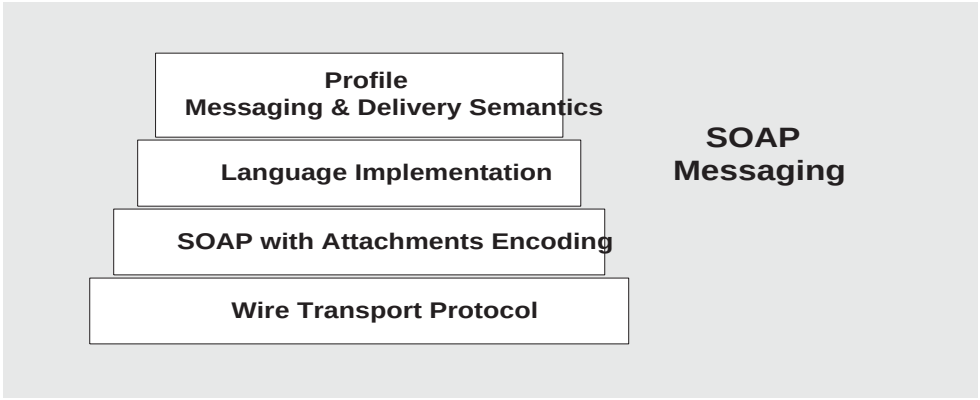
The SOAP protocol is fully described at `http://www.w3.org/TR/SOAP`. This section restricts itself to discussing the reasons why you would use SOAP and to describing basic concepts that will make it easier to work with SOAP messages.

SOAP with Attachments API for Java

The Soap with Attachments API for Java (SAAJ) is a JAVA-based API that enforces compliance to the SOAP standard. When you use this API to assemble and disassemble SOAP messages, it ensures the construction of syntactically correct SOAP messages. SAAJ also makes it possible to automate message processing when several applications need to handle different parts of a message before forwarding it to the next recipient.

Figure 5-1 shows the layers that can come into play in the implementation of SOAP messaging. This chapter focuses on the SOAP and language implementation layers.

Figure 5-1 SOAP Messaging Layers



The sections that follow describe each layer shown in the preceding figure in greater detail. The rest of this chapter focuses on the SOAP and language implementation layers.

The Transport Layer

Underlying any messaging system is the transport or wire protocol that governs the serialization of the message as it is sent across a wire and the interpretation of the message bits when it gets to the other side. Although SOAP messages can be sent using any number of protocols, the SOAP specification defines only the binding with HTTP. SOAP uses the HTTP request/response message model. It provides SOAP request parameters in an HTTP request and SOAP response parameters in an HTTP response. The HTTP binding has the advantage of allowing SOAP messages to go through firewalls.

The SOAP Layer

Above the transport layer is the SOAP layer. This layer, which is defined in the SOAP Specification, specifies the XML scheme used to identify the message parts: envelope, header, body, and attachments. All SOAP message parts and contents, except for the attachments, are written in XML. The following sample SOAP message shows how XML tags are used to define a SOAP message:

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle=
          "http://schemas.xmlsoap.org/soap/encoding/">
      <SOAP-ENV:Body>
          <m:GetLastTradePrice xmlns:m="Some-URI">
              <symbol>DIS</symbol>
          </m:GetLastTradePrice>
      </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The wire transport and SOAP layers are actually sufficient to do SOAP messaging. You could create an XML document that defines the message you want to send, and you could write HTTP commands to send the message from one side and to receive it on the other. In this case, the client is limited to sending synchronous messages to a specified URL. Unfortunately, the scope and reliability of this kind of messaging is severely restricted. To overcome these limitations, the *provider* and *profile* layers are added to SOAP messaging.

The Language Implementation Layer

A language implementation allows you to create XML messages that conform to SOAP, using API calls. For example, the SAAJ implementation of SOAP, allows a Java client to construct a SOAP message and all its parts as Java objects. The client would also use SAAJ to create a connection and use it to send the message. Likewise, a web service written in Java could use the same implementation (SAAJ), or any other language implementation, to receive the message, to disassemble it, and to acknowledge its receipt.

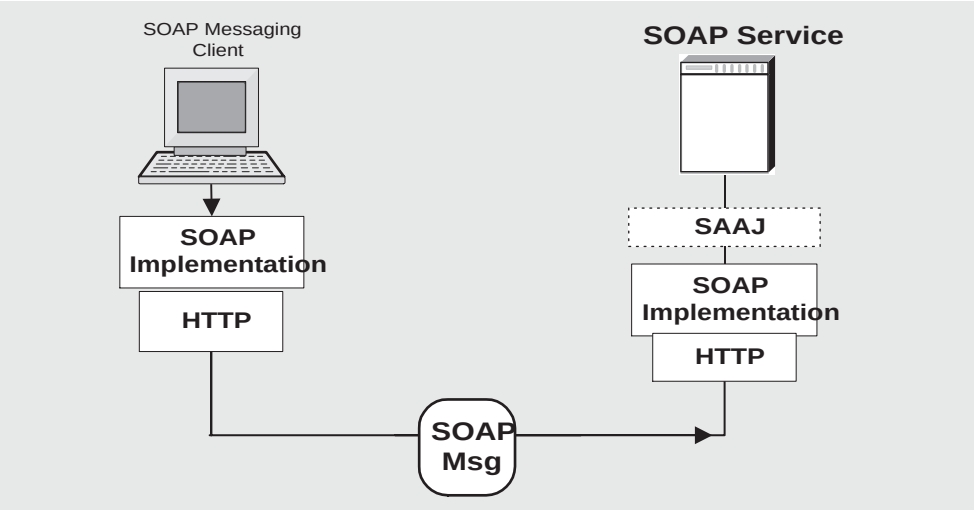
The Profiles Layer

In addition to a language implementation, a SOAP implementation can offer services that relate to message delivery. These could include reliability, persistence, security, and administrative control, and are typically delievered by a SOAP messaging provider. These services will be provided for SOAP messaging by Message Queue in future releases.

Interoperability

Because SOAP providers must all construct and deconstruct messages as defined by the SOAP specification, clients and services using SOAP are interoperable. That is, as shown in Figure 5-2, the client and the service doing SOAP messaging do not need to be written in the same language nor do they need to use the same SOAP provider. It is only the packaging of the message that must be standard.

Figure 5-2 SOAP Interoperability



In order for a SAAJ client or service to interoperate with a service or client using a different implementation, the parties must agree on two things:

- They must use the same transport bindings--that is, the same wire protocol.
- They must use the same profile in constructing the SOAP message being sent

The SOAP Message

Having surveyed the SOAP messaging layers, let's examine the SOAP message itself. Although the work of rendering a SOAP message in XML is taken care of by the SAAJ implementation, you must still understand its structure in order to make the SAAJ calls in the right order.

A *SOAP message* is an XML document that consists of a SOAP envelope, an optional SOAP header, and a SOAP body. The SOAP message header contains information that allows the message to be routed through one or more intermediate nodes before it reaches its final destination.

- The *envelope* is the root element of the XML document representing the message. It defines the framework for how the message should be handled and by whom. Once it encounters the Envelope element, the SOAP processor knows that the XML is a SOAP message and can then look for the individual parts of the message.

- The *header* is a generic mechanism for adding features to a SOAP message. It can contain any number of child elements that define extensions to the base protocol. For example, header child elements might define authentication information, transaction information, locale information, and so on. The *actors*, the software that handle the message may, without prior agreement, use this mechanism to define who should deal with a feature and whether the feature is mandatory or optional.
- The *body* is a container for mandatory information intended for the ultimate recipient of the message.

A SOAP message may also contain an attachment, which does not have to be in XML. For more information, see "SOAP Packaging Models" next.

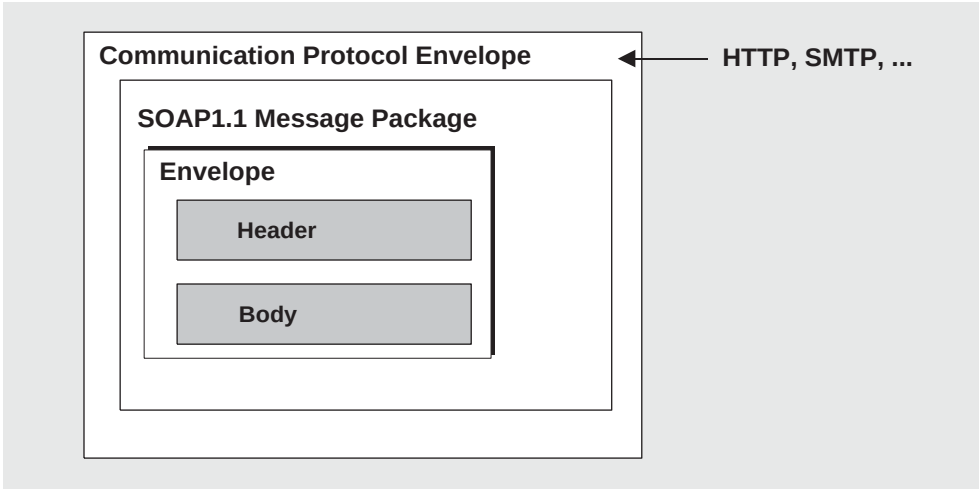
A SOAP message is constructed like a nested matrioshka doll. When you use SAAJ to assemble or disassemble a message, you need to make the API calls in the appropriate order to get to the message part that interests you. For example, in order to add content to the message, you need to get to the body part of the message. To do this you need to work through the nested layers: SOAP part, SOAP envelope, SOAP body, until you get to the SOAP body element that you will use to specify your data. For more information, see "The SOAP Message Object" on page 107.

SOAP Packaging Models

The SOAP specification describes two models of SOAP messages: one that is encoded entirely in XML and one that allows the sender to add an attachment containing non-XML data. You should look over the following two figures and note the parts of the SOAP message for each model. When you use SAAJ to define SOAP messages and their parts, it will be helpful for you to be familiar with this information.

Figure 5-3 shows the SOAP model without attachments. This package includes a SOAP envelope, a header, and a body. The header is optional.

Figure 5-3 SOAP Message Without Attachments

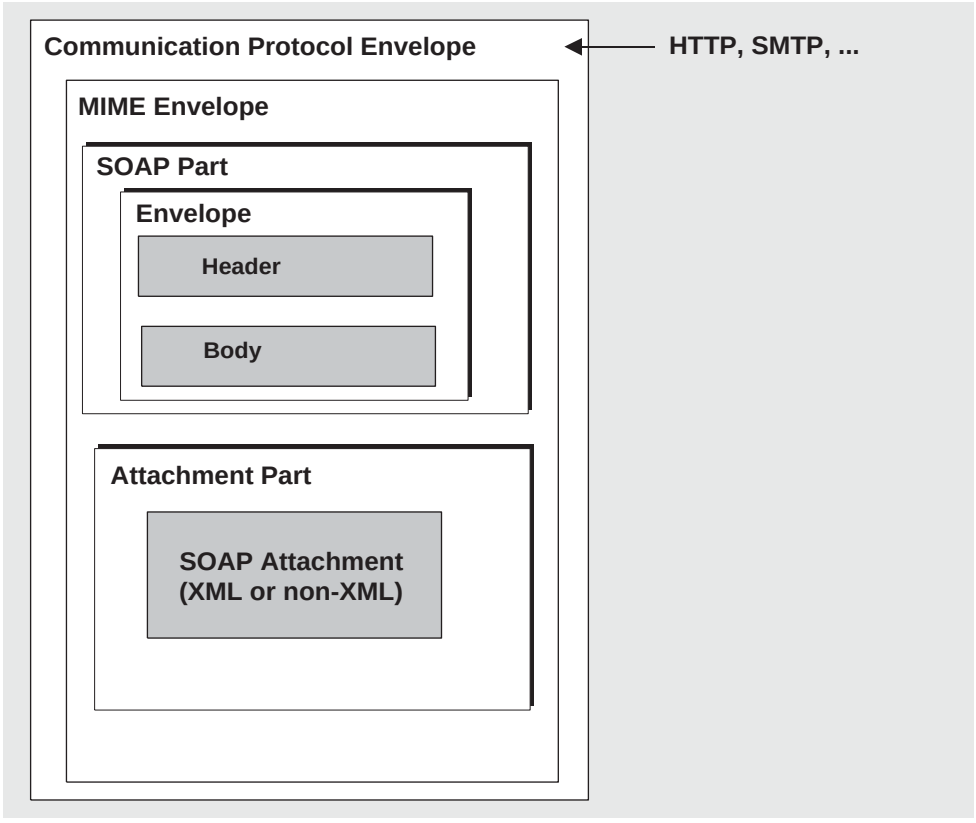


When you construct a SOAP message using SAAJ, you do not have to specify which model you're following. If you add an attachment, a message like that shown in Figure 5-4 is constructed; if you don't, a message like that shown in Figure 5-3 is constructed.

Figure 5-4 shows a SOAP Message with attachments. The attachment part can contain any kind of content: image files, plain text, and so on. The sender of a message can choose whether to create a SOAP message with attachments. The message receiver can also choose whether to consume an attachment.

A message that contains one or more attachments is enclosed in a MIME envelope that contains all the parts of the message. In SAAJ, the MIME envelope is automatically produced whenever the client creates an attachment part. If you add an attachment to a message, you are responsible for specifying (in the MIME header) the type of data in the attachment.

Figure 5-4 SOAP Message with Attachments



SOAP Messaging in JAVA

The SOAP specification does not provide a programming model or even an API for the construction of SOAP messages; it simply defines the XML schema to be used in packaging a SOAP message.

SAAJ is an application programming interface that can be implemented to support a programming model for SOAP messaging and to furnish Java objects that application or tool writers can use to construct, send, receive, and examine SOAP messages. SAAJ defines two packages:

- `javax.xml.soap`: you use the objects in this package to define the parts of a SOAP message and to assemble and disassemble SOAP messages. You can also use this package to send a SOAP message without the support of a provider.
- `javax.xml.messaging`: you use the objects in this package to send a SOAP message using a provider and to receive SOAP messages.

This chapter focuses on the `javax.xml.soap` package and how you use the objects and methods it defines

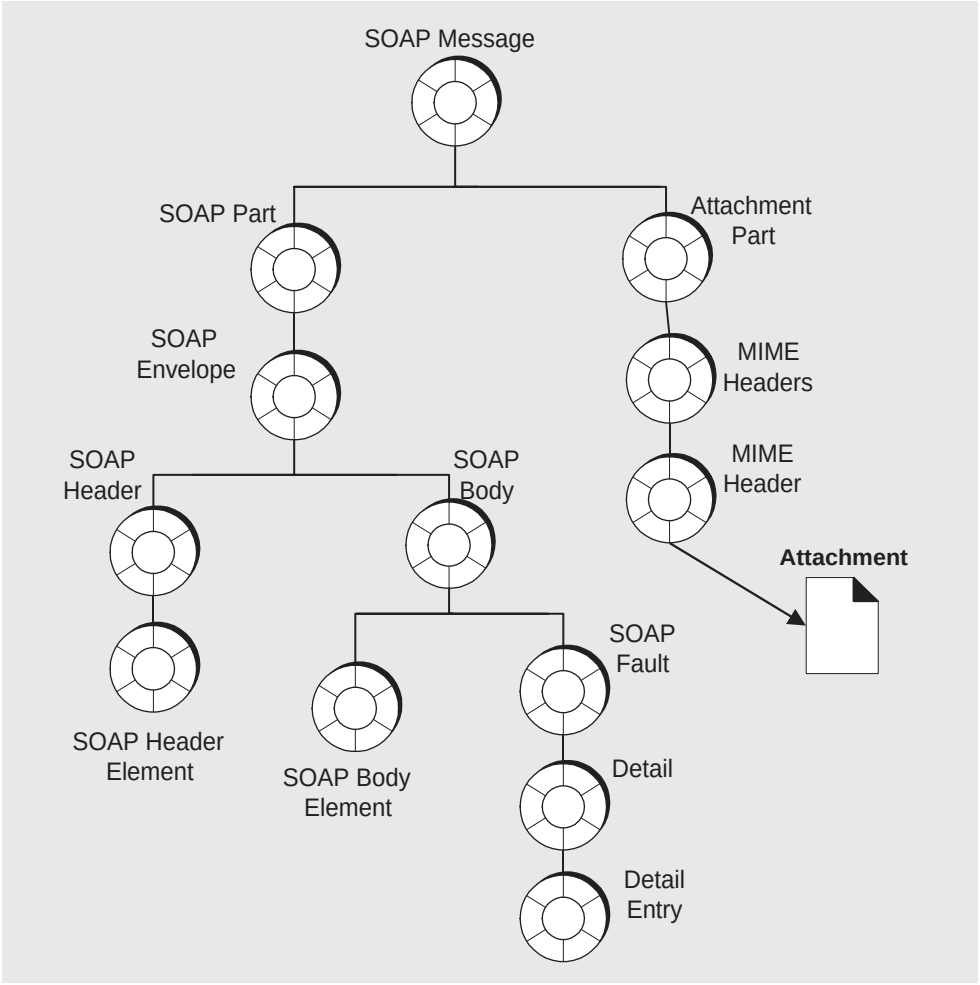
- to assemble and disassemble SOAP messages
- to send and receive these messages

It also explains how you can use the JMS API and Message Queue to send and receive JMS messages that carry SOAP message payloads.

The SOAP Message Object

A SOAP Message Object is a tree of objects as shown in Figure 5-5. The classes or interfaces from which these objects are derived are all defined in the `javax.xml.soap` package.

Figure 5-5 SOAP Message Object



As shown in the figure, the `SOAPMessage` object is a collection of objects divided in two parts: a SOAP part and an attachment part. The main thing to remember is that the attachment part can contain non-xml data.

The SOAP part of the message contains an envelope that contains a body (which can contain data or fault information) and an optional header. When you use SAAJ to create a SOAP message, the SOAP part, envelope, and body are created for you: you need only create the body elements. To do that you need to get to the parent of the body element, the SOAP body.

In order to reach any object in the SOAPMessage tree, you must traverse the tree starting from the root, as shown in the following lines of code. For example, assuming the SOAPMessage is `MyMsg`, here are the calls you would have to make in order to get the SOAP body:

```
SOAPPart MyPart = MyMsg.getSOAPPart();

SOAPEnvelope MyEnv = MyPart.getEnvelope();

SOAPBody MyBody = envelope.getBody();
```

At this point, you can create a name for a body element (as described in "Namespaces" on page 110) and add the body element to the SOAPMessage.

For example, the following code line creates a name (a representation of an XML tag) for a body element:

```
Name bodyName = envelope.createName("Temperature");
```

The next code line adds the body element to the body:

```
SOAPBodyElement myTemp = MyBody.addBodyElement(bodyName);
```

Finally, this code line defines some data for the body element `bodyName`:

```
myTemp.addTextNode("98.6");
```

Inherited Methods

The elements of a SOAP message form a tree. Each node in that tree implements the `Node` interface and, starting at the envelope level, each node implements the `SOAPElement` interface as well. The resulting shared methods are described in Table 5-1.

Table 5-1 Inherited Methods

Inherited From	Method Name	Purpose
`SOAPElement`	`addAttribute(Name, String)`	Add an attribute with the specified `Name` object and string value.

	`addChildElement(Name)`
	`addChildElement(String, String)`
	`addChildElement(String, String, String)`
		Create a new `SOAPElement` object, initialized with the given `Name` object, and add the new element.
		(Use the `Envelope.createName` method to create a `Name` object.)

	`addNameSpaceDeclaration (String, String)`	Add a namespace declaration with the specified prefix and URI.
	`addTextnode(String)`	Create a new `Text` object initialized with the given `String` and add it to this `SOAPElement` object.

Table 5-1 Inherited Methods *(Continued)*

Inherited From	Method Name	Purpose
	`getAllAttributes()`	Return an iterator over all the attribute names in this object.
	`getAttributeValue(Name)`	Return the value of the specified attribute.
	`getChildElements()`	Return an iterator over all the immediate content of this element.
	`getChildElements(Name)`	Return an iterator over all the child elements with the specified name.
	`getElementName()`	Return the name of this object.
	`getEncodingStyle()`	Return the encoding style for this object.
	`getNameSpacePrefixes()`	Return an iterator of namespace prefixes.
	`getNamespaceURI(String)`	Return the URI of the namespace with the given prefix.
	`removeAttribute(Name)`	Remove the specified attribute.
	`removeNamespaceDeclaration (String)`	Remove the namespace declaration that corresponds to the specified prefix.
	`setEncodingStyle(String)`	Set the encoding style for this object to that specified by `String`.
`Node`	`detachNode()`	Remove this `Node` object from the tree.
	`getParentElement()`	Return the parent element of this `Node` object.
	`getValue`	Return the value of the immediate child of this `Node` object if a child exists and its value is `text`.
	`recycleNode()`	Notify the implementation that his `Node` object is no longer being used and is free for reuse.
	`setParentElement (SOAPElement)`	Set the parent of this object to that specified by the `SOAPElement` parameter.

Namespaces

An *XML namespace* is a means of qualifying element and attribute names to disambiguate them from other names in the same document. This section provides a brief description of XML namespaces and how they are used in SOAP. For complete information, see http://www.w3.org/TR/REC-xml-names/.

An explicit XML namespace declaration takes the following form

<*prefix*:*myElement*

xmlns:*prefix* ="*URI*">

The declaration defines *prefix* as an alias for the specified URI. In the element `myElement`, you can use *prefix* with any element or attribute to specify that the element or attribute name belongs to the namespace specified by the URI.

The following is an example of a namespace declaration:

```
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
```

This declaration defines `SOAP_ENV` as an alias for the namespace

`http://schemas.xmlsoap.org/soap/envelope/`

After defining the alias, you can use it as a prefix to any attribute or element in the `Envelope` element. In Code Example 5-1, the elements `<Envelope>` and `<Body>` and the attribute `encodingStyle` all belong to the SOAP namespace specified by the URI `"http://schemas.xmlsoap.org/soap/envelope/"`.

Code Example 5-1 Explicit Namespace Declarations

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle=
                     "http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Header>
        <HeaderA
 xmlns="HeaderURI"
 SOAP-ENV:mustUnderstand="0">
     The text of the header
     </HeaderA>
 </SOAP-ENV:Header>
   <SOAP-ENV:Body>
.
.
.
   </SOAP-ENV:Body>
 </SOAP-ENV:Envelope>
```

Note that the URI that defines the namespace does not have to point to an actual location; its purpose is to disambiguate attribute and element names.

Pre-defined SOAP Namespaces

SOAP defines two namespaces:

- The SOAP envelope, the root element of a SOAP message, has the following namespace identifier:

  `"http://schemas.xmlsoap.org/soap/envelope"`

- The SOAP serialization, the URI defining SOAP's serialization rules, has the following namespace identifier:

  `"http://schemas.xmlsoap.org/soap/encoding"`

When you use SAAJ to construct or consume messages, you are responsible for setting or processing namespaces correctly and for discarding messages that have incorrect namespaces.

Using Namespaces when Creating a SOAP Name

When you create the body elements or header elements of a SOAP message, you must use the `Name` object to specify a well-formed name for the element. You obtain a `Name` object by calling the method `SOAPEnvelope.createName`.

When you call this method, you can pass a local name as a parameter or you can specify a local name, prefix, and URI. For example, the following line of code defines a name object `bodyName`.

```
Name bodyName = MyEnvelope.createName("TradePrice",
                                      "GetLTP",
                                         "http://foo.eztrade.com");
```

This would be equivalent to the namespace declaration:

```
<GetLTP:TradePrice xmlns:GetLTP= "http://foo.eztrade.com">
```

The following code shows how you create a name and associate it with a `SOAPBody` element. Note the use and placement of the `createName` method.

```
SoapBody body = envelope.getBody();//get body from envelope

Name bodyName = envelope.createName("TradePrice", "GetLTP",
                                       "http://foo.eztrade.com");

SOAPBodyElement gltp = body.addBodyElement(bodyName);
```

Parsing Name Objects

For any given `Name` object, you can use the following `Name` methods to parse the name:

- `getQualifiedName` returns "*prefix:LocalName*", for the given name, this would be `GetLTP:TradePrice`.
- `getURI` would return "`http://foo.eztrade.com`".
- `getLocalName` would return "`TradePrice`".
- `getPrefix` would return "`GetLTP`".

Destination, Message Factory, and Connection Objects

SOAP messaging occurs when a SOAP message, produced by a *message factory*, is sent to an *endpoint* via a *connection*.

- If you are working without a provider, you must do the following:
  - Create a `SOAPConnectionFactory` object.
  - Create a `SOAPConnection object`.
  - Create an `Endpoint` object that represents the message's destination.
  - Create a `MessageFactory` object and use it to create a message.
  - Populate the message.
  - Send the message.
- If you are working with a provider, you must do the following:
  - Create a `ProviderConnectionFactory` object.
  - Get a `ProviderConnection` object from the provider connection factory.
  - Get a `MessageFactory` object from the provider connection and use it to create a message.
  - Populate the message.
  - Send the message.

The following three sections describe endpoint, message factory, and connection objects in greater detail.

Endpoint

An *endpoint* identifies the final destination of a message. An endpoint is defined either by the `Endpoint` class (if you use a provider) or by the `URLEndpoint` class (if you don't use a provider).)

Constructing an Endpoint

You can initialize an endpoint either by calling its constructor or by looking it up in a naming service. For information about creating administered objects for endpoints, see "Using SOAP Administered Objects" on page 115.

The following code uses a constructor to create a `URLEndpoint`:

```
myEndpoint = new URLEndpoint("http://somehost/myServlet");
```

Using the Endpoint to Address a Message

If you are using a provider, the Message Factory creating the message includes the endpoint specification in the message header.

If you do not use a provider, you can specify the endpoint as a parameter to the `SOAPConnection.call` method, which you use to send a SOAP message.

Sending a Message to Multiple Endpoints

If you are using an administered object to define an endpoint, note that it is possible to associate that administered object with multiple URLs--each URL, is capable of processing incoming SOAP messages. The code sample below associates the endpoint whose lookup name is `myEndpoint` with two URLs: `http://www.myServlet1/` and `http://www.myServlet2/`.

```
imqobjmgr add
    -t e
    -l "cn=myEndpoint"
    -o "imqSOAPEndpointList=http://www.myServlet1/
                                    http://www.myServlet2/"
```

This syntax allows you to use a SOAP connection to publish a SOAP message to multiple endpoints. For additional information about the endpoint administered object, see "Using SOAP Administered Objects" on page 115.

Message Factory

You use a Message Factory to create a SOAP message.

To instantiate a message factory directly, use a statement like the following:

```
MessageFactory mf = MessageFactory.newInstance();
```

Connection

To send a SOAP message using SAAJ, you must obtain a `SOAPConnection`. You can also transport a SOAP message using Message Queue; for more information, see "Integrating SOAP and Message Queue" on page 130.

SOAP Connection

A `SOAPConnection` allows you to send messages directly to a remote party. You can obtain a `SOAPConnection` object simply by calling the static method `SOAPConnectionFactory.newInstance()`. Neither reliability nor security are guaranteed over this type of connection.

Using SOAP Administered Objects

Administered objects are objects that encapsulate provider-specific configuration and naming information. For endpoint objects, you have the choice either to instantiate such an object or to create an administered object and associate it with an endpoint object instance.

The main benefit of creating an endpoint through a JNDI lookup is to isolate endpoint URLs from the code, allowing the application to switch the destination without recompiling the code. A secondary benefit is provider independence.

Creating an administered object for a SOAP element is the same as creating an administered object in Message Queue: you use the Object Manager (`imqobjmgr`) utility to specify the lookup name of the object, its attributes, and its type.

Table 5-2 lists and describes the attributes and other information that you need to specify when you create an endpoint administered object. Remember to specify all attributes as strings.

Table 5-2 SOAP Administered Object Information

Option	Description
-o "*attribute=val*"	Use this option to specify three possible attributes for an endpoint administered object: • A URL list -o "`imqSOAPEndpointList` = "*url1 url2 ....urln*" The list may contain one or more space-separated URLs. If it contains more than one, the message is broadcast to all the URLs. Each URL should be associated with a servlet that can receive and process a SOAP message. • A name -o "`imqEndpointName=`*SomeName*" If you don't specify a name, the name `Untitled_Endpoint_Object` is used by default. • A description `-o "imqEndpointDescription=my endpoints for broadcast"` If you don't specify a description, the default value `"A description for the endpoint object"` is supplied by default.
-l "cn=*lookupName*"	Use this option to specify the lookup name of the endpoint.
-t *type*	Use this option to specify the object's type. This is always e for an endpoint.
-i *filename*	Use this option to specify the name of an input file containing `imqobjmgr` commands. Such an input file is typically used to specify object store attributes.
-j "*attribute=val*"	Use this option to specify object store attributes. You can also specify these in an input file. Use the -i option to specify the input file.

Code Example 5-2 shows how you use the `imqobjmgr` command to create an administered object for an endpoint and add it to an object store. The -i option specifies the name of an input file that defines object store attributes (`-j` option).

Code Example 5-2 Adding an Endpoint Administered Object

```
imqobjmgr add
     -t e
     -l "cn=myEndpoint"
     -o "imqSOAPEndpointList=http://www.myServlet/
                             http://www.myServlet2/"
     -o "imqEndpointName=MyBroadcastEndpoint"
     -i MyObjStoreAttrs
```

Having created the administered object and added it to an object store, you can now use it when you want to use an endpoint in your SAAJ application. In Code Example 5-3, you first create an initial context for the JNDI lookup and then you look up the desired object.

Code Example 5-3 Looking up an Endpoint Administered Object

```
Hashtable env = new Hashtable();
env.put (Context.INITIAL_CONTEXT_FACTORY,
     "com.sun.jndi.fscontext.RefFSContextFactory");
env.put (Context.PROVIDER_URL,
     "file:///c:/imq_admin_objects");
Context ctx = new InitialContext(env);
Endpoint mySOAPEndpoint = (Endpoint)
      ctx.lookup("cn=myEndpoint");
```

You can also list, delete, and update administered objects. For additional information, please see the *Message Queue Administration Guide*.

SOAP Messaging Models and Examples

This section explains how you use SAAJ to send and receive a SOAP message. It is also possible to construct a SOAP message using SAAJ and to send it as the payload of a JMS message. For information, see "Integrating SOAP and Message Queue" on page 130.

SOAP Messaging Programming Models

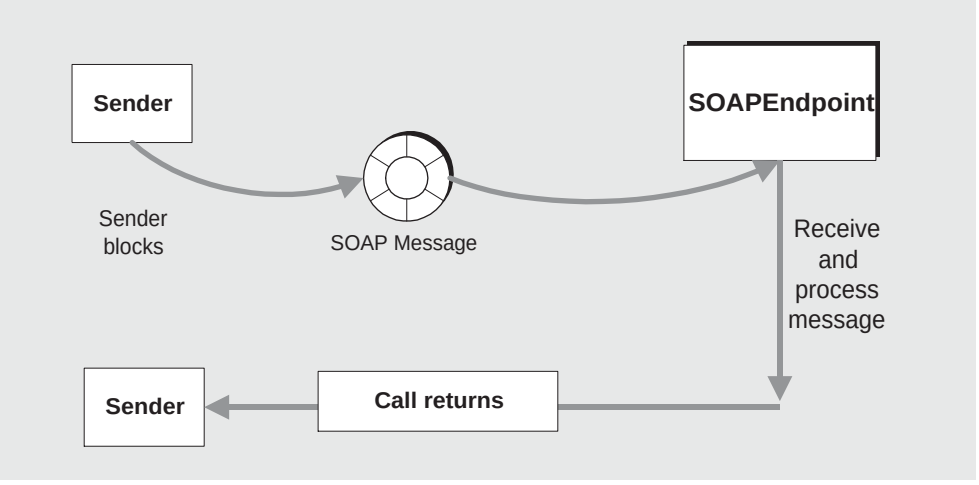
This section provides a brief summary of the programming models used in SOAP messaging using SAAJ.

A SOAP message is sent to an endpoint by way of a point-to-point connection (implemented by the `SOAPConnection` class).

Point-to-Point Connections

You use point-to-point connections to establish a request-reply messaging model. The request-reply model is illustrated in Figure 5-6.

Figure 5-6 Request-Reply Messaging



Using this model, the client does the following:

- Creates an endpoint that specifies the URL that will be passed to the `SOAPConnection.call` method that sends the message.

  See "Endpoint" on page 114 for a discussion of the different ways of creating an endpoint.

- Creates a SOAPConnection factory and obtains a SOAP connection.
- Creates a message factory and uses it to create a SOAP message.

- Creates a name for the content of the message and adds the content to the message.
- Uses the `SOAPConnection.call` method to send the message.

It is assumed that the client will ignore the `SOAPMessage` object returned by the call method because the only reason this object is returned is to unblock the client.

The SOAP service listening for a request-reply message uses a `ReqRespListener` object to receive messages.

For a detailed example of a client that does point-to-point messaging, see "Writing a SOAP Client" on page 120.

Working with Attachments

If a message contains any data that is not XML, you must add it to the message as an attachment. A message can have any number of attachment parts. Each attachment part can contain anything from plain text to image files.

To create an attachment, you must create a URL object that specifies the location of the file that you want to attach to the SOAP message. You must also create a data handler that will be used to interpret the data in the attachment. Finally, you need to add the attachment to the SOAP message.

To create and add an attachment part to the message, you need to use the JavaBeans Activation Framework (JAF) API. This API allows you to determine the type of an arbitrary piece of data, encapsulate access to it, discover the operations available on it, and activate a bean that can perform these operations. You must include the `activation.jar` library in your application code in order to work with the JavaBeans Activation Framework.

➤ **To Create and Add an Attachment**

1. Create a URL object and initialize it to contain the location of the file that you want to attach to the SOAP message.

   ```
   URL url = new URL("http://wombats.com/img.jpg");
   ```

2. Create a data handler and initialize it with a default handler, passing the URL as the location of the data source for the handler.

   ```
   DataHandler dh = new DataHandler(url);
   ```

3. Create an attachment part that is initialized with the data handler containing the URL for the image.

   ```
   AttachmentPart ap1 = message.createAttachmentPart(dh);
   ```

4. Add the attachment part to the SOAP message.

   ```
   myMessage.addAttachmentPart(ap1);
   ```

After creating the attachment and adding it to the message, you can send the message in the usual way.

If you are using JMS to send the message, you *can* use the `SOAPMessageIntoJMSMessage` conversion utility to convert a SOAP message that has an attachment into a JMS message that you can send to a JMS queue or topic using Message Queue.

Exception and Fault Handling

A SOAP application can use two error reporting mechanisms: SOAP exceptions and SOAP faults:

- Use a SOAP exception to handle errors that occur on the client side during the generation of the soap request or the unmarshalling of the response.
- Use a SOAP fault to handle errors that occur on the server side when unmarshalling the request, processing the message, or marshalling the response. In response to such an error, server-side code should create a SOAP message that contains a fault element, rather than a body element, and then it should send that SOAP message back to the originator of the message. If the message receiver is not the ultimate destination for the message, it should identify itself as the `soapactor` so that the message sender knows where the error occurred. For additional information, see "Handling SOAP Faults" on page 126.

Writing a SOAP Client

The following steps show the calls you have to make to write a SOAP client for point-to-point messaging.

1. Get an instance of a `SOAPConnectionFactory`:

   ```
   SOAPConnectionFactory myFct = SOAPConnectionFactory.newInstance();
   ```

2. Get a SOAP connection from the `SOAPConnectionFactory` object:

   ```
   SOAPConnection myCon = myFct.createConnection();
   ```

   The `myCon` object that is returned will be used to send the message.

3. Get a `MessageFactory` object to create a message:

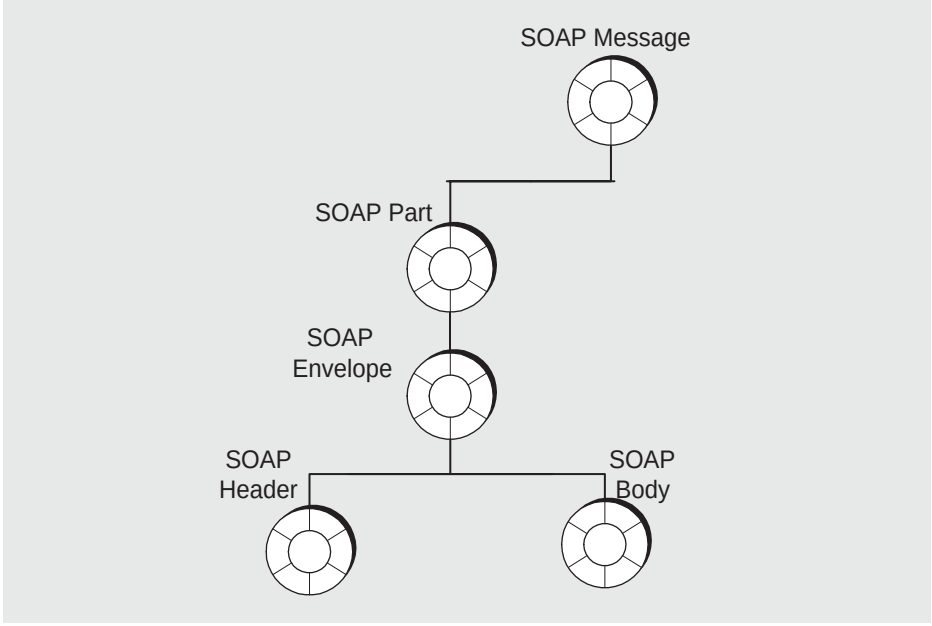
```
MessageFactory myMsgFct = MessageFactory.newInstance();
```

4. Use the message factory to create a message:

```
SOAPMessage message = myMsgFct.createMessage();
```

The message that is created has all the parts that are shown in Figure 5-7.

Figure 5-7 SOAP Message Parts



At this point, the message has no content. To add content to the message, you need to create a SOAP body element, define a name and content for it, and then add it to the SOAP body.

Remember that to access any part of the message, you need to traverse the tree, calling a `get` method on the parent element to obtain the child. For example, to reach the SOAP body, you start by getting the SOAP part and SOAP envelope:

```
SOAPPart mySPart = message.getSOAPPart();
SOAPEnvelope myEnvp = mySPart.getEnvelope();
```

5. Now, you can get the body element from the `myEnvp` object:

   ```
   SOAPBody body = myEnvp.getBody();
   ```

   The children that you will add to the body element define the content of the message. (You can add content to the SOAP header in the same way.)

6. When you add an element to a SOAP body (or header), you must first create a name for it by calling the `envelope.createName` method. This method returns a `Name` object, which you must then pass as a parameter to the method that creates the body element (or the header element).

   ```
   Name bodyName = envelope.createName("GetLastTradePrice", "m",
                                       "http://eztrade.com")

   SOAPBodyElement gltp = body.addBodyElement(bodyName);
   ```

7. Now create another body element to add to the `gltp` element:

   ```
   Name myContent = envelope.createName("symbol");

   SOAPElement mySymbol = gltp.addChildElement(myContent);
   ```

8. And now you can define data for the body element `mySymbol`:

   ```
   mySymbol.addTextNode("SUNW");
   ```

   The resulting SOAP message object is equivalent to this XML scheme:

   ```
   <SOAP-ENV: Envelope
       xmlns:SOAPENV="http://schemas.xmlsoap.org/soap/envelope/">
       <SOAP-ENV:Body>
           <m:GetLastTradePrice xmlns:m="http://eztrade.com">
               <symbol>SUNW</symbol>
           </m:GetLastTradePrice>
       </SOAP-ENV:Body>
   </SOAP-ENV: Envelope>
   ```

9. Every time you send a message or write to it, the message is automatically saved. However if you change a message you have received or one that you have already sent, this would be the point when you would need to update the message by saving all your changes. For example:

   ```
   message.saveChanges();
   ```

10. Before you send the message, you must create a `URLEndpoint` object with the URL of the endpoint to which the message is to be sent. (If you use a profile that adds addressing information to the message header, you do not need to do this.)

```
URLEndpoint endPt = new
                        URLEndpoint("http://eztrade.com//quotes");
```

11. Now, you can send the message:

```
SOAPMessage reply = myCon.call(message, endPt);
```

    The reply message (`reply`) is received on the same connection.

12. Finally, you need to close the `SOAPConnection` object when it is no longer needed:

```
myCon.close();
```

Writing a SOAP Service

A SOAP service represents the final recipient of a SOAP message and should currently be implemented as a servlet. You can write your own servlet or you can extend the `JAXMServlet` class, which is furnished in the `soap.messaging` package for your convenience. This section describes the task of writing a SOAP service based on the `JAXMServlet` class.

Your servlet must implement either the `ReqRespListener` or `OneWayListener` interfaces. The difference between these two is that `ReqRespListener` requires that you return a reply.

Using either of these interfaces, you must implement a method called `onMessage(SOAPMsg)`. `JAXMservlet` will call `onMessage` after receiving a message using the `HTTP POST` method, which saves you the work of implementing your own `doPost()` method to convert the incoming message into a SOAP message.

Code Example 5-4 shows the basic structure of a SOAP service that uses the `JAXMServlet` utility class.

Code Example 5-4 Skeleton Message Consumer

```
public class MyServlet extends JAXMServlet implements
                                ReqRespListener
{
    public SOAPMessage onMessage(SOAP Message msg)
    { //Process message here
    }
}
```

Code Example 5-5 shows a simple ping message service:

Code Example 5-5 A Simple Ping Message Service

```
public class SOAPEchoServlet extends JAXMServlet
                         implements ReqRespListener{

    public SOAPMessage onMessage(SOAPMessage mySoapMessage) {
    return mySoapMessage
    }
}
```

Table 5-3 describes the methods that the JAXM servlet uses. If you were to write your own servlet, you would need to provide methods that performed similar work. In extending `JAXMServlet`, you may need to override the `Init` method and the `SetMessageFactory` method; you *must* implement the `onMessage` method.

Table 5-3 `JAXMServlet` Methods

Method	Description
`void init (ServletConfig)`	Passes the `ServletConfig` object to its parent's constructor and creates a default `messageFactory` object. If you want incoming messages to be constructed according to a certain profile, you must call the `SetMessageFactory` method and specify the profile it should use in constructing SOAP messages.
`void doPost (HTTPRequest, HTTPResponse`	Gets the body of the HTTP request and creates a SOAP message according to the default or specified MessageFactory profile. Calls the `onMessage()` method of an appropriate listener, passing the SOAP message as a parameter. It is recommended that you do not override this method.

Table 5-3 `JAXMServlet` Methods *(Continued)*

Method	Description
`void setMessageFactory (MessageFactory)`	Sets the `MessageFactory` object. This is the object used to create the SOAP message that is passed to the `onMessage` method.
`MimeHeaders getHeaders (HTTPRequest)`	Returns a `MimeHeaders` object that contains the headers in the given HTTPRequest object.
`void putHeaders (mimeHeaders, HTTPresponse)`	Sets the given `HTTPResponse` object with the headers in the given `MimeHeaders` object
`onMessage (SOAPMesssage)`	User-defined method that is called by the servlet when the SOAP message is received. Normally this method needs to disassemble the SOAP message passed to it and to send a reply back to the client (if the servlet implements the `ReqRespListener` interface.)

Disassembling Messages

The `onMessage` method needs to disassemble the SOAP message that is passed to it by the servlet and process its contents in an appropriate manner. If there are problems in the processing of the message, the service needs to create a SOAP fault object and send it back to the client as described in "Handling SOAP Faults" on page 126.

Processing the SOAP message may involve working with the headers as well as locating the body elements and dealing with their contents. The following code sample shows how you might disassemble a SOAP message in the body of your `onMessage` method. Basically, you need to use a Document Object Model (DOM) API to parse through the SOAP message.

See `http://xml.coverpages.org/dom.html` for more information about the DOM API.

Code Example 5-6 Processing a SOAP Message

```
{http://xml.coverpages.org/dom.html
    SOAPEnvelope env = reply.getSOAPPart().getEnvelope();
    SOAPBody sb = env.getBody();

    // create Name object for XElement that we are searching for
    Name ElName = env.createName("XElement");

    //Get child elements with the name XElement
    Iterator it = sb.getChildElements( ElName );

    //Get the first matched child element.
    //We know there is only one.
    SOAPBodyElement sbe = (SOAPBodyElement) it.next();

    //Get the value for XElement
    MyValue =   sbe.getValue();
}
```

Handling Attachments

A SOAP message may have attachments. For sample code that shows you how to create and add an attachment, see Code Example 5-7 on page 136. For sample code that shows you how to receive and process an attachment, see Code Example 5-8 on page 138.

In handling attachments, you will need to use the Java Activation Framework API. See `http://java.sun.com/products/javabeans/glasgow/jaf.html` for more information.

Replying to Messages

In replying to messages, you are simply taking on the client role, now from the server side.

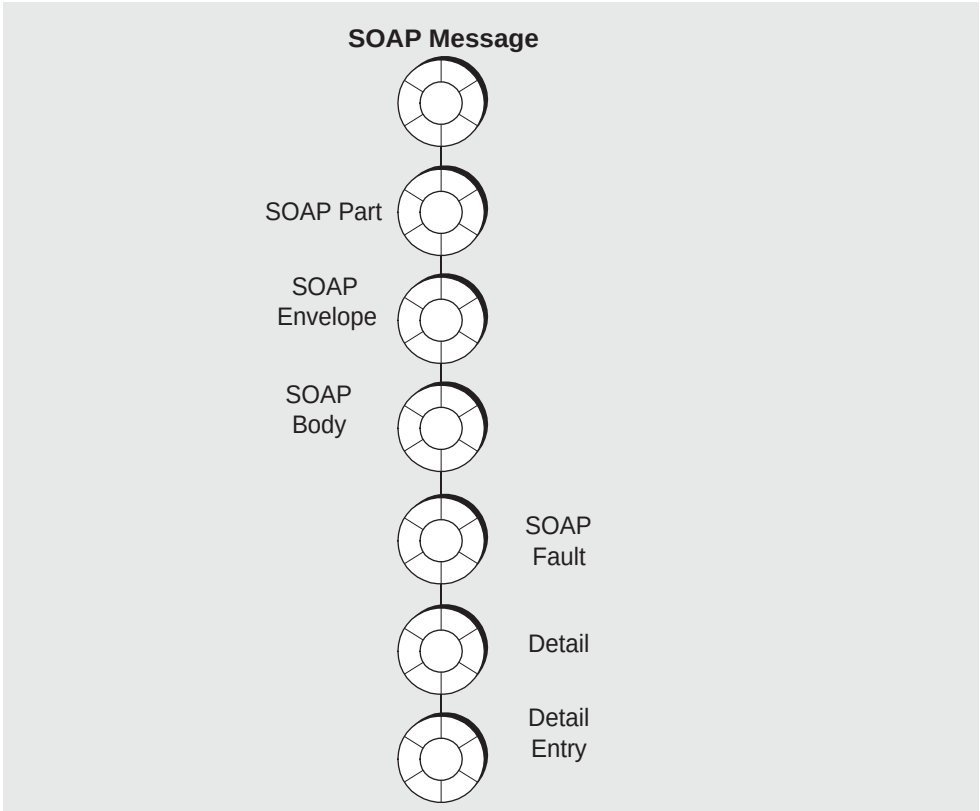
Handling SOAP Faults

Server-side code must use a SOAP fault object to handle errors that occur on the server side when unmarshalling the request, processing the message, or marshalling the response. The `SOAPFault` interface extends the `SOAPBodyElement` interface.

SOAP messages have a specific element and format for error reporting on the server side: a SOAP message body can include a SOAP fault element to report errors that happen during the processing of a request. Created on the server side and sent from the server back to the client, the SOAP message containing the `SOAPFault` object reports any unexpected behavior to the originator of the message.

Within a SOAP message object, the SOAP fault object is a child of the SOAP body, as shown in Figure 5-8. Detail and detail entry objects are only needed if one needs to report that the body of the received message was malformed or contained inappropriate data. In such a case, the detail entry object is used to describe the malformed data.

Figure 5-8 SOAP Fault Element



The SOAP Fault element defines the following four sub-elements:

- `faultcode`

  A code (qualified name) that identifies the error. The code is intended for use by software to provide an algorithmic mechanism for identifying the fault. Predefined fault codes are listed in Table 5-4 on page 128. This element is required.

- `faultstring`

  A string that describes the fault identified by the fault code. This element is intended to provide an explanation of the error that is understandable to a human. This element is required.

- `faultactor`

  A URI specifying the source of the fault: the actor that caused the fault along the message path. This element is not required if the message is sent to its final destination without going through any intermediaries. If a fault occurs at an intermediary, then that fault must include a `faultactor` element.

- `detail`

  This element carries specific information related to the Body element. It must be present if the contents of the Body element could not be successfully processed. Thus, if this element is missing, the client should infer that the body element was processed. While this element is not required for any error except a malformed payload, you can use it in other cases to supply additional information to the client.

Predefined Fault Codes

The SOAP specification lists four predefined `faultcode` values. The namespace identifier for these is `http://schemas.xmlsoap.org/soap/envelope/`.

Table 5-4 SOAP Faultcode Values

Faultcode Name	Meaning
`VersionMismatch`	The processing party found an invalid namespace for the SOAP envelope element; that is, the namespace of the SOAP envelope element was not `http://schemas.xmlsoap.org/soap/envelope/`.
`MustUnderstand`	An immediate child element of the SOAP Header element was either not understood or not appropriately processed by the recipient. This element's `mustUnderstand` attribute was set to 1 (true).

Table 5-4 SOAP Faultcode Values *(Continued)*

Faultcode Name	Meaning
`Client`	The message was incorrectly formed or did not contain the appropriate information. For example, the message did not have the proper authentication or payment information. The client should interpret this code to mean that the message must be changed before it is sent again. If this is the code returned, the `SOAPFault` object should probably include a `detailEntry` object that provides additional information about the malformed message.
`Server`	The message could not be processed for reasons that are not connected with its content. For example, one of the message handlers could not communicate with another message handler that was upstream and did not respond. Or, the database that the server needed to access is down. The client should interpret this error to mean that the transmission could succeed at a later point in time.

These standard fault codes represent classes of faults. You can extend these by appending a period to the code and adding an additional name. For example: you could define a `Server.OutOfMemory` code, a `Server.Down` code, etc.

Defining a SOAP Fault

Using SAAJ you can specify the value for `faultcode`, `faultstring`, and `faultactor` using methods of the `SOAPFault` object. The following code creates a SOAP fault object and sets the `faultcode`, `faultstring`, and `faultactor` attributes:

```
SOAPFault fault;
reply = factory.createMessage();
envp = reply.getSOAPPart().getEnvelope(true);
someBody = envp.getBody();
fault = someBody.addFault():
fault.setFaultCode("Server");
fault.setFaultString("Some Server Error");
fault.setFaultActor(http://xxx.me.com/list/endpoint.esp/)
reply.saveChanges();
```

The server can return this object in its reply to an incoming SOAP message in case of a server error.

The next code sample shows how to define a detail and detail entry object. Note that you must create a name for the detail entry object.

```
SOAPFault fault = someBody.addFault();
fault.setFaultCode("Server");
fault.setFaultActor("http://foo.com/uri");
fault.setFaultString ("Unkown error");
Detail myDetail = fault.addDetail();
detail.addDetailEntry(envelope.createName("125detail", "m",
    "Someuri")).addTextNode("the message cannot contain
    the string //");
reply.saveChanges();
```

Integrating SOAP and Message Queue

This section explains how you can send, receive, and process a JMS message that contains a SOAP payload.

Message Queue provides a utility to help you send and receive SOAP messages using the JMS API. With the support it provides, you can convert a SOAP message into a JMS message and take advantage of the reliable messaging service offered by Message Queue. You can then convert the message back into a SOAP message on the receiving side and use SAAJ to process it.

To send, receive, and process a JMS message that contains a SOAP payload, you must do the following:

- Import the library `com.sun.messaging.xml.MessageTransformer`. This is the utility whose methods you will use to convert SOAP messages to JMS messages and vice versa.
- Before you transport a SOAP message, you must call the `MessageTransformer.SOAPMessageIntoJMSMessage` method. This method transforms the SOAP message into a JMS message. You then send the resulting JMS message as you would a normal JMS message. For programming simplicity, it would be best to select a destination that is dedicated to receiving SOAP messages. That is, you should create a particular queue or topic as a destination for your SOAP message and then send only SOAP messages to this destination.

```
Message myMsg= MessageTransformer.SOAPMessageIntoJMSMessage
                          (SOAPMessage, Session);
```

The `Session` argument specifies the session to be used in producing the `Message`.

- On the receiving side, you get the JMS message containing the SOAP payload as you would a normal JMS message. You then call the `MessageTransformer.SOAPMessageFromJMSMessage` utility to extract the SOAP message, and then use SAAJ to disassemble the SOAP message and do any further processing. For example, to obtain the SOAPMessage make a call like the following

  ```
  SOAPMessage myMsg= MessageTransformer.SOAPMessageFromJMSMessage
                            (Message, MessageFactory);
  ```

  The `MessageFactory` argument specifies a message factory that the utility should use to construct the `SOAPMessage` from the given JMS `Message`.

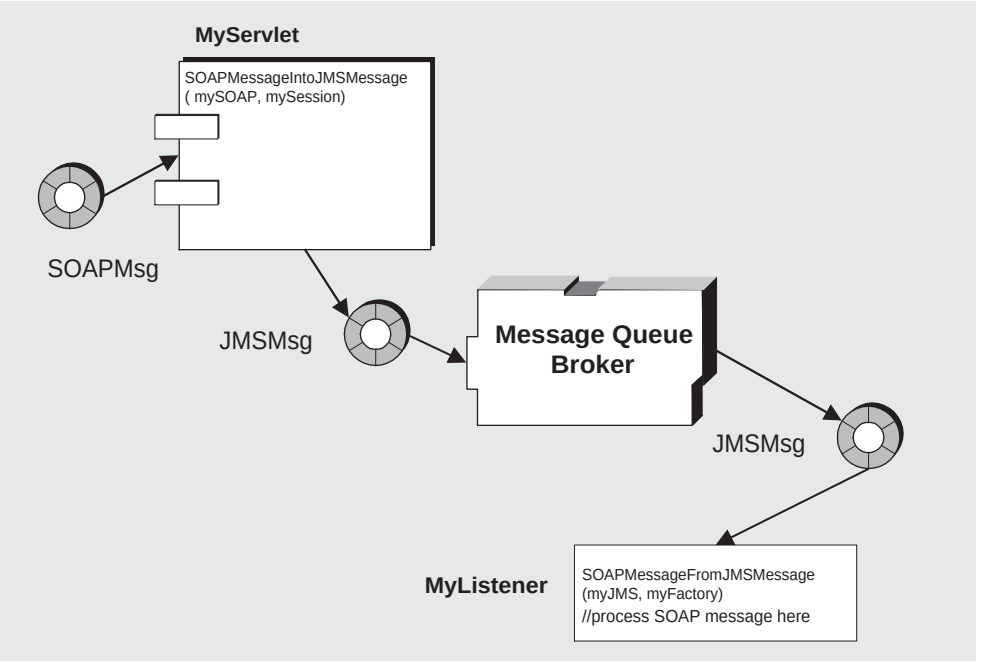
The following sections offer several use cases and code examples to illustrate this process.

Example 1: Deferring SOAP Processing

In the first example, illustrated in Figure 5-9, an incoming SOAP message is received by a servlet. After receiving the SOAP message, the servlet `MyServlet` uses the `MessageTransformer` utility to transform the message into a JMS message, and (reliably) forwards it to an application that receives it, turns it back into a SOAP message, and processes the contents of the SOAP message.

For information on how the servlet receives the SOAP message, see "Writing a SOAP Service" on page 123.

Figure 5-9 Deferring SOAP Processing



➤ **To Transform the SOAP Message into a JMS Message and Send the JMS Message**

1. Instantiate a `ConnectionFactory` object and set its attribute values, for example:

```
QueueConnectionFactory myQConnFact =
        new com.sun.messaging.QueueConnectionFactory();
```

2. Use the `ConnectionFactory` object to create a `Connection` object.

```
QueueConnection myQConn =
         myQConnFact.createQueueConnection();
```

3. Use the `Connection` object to create a `Session` object.

```
QueueSession myQSess = myQConn.createQueueSession(false,
         Session.AUTO_ACKNOWLEDGE);
```

4. Instantiate a Message Queue Destination administered object corresponding to a physical destination in the Message Queue message service. In this example, the administered object is `mySOAPQueue` and the physical destination to which it refers is `myPSOAPQ`.

   ```
   Queue mySOAPQueue = new com.sun.messaging.Queue("myPSOAPQ");
   ```

5. Use the `MessageTransformer` utility, as shown, to transform the SOAP message into a JMS message. For example, given a SOAP message named `MySOAPMsg`,

   ```
   Message MyJMS = MessageTransformer.SOAPMessageIntoJMSMessage
                                       (MySOAPMsg, MyQSess);
   ```

6. Create a `QueueSender` message producer.

   This message producer, associated with `mySOAPQueue`, is used to send messages to the queue destination named `myPSOAPQ`.

   ```
   QueueSender myQueueSender = myQSess.createSender(mySOAPQueue);
   ```

7. Send a message to the queue.

   ```
   myQueueSender.send(myJMS);
   ```

➤ **To Receive the JMS Message, Transform it into a SOAP Message, and Process It**

1. Instantiate a `ConnectionFactory` object and set its attribute values.

   ```
   QueueConnectioFactory myQConnFact = new
          com.sun.messaging.QueueConnectionFactory();
   ```

2. Use the `ConnectionFactory` object to create a `Connection` object.

   ```
   QueueConnection myQConn = myQConnFact.createQueueConnection();
   ```

3. Use the `Connection` object to create one or more `Session` objects.

   ```
   QueueSession myRQSess = myQConn.createQueueSession(false,
           session.AUTO_ACKNOWLEDGE);
   ```

4. Instantiate a `Destination` object and set its name attribute.

   ```
   Queue myRQueue = new com.sun.messaging.Queue("mySOAPQ");
   ```

5. Use a `Session` object and a `Destination` object to create any needed `MessageConsumer` objects.

   ```
   QueueReceiver myQueueReceiver =
      myRQSess.createReceiver(myRQueue);
   ```

6. If needed, instantiate a `MessageListener` object and register it with a `MessageConsumer` object.
7. Start the `QueueConnection` you created in Step 2. Messages for consumption by a client can only be delivered over a connection that has been started.

   ```
   myQConn.start();
   ```

8. Receive a message from the queue

   The code below is an example of a synchronous consumption of messages.

   ```
   Message myJMS = myQueueReceiver.receive();
   ```

9. Use the Message Transformer to convert the JMS message back to a SOAP message.

   ```
   SOAPMessage MySoap =
           MessageTransformer.SOAPMessageFromJMSMessage
              (myJMS, MyMsgFactory);
   ```

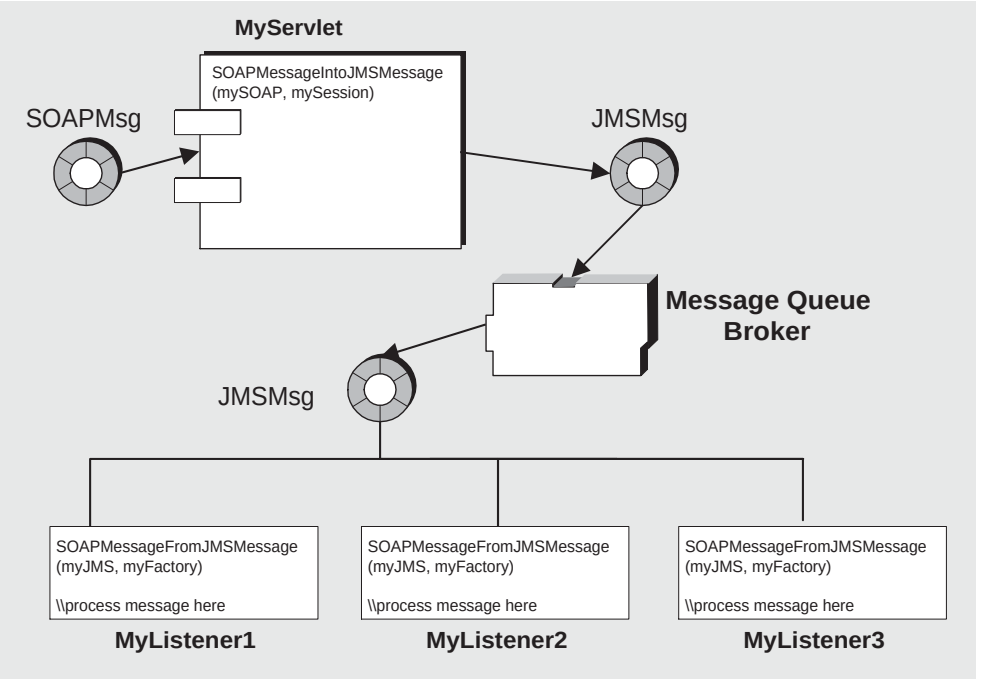
   If you specify null for the `MessageFactory` argument, the default Message Factory is used to construct the SOAP Message.

10. Disassemble the SOAP message in preparation for further processing. See "The SOAP Message Object" on page 107 for information.

Example 2: Publishing SOAP Messages

In the next example, illustrated in Figure 5-10, an incoming SOAP message is received by a servlet. The servlet packages the SOAP message as a JMS message and (reliably) forwards it to a topic. Each application that subscribes to this topic, receives the JMS message, turns it back into a SOAP message, and processes its contents.

Figure 5-10 Publishing a SOAP Message



The code that accomplishes this is exactly the same as in the previous example, except that instead of sending the JMS message to a queue, you send it to a topic. For an example of publishing a SOAP message using Message Queue, see Code Example 5-7 on page 136.

Code Samples

This section includes and describes two code samples: one that sends a JMS message with a SOAP payload, and another that receives the JMS/SOAP message and processes the SOAP message.

Code Example 5-7 illustrates the use of the JMS API, the SAAJ API, and the JAF API to send a SOAP message with attachments as the payload to a JMS message. The code shown for the `SendSOAPMessageWithJMS` includes the following methods:

- a constructor that calls the `init` method to initialize all the JMS objects required to publish a message

- a `send` method that creates the SOAP message and an attachment, converts the SOAP message into a JMS message, and publishes the JMS message
- a `close` method that closes the connection
- a `main` method that calls the send and close methods

Code Example 5-7 Sending a JMS Message with a SOAP Payload

```
//Libraries needed to build SOAP message
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPPart;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.AttachmentPart;
import javax.xml.soap.Name

//Libraries needed to work with attachments (Java Activation Framework API)
import java.net.URL;
import javax.activation.DataHandler;

//Libraries needed to convert the SOAP message to a JMS message and to send it
import com.sun.messaging.xml.MessageTransformer;
import com.sun.messaging.BasicConnectionFactory;

//Libraries needed to set up a JMS connection and to send a message
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicConnection;
import javax.jms.JMSException;
import javax.jms.Session;
import javax.jms.Message;
import javax.jms.TopicSession;
import javax.jms.Topic;
import javax.jms.TopicPublisher;

//Define class that sends JMS message with SOAP payload
public class SendSOAPMessageWithJMS{

    TopicConnectionFactory tcf = null;
    TopicConnection tc = null;
    TopicSession session = null;
    Topic topic = null;
    TopicPublisher publisher = null;

//default constructor method
public SendSOAPMessageWithJMS(String topicName){
    init(topicName);
    }

//Method to nitialize JMS Connection, Session, Topic, and Publisher
public void init(String topicName) {
    try {
```

Code Example 5-7 Sending a JMS Message with a SOAP Payload *(Continued)*

```
    tcf = new com.sun.messaging.TopicConnectionFactory();
    tc = tcf.createTopicConnection();
    session = tc.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
    topic = session.createTopic(topicName);
    publisher = session.createPublisher(topic);
    }

//Method to create and send the SOAP/JMS message
public void send() throws Exception{
    MessageFactory mf = MessageFactory.newInstance(); //create default factory
    SOAPMessage soapMessage=mfcreateMessage(); //create SOAP message object
    SOAPPart soapPart = soapMessage.getSOAPPart();//start to drill down to body
    SOAPEnvelope soapEnvelope = soapPart.getEnvelope(); //first the envelope
    SOAPBody soapBody = soapEnvelope.getBody();
    Name myName = soapEnvelope.createName("HelloWorld", "hw",
                          http;//www.sun.com/imq'); //name for body element
    SOAPElement element = soapBody.addChildElement(myName); //add body element
    element.addTextNode("Welcome to SUnOne Web Services."); //add text value

    //Create an attachment with the Java Framework Activation API
    URL url = new URL("http://java.sun.com/webservices/");
    DataHandler dh = new DataHnadler (url);
    AttachmentPart ap = soapMessage.createAttachmentPart(dh);

    //Set content type and ID
    ap.setContentType("text/html");
    ap.setContentID('cid-001");

    //Add attachment to the SOAP message
    soapMessage.addAttachmentPart(ap);
    soapMessage.saveChanges();

    //Convert SOAP to JMS message.
    Message m = MessageTransformer.SOAPMessageIntoJMSMessage(soapMessage,
                          session);

//Publish JMS message
    publisher.publish(m);

//Close JMS connection
    public void close() throws JMSException {
    tc.close();
    }

//Main program to send SOAP message with JMS
public static void main (String[] args) {
    try {
    String topicName = System.getProperty("TopicName");
    if(topicName == null) {
        topicName = "test";
    }

    SendSOAPMEssageWithJMS ssm = new SendSOAPMEssageWithJMS(topicName);
```

Code Example 5-7 Sending a JMS Message with a SOAP Payload *(Continued)*

```
    ssm.send();
    ssm.close();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    }
}
```

Code Example 5-8 illustrates the use of the JMS API, SAAJ, and the DOM API to receive a SOAP message with attachments as the payload to a JMS message. The code shown for the `ReceiveSOAPMessageWithJMS` includes the following methods:

- A constructor that calls the `init` method to initialize all the JMS objects needed to receive a message.
- An `onMessage` method that delivers the message and which is called by the listener. The `onMessage` method also calls the message transformer utility to convert the JMS message into a SOAP message and then uses SAAJ to process the SOAP body and uses SAAJ and the DOM API to process the message attachments.
- A `main` method that initializes the `ReceiveSOAPMessageWithJMS` class.

Code Example 5-8 Receiving a JMS Message with a SOAP Payload

```
//Libraries that support SOAP processing
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.AttachmentPart

//Library containing the JMS to SOAP transformer
import com.sun.messaging.xml.MessageTransformer;

//Libraries for JMS messaging support
import com.sun.messaging.TopicConnectionFactory

//Interfaces for JMS messaging
import javax.jms.MessageListener;
import javax.jms.TopicConnection;
import javax.jms.TopicSession;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.Topic;
import javax.jms.JMSException;
import javax.jms.TopicSubscriber
```

Code Example 5-8 Receiving a JMS Message with a SOAP Payload *(Continued)*

```
//Library to support parsing attachment part (from DOM API)
import java.util.iterator;

public class ReceiveSOAPMessageWithJMS implements MessageListener{
    TopicConnectionFactory tcf = null;
    TopicConnection tc = null;
    TopicSession session = null;
    Topic topic = null;
    TopicSubscriber subscriber = null;
    MessageFactory messageFactory = null;

//Default constructor
public ReceiveSOAPMessageWithJMS(String topicName) {
    init(topicName);
}
//Set up JMS connection and related objects
public void init(String topicName){
    try {
    //Construct default SOAP message factory
    messageFactory = MessageFactory.newInstance();

    //JMS set up
    tcf = new. com.sun.messaging.TopicConnectionFactory();
    tc = tcf.createTopicConnection();
    session = tc.createTopicSesstion(false, Session.AUTO_ACKNOWLEDGE);
    topic = session.createTopic(topicName);
    subscriber = session.createSubscriber(topic);
    subscriber.setMessageListener(this);
    tc.start();

    System.out.println("ready to receive SOAP m essages...");
    }catch (Exception jmse){
    jmse.printStackTrace();
    }
    }

//JMS messages are delivered to the onMessage method
public void onMessage(Message message){
    try {
    //Convert JMS to SOAP message
    SOAPMessage soapMessage = MessageTransformer.SOAPMessageFromJMSMessage
                (message, messageFactory);


    //Print attchment counts
    System.out.println("message received! Attachment counts:
                " + soapMessage.countAttachments());

    //Get attachment parts of the SOAP message
    Iterator iterator = soapMessage.getAttachments();
    while (iterator.hasNext()) {
        //Get next attachment
```

Code Example 5-8 Receiving a JMS Message with a SOAP Payload *(Continued)*

```
        AttachmentPart ap = (AttachmentPart) iterator.next();

        //Get content type
        String contentType = ap.getContentType();
        System.out.println("content type: " + conent TYpe);

        //Get content id
        String contentID = ap.getContentID();
        System.out.println("content Id:" + contentId);

        //Check to see if this is text
        if(contentType.indexOf"text")>=0 {
            //Get and print string content if it is a text attachment
            String content = (String) ap.getContent();
            System.outprintln("*** attachment content: " + content);
        }
    }
    }catch (Exception e) {
    e.printStackTrace();
    }
}

//Main method to start sample receiver
public static void main (String[] args){
    try {
    String topicName = System.getProperty("TopicName");
    if( topicName == null) {
        topicName = "test";
    }
    ReceiveSOAPMessageWithJMS rsm = new ReceiveSOAPMessageWithJMS(topicName);
    }catch (Exception e) {
    e.printStackTrace();
    }
    }
}
```

Appendix A

Warning Messages and Client Error Codes

This appendix provides reference information for warning messages and for error codes returned by the Message Queue client runtime when it raises a JMS exception.

- A *warning message* is a message output when the MQ Java client runtime experiences a problem that should not occur under normal operating conditions. The message is displayed where the application displays its output. Usually, this is the window from which the application is started. Table A-1 lists Message Queue warning messages.

  In general, a warning message does not cause message loss or affect reliability. issues. But when warning messages appear constantly on the application's console, the user should contact MQ technical support to diagnose the cause of the warning messages.

- *Error codes* and messages are returned by the client runtime when it raises an exception. You can obtain the error code and its corresponding message using the `JMSException.getErrorCode()` method and the `JMSException.getMessage()` method. Table A-2 lists Message Queue error codes.

Note that warning messages and error codes are not defined in the JMS specification, but are specific to each JMS provider. Applications that rely on these error codes in their programming logic are not portable across JMS providers.

Table A-1 Message Queue Warning Message Codes

Code	Message and Description
W2000	**Message** Warning: Received unknown packet: *mq-packet-dump*.
	Cause The Message Queue client runtime received an unrecognized Message Queue packet, where *mq-packet-dump* is replaced with the specific Message Queue packet dump that caused this warning message.
	The Message Queue broker may not be fully compatible with the client runtime version.
W2001	**Message** Warning: pkt not processed, no message consumer:*mq-packet-dum*p.
	Cause The Message Queue client runtime received an unexpected Message Queue acknowledge message. The variable *mq-packet-dump* is replaced with the specific Message Queue packet dump that caused this warning message.
W2003	**Message** Warning: Broker not responding *X* for *Y* seconds. Still trying....
	Cause The Message Queue client runtime has not received a response from the broker for more than 2 minutes (default). In the actual message, the *X* variable is replaced with the Message Queue packet type that the client runtime is waiting for, and the *Y* variable is replaced with the number of seconds that the client runtime has been waiting for the packet.

Table A-2 lists the error codes in numerical order. For each code listed, it supplies the error message and a probable cause.

Each error message returned has the following format:

`[`*`Code`*`]: “`*`Message`* `-cause` *`Root-cause-exception-message`*`.”`

Message text provided for `-cause` is only appended to the message if there is an exception linked to the JMS exception. For example, a JMS exception with error code `C4003` returns the following error message:

```
[C4003]: Error occurred on connection creation [localhost:7676] - cause:
java.net.ConnectException: Connection refused: connect
```

Table A-2 Message Queue Client Error Codes

Code	Message and Description
C4000	**Message** Packet acknowledge failed. **Cause** The client runtime was not able to receive or process the expected acknowledgment sent from the broker.
C4001	**Message** Write packet failed. **Cause** The client runtime was not able to send information to the broker. This might be caused by an underlying network I/O failure or by the JMS connection being closed.
C4002	**Message** Read packet failed. **Cause** The client runtime was not able to process inbound message properly. This might be caused by an underlying network I/O failure.
C4003	**Message** Error occurred on connection creation [*host*, *port*]. **Cause** The client runtime was not able to establish a connection to the broker with the specified host name and port number.
C4004	**Message** An error occurred on connection close. **Cause** The client runtime encountered one or more errors when closing the connection to the broker.
C4005	**Message** Get properties from packet failed. **Cause** The client runtime was not able to retrieve a property object from the Message Queue packet.
C4006	**Message** Set properties to packet failed. **Cause** The client runtime was not able to set a property object in the Message Queue packet.
C4007	**Message** Durable subscription {0} in use. *{0} is replaced with the subscribed destination name.* **Cause** The client runtime was not able to unsubscribe the durable subscriber because it is currently in use by another consumer.
C4008	**Message** Message in read-only mode. **Cause** An attempt was made to write to a JMS Message that is in read-only mode.
C4009	**Message** Message in write-only mode. **Cause** An attempt was made to read a JMS Message that is in write-only mode.
C4010	**Message** Read message failed. **Cause** The client runtime was not able to read the stream of bytes from a `BytesMessage` type message.

Table A-2 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
C4011	**Message** Write message failed.
	Cause The client runtime was not able to write the stream of bytes to a `BytesMessage` type message.
C4012	**Message** message failed.
	Cause The client runtime encountered an error when processing the `reset()` method for a `BytesMessage` or `StreamMessage` type message.
C4013	**Message** Unexpected end of stream when reading message.
	Cause The client runtime reached end-of-stream when processing the readXXX() method for a `BytesMessage` or `StreamMessage` type message.
C4014	**Message** Serialize message failed.
	Cause The client runtime encountered an error when processing the serialization of an object, such as `ObjectMessage.setObject(java.io.Serializable object)`.
C4015	**Message** Deserialize message failed.
	Cause The client runtime encountered an error when processing the deserialization of an object, for example, when processing the method `ObjectMessage.getObject()`.
C4016	**Message** Error occurred during message acknowledgment.
	Cause The client runtime encountered an error during the process of message acknowledgment in a session.
C4017	**Message** Invalid message format.
	Cause The client runtime encountered an error when processing a JMS Message; for example, during data type conversion.
C4018	**Message** Error occurred on request message redeliver.
	Cause The client runtime encountered an error when processing `recover()` or `rollback()` for the JMS session.
C4019	**Message** Destination not found: {0}. *{0} is replaced with the destination name specified in the API parameter.*
	Cause The client runtime was unable to process the API request due to an invalid destination specified in the API, for example, the call `MessageProducer.send (null, message)` raises `JMSException` with this error code and message.

Table A-2 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
C4020	**Message** Temporary destination belongs to a closed connection or another connection - {0}. *{0} is replaced with the temporary destination name specified in the API parameter.*
	Cause An attempt was made to use a temporary destination that is not valid for the message producer.
C4021	**Message** Consumer not found.
	Cause The Message Queue session could not find the message consumer for a message sent from the broker. The message consumer may have been closed by the application or by the client runtime before the message for the consumer was processed.
C4022	**Message** Selector invalid: {0}. *{0} is replaced with the selector string specified in the API parameter.*
	Cause The client runtime was unable to process the JMS API call because the specified selector is invalid.
C4023	**Message** Client unacknowledged messages over system defined limit.
	Cause The client runtime raises a `JMSException` with this error code and message if unacknowledged messages exceed the system defined limit in a `CLIENT_ACKNOWLEDGE` session.
C4024	**Message** The session is not transacted.
	Cause An attempt was made to use a transacted session API in a non-transacted session. For example, calling the methods `commit()` or `rollback` in a `AUTO_ACKNOWLEDGE` session.
C4025	**Message** Cannot call this method from a transacted session.
	Cause An attempt was made to call the `Session.recover()` method from a transacted session.
C4026	**Message** Client non-committed messages over system defined limit.
	Cause The client runtime raises a `JMSException` with this error code and message if non committed messages exceed the system -defined limit in a transacted session.
C4027	**Message** Invalid transaction ID: {0}. *{0} is replaced with the Message Queue internal transaction ID.*
	Cause An attempt was made to commit or rollback a transacted session with a transaction ID that is no longer valid.

Table A-2 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
C4028	**Message** Transaction ID {0} in use. *{0} is replaced with the Message Queue internal transaction ID.*
	Cause The internal transaction ID is already in use by the system. An application should not receive a `JMSException` with this error code under normal operations.
C4029	**Message** Invalid session for ServerSession.
	Cause An attempt was made to use an invalid JMS session for the `ServerSession` object, for example, no message listener was set for the session.
C4030	**Message** Illegal maxMessages value for ServerSession: {0}. *{0} was replaced with `maxMessages` value used by the application.*
	Cause The configured `maxMessages` value for `ServerSession` is less than 0.
C4031	**Message** MessageConsumer and ServerSession session conflict.
	Cause An attempt was made to create a message consumer for a session already used by a `ServerSession` object.
C4032	**Message** Can not use receive() when message listener was set.
	Cause An attempt was made to do a synchronous receive with an asynchronous message consumer.
C4033	**Message** Authentication type does not match: {0} and {1}. *{0} is replaced with the authentication type used by the client runtime.* *{1} is replaced with the authentication type requested by the broker.*
	Cause The authentication type requested by the broker does not match the authentication type in use by the client runtime.
C4034	**Message** Illegal authentication state.
	Cause The authentication hand-shake failed between the client runtime and the broker.
C4035	**Message** Received `AUTHENTICATE_REQUEST` status code `FORBIDDEN`.
	Cause The client runtime authentication to the broker failed.
C4036	**Message** A server error occurred.
	Cause A generic error code indicating that the client's requested operation to the broker failed.
C4037	**Message** Server unavailable or server timeout.
	Cause The client runtime was unable to establish a connection to the broker.

Table A-2 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
C4038	**Message** [4038] - cause: {0} *{0} is replaced with a root cause exception message.*
	Cause The client runtime caught an exception thrown from the JVM. The client runtime throws `JMSException` with the "root cause exception" set as the linked exception.
C4039	**Message** Cannot delete destination.
	Cause The client runtime was unable to delete the specified temporary destination. Please see `TemporaryTopic.delete()` and `TemporaryQueue.delete()` API Javadoc for constraints on deleting a temporary destination.
C4040	**Message** Invalid ObjectProperty type.
	Cause An attempt was made to set a non-primitive Java object as a JMS message property. Please see `Message.setObjectProperty()` API Javadoc for valid object property types.
C4041	**Message** Reserved word used as property name - {0}.
	Cause An attempt was made to use a reserved word, defined in the JMS Message API Javadoc, as the message property name, for example, `NULL`, `TRUE`, `FALSE`.
C4042	**Message** Illegal first character of property name - {0} *{0} is replaced with the illegal character.*
	Cause An attempt was made to use a property name with an illegal first character. See JMS Message API Javadoc for valid property names.
C4043	**Message** Illegal character used in property name - {0} *{0} is replaced with the illegal character used.*
	Cause An attempt was made to use a property name containing an illegal character. See JMS Message API Javadoc for valid property names.
C4044	**Message** Browser timeout.
	Cause The queue browser was unable to return the next available message to the application within the system's predefined timeout period.
C4045	**Message** No more elements.
	Cause In `QueueBrowser`, the enumeration object has reached the end of element but `nextElement()` is called by the application.
C4046	**Message** Browser closed.
	Cause An attempt was made to use `QueueBrowser` methods on a closed `QueueBrowser` object.

Table A-2 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
C4047	**Message** Operation interrupted.
	Cause `ServerSession` was interrupted. The client runtime throws `RuntimeException` with the above exception message when it is interrupted in the `ServerSession`.
C4048	**Message** ServerSession is in progress.
	Cause Multiple threads attempted to operate on a server session concurrently.
C4049	**Message** Can not call Connection.close(), stop(), etc from message listener.
	Cause An attempt was made to call `Connection.close()`, `...stop()`, etc from a message listener.
C4050	**Message** Invalid destination name - {0} *{0} is replaced with the invalid destination name used.*
	Cause An attempt was made to use an invalid destination name, for example, `NULL`.
C4051	**Message** Invalid delivery parameter. {0} : {1} *{0} is replaced with delivery parameter name, such as "DeliveryMode".* *{1} is replaced with delivery parameter value used by the application.*
	Cause An attempt was made to use invalid JMS delivery parameters in the API, for example, values other than `DeliveryMode.NON_PERSISTENT` or `DeliveryMode.PERSISTENT` were used to specify the delivery mode.
C4052	**Message** Client ID is already in use - {0} *{0} is replaced with the client ID that is already in use.*
	Cause An attempt was made to set a client ID to a value that is already in use by the system.
C4053	**Message** Invalid client ID - {0} *{0} is replaced with the client ID used by the application.*
	Cause An attempt was made to use an invalid client ID, for example, `null` or empty client ID.
C4054	**Message** Can not set client ID, invalid state.
	Cause An attempt was made to set a connection's client ID at the wrong time or when it has been administratively configured.
C4055	**Message** Resource in conflict. Concurrent operations on a session.
	Cause An attempt was made to concurrently operate on a session with multiple threads.
C4056	**Message** Received goodbye message from broker.
	Cause A Message Queue client received a `GOOD_BYE` message from broker.

Table A-2 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
C4057	**Message** No username or password. **Cause** An attempt was made to use a null object as a user name or password for authentication.
C4058	**Message** Cannot acknowledge message for closed consumer. **Cause** An attempt was made to acknowledge message(s) for a closed consumer.
C4059	**Message** Cannot perform operation, session is closed. **Cause** An attempt was made to call a method on a closed session.
C4060	**Message** Login failed: {0} `{0} message was replaced with user name.` **Cause** Login with the specified user name failed.
C4061	**Message** Connection recovery failed, cannot recover connection. **Cause** The client runtime was unable to recover the connection due to internal error.
C4062	**Message** Cannot perform operation, connection is closed. **Cause** An attempt was made to call a method on a closed connection.
C4063	**Message** Cannot perform operation, consumer is closed. **Cause** An attempt was made to call a method on a closed message consumer.
C4064	**Message** Cannot perform operation, producer is closed. **Cause** An attempt was made to call a method on a closed message producer.
C4065	**Message** Incompatible broker version encountered. Client version {0}.Broker version {1} *{0} is replaced with client version number.* *{1} is replaced with broker version number.* **Cause** An attempt was made to connect to a broker that is not compatible with the client version.
C4066	**Message** Invalid or empty Durable Subscription Name was used: {0} `{0} is replaced with the durable subscription name used by the application.` **Cause** An attempt was made to use a null or empty string to specify the name of a durable subscription.
C4067	**Message** Invalid session acknowledgment mode: {0} *{0} is replaced with the acknowledge mode used by the application.* **Cause** An attempt was made to use a non-transacted session mode that is not defined in the JMS Session API.

Table A-2 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
C4068	**Message** Invalid Destination Classname: {0}.
	Cause An attempt was made to create a message producer or message consumer with an invalid destination class type. The valid class type must be either `Queue` or `Topic`.
C4069	**Message** Cannot commit or rollback on an XASession.
	Cause The application tried to make a `session.commit()` or a `session.rollback()` call in an application server component whose transactions are being managed by the Transaction Manager via the XAResource. These calls are not allowed in this context.
C4070	**Message** Error when converting foreign message.
	Cause The client runtime encountered an error when processing a non-Message Queue JMS message.
C4071	**Message** Invalid method in this domain: {0} `{0} is replaced with the method name used.`
	Cause An attempt was made to use a method that does not belong to the current messaging domain. For example calling `TopicSession.createQueue()` will raise a `JMSException` with this error code and message.
C4072	**Message** Illegal property name - "" or null.
	Cause An attempt was made to use a null or empty string to specify a property name.
C4073	**Message** A JMS destination limit was reached. Too many Subscribers/Receivers for {0} : {1} *{0} is replaced with "Queue" or "Topic"* *{1} is replaced with the destination name.*
	Cause The client runtime was unable to create a message consumer for the specified domain and destination due to a broker resource constraint.
C4074	**Message** Transaction rolled back due to provider connection failover.
	Cause An attempt was made to call `Session.commit()` after connection failover occurred. The transaction is rolled back automatically.
C4075	**Message** Cannot acknowledge messages due to provider connection failover. Subsequent acknowledge calls will also fail until the application calls `session.recover()`.
	Cause As stated in the message.
C4076	**Message** Client does not have permission to create producer on destination: {0} *{0} is replaced with the destination name that caused the exception.*
	Cause The application client does not have permission to create a message producer with the specified destination.

Table A-2 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
C4077	**Message** Client is not authorized to create destination : {0} *{0} is replaced with the destination name that caused the exception.*
	Cause The application client does not have permission to create the specified destination.
C4078	**Message** Client is unauthorized to send to destination: {0} *{0} is replaced with the destination name that caused the exception*.
	Cause The application client does not have permission to produce messages to the specified destination.
C4079	**Message** Client does not have permission to register a consumer on the destination: {0} *{0} is replaced with the destination name that caused the exception.*
	Cause The application client does not have permission to create a message consumer with the specified destination name.
C4080	**Message** Client does not have permission to delete consumer: {0} *{0} is replaced with the Message Queue consumer ID for the consumer to be deleted.*
	Cause The application does not have permission to remove the specified consumer from the broker.
C4081	**Message** Client does not have permission to unsubscribe: {0} *{0} was replaced with the name of the subscriber to unsubscribe.*
	Cause The client application does not have permission to unsubscribe the specified durable subscriber.
C4082	**Message** Client is not authorized to access destination: {0} *{0} is replaced with the destination name that caused the exception.*
	Cause The application client is not authorized to access the specified destination.
C4083	**Message** Client does not have permission to browse destination: {0} {0} was replaced with the destination name that caused the exception.
	Cause The application client does not have permission to browse the specified destination.
C4084	**Message** User authentication failed: {0} *{0} is replaced with the user name.*
	Cause User authentication failed.
C4085	**Message** Delete consumer failed. Consumer was not found: {0} *{0} is replaced with name of the consumer that could not be found.*
	Cause The attempt to close a message consumer failed because the broker was unable to find the specified consumer.

Table A-2 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
C4086	**Message** Unsubscribe failed. Subscriber was not found: {0} *{0} is replaced with name of the durable subscriber.*
	Cause An attempt was made to unsubscribe a durable subscriber with a name that does not exist in the system.
C4087	**Message** Set Client ID operation failed. Invalid Client ID: {0} *{0} is replaced with the ClientID that caused the exception.*
	Cause Client is unable to set Client ID on the broker and receives a `BAD_REQUEST` status from broker.
C4088	**Message** A JMS destination limit was reached. Too many producers for {0} : {1} *{0} is replaced with `Queue` or `Topic`* *{1} is replaced with the destination name for which the limit was reached.*
	Cause The client runtime was not able to create a message producer for the specified domain and destination due to limited broker resources.
C4089	**Message** Caught JVM Error: {0} *{0} is replaced with root cause error message.*
	Cause The client runtime caught an error thrown from the JVM; for example, `OutOfMemory` error.
C4090	**Message** Invalid port number. Broker is not available or may be paused:{0} *{0} is replaced with "[host, port]" information.*
	Cause The client runtime received an invalid port number (0) from the broker. Broker service for the request was not available or was paused.
C4091	**Message** Cannot call `Session.recover()` from a `NO_ACKNOWLEDGE` session.
	Cause The application attempts to call `Session.recover()` from a `NO_ACKNOWLEDGE` session.
C4092	**Message** Broker does not support `Session.NO_ACKNOWLEDGE` mode, broker version: {0}
	{0} is replaced with the version number of the broker to which the Message Queue application is connected.
	Cause The application attempts to create a `NO_ACKNOWLEDGE` session to a broker with version # less than 3.6.
C4093	**Message** Received wrong packet type. Expected: {0}, but received: {1}
	{0} is replaced with the packet type that the Message Queue client runtime expected to receive from the broker. *{1} is replaced with the packet type that the Message Queue client runtime actually received from the broker.*
	Cause The Message Queue client runtime received an unexpected Message Queue packet from broker.

Table A-2 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
C4094	**Message** The destination this message was sent to could not be found: {0} *{0} is replaced with the destination name that caused the exception.* **Cause**: A destination to which a message was sent could not be found.
C4095	**Message**: Message exceeds the single message size limit for the server or destination: {0} *{0} is replaced with the destination name that caused the exception.* **Cause**: A message exceeds the single message size limit for the server or destination.
C4096	**Message**: Destination is full and is rejecting new messages: {0} *{0} is replaced with the destination name that caused the exception.* **Cause**: A destination is full and is rejecting new messages

Index

A

administered objects
- about 37
- instantiation of 41
- JNDI lookup of 38
- provider independence, and 39
- SAAJ, for 115
- types of 37

AUTO_ACKNOWLEDGE mode 73

auto-reconnect
- behavior 72, 73
- configurable attributes 74
- limitations 73

B

broker cluster 35, 71, 75

broker, metrics for 85

C

C clients, communicating with 80

checklist for client deployment 35

CLASSPATH environment variable 26

client acknowledgement modes
- AUTO_ACKNOWLEDGE 73
- CLIENT_ACKNOWLEDGE 75
- NO_ACKNOWLEDGE 78

client acknowledgements 75

client applications, *See* JMS clients

client identifier (ClientID) 49

client threads
- managing use of 53
- performance 53

CLIENT_ACKNOWLEDGE mode 75

clustered broker configuration 35, 71, 75

code, sample 34

compiling JMS clients 32

connection factory administered objects
- ClientID, and 49
- instantiation of 41
- JNDI lookup of 39
- overriding attribute values 44

connections
- reconnecting 70
- thread use by 53

consumers
- dedicated 61
- message loss, correcting 62
- pinging 61
- synchronous 62

custom client acknowledgement 75, 78

D

dead message queue 57

delivery modes 64

deployment checklist for client applications 35

destination administered objects
instantiation of 43
lookup of 40
destination metrics 87
directory variables
IMQ_HOME 16
IMQ_JAVAHOME 17
IMQ_VARHOME 16
distributed applications and synchronous consumers 62
domains 46
durable subscriptions
ClientID, and 49
performance impact of 67

E

environment variables, *See* directory variables
examples, code 34

F

FLOW_CONTROL property 61

H

hashtable for destination-list metrics 87, 94
heap space, JVM 54

I

IMQ_HOME directory variable 16
IMQ_JAVAHOME directory variable 17
IMQ_VARHOME directory variable 16
imqAddressList attribute 71, 74
imqAddressListBehavior attribute 75
imqAddressListIterations attribute 71
imqPingInterval attribute 62
imqReconnectAttempts attribute 71
imqReconnectEnabled attribute 70
imqReconnectInterval attribute 71, 74
imqReconnectListBehavior attribute 71

J

JAF API 119
JAXM servlet 123
JMS clients
avoiding deadlock 53
compiling 32
deployment checklist 35
development steps 29
factors impacting performance 63
portability of 45
provider independence 46
requirements for deployment 35
running 32
setup summary 24
system properties, and 44
JMS specification 13, 20
JMS_SUN_COMPRESS property 55
JMS_SUN_COMPRESSED_SIZE property 55
JMS_SUN_DMQ_BODY_TRUNCATED property 60
JMS_SUN_DMQ_PRODUCING_BROKER property 60
JMS_SUN_DMQ_UNDELIVERED_COMMENTS property 60
JMS_SUN_DMQ_UNDELIVERED_EXCEPTION property 60
JMS_SUN_DMQ_UNDELIVERED_REASON property 60
JMS_SUN_DMQ_UNDELIVERED_TIMESTAMP property 59
JMS_SUN_LOG_DEAD_MESSAGES property 58
JMS_SUN_PRESERVE_UNDELIVERED property 58
JMS_SUN_TRUNCATE_MSG_BODY property 59
JMS_SUN_UNCOMPRESSED_SIZE property 55
JMSXDeliveryCount property 59
JNDI
connection factory lookup 39
destination lookup 40

JVM
heap space 54
metrics for 87

L

LDAP object store 39

M

master broker 75
memory management 54
message delivery models 46
Message Queue libraries 23
message selector 50
Message.setBooleanProperty() method 55
message-based monitoring 81
MessageFactory object 125
messages
acknowledgement, *See* client acknowledgements or broker acknowledgements
body type, and performance 69
compression 49, 55
delivery models 46
ordering of 50
prioritizing 50
selectors 50
sequencing 54
size of 54, 55
size, and performance 68
SOAP payloads, with 130
metrics messages
format of 82, 85
properties of 85
type 82
metrics-based monitoring
administration of 83
creating client for 84
examples of 90
implementation of 83
introduced 81
MimeHeaders object 125
mq.metrics.broker topic 82
mq.metrics.destination.queue.*monitoredDestName* topic 82
mq.metrics.destination.topic.*monitoredDestName* topic 82
mq.metrics.destination_list topic 82
mq.metrics.jvm topic 82
mq.sys.dmq queue 58

N

namespaces, in SOAP 110
NO_ACKNOWLEDGE mode 78

O

object stores
administered objects, for 38
file system 39
LDAP server 39
onMessage() method 124
OutOfMemory error 54

P

performance
factors impacting, *See* performance impact factors
performance and reliability 63
performance impact factors
acknowledgement mode 66
delivery mode 64
durable subscriptions 67
message body type 69
message size 68

performance impact factors *(continued)*
selectors 68
transactions 65
physical destination properties 61
ping interval 61
programming domains 46
provider independence
about 46
administered objects, and 39

R

REJECT_NEWEST property 61
reliability and performance 63
REMOVE_LOW_PRIORITY property 61
REMOVE_OLDESTproperty 61
ReqRespListener object 119
running JMS clients 32

S

SAAJ API
about 107
client code 120
exception handling 120
fault handling 120, 126
javax.xml.messaging package 107
javax.xml.soap package 107
JAXM servlet 123
programming model for SOAP 100, 107, 118
service code 123
SAAJ specification 13, 20
sample code 34
selectors 50, 68
sequencing partial messages 54
ServletConfig object 124
sessions
threading restrictions in 52
work done by 48
SOAP message
attachments to 119
disassembling 125
envelope 103
header 104
MIME envelope for 105
models of 104
Name object 113
payload to JMS message, as 130
SOAPMessage object 107
structure of 104
SOAP messaging
attachments, using 119
client code 120
connections 115
endpoints 114
exception handling 120
fault codes 128
fault handling 120, 126
layers of 100
message factories 115
namespaces 110
point-to-point connections 118
programming models 118
protocol for 100
service code 123
SOAPMessageFromJMSMessage method 131
SOAPMessageIntoJMSMessage utility 130
system properties, setting 44

T

threads, *See* client threads
transactions
and custom client acknowledgement 77
performance impact of 65

U

URLEndpoint object 123

W

warning messages 141
web services 100