Sun Java™ System

Message Queue 3.5 Java Client Developer's Guide

Service Pack 1

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

Part No: 817-6026-10

Contents

List of Figures 9

List of Tables 11

List of Procedures 13

List of Code Examples 15

Preface 17
Audience for This Guide 17
Organization of This Guide 18
Conventions 19
Text Conventions 19
Directory Variable Conventions 20
Other Documentation Resources 21
The Message Queue Documentation Set 22
JavaDoc 22
Example Client Applications 23
The Java Message Service (JMS) Specification 23
The Java XML Messaging (JAXM) Specification 23
Books on JMS Programming 24

Chapter 1 Overview 25
What Is Sun Java System Message Queue? 25
Product Editions 27
Platform Edition 27
Enterprise Edition 28
Message Queue Service Architecture 28

The JMS Programming Model 30
JMS Programming Interface 30
Message 30
Destination 33
ConnectionFactory 33
Connection 33
Session 33
Message Producer 34
Message Consumer 34
Message Listener 34
Administered Objects 34
JMS Client Setup Operations 35
To Set Up a Client to Produce Messages 35
To Set Up a Client to Consume Messages 36
JMS Client Design Issues 36
Programming Domains 36
JMS Provider Independence 38
Client Identifiers 39
Reliable Messaging 39
Acknowledgements/Transactions 40
Persistent Storage 41
Performance Trade-offs 42
Message Consumption: Synchronous and Asynchronous 42
Message Selection 43
Message Order and Priority 43
JMS/J2EE Programming: Message-Driven Beans 43
Message-Driven Beans 44
J2EE Application Server Support 46

Chapter 2 Quick Start Tutorial 47
Setting Up Your Environment 47
Starting and Testing the Message Server 49
To Start a Broker 50
To Test a Broker 50
Developing a Simple Client Application 51
Compiling and Running a Client Application 54
To Compile and Run the HelloWorldMessage Application 54
Example Application Code 55

Chapter 3 Using Administered Objects 57
JNDI Lookup of Administered Objects 58
Looking Up ConnectionFactory Objects 59
To Perform a JNDI Lookup of a ConnectionFactory Object 59
Looking Up Destination Objects 60
To Perform a JNDI Lookup of a Destination Object 60
Instantiating Administered Objects 61
Instantiating ConnectionFactory Objects 61
To Directly Instantiate and Configure a ConnectionFactory Object 62
Instantiating Destination Objects 63
To Directly Instantiate and Configure a Destination Object 63
Starting Client Applications With Overrides 64

Chapter 4 Configuring the Message Queue Client Runtime 65
Message Production and Consumption 65
Message Production 66
Message Consumption 67
Client Runtime Configurable Properties 69
Connection Handling 71
Specifying a Message Server Address 71
Connecting to a Message Server 73
Automatic Reconnect to a Message Server (Enterprise Edition) 73
Auto-reconnect Behavior 74
Message Queue 3.0 Connection Handling 76
Client Identification 77
Message Header Overrides 78
Reliability And Flow Control 79
Queue Browser Behavior and Server Session 82
JMS-Defined Properties Support 83
Managing Reliability and Performance 84
Delivery Mode 84
Client Acknowledgement Mode 84
Message Flow Metering 85
Message Flow Limits 86

Chapter 5 Message Queue Client Programming Techniques 89
Custom Client Acknowledgement 89
Message-Based Monitoring API 92
Format of Metrics Messages 93
Broker Metrics 94
JVM Metrics 95
Destination-List Metrics 96
Destination Metrics 97

Configuring Metrics Message Production on the Broker 99
Using the Message-Based Monitoring API 99
Metrics Monitoring Client Code Examples 101
A Broker Metrics Example 101
A Destination List Metrics Example 103
A Destination Metrics Example 106
Client Connection Failover (Auto-reconnect) 108
Enabling Auto-reconnect 109
Auto-reconnect Behaviors 110
Auto-reconnect Limitations 110
Auto-reconnect Configuration Examples 111
Single-Broker Auto-reconnect 111
Parallel Broker Auto-reconnect 112
Clustered-Broker Auto-reconnect 112
Other Programming Topics 113
Managing Memory and Message Size 114
Using Secure HTTP Connections (HTTPS) 114
In Case of Server or Broker Failure 115
Repairing an HTTPS Tunnel Servlet Connection 115
Managing Client Threads 115
Synchronous Consumption in Distributed Applications 117
Client Application Deployment Considerations 118

Chapter 6 Working With SOAP Messages 119
What is SOAP? 120
SOAP and the JAVA for XML Messaging API 120
The Transport Layer 121
The SOAP Layer 121
The Provider Layer 122
The Profiles Layer 123
The SOAP Message 124
SOAP Packaging Models 125
SOAP Messaging in JAVA 127
The SOAP Message Object 127
Inherited Methods 129
Namespaces 130
Destination, Message Factory, and Connection Objects 133
Endpoint 134
Message Factory 135
Connection 135

Using JAXM Administered Objects 136
SOAP Messaging Models and Examples 138
  SOAP Messaging Programming Models 138
    Point-to-Point Connections 138
    Provider Connections 140
  Working with Attachments 141
    To Create and Add an Attachment 141
  Exception and Fault Handling 142
  Writing a SOAP Client 142
  Writing a SOAP Service 145
    Disassembling Messages 147
    Handling Attachments 148
    Replying to Messages 148
    Handling SOAP Faults 148
Integrating SOAP and Message Queue 152
  Example 1: Deferring SOAP Processing 153
    To Transform the SOAP Message into a JMS Message and Send the JMS Message 154
    To Receive the JMS Message, Transform it into a SOAP Message, and Process It 155
  Example 2: Publishing SOAP Messages 156
  Code Samples 157

Appendix A Administered Object Attributes 163
ConnectionFactory Administered Object 163
Destination Administered Objects 165
Endpoint Administered Objects 166

Appendix B Client Error Codes 167

Index 179

List of Figures

Figure 1-1 Message Queue System Architecture 29
Figure 1-2 JMS Programming Objects 31
Figure 1-3 Messaging with MDBs 45
Figure 4-1 Messaging Operations 66
Figure 4-2 Message Delivery to Message Queue Client Runtime 67
Figure 6-1 SOAP Messaging Layers 121
Figure 6-2 SOAP Interoperability 123
Figure 6-3 SOAP Message Without Attachments 125
Figure 6-4 SOAP Message with Attachments 126
Figure 6-5 SOAP Message Object 128
Figure 6-6 Request-Reply Messaging 139
Figure 6-7 One-Way Messaging 140
Figure 6-8 SOAP Message Parts 143
Figure 6-9 SOAP Fault Element 149
Figure 6-10 Deferring SOAP Processing 153
Figure 6-11 Publishing a SOAP Message 156

List of Tables

Table 1	Book Contents	18
Table 2	Document Conventions	19
Table 3	Message Queue Directory Variables	20
Table 4	Message Queue Documentation Set	22
Table 1-1	JMS-Defined Message Header	31
Table 1-2	Message Body Types	32
Table 1-3	JMS Programming Objects	37
Table 2-1	jar File Locations	48
Table 2-2	jar Files Needed in `CLASSPATH`	48
Table 2-3	Location of Message Queue Executables	50
Table 2-4	Example Programs	55
Table 4-1	Message Server Address Schemes and Syntax	71
Table 4-2	Message Server Address Examples	72
Table 4-3	Connection Factory Attributes: Connection Handling	75
Table 4-4	Supported Message Queue 3.0 Connection Handling Attributes	76
Table 4-5	Connection Factory Attributes: Client Identification	78
Table 4-6	Connection Factory Attributes: Message Header Overrides	79
Table 4-7	Connection Factory Attributes: Reliability and Flow Control	80
Table 4-8	Connection Factory Attributes: Queue Browser Behavior	82
Table 4-9	Connection Factory Attributes: JMS-defined Properties Support	83
Table 5-1	Metrics Topic Destinations	92
Table 5-2	Broker Metrics Message Properties	94
Table 5-3	Data in the Body of a Broker Metrics Message	94
Table 5-4	JVM Metrics Message Properties	95
Table 5-5	Data in the Body of a JVM Metrics Message	95
Table 5-6	Destination-List Message Properties	96
Table 5-7	Data in the Body of a Destination-List Metrics Message	97

Table 5-8 Destination Metrics Message Properties 97
Table 5-9 Data in the Body of a Destination Metrics Message 98
Table 5-10 Starter Checklist for the Message Queue Administrator 118
Table 6-1 Inherited Methods 129
Table 6-2 SOAP Administered Object Information 136
Table 6-3 JAXMServlet Methods 146
Table 6-4 SOAP Faultcode Values 150
Table A-1 Connection Factory Attributes 163
Table A-2 Destination Attributes 165
Table A-3 Endpoint Attributes 166
Table B-1 Message Queue Client Error Codes 167

List of Procedures

To Set Up a Client to Produce Messages 35
To Set Up a Client to Consume Messages 36
To Start a Broker 50
To Test a Broker 50
To Compile and Run the HelloWorldMessage Application 54
To Perform a JNDI Lookup of a ConnectionFactory Object 59
To Perform a JNDI Lookup of a Destination Object 60
To Directly Instantiate and Configure a ConnectionFactory Object 62
To Directly Instantiate and Configure a Destination Object 63
To Create and Add an Attachment 141
To Transform the SOAP Message into a JMS Message and Send the JMS Message 154
To Receive the JMS Message, Transform it into a SOAP Message, and Process It 155

List of Code Examples

Code Example 3-1 Looking Up a ConnectionFactory Object 60
Code Example 3-2 Instantiating a ConnectionFactory Object 62
Code Example 3-3 Instantiating a Destination Object 63
Code Example 5-1 Syntax for `acknowledgeThisMessage()` Method 90
Code Example 5-2 Example of Custom Client Acknowledgement Code 91
Code Example 5-3 Example of Subscribing to a Broker Metrics Topic 101
Code Example 5-4 Example of Processing a Broker Metrics Message 102
Code Example 5-5 Example of Subscribing to the Destination List Metrics Topic 103
Code Example 5-6 Example of Processing a Destination List Metrics Message 104
Code Example 5-7 Example of Extracting Destination Information From a Hashtable 105
Code Example 5-8 Example of Subscribing to a Destination Metrics Topic 106
Code Example 5-9 Example of Processing a Destination Metrics Message 107
Code Example 5-10 Example of Command to Configure a Single Broker 111
Code Example 5-11 Example of Command to Configure Parallel Brokers 112
Code Example 5-12 Example of Command to Configure a Broker Cluster 113
Code Example 6-1 Explicit Namespace Declarations 131
Code Example 6-2 Adding an Endpoint Administered Object 137
Code Example 6-3 Looking up an Endpoint Administered Object 137
Code Example 6-4 Skeleton Message Consumer 145
Code Example 6-5 A Simple Ping Message Service 146
Code Example 6-6 Processing a SOAP Message 147
Code Example 6-7 Sending a JMS Message with a SOAP Payload 157
Code Example 6-8 Receiving a JMS Message with a SOAP Payload 160

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:

- "Audience for This Guide" on page 17
- "Organization of This Guide" on page 18
- "Conventions" on page 19
- "Other Documentation Resources" on page 21

Audience for This Guide

This guide is meant principally for developers of Java applications that exchange messages using a Message Queue messaging system.

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

This *Message Queue Java Client Developer's Guide* 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 the JAXM APIs or with JAXM programming guidelines. This material is described in Chapter 6, "Working With SOAP Messages," and only assumes basic knowledge of XML.

Organization of This Guide

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 Message Queue and of JMS concepts and programming issues.
Chapter 2, "Quick Start Tutorial"	A tutorial that acquaints you with the Message Queue development environment using a simple example JMS client application.
Chapter 3, "Using Administered Objects"	Describes how to use Message Queue administered objects in both a provider- independent and provider-specific way.
Chapter 4, "Configuring the Message Queue Client Runtime"	Explains features of the Message Queue client runtime and how they can be used to optimize client applications.
Chapter 5, "Message Queue Client Programming Techniques"	Covers a number of topics that illustrate how to write client applications that use Message Queue-specific features.
Chapter 6, "Working With SOAP Messages"	Explains how you send and receive SOAP messages with and without Message Queue support.
Appendix A, "Administered Object Attributes"	Summarizes and documents administered object attributes.
Appendix B, "Client Error Codes"	Provides reference information for error codes returned by the Message Queue client runtime when it raises a JMS exception.

Conventions

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 Solaris, for Sun Java™ System Application Server (formerly Sun™ ONE Application Server), the root Message Queue installation directory is `/imq`, under the Application Server base directory. • On Windows, the root Message Queue installation directory is set by the Message Queue installer (by default, as `C:\Program Files\Sun\MessageQueue3`). • On Windows, for Sun Java System Application Server, the root Message Queue installation directory is `/imq`, under the Application Server base directory. • 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.
`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 Solaris, for Sun Java System Application Server, `IMQ_VARHOME` defaults to the `IMQ_HOME/var` directory. • On Windows `IMQ_VARHOME` defaults to the `IMQ_HOME\var` directory. • On Windows, for Sun Java System Application Server, `IMQ_VARHOME` defaults to the `IMQ_HOME\var` directory. • On Linux, `IMQ_VARHOME` defaults to the `/var/opt/imq` 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` defaults to the `/usr/j2se/jre` directory, but a user can optionally set the value to wherever the required JRE resides. • On Windows, `IMQ_JAVAHOME` defaults to `IMQ_HOME\jre`, but a user can optionally set the value to wherever the required JRE resides. • On Linux, Message Queue first looks for the Java runtime in the `/usr/java/j2sdk`*Version* directory, and then looks in the `/usr/java/j2re`*Version* directory, but a user can optionally set the value of `IMQ_JAVAHOME` 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 (`/`).

Other Documentation Resources

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 Java Client Developer's Guide	Developers	Provides a quick-start tutorial and programming information for developers of Java client programs using the Message Queue implementation of the JMS and SOAP/JAXM specifications.
Message Queue C Client Developer's Guide	Developers	Provides programming and reference documentation for developers of C client programs using the C interface (C-API) to the Message Queue service.
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/imq/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 3, “Using Administered Objects”).

Example Client Applications

A number of example applications that provide sample Java client application code are included in the following directories:

Platform	Location
Solaris	`/usr/demo/imq/`
Linux	`/opt/imq/demo/`
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 Java XML Messaging (JAXM) Specification

The JAXM specification can be found at the following location:

`http://java.sun.com/xml/downloads/jaxm.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

Chapter 1

Overview

This chapter provides an overall introduction to Sun Java™ System Message Queue (formerly Sun™ ONE Message Queue) and to JMS concepts and programming issues of interest to developers.

The chapter covers the following topics:

- "What Is Sun Java System Message Queue?" on page 25
- "Product Editions" on page 27
- "Message Queue Service Architecture" on page 28
- "The JMS Programming Model" on page 30
- "JMS Client Design Issues" on page 36
- "JMS/J2EE Programming: Message-Driven Beans" on page 43

What Is Sun Java System Message Queue?

The Message Queue product is a standards-based solution for reliable, asynchronous messaging for distributed applications. Message Queue is an enterprise messaging system that implements the Java™ Message Service (JMS) open standard: in fact it serves as the JMS Reference Implementation. However Message Queue is also a full-featured JMS provider with enterprise-strength features.

The JMS specification describes a set of messaging semantics and behaviors, and an application programming interface (API), that provide a common way for Java language applications to create, send, receive, and read messages in a distributed environment (see "The JMS Programming Model" on page 30). In addition to supporting Java messaging applications, Message Queue also provides a C language interface to the Message Queue service (the Message Queue C-API).

With Sun Java System Message Queue software, processes running on different platforms and operating systems can connect to a common Message Queue service to send and receive information. Application developers are free to focus on the business logic of their applications, rather than on the low-level details of how their applications reliably communicate across a network.

Message Queue has features that exceed the minimum requirements of the JMS specification. Among these features are the following:

Centralized administration. Provides both command-line and GUI tools for administering a Message Queue service and managing application-dependent entities, such as destinations, transactions, durable subscriptions, and security. Message Queue also supports remote monitoring of the Message Queue service.

Scalable message service. Allows you to service increasing numbers of Message Queue clients (components or applications) by balancing the load among a number of Message Queue message server components (*brokers*) working in tandem (multi-broker cluster).

Client connection failover. Automatically restores a failed client connection to a Message Queue message server.

Tunable performance. Lets you increase performance of the Message Queue service when less reliability of delivery is acceptable.

Multiple transports. Supports the ability of Message Queue clients to communicate with the Message Queue message server over a number of different transports, including TCP and HTTP, and using secure (SSL) connections.

JNDI support. Supports both file-based and LDAP implementations of the Java Naming and Directory Interface (JNDI) as object stores and user repositories.

SOAP messaging support. Supports creation and delivery of SOAP messages—messages that conform to the Simple Object Access Protocol (SOAP) specification— *via* JMS messaging. SOAP allows for the exchange of structured XML data between peers in a distributed environment. See the Chapter 6, "Working With SOAP Messages" for more information.

See the *Message Queue Administration Guide* for documentation of JMS compliance-related issues.

Product Editions

Sun Java System Message Queue is available in two editions: Platform and Enterprise—each corresponding to a different feature set and licensed capacity, as described below. (Instructions for upgrading Message Queue from one edition to another are in the *Message Queue Installation Guide*.)

Platform Edition

This edition can be downloaded free from the Sun website and is also bundled with the Sun Java™ System Application Server platform. The Platform Edition places no limit on the number of client connections supported by the Message Queue message server. It comes with two licenses, as described below:

- **a basic license.** This license provides basic JMS support (it's a full JMS provider), but does *not* include such enterprise features as load balancing (multi-broker message service), HTTP/HTTPS connections, secure connection services, scalable connection capability, client connection failover, queue delivery to multiple consumers, remote message-based monitoring, and C-API support. The license has an unlimited duration, and can therefore be used in less demanding production environments.
- **a 90-day trial enterprise license.** This license includes all enterprise features (such as support for multi-broker message services, HTTP/HTTPS connections, secure connection services, scalable connection capability, client connection failover, queue delivery to multiple consumers, remote message-based monitoring, and C-API support) not included in the basic license. However, the license has a limited 90-day duration enforced by the software, making it suitable for evaluating the enterprise features available in the Enterprise Edition of the product (see "Enterprise Edition").

NOTE The 90-day trial license can be enabled by starting the Message Queue service—a broker instance—with the `-license` command line option (see the *Message Queue Administration Guide*) and passing "`try`" as the license to use:

```
imqbrokerd -license try
```

You must use this option each time you start the broker instance, otherwise it defaults back to the basic Platform Edition license.

Enterprise Edition

This edition is for deploying and running messaging applications in a production environment. It includes support for multi-broker message services, HTTP/HTTPS connections, secure connection services, scalable connection capability, client connection failover, queue delivery to multiple consumers, remote message-based monitoring, and C-API support. You can also use the Enterprise Edition for developing, debugging, and load testing messaging applications and components. The Enterprise Edition has an unlimited duration license that places no limit on the number of brokers in a multi-broker message service, but is based on the number of CPUs that are used.

NOTE For all editions of Message Queue, a portion of the product—the client runtime—can be freely redistributed for commercial use. All other files in the product *cannot* be redistributed. The portion that can be freely redistributed allows a licensee to develop a Java client application (one which can be connected to a Message Queue service) that can be sold to a third party without incurring any Message Queue licensing fees. The third party will either need to purchase Message Queue to access a Message Queue service or make a connection to yet another party that has a Message Queue service installed and running.

Message Queue Service Architecture

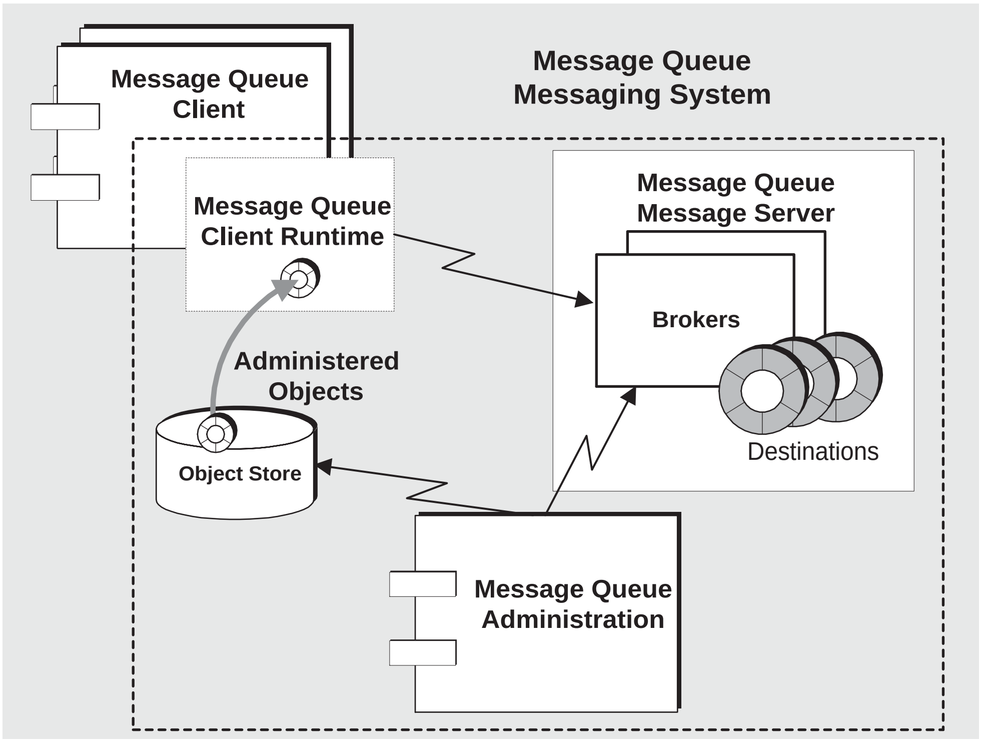
This section briefly describes the main parts of a Message Queue service. While as a developer, you do not need to be familiar with the details of all of these parts or how they interact, a high-level understanding of the basic architecture will help you understand features of the system that impact client application design and development.

The main parts of a Message Queue service, shown in Figure 1-1, are the following:

Message Queue server The Message Queue server is the heart of a messaging system. It consists of one or more brokers which provide delivery services for the system. These services include connections to clients, message routing and delivery, persistence, security, and logging. The message server maintains physical destinations to which clients send messages, and from which the messages are delivered to consuming clients. The Message Queue server is described in detail in the *Message Queue Administration Guide*.

Message Queue client runtime The Message Queue client runtime provides clients with an interface to the Message Queue service—it supplies clients with all the JMS programming objects introduced in "The JMS Programming Model" on page 30. It supports all operations needed for clients to send messages to destinations and to receive messages from such destinations. The Message Queue client runtime is described in detail in Chapter 4, "Configuring the Message Queue Client Runtime."

Figure 1-1 Message Queue System Architecture


Message Queue Client
Message Queue Messaging System
Message Queue Client Runtime
Message Queue Message Server
Brokers
Administered Objects
Destinations
Object Store
Message Queue Administration

Message Queue administered objects Administered Objects encapsulate provider-specific implementation and configuration information in objects that are used by Message Queue clients. Administered objects are generally created and configured by an administrator, stored in a name service, accessed by clients through standard JNDI lookup code, and then used in a provider-independent manner. They can also be instantiated by clients, in which case they are used in a provider-specific manner. Configuration of the client runtime is performed through administered object attributes, as described in Chapter 4, "Configuring the Message Queue Client Runtime."

Message Queue administration Message Queue provides a number of administration tools for managing a Message Queue service. These tools are used to manage the message server, create and store administered objects, manage security, manage messaging application resources, and manage persistent data. These tools are generally used by administrators and are described in the *Message Queue Administration Guide*.

The JMS Programming Model

This section briefly describes the programming model of the JMS specification. It is meant as a review of the most important concepts and terminology used in programming JMS clients.

JMS Programming Interface

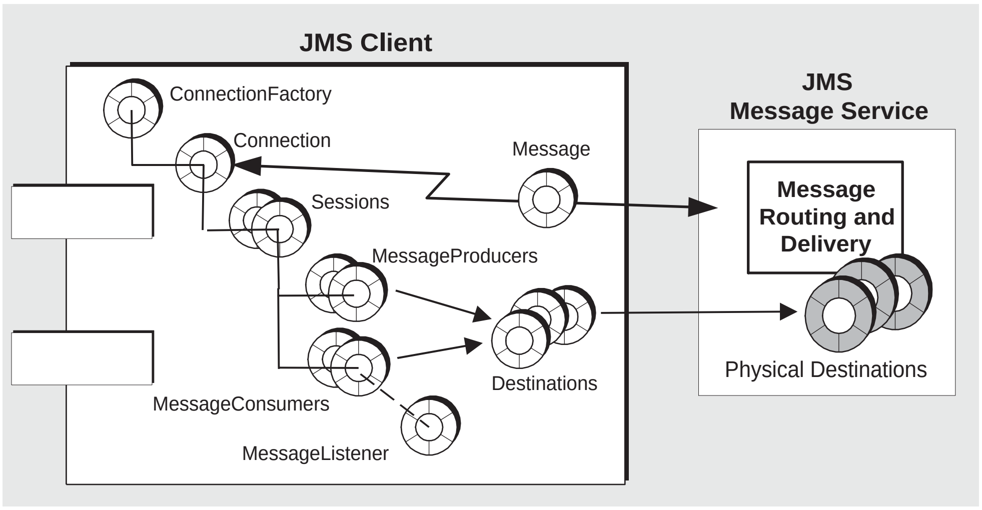
In the JMS programming model, JMS clients (components or applications) interact using a JMS application programming interface (API) to send and receive messages. This section introduces the objects that implement the JMS API and that are used to set up a client for delivery of messages (see "JMS Client Setup Operations" on page 35). The main interface objects are shown in Figure 1-2 and described in the following paragraphs.

Message

In the Message Queue product, data is exchanged using JMS messages—messages that conform to the JMS specification. According to the JMS specification, a message is composed of three parts: a header, properties, and a body.

Properties are optional—they provide values that clients can use to filter messages. A body is also optional—it contains the actual data to be exchanged.

Figure 1-2 JMS Programming Objects



Header

A header is required of every message. Header fields contain values used for routing and identifying messages.

Some header field values are set automatically by Message Queue during the process of producing and delivering a message, some depend on settings of message producers specified when the message producers are created in the client, and others are set on a message by message basis by the client using JMS APIs. The following table lists the header fields defined (and required) by JMS, as well as how they are set.

Table 1-1 JMS-Defined Message Header

Header Field	Set By:	Default
`JMSDestination`	Client, for each message producer or message	
`JMSDeliveryMode`	Client, for each message producer or message	`Persistent`
`JMSExpiration`	Client, for each message producer or message	time to live is `0` (no expiration)
`JMSPriority`	Client, for each message producer or message	4 (normal)
`JMSMessageID`	Provider, automatically	

Table 1-1 JMS-Defined Message Header *(Continued)*

Header Field	Set By:	Default
`JMSTimestamp`	Provider, automatically	
`JMSRedelivered`	Provider, automatically	
`JMSCorrelationID`	Client, for each message	
`JMSReplyTo`	Client, for each message	
`JMSType`	Client, for each message	

Properties

When data is sent between two processes, other information besides the payload data can be sent with it. These descriptive fields, or properties, can provide additional information about the data, including which process created it, the time it was created, and information that uniquely identifies the structure of each piece of data. Properties (which can be thought of as an extension of the header) consist of property name and property value pairs, as specified by JMS client code.

Having registered an interest in a particular destination, consuming clients can fine-tune their selection by specifying certain property values as selection criteria. For instance, a client might indicate an interest in Payroll messages (rather than Facilities) but only Payroll items concerning part-time employees located in New Jersey. Messages that do not meet the specified criteria are not delivered to the consumer.

Message Body Types

JMS specifies six classes (or types) of messages that a JMS provider must support, as described in the following table:

Table 1-2 Message Body Types

Type	Description
Message	A message without a message body.
StreamMessage	A message whose body contains a stream of Java primitive values. It is filled and read sequentially.
MapMessage	A message whose body contains a set of name-value pairs. The order of entries is not defined.
TextMessage	A message whose body contains a Java string, for example an XML message.

Table 1-2 Message Body Types *(Continued)*

Type	Description
ObjectMessage	A message whose body contains a serialized Java object.
BytesMessage	A message whose body contains a stream of uninterpreted bytes.

Destination

A `Destination` is a JMS administered object (see "Administered Objects" on page 34) that identifies a *physical* destination in a JMS message service. A physical destination is a JMS message service entity to which producers send messages and from which consumers receive messages. The message service provides the routing and delivery for messages sent to a physical destination. A `Destination` administered object encapsulates provider-specific naming conventions for physical destinations. This lets clients be provider independent.

ConnectionFactory

A `ConnectionFactory` is a JMS administered object (see "Administered Objects" on page 34) that encapsulates provider-specific connection configuration information. A client uses it to create a connection over which messages are delivered. JMS administered objects can either be acquired through a Java Naming and Directory Service (JNDI) lookup or directly instantiated using provider-specific classes.

Connection

A `Connection` is a client's active connection to a JMS message service. Both allocation of communication resources and authentication of a client take place when a connection is created. Hence it is a relatively heavy-weight object, and most clients do all their messaging with a single connection. A connection is used to create sessions.

Session

A `Session` is a single-threaded context for producing and consuming messages. While there is no restriction on the number of threads that can use a session, the session should not be used *concurrently* by multiple threads. It is used to create the message producers and consumers that send and receive messages, and defines a serial order for the messages it delivers. A session supports reliable delivery through a number of acknowledgement options or by using transactions. A transacted session can combine a series of sequential operations into a single transaction that can span a number of producers and consumers.

Message Producer

A client uses a `MessageProducer` to send messages to a physical destination. A `MessageProducer` object is normally created by passing a `Destination` administered object to a session's methods for creating a message producer. (If you create a message producer that does not reference a specific destination, then you must specify a destination for each message you produce.) A client can specify a default delivery mode, priority, and time-to-live for a message producer. These values govern all messages sent by a producer, except when explicitly over-ridden.

Message Consumer

A client uses a `MessageConsumer` to receive messages from a physical destination. It is created by passing a `Destination` administered object to a session's methods for creating a message consumer. A message consumer can have a message selector that allows the message service to deliver only those messages that match the selection criteria. A message consumer can support either synchronous or asynchronous consumption of messages (see "Message Consumption: Synchronous and Asynchronous" on page 42).

Message Listener

A client uses a `MessageListener` object to consume messages asynchronously. The MessageListener is registered with a message consumer. A client consumes a message when a session thread invokes the `onMessage()` method of the `MessageListener` object.

Administered Objects

The JMS specification facilitates provider-independent clients by specifying *administered objects* that encapsulate provider-specific configuration information.

Two of the objects described in the "The JMS Programming Model" on page 30 depend on how a JMS provider implements a JMS message service. The connection factory object depends on the underlying protocols and mechanisms used by the provider to deliver messages, and the destination object depends on the specific naming conventions and capabilities of the physical destinations used by the provider.

Normally these provider-specific characteristics would make client code dependent on a specific JMS implementation. However, the JMS specification requires that provider-specific implementation and configuration information be encapsulated in connection factory and destination objects that can then be accessed in a standardized, non-provider-specific way.

Administered objects are created and configured by an administrator, stored in a name service, and accessed by clients through standard Java Naming and Directory Service (JNDI) lookup code. Using administered objects in this way makes client code provider-independent.

The two types of administered objects, connection factories and destinations, encapsulate provider-specific information, but they have very different uses within a client. A connection factory is used to create connections to the message server, while destination objects are used to identify physical destinations.

For more information on administered objects, see Chapter 3, "Using Administered Objects."

JMS Client Setup Operations

There is a general approach within the JMS programming model for setting up a JMS client to produce or consume messages. It uses the JMS programming interface objects described in the previous section.

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

➤ To Set Up a Client to Produce Messages

1. Use JNDI to find a `ConnectionFactory` object. (You can also directly instantiate a `ConnectionFactory` object and set its attribute values.)
2. Use the `ConnectionFactory` object to create a `Connection` object.
3. Use the `Connection` object to create one or more `Session` objects.
4. Use JNDI to find one or more `Destination` objects. (You can also directly instantiate a `Destination` object and set its name attribute.)
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.

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

1. Use JNDI to find a `ConnectionFactory` object. (You can also directly instantiate a `ConnectionFactory` object and set its attribute values.)
2. Use the `ConnectionFactory` object to create a `Connection` object.
3. Use the `Connection` object to create one or more `Session` objects.
4. Use JNDI to find one or more `Destination` objects. (You can also directly instantiate a `Destination` object and set its name attribute.)
5. Use a `Session` object and a `Destination` object to create any needed `MessageConsumer` objects.
6. If needed, 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.

At this point the client has the basic setup needed to consume messages.

JMS Client Design Issues

This section is a review of a number of JMS messaging issues that impact JMS client design.

Programming Domains

JMS supports two distinct message delivery models: point-to-point and publish/subscribe.

Point-to-Point (Queue Destinations) A message is delivered from a producer to one consumer. In this delivery model, the destination is a *queue*. Messages are first delivered to the queue destination, then delivered from the queue, one at a time, depending on the queue's delivery policy (see Chapter 2 in the *Message Queue Administration Guide*), to one of the consumers registered for the queue. Any number of producers can send messages to a queue destination, but each message is guaranteed to be delivered to—and successfully consumed by—only *one* consumer. If there are no consumers registered for a queue destination, the queue holds messages it receives, and delivers them when a consumer registers for the queue.

Publish/Subscribe (Topic destinations) A message is delivered from a producer to any number of consumers. In this delivery model, the destination is a *topic*. Messages are first delivered to the topic destination, then delivered to *all* active consumers that have *subscribed* to the topic. Any number of producers can send messages to a topic destination, and each message can be delivered to any number of subscribed consumers. Topic destinations also support the notion of *durable subscriptions.* A durable subscription represents a consumer that is registered with the topic destination but can be inactive at the time that messages are delivered. When the consumer subsequently becomes active, it receives the messages. If there are no consumers registered for a topic destination, the topic does not hold messages it receives, unless it has durable subscriptions for inactive consumers.

These two message delivery models are handled using different API objects—with slightly different semantics—representing different programming domains, as shown in Table 1-3.

Table 1-3 JMS Programming Objects

Base Type (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.

You can program both point-to-point and publish/subscribe messaging using the unified domain objects that conform to the JMS 1.1 specification (shown in the first column of Table 1-3). The JMS 1.1 specification, provides a simplified approach to JMS client programming as compared to JMS 1.02. In particular, a client can perform both point-to-point and publish/subscribe messaging over the same connection and within the same session, and can include both queues and topics in the same transaction.

In short, a client developer need not make a choice between the separate point-to-point and publish/subscribe programming domains of JMS 1.0.2, opting instead for the simpler, unified domain approach of JMS 1.1. This is the preferred approach, however the JMS 1.1 specification continues to support the separate JMS 1.02 programming domains. (In fact, 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.)

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 (see "Message-Driven Beans" on page 44)—must be based on the domain-specific JMS APIs.

JMS Provider Independence

JMS specifies the use of administered objects (see "Administered Objects" on page 34) to support the development of clients that are portable to other JMS providers. Administered objects allow clients to use logical names to look up and reference provider-specific objects. In this way application does not need to know specific naming or addressing syntax or configurable properties used by a provider. This makes the code provider-independent.

Administered objects are Message Queue system objects created and configured by a Message Queue administrator. These objects are placed in a JNDI directory service, and a JMS client accesses them using a JNDI lookup.

Message Queue administered objects can also be instantiated by the client, rather than looked up in a JNDI directory service. This has the drawback of requiring the application developer to use provider-specific APIs. It also undermines the ability of a Message Queue administrator to successfully control and manage a Message Queue service.

For more information on administered objects, see Chapter 3, "Using Administered Objects."

Client Identifiers

JMS providers must support the notion of a *client identifier*, which associates a JMS client's connection to a message service with state information maintained by the message service on behalf of the client. By definition, a client identifier is unique, and applies to only one user at a time. Client identifiers are used in combination with a durable subscription name (see "Publish/Subscribe (Topic destinations)" on page 37) to make sure that each durable subscription corresponds to only one user.

The JMS specification allows client identifiers to be set by the client through an API method call, but recommends setting it administratively using a connection factory administered object (see "Administered Objects" on page 34). If hard wired into a connection factory, however, each user would then need an individual connection factory to have a unique identity.

Message Queue provides a way for the client identifier to be both ConnectionFactory and user specific using a special variable substitution syntax that you can configure in a `ConnectionFactory` object (see "Client Identification" on page 77). When used this way, a single `ConnectionFactory` object can be used by multiple users who create durable subscriptions, without fear of naming conflicts or lack of security. A user's durable subscriptions are therefore protected from accidental erasure or unavailability due to another user having set the wrong client identifier.

For deployed applications, the client identifier must either be programmatically set by the client, using the JMS API, or administratively configured in the `ConnectionFactory` objects used by the client.

In any case, in order to create a durable subscription, a client identifier must be either programmatically set by the client, using the JMS API, or administratively configured in the `ConnectionFactory` objects used by the client.

Reliable Messaging

JMS defines two *delivery modes*:

Persistent messages These messages are guaranteed to be delivered and successfully consumed once and only once. Reliability is at a premium for such messages.

Non-persistent messages These messages are guaranteed to be delivered at most once. Reliability is not a major concern for such messages.

There are two aspects of assuring reliability in the case of *persistent* messages. One is to assure that their delivery to and from a message service is successful. The other is to assure that the message service does not lose persistent messages before delivering them to consumers.

Acknowledgements/Transactions

Reliable messaging depends on guaranteeing the successful delivery of persistent messages to and from a destination. This can be achieved using either of two general mechanisms supported by a Message Queue session: acknowledgements or transactions. In the case of transactions, these can either be local or distributed, under the control of a distributed transaction manager.

Acknowledgements

A session can be configured to use acknowledgements to assure reliable delivery.

In the case of a producer, this means that the message service acknowledges delivery of a persistent message to its destination before the producer's `send()` method returns. In the case of a consumer, this means that the client acknowledges delivery and consumption of a persistent message from a destination before the message service deletes the message from that destination.

Local Transactions

A session can also be configured as *transacted*, in which case the production and/or consumption of one or more messages can be grouped into an atomic unit—a *transaction*. The JMS API provides methods for initiating, committing, or rolling back a transaction.

As messages are produced or consumed within a transaction, the broker tracks the various sends and receives, completing these operations only when the client issues a call to commit the transaction. If a particular send or receive operation within the transaction fails, an exception is raised. The application can handle the exception by ignoring it, retrying the operation, or rolling back the entire transaction. When a transaction is committed, all the successful operations are completed. When a transaction is rolled back, all successful operations are cancelled.

The scope of a local transaction is always a single session. That is, one or more producer or consumer operations performed in the context of a single session can be grouped into a single local transaction.

Since transactions span only a single session, you cannot have an end-to-end transaction encompassing both the production and consumption of a message. (In other words, the delivery of a message to a destination and the subsequent delivery of the message to a client cannot be placed in a single transaction.)

Distributed Transactions

Message Queue also supports *distributed* transactions. That is, the production and consumption of messages can be part of a larger, distributed transaction that includes operations involving other resource managers, such as database systems. In distributed transactions, a distributed transaction manager tracks and manages operations performed by multiple resource managers (such as a message service and a database manager) using a two-phase commit protocol defined in the Java Transaction API (JTA), *XA Resource* API specification. In the Java world, interaction between resource managers and a distributed transaction manager are described in the JTA specification.

Support for distributed transactions means that messaging clients can participate in distributed transactions through the XAResource interface defined by JTA. This interface defines a number of methods for implementing two-phase commit. While the API calls are made on the client side, the Message Queue broker tracks the various send and receive operations within the distributed transaction, tracks the transactional state, and completes the messaging operations only in coordination with a distributed transaction manager—provided by a Java Transaction Service (JTS).

As with local transactions, the client can handle exceptions by ignoring them, retrying operations, or rolling back an entire distributed transaction.

Message Queue implements support for distributed transactions through an XA connection factory, which lets you create XA connections, which in turn lets you create XA sessions (see “The JMS Programming Model” on page 30). In addition, support for distributed transactions requires either a third party JTS or a J2EE-compliant Application Server (that provides JTS).

Persistent Storage

The other important aspect of reliability is assuring that once persistent messages are delivered to their destinations, the message service does not lose them before they are delivered to consumers. This means that upon delivery of a persistent message to its destination, the message service must place it in a persistent data store. If the message service goes down for any reason, it can recover the message and deliver it to the appropriate consumers. While this adds overhead to message delivery, it also adds reliability.

A message service must also store durable subscriptions. This is because to guarantee delivery in the case of topic destinations, it is not sufficient to recover only persistent messages. The message service must also recover information about durable subscriptions for a topic, otherwise it would not be able to deliver a message to durable subscribers when they become active.

Messaging applications that are concerned about guaranteeing delivery of persistent messages must either employ queue destinations or employ durable subscriptions to topic destinations.

Performance Trade-offs

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. Between these extremes are a number of options, depending on the needs of an application, including the use of Message Queue-specific persistence and acknowledgement properties (see "Managing Reliability and Performance" on page 84).

Message Consumption: Synchronous and Asynchronous

There are two ways a JMS client can consume messages: either synchronously or asynchronously.

In synchronous consumption, a client gets a message by invoking the `receive()` method of a `MessageConsumer` object. The client thread blocks until the method returns. This means that if no message is available, the client blocks until a message does become available or until the `receive()` method times out (if it was called with a time-out specified). In this model, a client thread can only consume messages one at a time (synchronously).

In asynchronous consumption, a client registers a `MessageListener` object with a message consumer. The message listener is like a call-back object. A client consumes a message when the session invokes the `onMessage()` method of the `MessageListener` object. In this model, the client thread does not block (message is asynchronously consumed) because the thread listening for and consuming the message belongs to the Message Queue client runtime.

Message Selection

JMS provides a mechanism by which a message service can perform message filtering and routing based on criteria placed in message selectors. A producing client can place application-specific properties in the message, and a consuming client can indicate its interest in messages using selection criteria based on such properties. This simplifies the work of the client and eliminates the overhead of delivering messages to clients that don't need them. However, it adds some additional overhead to the message service processing the selection criteria. Message selector syntax and semantics are outlined in the JMS specification.

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.

JMS/J2EE Programming: Message-Driven Beans

In addition to the general client programming model introduced in "The JMS Programming Model" on page 30, there is a more specialized adaptation of JMS used in the context of Java 2 Enterprise Edition (J2EE) applications. This specialized JMS client is called a *message-driven bean* and is one of a family of Enterprise JavaBeans (EJB) components specified in the EJB 2.0 Specification (`http://java.sun.com/products/ejb/docs.html`).

The need for message-driven beans arises out of the fact that other EJB components (session beans and entity beans) can only be called synchronously. These EJB components have no mechanism for receiving messages asynchronously, since they are only accessed through standard EJB interfaces.

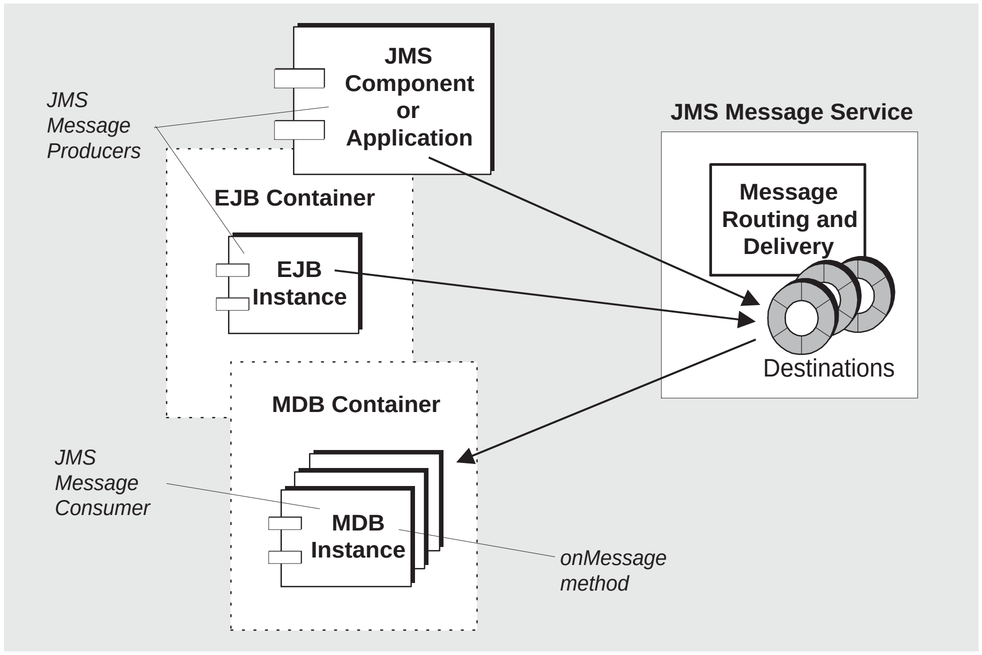
However, asynchronous messaging is a requirement of many enterprise applications. Most such applications require that server-side components be able to communicate and respond to each other without tying up server resources. Hence, the need for an EJB component that can receive messages and consume them without being tightly coupled to the producer of the message. This capability is needed for any application in which server-side components must respond to application events. In enterprise applications, this capability must also scale under increasing load.

Message-Driven Beans

A message-driven bean (MDB) is a specialized EJB component supported by a specialized EJB container (a software environment that provides distributed services for the components it supports).

Message-driven Bean The MDB is a JMS message consumer that implements the JMS `MessageListener` interface. The `onMessage` method (written by the MDB developer) is invoked when a message is received by the MDB container. The `onMessage()` method consumes the message, just as the `onMessage()` method of a standard `MessageListener` object would. You do not remotely invoke methods on MDBs—like you do on other EJB components—therefore there are no home or remote interfaces associated with them. The MDB can consume messages from a single destination. The messages can be produced by standalone JMS applications, JMS components, EJB components, or Web components, as shown in Figure 1-3.

Figure 1-3 Messaging with MDBs



MDB Container The MDB is supported by a specialized EJB container, responsible for creating instances of the MDB and setting them up for asynchronous consumption of messages. This involves setting up a connection with the message service (including authentication), creating a pool of sessions associated with a given destination, and managing the distribution of messages as they are received among the pool of sessions and associated MDB instances. Since the container controls the life-cycle of MDB instances, it manages the pool of MDB instances so as to accommodate incoming message loads.

Associated with an MDB is a deployment descriptor that specifies the JNDI lookup names for the administered objects used by the container in setting up message consumption: a connection factory and a destination. The deployment descriptor might also include other information that can be used by deployment tools to configure the container. Each such container supports instances of only a single MDB.

J2EE Application Server Support

In J2EE architecture (see the J2EE Platform Specification located at http://java.sun.com/j2ee/download.html#platformspec), EJB containers are hosted by J2EE application servers. An application server provides resources needed by the various containers: transaction managers, persistence managers, name services, and, in the case of messaging and MDBs, a JMS provider.

In Sun Java System Application Server, JMS messaging resources are provided by Sun Java System Message Queue:

- For Sun Java System Application Server 7, a Message Queue messaging system is integrated into the application server as its native JMS provider.
- For the Sun J2EE 1.4 Application Server, Message Queue is plugged into the application server as an embedded JMS resource adapter (see Appendix F, "The MQ Resource Adapter," in the *Message Queue Administration Guide* for details).
- For future releases of Application Server, Message Queue will be plugged into the application server using standard resource adapter deployment and configuration methods.

Chapter 2

Quick Start Tutorial

This chapter provides a quick introduction to JMS client programming in a Sun Java System Message Queue environment. It consists of a tutorial-style description of procedures used to create, compile, and run a simple HelloWorldMessage example application.

This chapter covers the following topics:

- "Setting Up Your Environment" on page 47
- "Starting and Testing the Message Server" on page 49
- "Developing a Simple Client Application" on page 51
- "Compiling and Running a Client Application" on page 54
- "Example Application Code" on page 55

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 1.2.2.

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 is a SOAP client or a SOAP servlet
- whether your application uses the SOAP/JMS transformer utilities
- the JDK version you are using (which affects JNDI support).

Table 2-1 specifies the directories where jar files are to be found on the different platforms:

Table 2-1 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/imq/lib/`
Windows	`IMQ_HOME\lib\`

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

Table 2-2 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 6, "Working With SOAP Messages"

SOAP Servlet	`jaxm-api.jar`
	`saaj-api.jar`
	`activation.jar`
			SOAP servlets can run in the App Server 7 without additional runtime support.

Table 2-2 jar Files Needed in `CLASSPATH` *(Continued)*

Code	To Compile	To Run	Discussion
code using SOAP/JMS transformer utilities	`imqxm.jar` (and jars for JMS and SOAP clients)		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 methods belonging to the `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 context (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 2-3 Location of Message Queue Executables

Platform	Location
Solaris	`/usr/bin/`
Linux	`/opt/imq/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 2-3).

2. Run `imqcmd` with the arguments shown below.

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

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

```
% imqcmd query bkr -u admin -p admin

Querying the broker specified by:

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

Version                                            3.5 SP1
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.*;
   ```

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 59). 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 60). 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 (see "Message Consumption: Synchronous and Asynchronous" on page 42).

    ```
    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, though versions 1.3 and 1.2 are 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 47, before attempting to compile or run a client application.

The following instructions are based on the HelloWorldMessage application, as created in "Developing a Simple Client Application" on page 51, and located in the Message Queue 3.5 SP1 JMS example applications directory (see Table 2-4). 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 3.5 SP1 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 3.5 SP1 consist of both JMS messaging applications as well as JAXM messaging examples (see "Working With SOAP Messages" on page 119 for more information).

Directories containing example application code are set as follows:

- Solaris: `/usr/demo/imq`
- Linux: `/opt/imq/demo`
- 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 2-4 lists and describes the contents of the directories of interest to Message Queue Java clients.

Table 2-4 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.

Table 2-4 Example Programs *(Continued)*

Directory	Contents
`applications`	Two 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.
`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.

Chapter 3

Using Administered Objects

Administered objects encapsulate provider-specific implementation and configuration information in objects that are used by Message Queue clients.

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.

`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 51). The JAXM endpoint administered object is used to send SOAP messages (see Chapter 6, "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` (and `XAConnectionFactory`) 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 "Starting Client Applications With Overrides" on page 64.

While it is possible for clients to instantiate `ConnectionFactory` (and `XAConnectionFactory`) 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 "Client Runtime Configurable Properties" on page 69 and for a complete list of attributes, see "ConnectionFactory Administered Object" on page 163.

3. Use the ConnectionFactory to create a connection object.

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

The code in the previous steps is shown in Code Example 3-1. (The directory represented by `c:/`*imq_admin_objects* must be an existing directory.)

Code Example 3-1 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();
```

The code in the previous steps is shown in Code Example 3-1.

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 supported by 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 supported by other JMS providers.

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`).

➤ To Directly Instantiate and Configure a ConnectionFactory Object

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

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

2. Configure the `ConnectionFactory` object.

   ```
   myQConnFactory.setProperty("imqBrokerHostName", "new_hostname");
   myQConnFactory.setProperty("imqBrokerHostPort", "7878");
   ```

   For a discussion of `ConnectionFactory` configuration properties, see "Client Runtime Configurable Properties" on page 69 and for a complete list of properties, see "ConnectionFactory Administered Object" on page 163.

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

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

The code in the previous steps is shown in Code Example 3-2.

Code Example 3-2 Instantiating a ConnectionFactory Object

```
com.sun.messaging.QueueConnectionFactory myQConnFactory =
    new com.sun.messaging.QueueConnectionFactory();
try {
    myQConnFactory.setProperty("imqBrokerHostName", "new_host");
    myQConnFactory.setProperty("imqBrokerHostPort", "7878");
} catch (JMSException je) {
}
QueueConnection myQConn =
    myQConnFactory.createQueueConnection();
```

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((Queue)myQueue);
   ```

The code is shown in Code Example 3-3.

Code Example 3-3 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((Queue)myQueue);
...
```

Starting Client Applications With Overrides

As with any Java application, you can start messaging applications 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 [[-Dattribute=value ]...] clientAppName
```

where `attribute` corresponds to any of the `ConnectionFactory` administered object attributes documented in "Client Runtime Configurable Properties" on page 69.

For example, if you want a client to connect to a different broker than that specified in a ConnectionFactory administered object accessed in the application code, you can start up the client using command line overrides to set the `imqBrokerHostName` and `imqBrokerHostPort` of another broker.

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 4

Configuring the Message Queue Client Runtime

The performance of client applications depends both on the inherent design of these applications and on the features and capabilities of the Message Queue client runtime.

This chapter describes how the Message Queue client runtime supports the messaging capabilities of client applications, with special emphasis on properties and behaviors that you can configure to improve performance and message throughput.

The chapter covers the following topics:

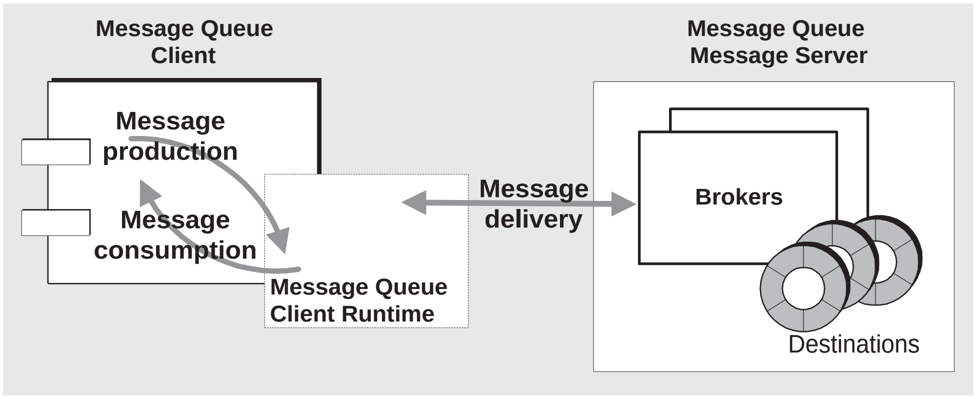
- "Message Production and Consumption" on page 65
- "Client Runtime Configurable Properties" on page 69
- "Managing Reliability and Performance" on page 84

Message Production and Consumption

The Message Queue client runtime provides client applications with an interface to the Message Queue service—it supplies these clients with all the JMS programming objects introduced in "The JMS Programming Model" on page 30. It supports all operations needed for clients to send messages to destinations and to receive messages from such destinations.

This section provides a high level description of how the Message Queue client runtime supports message production and consumption. Figure 4-1 on page 66 illustrates how message production and consumption involve an interaction between clients and the client runtime, while message delivery involves an interaction between the client runtime and the message server.

Figure 4-1 Messaging Operations



Once a client has created a connection to a broker, created a session as a single-threaded context for message delivery, and created the MessageProducer and MessageConsumer objects needed to access particular destinations in a message server, production (sending) and consumption (receiving) of messages can proceed.

Message Production

In message production, a message is created by the client, and sent over a connection to a destination on a broker. If the message delivery mode of the MessageProducer object has been set to persistent (guaranteed delivery, once and only once), the client thread blocks until the broker acknowledges that the message was delivered to its destination and stored in the broker's persistent data store. If the message is not persistent, no broker acknowledgement message (referred to as "Ack" in property names) is returned by the broker, and the client thread does not block.

In the case of persistent messages, to increase throughput, you can set the connection to *not* require broker acknowledgement (see `imqAckOnProduce` property, Table 4-9 on page 83), but this eliminates the guarantee that persistent messages are reliably delivered.

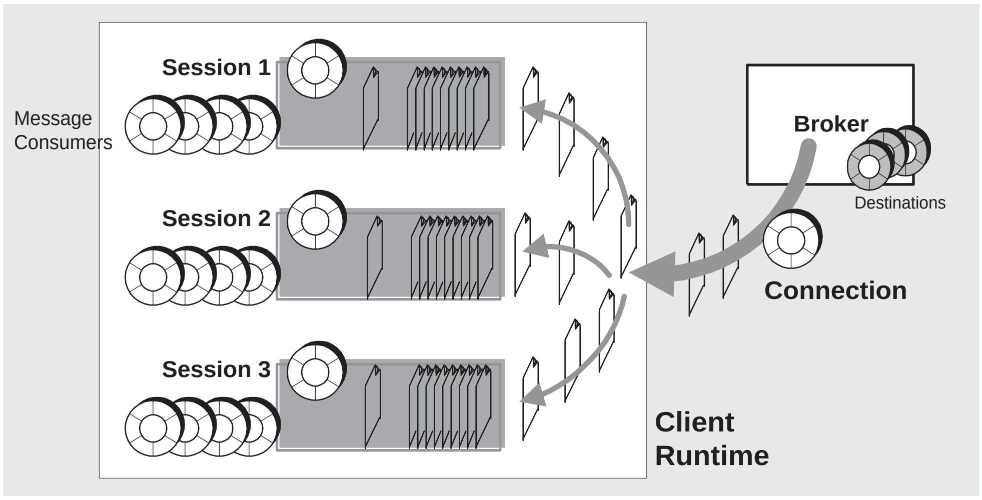
Message Consumption

Message consumption is more complex than production. Messages arriving at a destination on a broker are delivered over a connection to the client runtime under the following conditions:

- the client has set up a consumer for the given destination
- the selection criteria for the consumer, if any, match that of messages arriving at the given destination
- the connection has been told to start delivery of messages.

Messages delivered over the connection are distributed to the appropriate sessions, where they are queued up to be consumed by the appropriate MessageConsumer objects, as shown in Figure 4-2.

Figure 4-2 Message Delivery to Message Queue Client Runtime



NOTE The flow of messages delivered to the client runtime is metered at both the connection and consumer levels (see "Message Flow Metering" on page 85). By appropriately adjusting connection configuration properties, you can balance the flow of messages so that messages delivered to one session do not adversely affect the delivery of messages to other sessions on the same connection.

Messages are fetched off each session queue one at a time (a session is single threaded) and consumed either synchronously (by a client thread invoking the `receive` method) or asynchronously (by the session thread invoking the `onMessage` method of a MessageListener object).

When a broker delivers messages to the client runtime, it marks the messages accordingly, but does not really know if they have been consumed. Therefore, the broker waits for the client to acknowledge receipt of a message before deleting the message from the broker's destination. If a connection fails, and another connection is subsequently established, the broker will re-deliver all previously delivered but unconsumed messages, marking them with a `Redeliver` flag.

In accordance with the JMS specification, there are three acknowledgment modes that you can specify for a client session:

- `AUTO_ACKNOWLEDGE`: the session automatically acknowledges each message consumed by the client.
- `CLIENT_ACKNOWLEDGE`: the client explicitly acknowledges after one or more messages have been consumed. This mode gives the client the most control. This acknowledgement takes place by invoking the `acknowledge()` method of a message object, causing the session to acknowledge all messages that have been consumed by the session since the previous invocation of the method. (This could include messages consumed asynchronously by many different message listeners in the session, independent of the *order* in which they were consumed.)

NOTE Message Queue also provides a specific method you can use in `CLIENT_ACKNOWLEDGE` mode, by which you can acknowledge only the *individual* message on which you invoke the method, rather than the standard behavior. This is achieved using programming techniques described in"Custom Client Acknowledgement" on page 89.

- `DUPS_OK_ACKNOWLEDGE`: the session acknowledges after ten messages have been consumed (this value is not currently configurable) and doesn't guarantee that messages are delivered and consumed only once. Clients use this mode if they don't care if messages are processed more than once.

Each of the three acknowledgement modes requires a different level of processing and bandwidth overhead. `AUTO_ACKNOWLEDGE` consumes the most overhead and guarantees reliability on a message by message basis, while `DUPS_OK_ACKNOWLEDGE` consumes the least overhead, but allows for duplicate delivery of messages.

In the case of the `AUTO_ACKNOWLEDGE` or `CLIENT_ACKNOWLEDGE` modes, the threads performing the acknowledgement, or committing a transaction, will block, waiting for the broker to return a control message acknowledging receipt of the client acknowledgement. This broker acknowledgement (referred to as "Ack" in property names) guarantees that the broker has deleted the corresponding persistent message and will not send it twice—which could happen were the client or broker to fail, or the connection to fail, at the wrong time.

To increase throughput, you can set the connection to *not* require broker acknowledgement of client acknowledgements (see `imqAckOnAcknowledge` property, Table 4-7 on page 80), but this eliminates the guarantee that persistent messages are delivered once and only once.

NOTE In the `DUPS_OK_ACKNOWLEDGE` mode, the session does not wait for broker acknowledgements. This mode is used in clients in which duplicate messages are not a problem. Also, there is a JMS API (`recover Session`) by which a client can explicitly request redelivery of messages that have been received but not yet acknowledged by the client. When redelivering such messages, the broker marks them with a `Redeliver` flag.

Client Runtime Configurable Properties

The Message Queue client runtime supports all the operations described in "Message Production and Consumption" on page 65. It also provides a number of configurable properties that you can use to optimize resources, performance, and message throughput. These properties correspond to attributes of the `ConnectionFactory` object used to create physical connections between a client runtime and a message server.

NOTE If you wish to support distributed transactions (see "Distributed Transactions" on page 41), you need to use a special `XAConnectionFactory` object that supports distributed transactions.

A `ConnectionFactory` (or `XAConnectionFactory`) object has no physical representation in a broker—it is used simply to enable the client to establish connections with a broker and to specify behaviors of the connection and of the client runtime using the connection. (The `ConnectionFactory` object can also be used to manage Message Queue message server resources by overriding message header values set by clients—see "Message Header Overrides" on page 78.)

`ConnectionFactory` (and `XAConnectionFactory`) administered objects are created by adminstrators or instantiated in the application, as described in Chapter 3, "Using Administered Objects."

By configuring a `ConnectionFactory` (or `XAConnectionFactory`) administered object, you specify the attribute values (the properties) common to all the connections that it produces. `ConnectionFactory` and `XAConnectionFactory` objects share the same set of attributes. These attributes are grouped into a number of categories, depending on the behaviors they affect:

- Connection Handling
- Client Identification
- Message Header Overrides
- Reliability And Flow Control
- Queue Browser Behavior and Server Session
- JMS-Defined Properties Support

Each of these categories is discussed in the following sections with a description of the `ConnectionFactory` (or `XAConnectionFactory`) attributes each includes. The attribute values are set using Message Queue administration tools, as described in the *Message Queue Administration Guide*.

Connection Handling

Connections to a message server are specified by a broker host name, the port number at which the broker's Port Mapper resides (or at which a specific connection service resides), and the kind of connection service used to access the broker (see the *Message Queue Administration Guide* for a discussion of the various connection services provided by Message Queue.)

This information is provided in a message server address that is used in connecting the client runtime to a broker. In the case of a multi-broker cluster, you might specify multiple message server addresses: if a broker or a connection fails, the connection can be automatically re-established with a different broker using a different message server address.

Specifying a Message Server Address

The syntax for specifying a message server address depends upon the connection service used to access a broker, as follows:

scheme`://`*address_syntax*

where the *scheme* and *address_syntax* are described in Table 4-1.

Table 4-1 Message Server Address Schemes and Syntax

Scheme	Connection Service	Description	Address Syntax
`mq`	`jms` and ssljms	Message Queue client runtime makes a connection to the Message Queue Port Mapper at the specified host and port. The Port Mapper returns a list of the dynamically established connection service ports, and the client runtime then makes a connection to the port hosting the specified connection service.	[*hostName*][:*port*][/*serviceName*] Defaults:[1] *hostName* = `localhost` *port* = 7676 *serviceName* = `jms`
`mqtcp`	jms	Message Queue client runtime makes a tcp connection to the specified host and port (bypassing the Message Queue Port Mapper) to establish a connection.	*hostName*`:`*port*`/jms`
`mqssl`	ssljms	Message Queue client runtime makes a secure ssl connection to the specified host and port (bypassing the Message Queue Port Mapper) to establish a connection.	*hostName*`:`*port*`/ssljms`
`http`	httpjms	Message Queue client runtime makes an HTTP connection to a Message Queue tunnel servlet at the specified URL. (The broker must be configured to access the HTTP tunnel servlet, as described in the *Message Queue Administration Guide*.)	`http://`*hostName:port*`/`*contextRoot*`/tunnel`[2]

Table 4-1 Message Server Address Schemes and Syntax *(Continued)*

Scheme	Connection Service	Description	Address Syntax
`https`	httpsjms	Message Queue client runtime makes a secure HTTPS connection to the specified Message Queue tunnel servlet URL. (The broker must be configured to access the HTTPS tunnel servlet, as described in the *Message Queue Administration Guide*.)	`https://`*hostName:port*`/`*contextRoot*`/tunnel`[3]

1. Defaults only apply to the jms connection service. For the ssljms connection service, all variables need to be specified
2. If multiple broker instances are using the same tunnel servlet, then the syntax for connecting to a specific broker instance (rather than a randomly selected one) is: `http://`*hostName:port*`/`*contextRoot*`/tunnel?ServerName=`*hostName:instanceName*
3. If multiple broker instances are using the same tunnel servlet, then the syntax for connecting to a specific broker instance (rather than a randomly selected one) is: `https://`*hostName:port*`/`*contextRoot*`/tunnel?ServerName=`*hostName:instanceName*

To see how the message server address syntax applies in some typical cases, consult Table 4-2.

Table 4-2 Message Server Address Examples

Connection Service	Broker Host	Port	Example Address
Not Specified	Local Host	Not Specified	Default (`mq://localHost:7676/jms`)
Not Specified	Specified Host	Not Specified	`myBkrHost` (`mq://myBkrHost:7676/jms`)
Not Specified	Not Specified	Portmapper Port Specified	`1012` (`mq://localHost:1012/jms`)
ssljms	Local Host	Portmapper Port Not Specified	`mq://localHost:7676/ssljms`
ssljms	Specified Host	Portmapper Port	`mq://myBkrHost:7676/ssljms`
ssljms	Specified Host	Portmapper Port Specified	`mq://myBkrHost:1012/ssljms`
jms	Local Host	Service Port Specified	`mqtcp://localhost:1032/jms`
ssljms	Specified Host	Service Port Specified	`mqssl://myBkrHost:1034/ssljms`
httpjms	N/A	N/A	`http://websrvr1:8085/imq/tunnel`
httpsjms	N/A	N/A	`https://websrvr2:8090/imq/tunnel`

Connecting to a Message Server

Using a message server address provided by a `ConnectionFactory` (or `XAConnectionFactory`) attribute, a client runtime attempts to connect to the message server. If the message server address scheme involves the Message Queue Port Mapper (*scheme* = `mq`), then the Port Mapper dynamically assigns a port number and a connection is attempted to the specified port.

In some cases, you might wish to provide more than one message server address to which to make a connection. For example, in multi-broker cluster environments (Enterprise Edition only) when one broker might not be on line, you might wish to connect to another broker in the cluster. By specifying more than one address in the connection factory `imqAddressList` attribute, the system will automatically attempt a connection to a second address if a connection to the first address fails. The connection attempts continue until all addresses in a list are tried, after which the system recycles through the list a specified number of times, in attempting a connection.

Automatic Reconnect to a Message Server (Enterprise Edition)

Message Queue also provides an auto-reconnect capability, by which the client runtime can automatically reconnect to a broker if a connection fails. To enable this capability, you set the connection factory `imqReconnectEnabled` attribute.

While attempting to re-establish the connection, Message Queue maintains objects (sessions, message consumers, message producers, and so forth) provided by the client runtime. However, in circumstances where the client-side state cannot be fully restored on a broker upon reconnect (for example, when using transacted sessions or temporary destinations—which exist only for the duration of a connection), auto-reconnect will not take place, and the connection exception handler is called instead. (In such cases, application code has to catch the exception, reconnect, and restore state.)

A failed connection can be restored not only on the original broker, but also on a broker different from the original connection (that is, the reconnect is to the message server rather than to a specific broker instance within the message server cluster). To implement this behavior, you specify a list of message server addresses in the `imqAddressList` attribute.

When the client runtime needs to re-establish a connection to a message service, it will attempt a specified number of reconnect attempts (`imqReconnectAttempts`) to the original broker, each after a specified time interval (`imqReconnectInterval`). If these attempts fail, then the client runtime attempts to connect to other brokers in the list (the same number of times at the same time interval), until it finds an available broker or fails to find one. You can specify the number of times the client runtime iterates through the list in this way (`imqAddressListIterations`).

Because broker instances do not currently use a shared, highly available persistent store, persistent messages and other state information held by the failed (or disconnected) broker can be lost if a reconnect is to a broker instance different from the original. However, the ability of the client runtime to automatically reconnect to a different broker instance allows you to create recovery scenarios by which a backup broker or a broker cluster can be used for (less than complete) failover protection.

(If auto-reconnect is enabled, Message Queue persists *temporary* destinations when the associated connection fails, due to the possibility that clients might re-connect and access them again. After giving the client due time to reconnect and make use of these destinations, the broker will delete them.)

Auto-reconnect Behavior

The impact of auto-reconnect is different for message production and message consumption.

Message Production During reconnect, producers cannot send messages. The production of messages (or *any* operation that involves communication with the message server) is blocked until the connection is re-established.

Message Consumption Auto-reconnect 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. JMS application code can use this flag to determine if any message has already been consumed (but not yet acknowledged). In the case of non-durable subscribers, some messages might be lost. This is because the message server does not hold messages for non-durable subscribers once their connections have been closed. Thus, any messages produced for these subscribers while the connection is down can not be delivered when the connection is re-established.

The attributes that affect connection handling are described in Table 4-3. (Connection handling attributes used in earlier, Message Queue 3.0 versions, which continue to be supported by Message Queue 3.5 SP1, are described in "Message Queue 3.0 Connection Handling" on page 76.)

Table 4-3 Connection Factory Attributes: Connection Handling

Attribute/Property Name	Description
`imqAddressList`	Specifies a list of message server addresses (one or more), separated by commas, each corresponding to a different broker instance to which a client runtime can connect. Each address in the list specifies the host name, host port, and connection service for the connection (see "Specifying a Message Server Address" on page 71). Default: If no address is specified, this attribute defaults to an existing Message Queue 3.0 address (see "Message Queue 3.0 Connection Handling" on page 76), if any, or if not, to the first entry in Table 4-2 on page 72.
`imqAddressListBehavior`	Specifies whether connection attempts are in the order of addresses in the `imqAddressList` attribute (`PRIORITY`) or in a random order (`RANDOM`). If you have many clients attempting a connection using the same connection factory, you would use a random order to prevent them from all being connected to the same address. Default: `PRIORITY`
`imqAddressListIterations`	Specifies the number of times the client runtime will iterate through the `imqAddressList` in an effort to establish (or re-establish a connection). A value of `-1` indicates that the number of attempts is unlimited. Default: `5`
`imqReconnectEnabled`	If enabled (value = `true`), specifies that the client runtime will attempt to reconnect to a message server (or the list of addresses in `imqAddressList`) when a connection is lost. Default: `false`
`imqReconnectAttempts`	Specifies the number of attempts to connect (or reconnect) for each address in the `imqAddressList` before the client runtime moves on to try the next address in the list. A value of `-1` indicates that the number of reconnect attempts is unlimited (the client runtime will attempt to connect to the first address until it succeeds). Default: `0`
`imqReconnectInterval`	Specifies the interval between reconnect attempts. this applies for attempts on each address in the `imqAddressList` and for successive addresses in the list. If too short, this time interval does not give a broker time to recover. If too long, the reconnect might represent an unacceptable delay. Default: `3000 milliseconds`

Message Queue 3.0 Connection Handling

Connection handling attributes used in earlier, Message Queue 3.0 versions continue to be supported by Message Queue 3.5 SP1, for purposes of compatibility. These attributes are shown in Table 4-4.

The Message Queue 3.0 attributes should not be used. They correspond to a different connection handling approach that does not support multiple message server addresses for establishing a connection.

If an address is specified in the imqAddressList attribute (see Table 4-3), then any existing Message Queue 3.0 connection handling attributes will be ignored.

Table 4-4 Supported Message Queue 3.0 Connection Handling Attributes

Attribute/Property Name	Description
`imqConnectionType`	Specifies transport protocol of the connection service used by the client. Supported types are `TCP`, `TLS`, `HTTP`. Default: `TCP`
`imqBrokerHostName`	Specifies the broker host name to which to connect (if `imqConnectionType` is either `TCP` or `TLS`). Default: `localhost`
`imqBrokerHostPort`	Specifies the broker host port (if `imqConnectionType` is either `TCP` or `TLS`). Default: `7676`
`imqBrokerServicePort`	Specifies a port on which a connection should be attempted (if `imqConnectionType` is either `TCP` or `TLS`), bypassing a connection through the broker host port (Port Mapper port). This attribute is used mainly to provide for connections through a firewall, in which case you want to minimize the number of open ports. To use this feature, you have to start a specific service on a specific port using the broker's connection service configuration properties (see the *Message Queue Administration Guide*). Default: `0` (not used)
`imqSSLIsHostTrusted`	Specifies whether the client should trust the Message Queue broker and accept self-signed certificates from the broker when the `imqConnectionType` is `TLS`. If this is set to `false`, then either the broker's (self-signed) certificate must be installed in the client's keystore, or the broker's certificate must be signed by a Certificate Authority (CA) that is trusted by the client. (That is, the CA's root certificate is either one that is shipped with the JRE or is installed in the client's keystore.) Default: `true`

Table 4-4 Supported Message Queue 3.0 Connection Handling Attributes

Attribute/Property Name	Description
`imqConnectionURL`	Specifies the URL that will be used to connect to the message server (if `imqConnectionType` is `HTTP`). A typical value (HTTPS connection) might be `https://`*hostName:port*`/imq/tunnel` Default: `http://localhost/imq/tunnel`

Client Identification

Clients need to be identified to a broker both for authentication purposes and to keep track of durable subscriptions (see "Client Identifiers" on page 39).

For authentication purposes, Message Queue provides a default user name and password. These are a convenience for developers who do not wish to explicitly populate a user repository (see the *Message Queue Administration Guide*) to perform application testing.

To keep track of durable subscriptions, Message Queue uses a unique client identification (ClientID). If a durable subscriber is inactive at the time that messages are delivered to a topic destination, the broker retains messages for that subscriber and delivers them when the subscriber once again becomes active. The only way for the broker to identify the subscriber is through its ClientID.

There are a number of ways that the ClientID can be set for a connection. For example, application code can use the `setClientID()` method of a `Connection` object. The ClientID must be set before using the connection in any way; once the connection is used, the ClientID cannot be set or reset.

Setting the ClientID in application code, however, is not optimal. Each user needs a unique identification: this implies some centralized coordination. Message Queue therefore provides a `imqConfiguredClientID` attribute on the ConnectionFactory object. This attribute can be used to provide a unique ClientID to each user. To use this feature, the value of `imqConfiguredClientID` is set as follows:

`imqConfiguredClientID=${u}`*string*

where the special reserved characters, ${u}, provide a unique user identification during the user authentication stage of establishing a connection, and *string* is a text value unique to the ConnectionFactory object. When used properly, the message server will substitute `u:`*userName* for the `u`, resulting in a user-specific ClientID.

The `${u}` must be the first four characters of the attribute value. If anything other than "u" is encountered, it will result in an JMS exception upon connection creation. When `${}` is used anywhere else in the attribute value, it is treated as plain text and no variable substitution is performed.

An additional attribute, `imqDisableSetClientID`, can be set to `true` to disallow clients that use the connection factory from changing the configured ClientID through the `setClientID()` method of the `Connection` object.

It is required that you set the client identifier whenever using durable subscriptions in deployed applications, either programmatically using the `setClientID()` method or using the `imqConfiguredClientID` attribute of the `ConnectionFactory` object.

The attributes that affect client identification are described in Table 4-5.

Table 4-5 Connection Factory Attributes: Client Identification

Attribute/Property Name	Description
`imqDefaultUsername`	Specifies the default user name that will be used to authenticate with the broker. Default: `guest`
`imqDefaultPassword`	Specifies the default password that will be used to authenticate with the broker. Default: `guest`
`imqConfiguredClientID`	Specifies the value of an administratively configured ClientID. Default: `null`
`imqDisableSetClientID`	Specifies if client is prevented from changing the ClientID using the `setClientID()` method in the JMS API. Default: `false`

Message Header Overrides

A Message Queue administrator can override JMS message header fields that specify the persistence, lifetime, and priority of messages. Specifically, values in the following fields can be overridden (see "The Java XML Messaging (JAXM) Specification" on page 23):

- JMSDeliveryMode (message persistence/non-persistence)
- JMSExpiration (message lifetime)
- JMSPriority (message priority—an integer from 0 to 9)

The ability to override message header values gives an administrator more control over the resources of a message server. Overriding these fields, however, has the risk of interfering with application-specific requirements (for example, message persistence). So this capability should only be used in consultation with the appropriate application users or designers.

Message Queue allows message header overrides at the level of a connection: overrides apply to all messages produced in the context of a given connection, and are configured by setting attributes of the corresponding connection factory administered object. These attributes are described in Table 4-6.

Table 4-6 Connection Factory Attributes: Message Header Overrides

Attribute/Property Name	Description
`imqOverrideJMSDeliveryMode`	Specifies whether client-set `JMSDeliveryMode` field can be overridden. Default: `false`
`imqJMSDeliveryMode`	Specifies the override value of `JMSDeliveryMode`. Values are `1` (non-persistent) and `2` (persistent). Default: `2`
`imqOverrideJMSExpiration`	Specifies whether client-set `JMSExpiration` field can be overridden. Default: `false`
`imqJMSExpiration`	Specifies the override value of `JMSExpiration` (in milliseconds). Default: `0` (does not expire)
`imqOverrideJMSPriority`	Specifies whether client-set `JMSPriority` field can be overridden. Default: `false`
`imqJMSPriority`	Specifies the override value of `JMSPriority` (an integer from 0 to 9). Default: `4` (normal)
`imqOverrideJMSHeadersTo` `TemporaryDestinations`	Specifies whether overrides apply to temporary destinations. Default: `false`

Reliability And Flow Control

A number of attributes determine the use and flow of Message Queue control messages by the client runtime, especially broker acknowledgements (referred to as "Ack" in the attribute names).

The attributes that affect reliability and flow control are described in Table 4-7. For an extended discussion of these settings and the effect of various permutations, see "Managing Reliability and Performance" on page 84.

Table 4-7 Connection Factory Attributes: Reliability and Flow Control

Attribute/Property Name	**Description**
`imqAckTimeout`	Specifies the maximum time in milliseconds that the client runtime will wait for any broker acknowledgement before throwing an exception. A value of 0 means there is no time-out—the client runtime will wait forever. Default: `0` In some situations, for example, the first time a broker authenticates a user against an LDAP user repository over a secure (SSL) connection, it can take upwards of 30 seconds to complete authentication. If `imqAckTimeout` is set too small, the client runtime can time out.
`imqAckOnProduce`	Specifies broker acknowledgement of messages from producing client: If set to `true`, the broker acknowledges receipt of all JMS messages (persistent and non-persistent) from producing client, and producing client thread will block waiting for those acknowledgements (referred to as "Ack" in property name). If set to `false`, broker does not acknowledge receipt of any JMS message (persistent or non-persistent) from producing client, and producing client thread will not block waiting for broker acknowledgements. If not specified, broker acknowledges receipt of *persistent* messages only, and producing client thread will block waiting for those acknowledgements. Default: not specified
`imqAckOnAcknowledge`	Specifies broker response to a consuming client when the client acknowledges a consumed message: If set to true, broker acknowledges all consuming client acknowledgements, and consuming client thread will block waiting for such broker acknowledgements (referred to as "Ack" in property name). If set to false, broker does not acknowledge any consuming client acknowledgements, and consuming client thread will not block waiting for such broker acknowledgements. If not specified, broker acknowledges consuming client acknowledgements for AUTO_ACKNOWLEDGE and CLIENT_ACKNOWLEDGE mode (and consuming client thread will block waiting for such broker acknowledgements), but does not acknowledge consuming client acknowledgements for DUPES_OK_ACKNOWLEDGE mode (and consuming client thread will not block.) Default: not specified

Table 4-7 Connection Factory Attributes: Reliability and Flow Control *(Continued)*

Attribute/Property Name	Description
`imqConnectionFlowCount`	Specifies the number of JMS messages in a metered batch. When this number of JMS messages is delivered to the client runtime, delivery is temporarily suspended, allowing any control messages that had been held up to be delivered. Payload message delivery is resumed upon notification by the client runtime, and continues until the count is again reached. If the count is set to 0 then there is no restriction in the number of JMS messages in a metered batch. A non-zero setting allows the client runtime to meter message flow so that Message Queue control messages are not blocked by heavy JMS message delivery. Default: `100`
`imqConnectionFlowLimitEnabled`	If enabled (value = `true`), the value of `imqConnectionFlowLimit` is used to limit message flow at the connection level. Default: `false`
`imqConnectionFlowLimit`	Specifies a limit on the number of messages that can be delivered over a connection and buffered in the client runtime, waiting to be consumed. Note however, that unless `imqConnectionFlowIsLimited` is enabled, this limit is not checked. When the number of JMS messages delivered to the client runtime (in accordance with the flow metering governed by `imqConnectionFlowCount`) exceeds this limit, message delivery stops. It is resumed only when the number of unconsumed messages drops below the value set with this property. This limit prevents a consuming client that is taking a long time to process messages from being overwhelmed with pending messages that might cause it to run out of memory. Default: `1000`

Table 4-7 Connection Factory Attributes: Reliability and Flow Control *(Continued)*

Attribute/Property Name	Description
`imqConsumerFlowLimit`	Specifies a limit on the number of messages *per consumer* that can be delivered over a connection and buffered in the client runtime, waiting to be consumed. This limit is used to improve load-balancing among consumers in multi-consumer queue delivery situations (no one consumer can be sent a disproportionate number of messages). This limit can be overridden by a lower value set on the broker side for the queue's `consumerFlowLimit` attribute (see information on destination attributes in the *Message Queue Administration Guide*). This limit also helps prevent any one consumer on a connection from starving other consumers on the connection. When the number of JMS messages delivered to the client runtime exceeds this limit for *any* consumer, message delivery for that consumer stops. It is resumed only when the number of unconsumed messages for that consumer drops below the value set with `imqConsumerFlowThreshold`. (Note that if the total number of messages buffered for *all* consumers on a connection exceeds the `imqConnectionFlowLimit`, then delivery of messages through the connection will stop until that total drops below the connection limit.) Default: `100`
`imqConsumerFlow Threshold`	Specifies, as a percentage of `imqConsumerFlowLimit`, the number of messages *per consumer* buffered in the client runtime, below which delivery of messages for a consumer will resume. For more information, see "Message Flow Limits" on page 86. Default: `50`

Queue Browser Behavior and Server Session

The attributes that affect queue browsing for the client runtime are described in Table 4-8.

Table 4-8 Connection Factory Attributes: Queue Browser Behavior

Attribute/Property Name	Description
`imqQueueBrowserMax MessagesPerRetrieve`	Specifies the maximum number of messages that the client runtime will retrieve at one time, when browsing the contents of a queue destination. Default: `1000`

Table 4-8 Connection Factory Attributes: Queue Browser Behavior *(Continued)*

Attribute/Property Name	Description
`imqQueueBrowserRetrieveTimeout`	Specifies the maximum time that the client runtime will wait to retrieve messages, when browsing the contents of a queue destination, before throwing an exception. Default: `60000 milliseconds`.
`imqLoadMaxToServerSession`	Used only for JMS application server facilities. Specifies whether a Message Queue ConnectionConsumer should load up to the `maxMessages` number of messages into a ServerSession's session (value=`true`), or load only a single message at a time (value=`false`). Default: `true`

JMS-Defined Properties Support

JMS-defined properties are property names reserved by JMS, and which a JMS provider can choose to support (see "The Java XML Messaging (JAXM) Specification" on page 23). These properties enhance client programming capabilities.

The JMS-defined properties supported by Message Queue are described in Table 4-9.

Table 4-9 Connection Factory Attributes: JMS-defined Properties Support

Attribute/Property Name	Description
`imqSetJMSXUserID`	Specifies whether Message Queue should set the JMS-defined property, `JMSXUserID` (identity of user sending the message), on produced messages. Default: `false`
`imqSetJMSXAppID`	Specifies whether Message Queue should set the JMS-defined property, `JMSXAppID` (identity of application sending the message), on produced messages. Default: `false`
`imqSetJMSXProducerTXID`	Specifies whether Message Queue should set the JMS-defined property, `JMSXProducerTXID` (transaction identifier of the transaction within which this message was produced), on produced messages. Default: `false`
`imqSetJMSXConsumerTXID`	Specifies whether Message Queue should set the JMS-defined property, `JMSXConsumerTXID` (transaction identifier of the transaction within which this message was consumed), on consumed messages. Default: `false`

Table 4-9 Connection Factory Attributes: JMS-defined Properties Support *(Continued)*

Attribute/Property Name	Description
`imqSetJMSXRcvTimestamp`	Specifies whether Message Queue should set the JMS-defined property, `JMSXRcvTimestamp` (the time the message is delivered to the consumer), on consumed messages. Default: `false`

Managing Reliability and Performance

Because of the mechanisms by which messages are delivered to and from a broker, and because of the Message Queue control messages used to assure reliable delivery, there are a number of factors that affect reliability and performance. Some of these factors depend on messaging application design (delivery mode and acknowledgement mode) and some depend on client runtime behaviors (message flow metering and message flow limits).

Although these factors are quite distinct, their interactions can complicate the task of balancing reliability with performance. Specifically, because JMS messages and Message Queue control messages flow across the same connection between the client and the broker, you need to understand how to balance the requirement for reliability with the need for throughput.

This section describes the factors that affect reliability and performance, and the connection factory attributes that help manage message flow.

Delivery Mode

The delivery mode specifies whether a message is to be delivered at most once (non-persistent) or once and only once (persistent). These different reliability requirements imply different degrees of overhead. Specifically, the management of persistent messages requires greater use of broker control messages flowing across a connection.

Client Acknowledgement Mode

The setting of the client acknowledgement mode impacts reliability and affects the number of client and broker acknowledgement messages passing over a connection:

- In the AUTO_ACKNOWLEDGE mode, a client acknowledgement and broker acknowledgement (a confirmation of the client acknowledgement) are required for each consumed message, and the delivery thread blocks waiting for the broker acknowledgement.

  If this mode, with a synchronous receiver, it is possible for a message to be partially processed, but lost, if the system fails before the message is consumed. For increased reliability, you can use the CLIENT_ACKNOWLEDGE mode or a transacted session to guarantee no message is lost if the system fails.

- In the CLIENT_ACKNOWLEDGE mode client acknowledgements and broker acknowledgements are batched (rather than being sent one by one). This conserves connection bandwidth and generally reduces the overhead for broker acknowledgements, as compared to the AUTO_ACKNOWLEDGE mode. Of course, if in this mode, the client acknowledges each message, no batching will occur, and the acknowledgements are sent one by one.

- In the DUPS_OK_ACKNOWLEDGE mode, throughput is improved even further, because client acknowledgements are batched and because the client thread does not block (broker acknowledgements are not requested). However, in this case, the same message can be delivered and consumed more than once.

Message Flow Metering

Messages sent and received by clients (JMS messages) and Message Queue control messages pass over the same client-broker connection. Because of this, delays may occur in the delivery of control messages, such as broker acknowledgements, if these are held up by the delivery of JMS messages. To prevent this type of congestion, Message Queue meters the flow of JMS messages across a connection.

JMS messages are batched (as specified with the imqConnectionFlowCount property) so that only a set number are delivered; when the batch has been delivered, delivery of JMS messages is suspended, and pending control messages are delivered. This cycle repeats, as other batches of JMS messages are delivered, followed by queued up control messages.

The value of imqConnectionFlowCount should be kept low if the client is doing operations that require many responses from the broker; for example, the client is using the CLIENT_ACKNOWLEDGE or AUTO_ACKNOWLEDGE modes, persistent messages, transactions, queue browsers, or if the client is adding or removing consumers. If, on the other hand, the client has only simple consumers on a connection using DUPS_OK mode, you can increase imqConnectionFlowCount without compromising performance.

Message Flow Limits

There is a limit to the number of JMS messages that the Message Queue client runtime can handle before encountering local resource limitations, such as memory. When this limit is approached, performance suffers. Hence, Message Queue lets you limit the number of messages per consumer that can be delivered over a connection (`imqConsumerFlowLimit`) and buffered in the client runtime, waiting to be consumed.

When the number of JMS messages delivered to the client runtime exceeds this limit (`imqConsumerFlowLimit`) for any consumer, message delivery for that consumer stops. It is resumed only when the number of unconsumed messages for that consumer drops below the value set with `imqConsumerFlowThreshold`. The following example illustrates the use of these limits: consider the default settings for topic consumers

```
imqConsumerFlowLimit=1000
```

```
imqConsumerFlowThreshold=50
```

When the consumer is created, the broker delivers the initial batch of 1000 messages (providing they exist) to this consumer without pausing. After sending 1000 messages, the broker stops delivery until the client runtime asks for more messages. The client runtime holds these messages until the application processes them. The client runtime then allows the application to consume at least 50% (`imqConsumerFlowThreshold`) of the message buffer capacity (i.e. 500 messages) before asking the broker to send the next batch.

In the same situation, if the threshold were 10%, the client runtime would wait for the application to consume at least 900 messages before asking for the next batch. The next batch size is calculated as follows:

```
imqConsumerFlowLimit - (current # of pending msgs in buffer)
```

So, if `imqConsumerFlowThreshold` is 50%, the next batch size can fluctuate between 500 and 1000, depending on how fast the application can process the messages. Thus, the protocol guarantees two things:

- that the client runtime will never hold more than 1000 undelivered messages
- that the batch size will always be greater than the threshold

If the `imqConsumerFlowThreshold` is too high (close to 100%), the broker will tend to send smaller batches, which can lower message throughput. If the value is too low (close to 0%), the broker might be able to finish the remaining buffered messages before the broker delivers the next set. This can also cause message throughput degradation. Thus, for most applications, it only makes sense to tune the `imqConsumerFlowLimit` value because it controls memory requirements. Unless you have specific performance or reliability concerns, there is no need to fine tune the `imqConsumerFlowThreshold` attribute.

These consumer-based flow controls are the best way to manage memory in the client runtime. Generally, depending on the client application, you know the number of consumers you need to support on any connection, the size of the messages, and the total amount of memory that is available to the client runtime.

In the case of some client applications, however, the number of consumers might be indeterminate, depending on choices made by end users. In those cases, you can still manage memory, using connection-level flow limits.

Connection-level flow controls limit the total number of messages buffered for *all* consumers on a connection. If this number exceeds the `imqConnectionFlowLimit`, then delivery of messages through the connection will stop until that total drops below the connection limit. (The `imqConnectionFlowLimit` is only enabled if you set the `imqConnectionFlowLimitEnabled` property to `true`.)

The number of messages queued up in a session is a function of the number of message consumers using the session and the message load for each consumer. If a client is exhibiting delays in producing or consuming messages, you can normally improve performance by redesigning the application to distribute message producers and consumers among a larger number of sessions or to distribute sessions among a larger number of connections.

Chapter 5

Message Queue Client Programming Techniques

Some features and capabilities of Message Queue go beyond the JMS specification. If you want to write client applications that leverage the power of these features (which are specific to Message Queue), use the techniques described here. The chapter provides programming guidelines and examples for developing clients that make use of the following Message Queue service features:

- "Custom Client Acknowledgement" on page 89
- "Message-Based Monitoring API" on page 92
- "Client Connection Failover (Auto-reconnect)" on page 108
- "Other Programming Topics" on page 113
  - "Managing Memory and Message Size" on page 114
  - "Using Secure HTTP Connections (HTTPS)" on page 114
  - "Managing Client Threads" on page 115
  - "Synchronous Consumption in Distributed Applications" on page 117
  - "Client Application Deployment Considerations" on page 118

Custom Client Acknowledgement

As discussed in "Message Consumption" on page 67, Message Queue supports several JMS acknowledgement modes. These modes let message consumers in a session acknowledge the messages they have consumed. The different modes affect the performance and reliability of message delivery. 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 individual message at a time.

Observe the following rules when implementing custom client acknowledgement:

- When you code an acknowledgement of an individual message, call the `acknowledgeThisMessage()` method. When you code an acknowledgement of all messages consumed so far, call the `acknowledgeUpThroughThisMessage()` method. Both are shown in Code Example 5-1.

Code Example 5-1 Syntax for `acknowledgeThisMessage()` Method

```
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 try to mix custom-acknowledgement sessions and 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 5-2 demonstrates both types of custom client acknowledgement.

Code Example 5-2 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();

... [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();
```

Message-Based Monitoring API

By using the Message Queue metrics monitoring capability, a broker can write metrics data into messages which the broker then sends to one of a number of metrics topic destinations. The destination depends on the type of metrics data in a given message. You get access to this metrics data when you write a client application that does three things:

- Subscribes to the metrics topic destinations
- Consumes the messages in those destinations
- Processes the metrics data that the messages contain

The message-based monitoring API and other metrics monitoring tools are described in the *Message Queue Administration Guide*.

Table 5-1 shows the five metrics topic destinations and the type of metrics messages each destination can receive.

Table 5-1 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.`*monitored_destination_name*	Destination metrics for a queue of the specified name, such as number of consumers, message flow or volume, or disk usage
`mq.metrics.destination.topic.`*monitored_destination_name*	Similar destination metrics for a topic of the specified name

For an example of how these metrics messages can be useful, when a particular limit has been reached, you might want to program in an alert and a response action (such as sending mail to the administrator).

As explained in the *Message Queue Administration Guide* (look up "metrics messages" and "configuration files"), you can do the same thing manually by using the `metrics` command utility. However, if that manual approach isn't appropriate for your purposes, you can write your JMS client so that it automatically consumes metrics messages and displays output in a convenient format.

Monitoring topics have destination names beginning with `mq.` (Always include the period.) These names are reserved for use by Message Queue.

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

When a metrics subscriber is detected, the broker automatically creates the metrics topic. A metrics monitoring topic can't be created using an administrative command. Only the broker can publish messages to a metrics monitoring topic.

You specify how often to receive metrics information by configuring a property in the broker's `config.properties` file. All the destinations receiving metrics on that broker receive them at that same specified interval. (For information on how to set that interval, refer to the *Message Queue Administration Guide*.)

This API is designed for monitoring the broker. It's not designed for doing administrative tasks on the broker such as:

- Creating, managing, destroying, or purging physical destinations
- Configuring the broker or updating the broker's properties
- Shutting down or restarting the broker

For information on how to use Message Queue administration tools to do those tasks, refer to the *Message Queue Administration Guide*.

Format of Metrics Messages

Subscribers to metrics topics receive JMS messages of type `MapMessage`. (See "Message Body Types" on page 32 for details.) The header of a metrics message contains two properties: `type` and `timestamp`. The `type` property is useful if the same subscriber processes more than one type of metrics message—for example, messages from topics `mq.metrics.broker` and `mq.metrics.jvm`. The `timestamp` property is useful for calculating rates or drawing graphs.

The body of the message contains name-value pairs, and the data depends on the type of metrics message. The format of each metrics message type is explained in the following tables.

Notice these points:

- The names used for extracting data are case-sensitive. For example:
    - Incorrect: `NumMsgsOut`
    - Correct: `numMsgsOut`
- Each metrics message type has a defined set of name-value pairs. A name that is specific to a particular message type can be used only with that type. For example, the name `freeMemory` can't be used with a message received from the topic `mq.metrics.broker`; it can be used only with a message received from the topic `mq.metrics.jvm`.

Broker Metrics

The messages you receive when you subscribe to the topic `mq.metrics.broker` have the following message properties (Table 5-2) and metrics data in the message body (Table 5-3).

Table 5-2 Broker Metrics Message Properties

Property	Type	Value or Description
`type`	String	`mq.metrics.broker`
`timestamp`	long	Timestamp in milliseconds when metric sample was taken

Table 5-3 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

Table 5-3 Data in the Body of a Broker Metrics Message *(Continued)*

Metric Name	Value Type	Description
`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 following message properties (Table 5-4) and metrics data in the message body (Table 5-5):

Table 5-4 JVM Metrics Message Properties

Property	Type	Value or Description
`type`	String	`mq.metrics.jvm`
`timestamp`	long	Timestamp in milliseconds when the metric sample was taken

Table 5-5 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 following properties (Table 5-6):

Table 5-6 Destination-List Message Properties

Property	Type	Value or Description
`type`	String	`mq.metrics.destination_list`
`timestamp`	long	Timestamp in milliseconds when the metric sample was taken

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 5-7 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 5-7, 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 5-7 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* or the topic `mq.metrics.destination.topic.`*monitored_destination_name* have the following message properties (Table 5-8) and metrics data in the message body (Table 5-9):

Table 5-8 Destination Metrics Message Properties

Property	Type	Value or Description
`type`	String	`mq.metrics.destination.queue.`*monitored_destination_name* or `mq.metrics.destination.topic.`*monitored_destination_name*
`timestamp`	long	Timestamp in milliseconds when the metric sample was taken

Table 5-9 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)
`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

Table 5-9 Data in the Body of a Destination Metrics Message *(Continued)*

Metric Name	Value Type	Description
`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

Configuring Metrics Message Production on the Broker

When you use Message Queue, metrics message production is enabled by default for your client application. However, the Message Queue administrator must use the broker properties to set the reporting interval (to specify how often metrics updates are reported), and to specify whether metrics messages are persistent and how long they are to "live" in their destinations.

For details on configuring broker properties, refer to the *Message Queue Administration Guide*.

Also, without certain security features that Message Queue 3.5 SP1 provides, someone could obtain and misuse sensitive information about a broker and its resources. An administrator should take the approach described under "metrics monitoring tools" in the *Message Queue Administration Guide* to provide the proper access control to metrics topic destinations.

Using the Message-Based Monitoring API

You use the message-based monitoring API in the same way that you would write any JMS client, except that you subscribe to a special topic, you receive messages of a specific type and format, and you process the messages in a particular way.

A client that uses the message-based monitoring API to monitor broker metrics must perform the following basic tasks:

- Create or look up a `TopicConnectionFactory` object
- Create a `TopicConnection` to the Message Queue service
- Create a `TopicSession`
- Create a metrics `Topic` destination object

- Create a `TopicSubscriber`
- Register as an asynchronous listener to the topic, or invoke the synchronous `receive()` method to wait for incoming metrics messages
- 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 message-based monitoring API is specific to Message Queue, so 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 section 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 getter method in the class `javax.jms.MapMessage` to extract the name-value pairs. The getter method depends on the value type. Three examples follow:

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

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 Chapter 2, "Quick Start Tutorial."

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 5-3 shows how to subscribe to `mq.metrics.broker`.

Code Example 5-3 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 5-3 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 5-4.

Code Example 5-4 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 5-5 shows how to subscribe to `mq.metrics.destination_list`.

Code Example 5-5 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();

metricSession = metricConnection.createTopicSession(false,
```

Code Example 5-5 Example of Subscribing to the Destination List Metrics Topic

```
                      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 5-6:

Code Example 5-6 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");
        }
    } catch (Exception e)  {
```

Code Example 5-6 Example of Processing a Destination List Metrics Message

```
            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 93, 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 5-7 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 5-8 shows how to subscribe to the metrics topic for monitoring a specified destination.

Code Example 5-8 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();
metricConnection.start();

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

Code Example 5-8 Example of Subscribing to a Destination Metrics Topic *(Continued)*

```
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 5-9:

Code Example 5-9 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++] =

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

Code Example 5-9 Example of Processing a Destination Metrics Message *(Continued)*

```
             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
```

Client Connection Failover (Auto-reconnect)

Message Queue supports client connection failover. A failed connection can be restored not only on the original broker, but also on a different broker; that is, it can reconnect to the message service rather than to a specific broker instance. This reconnection does not apply in situations where the client-side state could not be fully restored on the broker upon reconnect (for example, when using transacted sessions or temporary destinations, which exist only for the duration of a connection).

Enabling Auto-reconnect

To enable this auto-reconnect behavior, you configure the connection factory `imqReconnectEnabled` attribute to `true`. You also configure the connection factory administered object 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 in the list, consider 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 any 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.

  Notice that the value `5` means "Try five times" and the value `-1` means "Don't stop trying."

- **The number of attempts to be made 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

Notice that 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, and the connection exception handler is called. Perhaps a resource that the client needs cannot be re-created. In this case, your client receives a `JMSException`, and must reconnect and restore state.

If the client is being auto-reconnected explicitly to a broker instance that is different from the original, then 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. The reason is that the various 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. Therefore, at that point, 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`.

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 the other session types, you might get more than one.
- 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

Next are examples that illustrate how to enable each type of auto-reconnect support.

Single-Broker Auto-reconnect

Configure your connection-factory object as follows:

Code Example 5-10 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`.

Note that you can ensure that the broker starts automatically with the machine 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 5-11 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 an 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 5-12 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.

Other Programming Topics

The rest of this chapter discusses the following miscellaneous topics:

- Managing Memory and Message Size
- Using Secure HTTP Connections (HTTPS)
- Managing Client Threads
- Synchronous Consumption in Distributed Applications
- Client Application Deployment Considerations

Managing Memory and Message Size

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 encounters `OutOfMemoryError` errors, chances are that not enough memory was provided to handle the size or the number of messages being consumed or produced.

The default JVM heap space is 64 meg, but your client might need more than that.

Consider the following guidelines:

- Evaluate the normal and peak system memory footprints when sizing heap space.
- You can start by doubling the heap size, as in the following command:

  ```
  java -Xmx128m MyClass
  ```
- The ideal 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.

For better manageability, break large messages into smaller parts, and use sequencing to ensure that the partial messages are concatenated properly.

Other methods of dealing with memory issues are explained in Chapter 4, "Configuring the Message Queue Client Runtime." They include metering the message flow over the client-broker connection and limiting the per-consumer message flow.

Using Secure HTTP Connections (HTTPS)

If you run your client applications in an environment secured by a firewall, you might need to have client applications communicate with brokers using the HTTP or HTTPS protocol rather than direct TCP connections. Web-based connections are usually allowed through firewalls.

The client runtime uses a transport driver and an HTTP proxy to send messages to the firewall. A tunnel servlet on the web server reaches through the firewall, pulls messages from the client's HTTP requests, and sends the messages to the broker.

Refer to the *Message Queue Administration Guide* for details on how to implement Message Queue support of HTTP and HTTPS in your JMS applications.

In Case of Server or Broker Failure

If the web server fails and is restarted, all connections are restored and there is no effect on clients. However, if the broker fails and is restarted, an exception is thrown and clients must re-establish their connections.

If both the web server and the broker fail, and the broker is not restarted, the web server restores client connections and continues waiting for a broker connection without notifying clients. To avoid this situation, always make sure the broker is restarted.

Repairing an HTTPS Tunnel Servlet Connection

If an HTTPS client can't connect to the broker through the tunnel servlet, do the following:

1. Start the servlet and the broker.
2. Use a browser to manually access the servlet through the HTTPS tunnel servlet URL.
3. Use the following administrative commands to pause and resume the connection:

   ```
   imqcmd pause svc -n httpsjms -u admin -p admin -f
   ```

   ```
   imqcmd resume svc -n httpsjms -u admin -p admin -f
   ```

When the service is resumed, your HTTPS client should be able to connect to the broker through the tunnel servlet.

Managing Client Threads

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

When the Message Queue client runtime creates a connection, two threads are created: one for consuming messages, and one to distribute and control flow for the connection. In addition, each JMS session creates a thread to deliver messages to message consumers. Thus, for example:

- If a connection has one session, three threads are created.
- If a connection has three sessions, five threads are created.

When you create several 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 message flow while inundating other consumers. So, if the message rate across these consumers is high enough to cause an imbalance, you might want to separate the consumers into different sessions.

Note that the JMS specification restricts a session for use by a single thread at a time. Violating this restriction can result in a deadlocked client.

You can reduce the number of threads by using fewer connections and fewer sessions. However, doing this might slow your application's throughput.

Finally, you might be able to use certain JVM runtime options to improve thread memory usage and performance. Refer to the JDK documentation for details.

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:

Option	**Result**
`Xss128K`	This decreases the memory size of the heap.
`xconcurrentIO`	This improves thread performance in the 1.3 VM.

Try to observe the following "golden rules" of thread management in your JMS applications:

- Don't use more than one thread at a time in a session.
- If a session has an asynchronous consumer, don't operate on that session outside the message listener except to close the session.

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 exits with a `null`.

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

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

Client Application Deployment Considerations

When you are ready to move your client application into production, you should make sure the administrator knows what the application requires. You can start with the following checklist if you like, but tailor it to your environment and your administrator's needs.

Table 5-10 Starter Checklist for the Message Queue Administrator

- Configuring administered objects:
  - Connection factories to be created
    - Type:
    - JNDI lookup name:
    - Physical name (if your administrator wants it):
    - Other attributes:
  - Destination objects to be created
    - Type (queue or topic):
    - JNDI lookup name:
    - Physical name (if your administrator wants it):
    - Other attributes:
- Configuring a broker or broker cluster:
    - Name:
    - Number of destinations:
    - Maximum number of messages expected:
    - Maximum size of messages expected:
    - Maximum message bytes expected:
    - Access control and other security requirements:
    - For broker cluster:
      - Load-balancing requirements:
      - Geographic distribution:
    - Auto-reconnect implementation model, if any:
- Configuring physical destinations:
    - Type:
    - Name:
    - Attributes:
    - Maximum number of messages expected:
    - Maximum size of messages expected:
    - Maximum message bytes expected:

In regard to geographic distribution, notice that clients can be grouped in different areas to minimize traffic over long links.

For details on configuration, refer to the *Message Queue Administration Guide*.

Chapter 6

Working With SOAP Messages

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 explains how you do the following:

- Send and receive SOAP messages without using Message Queue
- Send and receive JMS messages that contain a SOAP payload

This chapter begins with an overview of SOAP processing and describes the Java API for SOAP with attachments (JAXM). You need to know this information to process SOAP messages. The chapter concludes by explaining how you can create a JMS message that contains a SOAP message payload.

If you are familiar with the SOAP specification, you can skip the introductory section and start by reading "SOAP Messaging in JAVA" on page 127.

The remaining sections in this chapter cover the following topics:

- "Using JAXM Administered Objects" on page 136
- "SOAP Messaging Models and Examples" on page 138
- "Integrating SOAP and Message Queue" on page 152

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.

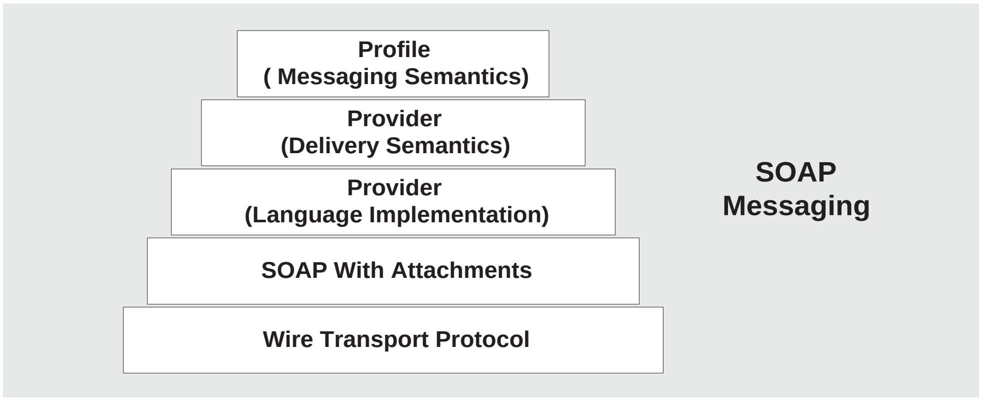
The SOAP protocol is fully described at `http://www.w3.org/TR/SOAP`. This section restricts itself to discussing the reasons why you would use SOAP and to describing some basic concepts that will make it easier to work with the JAXM API.

SOAP and the JAVA for XML Messaging API

The JAVA API for XML messaging (JAXM) 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. JAXM 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 6-1 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 6-1 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 Provider Layer

In Figure 6-1, the provider is shown as two pieces of functionality: a language implementation and delivery semantics.

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

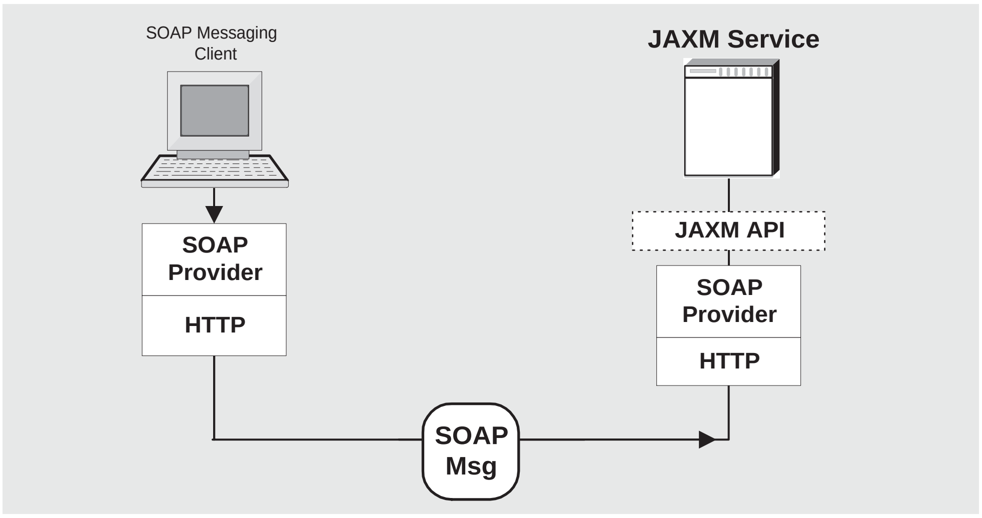
Messaging Semantics

In addition to a language implementation, a SOAP provider can offer services that relate to message delivery. These could include reliability, persistence, security, and administrative control. 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 6-2, 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 6-2 SOAP Interoperability



In order for a JAXM client or service to interoperate with a service or client using a different provider, 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.

Profiles provide additional processing information, as described next.

The Profiles Layer

The final, *profile,* layer of SOAP messaging governs messaging semantics between business partners who use SOAP messaging with SOAP providers. A *profile* is an industry standard, such as “ebxml”, which defines additional rules for message processing. A provider can add profile information to the header of a message when its message factory creates the message. (The SOAP message header is the primary means of SOAP messaging extensibility.) Support for the ebxml profile will be added in future releases of Message Queue.

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 JAXM libraries, you must still understand its structure in order to make the JAXM 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.

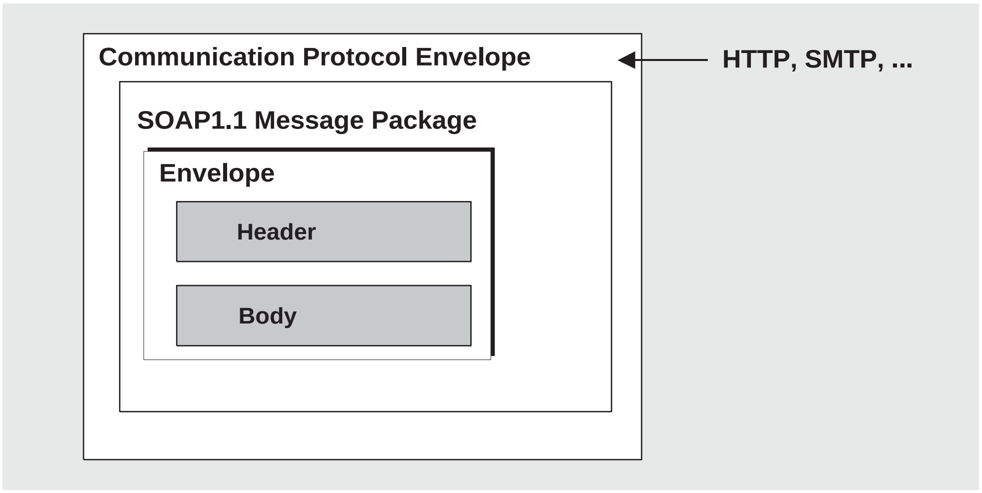
A SOAP message is constructed like a nested matrioshka doll. When you use JAXM to assemble or disassemble a message, you need to make the API calls in the appropriate order to get to the message part that interests you. For example, in order to add content to the message, you need to get to the body 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 127.

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 JAXM to define SOAP messages and their parts, it will be helpful for you to be familiar with this information.

Figure 6-3 shows the SOAP model without attachments. This package includes a SOAP envelope, a header, and a body. The header is optional.

Figure 6-3 SOAP Message Without Attachments

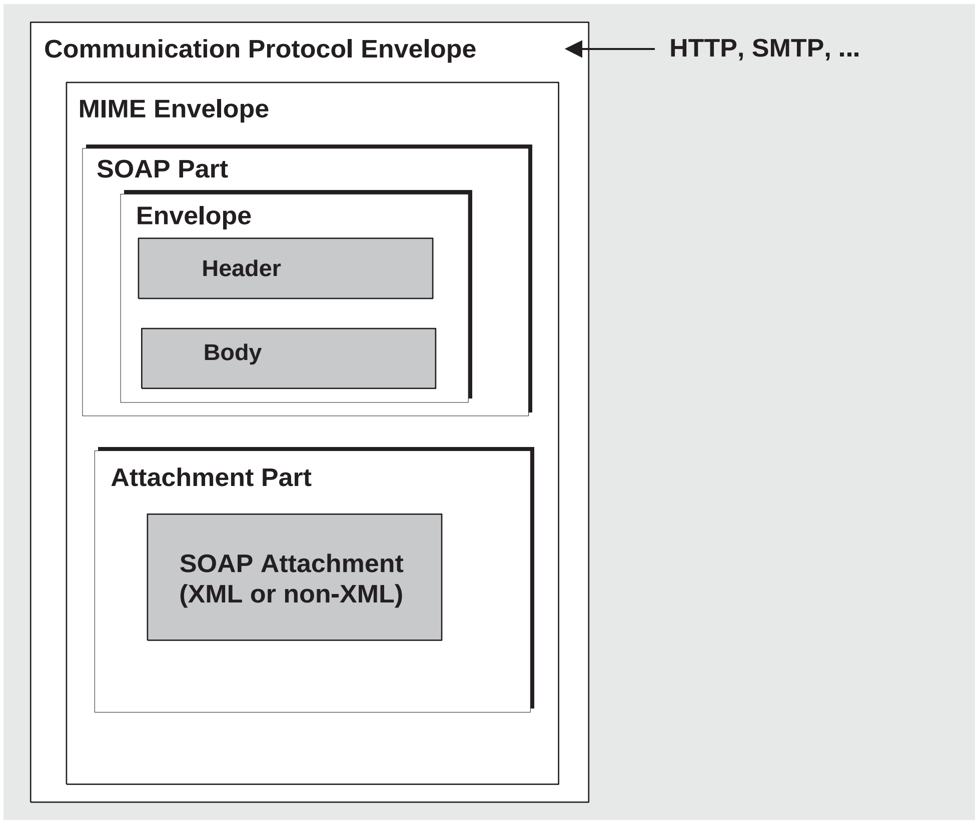


When you construct a SOAP message using JAXM, you do not have to specify which model you're following. If you add an attachment, a message like that shown in Figure 6-4 is constructed; if you don't, a message like that shown in Figure 6-3 is constructed.

Figure 6-4 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 JAXM, 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 6-4 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.

JAXM 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. JAXM 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

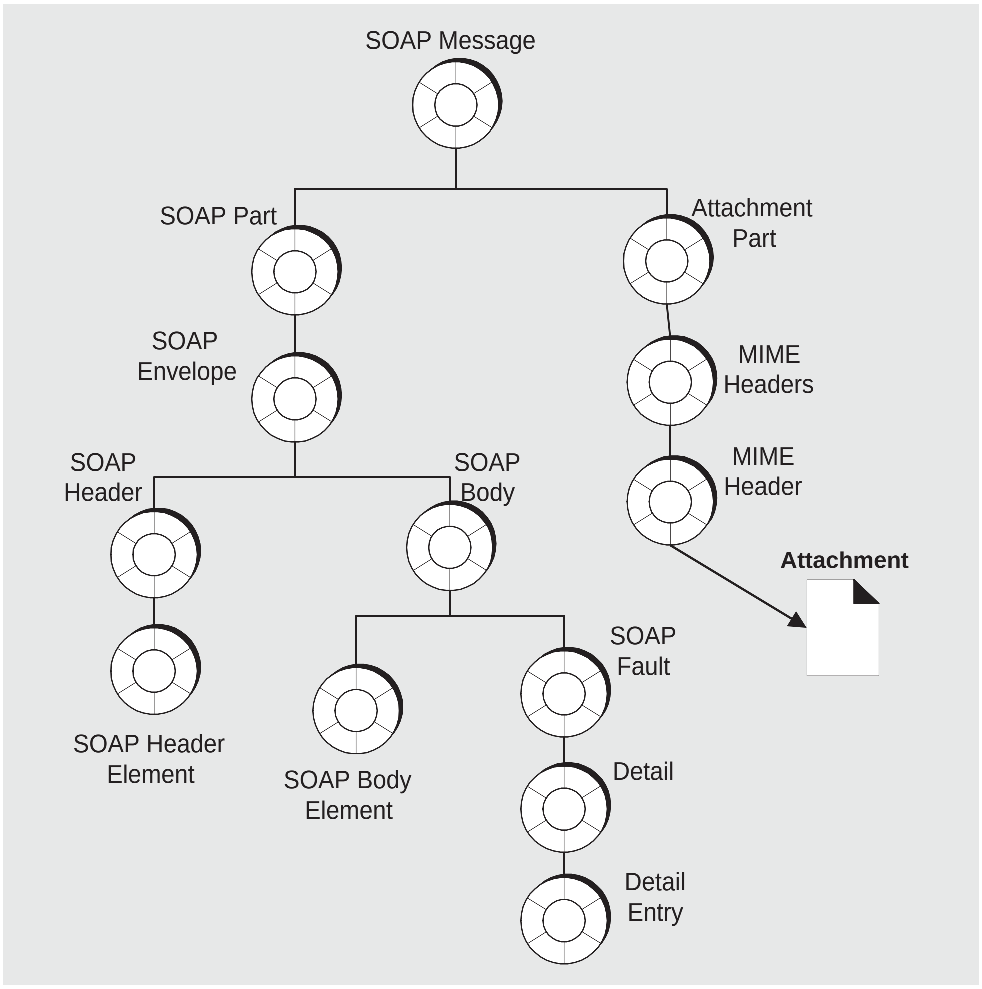
- to assemble and disassemble SOAP messages
- to send and receive these messages

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 6-5. The classes or interfaces from which these objects are derived are all defined in the `javax.xml.soap` package.

Figure 6-5 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 JAXM 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 130) 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 6-1.

Table 6-1 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 6-1 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 6-1, 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 6-1 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 JAXM 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 JAXM Administered Objects" on page 136.

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 JAXM Administered Objects" on page 136.

Message Factory

You use a Message Factory to create a SOAP message.

- If you are using a provider, you should create a message factory by using the `createMessageFactory` method of your provider connection. For example, if `con` is a provider connection, the following code creates a message factory, `mf`:

  ```
  MessageFactory mf = con.createMessageFactory(xProfile);
  ```

  The *profile* parameter you pass to the `createMessageFactory` method determines what addressing and other information is placed in the message header for messages created by the message factory.

- If you are not using a provider, you can instantiate a message factory directly; for example:

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

Connection

To send a SOAP message using JAXM, you must obtain either a `SOAPConnection` or a `ProviderConnection`. You can also transport a SOAP message using Message Queue; for more information, see "Integrating SOAP and Message Queue" on page 152.

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.

Provider Connection

A `ProviderConnection`, which you get from a `ProviderConnectionFactory`, creates a connection to a particular messaging provider. When you send a SOAP message using a provider, the message is forwarded to the provider, and then the provider is responsible for delivery to its final destination. The provider guarantees reliable, secure messaging. (Message Queue does not currently offer SOAP provider support.)

Using JAXM 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 6-2 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 6-2 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.

Table 6-2 SOAP Administered Object Information *(Continued)*

Option	Description
-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 6-2 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 6-2 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 JAXM application. In Code Example 6-3, you first create an initial context for the JNDI lookup and then you look up the desired object.

Code Example 6-3 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 JAXM to send and receive a SOAP message. It is also possible to construct a SOAP message using JAXM and to send it as the payload of a JMS message. For information, see "Integrating SOAP and Message Queue" on page 152.

JAXM supplies two models that you can use to do SOAP messaging: one uses the `SOAPConnection` object and the other uses the `ProviderConnection` object. Message Queue does not currently support the `ProviderConnection` object.

SOAP Messaging Programming Models

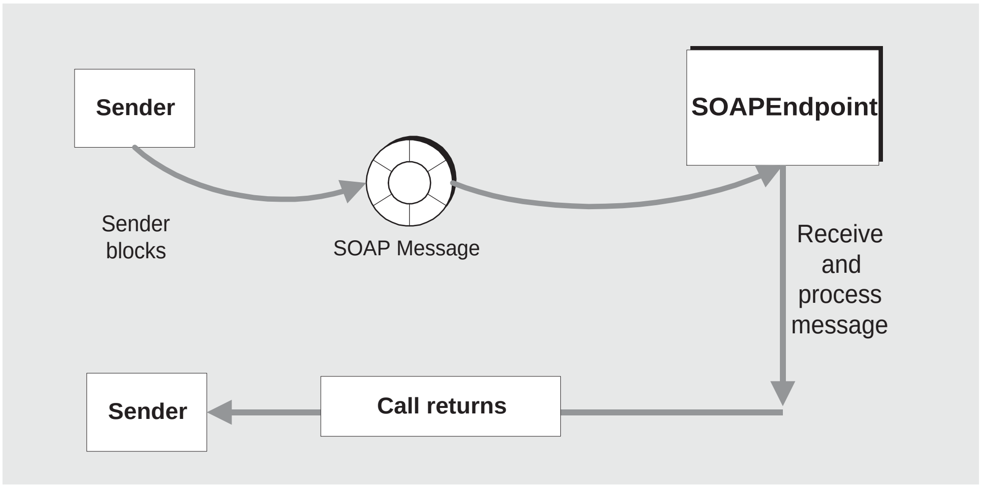
This section provides a brief summary of the programming models used in SOAP messaging using JAXM.

A SOAP message is sent to an endpoint by way of a connection. There are two types of connections: point-to-point connections (implemented by the `SOAPConnection` class) and provider connections (implemented by the `ProviderConnection` 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 6-6.

Figure 6-6 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 134 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.

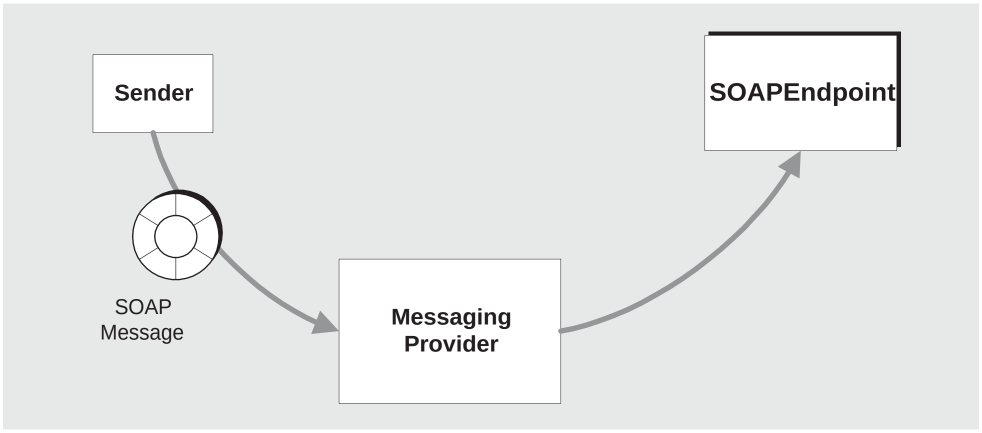
The JAXM service listening for a request-reply message uses a `ReqRespListener` object to receive messages.

For a detailed example of a client that does point-to-point messaging, see “Writing a SOAP Client” on page 142.

Provider Connections

You use a provider connection to implement one-way messaging. Figure 6-7 illustrates the one-way messaging model.

Figure 6-7 One-Way Messaging



As opposed to the point-to-point model, the final destination for the message is written into the message header by the provider. (When the administrator configures the messaging provider, she can supply a list of Endpoint objects. When a client uses the provider to send messages, the provider sends the messages only to those parties represented by Endpoint objects in its messaging provider's list.)

A message sent by means of a provider connection is always routed through an intermediate destination in the provider before it is forwarded to its final destination. The provider is also responsible for the reliability of the transmission and the privacy of the message.

Using this model, the client does the following:

- Creates a provider connection factory and gets a connection.
- Creates a message factory and creates a new message.
- Creates a name for the content and adds content to the message.
- Sends the message. (The send method is asynchronous and returns immediately.)

The JAXM service listening for a one way message uses a `OnewayListener` object to receive messages asynchronously.

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 148.

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:

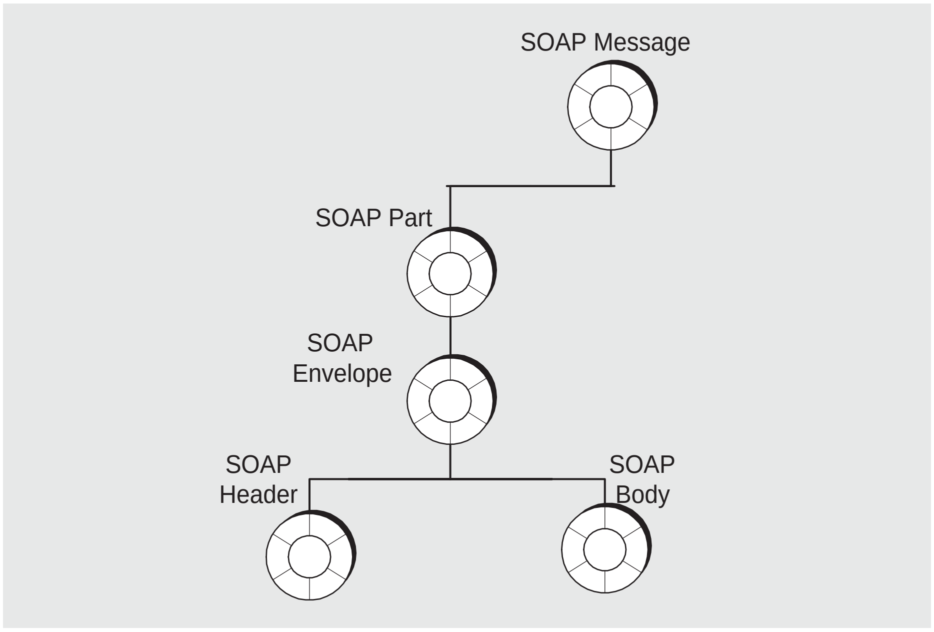
   ```
   MessageFactory myMsgFct = MessageFactory.newInstance();
   ```

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 6-8.

Figure 6-8 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 6-4 shows the basic structure of a SOAP service that uses the JAXM servlet utility class.

Code Example 6-4 Skeleton Message Consumer

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

Code Example 6-5 shows a simple ping message service:

Code Example 6-5 A Simple Ping Message Service

```
public class SOAPEchoServlet extends JAXMServlet
                          implements ReqRespListener{

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

Table 6-3 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 6-3 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.
`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

Table 6-3 JAXMServlet Methods *(Continued)*

Method	Description
`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 148.

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 6-6 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 6-7 on page 157. For sample code that shows you how to receive and process an attachment, see Code Example 6-8 on page 160.

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.

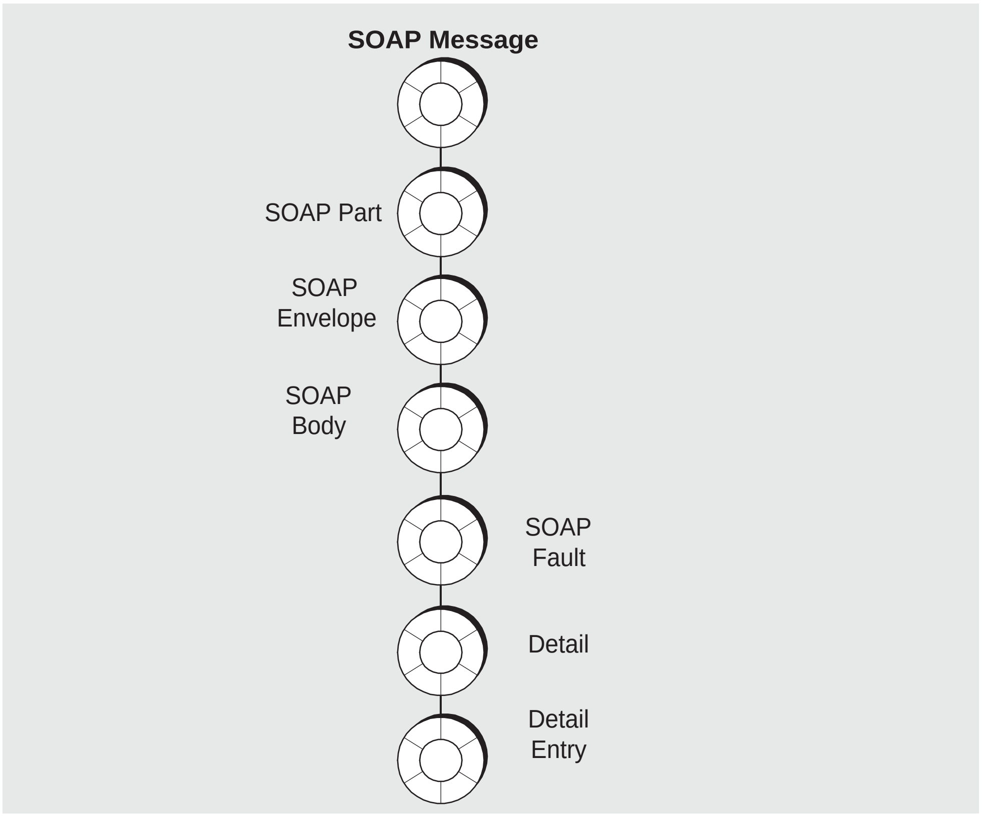
Handling SOAP Faults

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 6-9. 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 6-9 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 6-4 on page 150. 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 6-4 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).
`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 JAXM 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 the JAXM API 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.

  Transforming a SOAP message into a JMS message involves making a call like the following:

  ```
  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 JAXM 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`.

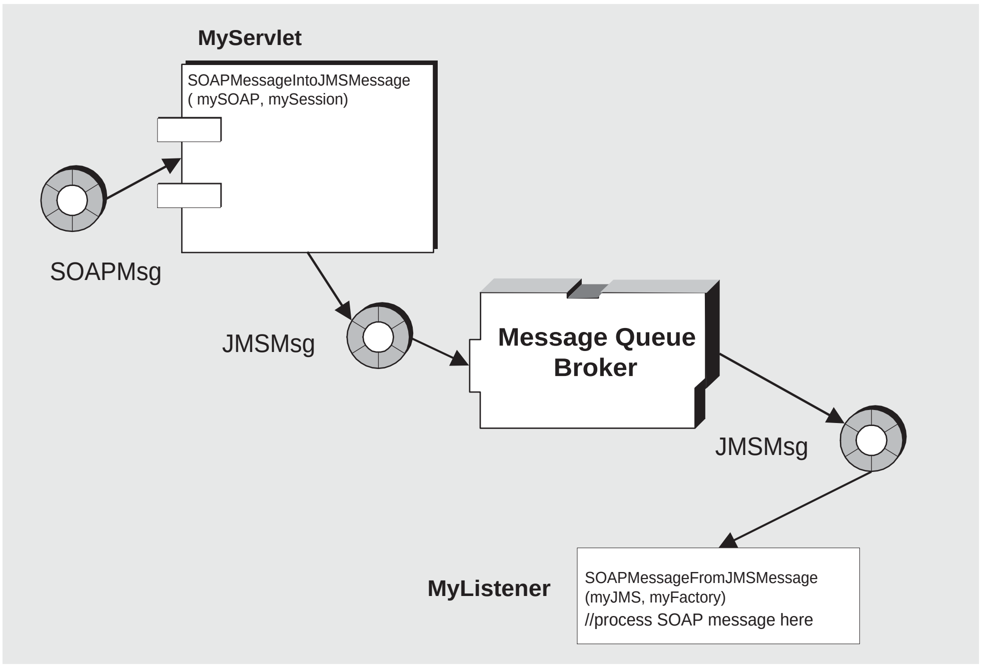
The following sections offer several use cases and code examples to illustrate this process.

Example 1: Deferring SOAP Processing

In the first example, illustrated in Figure 6-10, 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 145.

Figure 6-10 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);
   ```

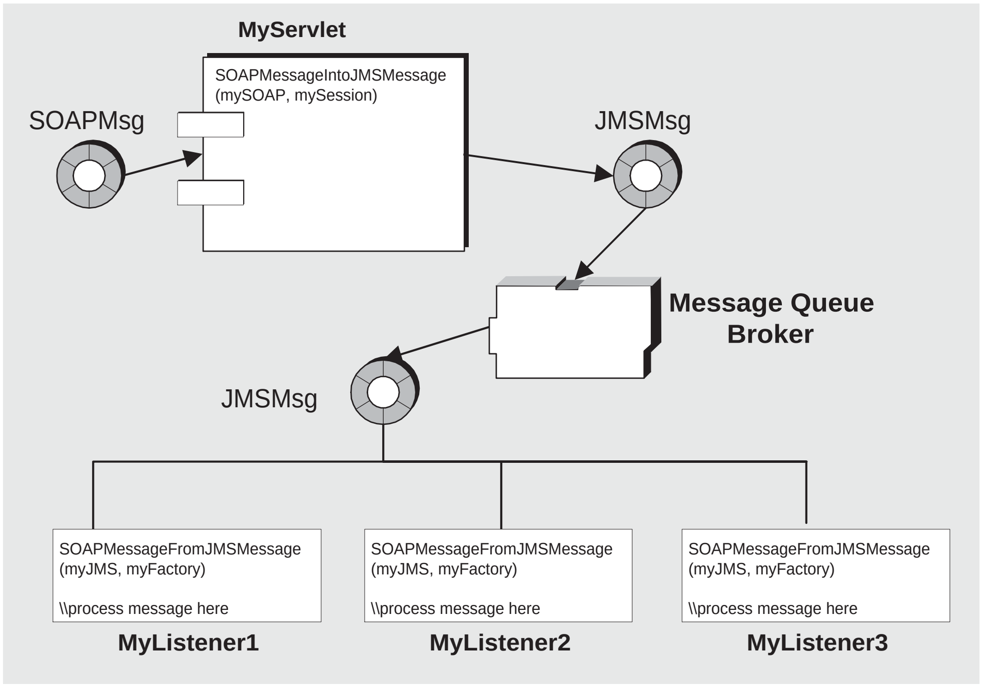
   If you specify null for the `MessageFactory` argument, the default Message Factory is used to construct the SOAP Message.

10. Disassemble the SOAP message in preparation for further processing. See "The SOAP Message Object" on page 127 for information.

Example 2: Publishing SOAP Messages

In the next example, illustrated in Figure 6-11, 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 6-11 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 6-7 on page 157.

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 6-7 illustrates the use of the JMS API, the JAXM 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 6-7 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
```

Code Example 6-7 Sending a JMS Message with a SOAP Payload *(Continued)*

```
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 {
    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
```

Code Example 6-7 Sending a JMS Message with a SOAP Payload *(Continued)*

```
    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);
    ssm.send();
    ssm.close();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    }
}
```

Code Example 6-8 illustrates the use of the JMS API, the JAXM API, 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 the JAXM API to process the SOAP body and the JAXM and DOM API to process the message attachments.
- A `main` method that initializes the `ReceiveSOAPMessageWithJMS` class.

Code Example 6-8 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

//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();
```

Code Example 6-8 Receiving a JMS Message with a SOAP Payload *(Continued)*

```
    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
        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";
    }
```

Code Example 6-8 Receiving a JMS Message with a SOAP Payload *(Continued)*

```
    ReceiveSOAPMessageWithJMS rsm = new ReceiveSOAPMessageWithJMS(topicName);
    }catch (Exception e) {
    e.printStackTrace();
    }
    }
}
```

Appendix A

Administered Object Attributes

This appendix provides reference tables for the attributes of the `ConnectionFactory`, `XAConnectionFactory`, destination, and endpoint administered objects.

ConnectionFactory Administered Object

Table A-1 summarizes the configurable properties of both `ConnectionFactory` and `XAConnectionFactory` administered objects. The attributes are presented in alphabetical order for quick reference. For groupings of these attributes in functional categories, and a description of each, see "Client Runtime Configurable Properties" on page 69.

Table A-1 Connection Factory Attributes

Attribute/Property Name	Type	Default Value	Reference
`imqAckOnAcknowledge`	String	`null: no default value`	Table 4-7 on page 80
`imqAckOnProduce`	String	`null: no default value`	Table 4-7 on page 80
`imqAckTimeout`	String	`0 millisecs`	Table 4-7 on page 80
`imqAddressList`	String	`null: no default value`	Table 4-3 on page 75
`imqAddressListIterations`	Integer	`1`	Table 4-3 on page 75
`imqAddressListBehavior`	String	`PRIORITY`	Table 4-3 on page 75
`imqBrokerHostName` (Message Queue 3.0)	String	`localhost`	Table 4-4 on page 76
`imqBrokerHostPort` (Message Queue 3.0)	Integer	`7676`	Table 4-4 on page 76

Table A-1 Connection Factory Attributes *(Continued)*

Attribute/Property Name	Type	Default Value	Reference
imqBrokerServicePort (Message Queue 3.0)	Integer	0	Table 4-4 on page 76
imqConfiguredClientID	String	null: no default value	Table 4-5 on page 78
imqConnectionFlowCount	Integer	100	Table 4-7 on page 80
imqConnectionFlowLimit	Integer	1000	Table 4-7 on page 80
imqConnectionFlowLimitEnabled	Boolean	false	Table 4-7 on page 80
imqConnectionType (Message Queue 3.0)	String	TCP	Table 4-4 on page 76
imqConnectionURL (Message Queue 3.0)	String	http://localhost/imq/tunnel	Table 4-4 on page 76
imqConsumerFlowLimit	Integer	100	Table 4-7 on page 80
imqConsumerFlowThreshold	Integer	50	Table 4-7 on page 80
imqDefaultPassword	String	guest	Table 4-5 on page 78
imqDefaultUsername	String	guest	Table 4-5 on page 78
imqDisableSetClientID	Boolean	false	Table 4-5 on page 78
imqJMSDeliveryMode	Integer	2 (persistent)	Table 4-6 on page 79
imqJMSExpiration	Long	0 (does not expire)	Table 4-6 on page 79
imqJMSPriority	Integer	4 (normal)	Table 4-6 on page 79
imqLoadMaxToServerSession	Boolean	true	Table 4-9 on page 83
imqOverrideJMSDeliveryMode	Boolean	false	Table 4-6 on page 79
imqOverrideJMSExpiration	Boolean	false	Table 4-6 on page 79
imqOverrideJMSHeadersTo TemporaryDestinations	Boolean	false	Table 4-6 on page 79
imqOverrideJMSPriority	Boolean	false	Table 4-6 on page 79
imqQueueBrowserMaxMessages PerRetrieve	Integer	1000	Table 4-8 on page 82
imqQueueBrowserRetrieveTimeout	Long	60,000 millisecs	Table 4-8 on page 82
imqReconnectAttempts	Integer	0	Table 4-3 on page 75
imqReconnectDelay	Integer	30,000 millisecs	Table 4-4 on page 76
imqReconnectEnabled	Boolean	false	Table 4-4 on page 76
imqReconnectInterval	Long	3000	Table 4-3 on page 75

Table A-1 Connection Factory Attributes *(Continued)*

Attribute/Property Name	Type	Default Value	Reference
`imqSetJMSXAppID`	Boolean	`false`	Table 4-9 on page 83
`imqSetJMSXConsumerTXID`	Boolean	`false`	Table 4-9 on page 83
`imqSetJMSXProducerTXID`	Boolean	`false`	Table 4-9 on page 83
`imqSetJMSXRcvTimestamp`	Boolean	`false`	Table 4-9 on page 83
`imqSetJMSXUserID`	Boolean	`false`	Table 4-9 on page 83
`imqSSLIsHostTrusted` (Message Queue 3.0)	Boolean	`true`	Table 4-4 on page 76

For more information on using `ConnectionFactory` administered objects see Chapter 3, "Using Administered Objects."

Destination Administered Objects

A destination administered object represents a physical destination (a queue or a topic) in a broker to which the publicly-named destination object corresponds. Its only attribute is the physical destination's internal, provider-specific name. By creating a destination object, you allow a client's `MessageConsumer` and/or `MessageProducer` objects to access the corresponding physical destination.

Table A-2 Destination Attributes

Attribute/Property Name	Type	Default
`imqDestinationDescription`	String	`A Description for the Destination Object`
`imqDestinationName`	String[1]	`Untitled_Destination_Object`

1. Destination names can contain only alphanumeric characters (no spaces) and must begin with an alphabetic character or the characters "_" and/or "$".

For more information on `Destination` administered objects see Chapter 3, "Using Administered Objects."

Endpoint Administered Objects

An endpoint administered object represents an endpoint object. By creating an administered object for an endpoint, you allow the endpoint to be accessed through a look-up operation while isolating specific endpoint information from application code or particular provider requirements. You can set one or more attributes for an endpoint administered object. These are described in Table A-3.

For additional information about endpoint administered objects, see "Using JAXM Administered Objects" on page 136.

Table A-3 Endpoint Attributes

Attribute Name	Type	Description
`imqSOAPEndpointList`	String	A list containing one or more URLs delimited by spaces. This list contains the URLs of all endpoints to which you want to broadcast a SOAP message. Each URL should be associated with a servlet that can receive and process a SOAP message.
`imqEndpointName`	String	The name of the endpoint object. Default: `Untitled_Endpoint_Object`
`imqEndpointDescription`	String	A description of the endpoint and its use. Default: `A description for the endpoint object`.

Appendix B

Client Error Codes

This appendix provides reference information for error codes returned by the Message Queue client runtime when it raises a JMS exception.

When client runtime code raises an exception, it returns a specific client error code and message. You can obtain the error code and message using the `JMSException.getErrorCode()` method and the `JMSException.getMessage()` method.

Note that error codes and error messages are not standardized in the JMS specification, but are specific to each JMS provider. Applications that rely on these error codes and messages in their programming logic are not portable across JMS providers.

Table B-1 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 B-1 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.

Table B-1 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
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.
C4011	**Message** Write message failed. **Cause** The client runtime was not able to write the stream of bytes to a `BytesMessage` type message.

Table B-1 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
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.
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.

Table B-1 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
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.
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.

Table B-1 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
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.
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.

Table B-1 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
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.
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.

Table B-1 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
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.
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.

Table B-1 Message Queue Client Error Codes *(Continued)*

Code	Message and Description

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.
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.

Table B-1 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
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.
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.

Table B-1 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
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.
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.

Table B-1 Message Queue Client Error Codes *(Continued)*

Code	Message and Description
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.

Index

A

acknowledgements
- about 40
- broker, *See* broker acknowledgements
- client, *See* client acknowledgements

administered objects
- about 29, 34, 57
- connection factory, *See* connection factory administered objects
- destination, *See* destination administered objects
- instantiation of 61
- JAXM, for 136
- JNDI lookup of 58
- provider independence, and 58
- types 34
- types of 57
- XA connection factory, *See* connection factory administered objects

administration tools 30

application servers 46

applications, *See* JMS clients

AUTO_ACKNOWLEDGE mode 68, 110

auto-reconnect
- behavior 74, 110
- configurable attributes 74, 111
- limitations 73, 110
- mechanism 73

B

broker acknowledgements
- about 66
- on produce 80, 163
- wait period for client 80, 163

broker cluster 109, 112, 118

broker failure and secure connections 115

broker metrics 92

C

checklist for client deployment 118

CLASSPATH environment variable 47

client acknowledgement modes
- AUTO_ACKNOWLEDGE 68, 110
- CLIENT_ACKNOWLEDGE 68, 89
- DUPS_OK_ACKNOWLEDGE 68

client acknowledgements
- about 68
- custom message acknowledgement 68, 89
- effect on performance 85
- modes, *See* client acknowledgement modes

client applications, *See* JMS clients

client identifier (ClientID)
- about 77
- setting in connection factory 77
- setting programmatically 77

client runtime
- about 29
- configurable properties 69
- message consumption, and 67
- message production, and 66

client threads 115

CLIENT_ACKNOWLEDGE mode 68, 89

clustered broker configuration 109, 112, 118

compiling JMS clients 54

components
- EJB 43
- MDB 44

connection factory administered objects
- about 33
- application server support attributes 83
- attributes 70
- client identification attributes 77
- ClientID, and 39
- connection handling attributes 71
- instantiation of 61
- JMS properties support attributes 83
- JNDI lookup 35
- JNDI lookup of 59
- overriding attribute values 64
- overriding message header fields 79
- queue browser behavior attributes 82
- reliability and flow control attributes 79

ConnectionConsumer behavior with auto-reconnect 111

connections
- about 33
- auto-reconnect, *See* auto-reconnect
- failover, *See* auto-reconnect
- message server address syntax 71
- secure 114
- server or broker failure 115
- specifying 71

consumers 34

containers
- EJB 45
- MDB 45

custom client acknowledgement 68, 89

D

delivery modes
- effects on performance 84
- non-persistent 39
- persistent 39

delivery, reliable 39

deployment checklist for client applications 118

destination administered objects
- about 33
- attributes 165
- instantiation of 63
- lookup of 60

destination metrics 92

destination-list metrics 92

directory variables
- IMQ_HOME 20
- IMQ_JAVAHOME 21
- IMQ_VARHOME 20

distributed applications and synchronous consumers 117

distributed transactions
- about 41
- XA resource manager 41
- *See also* XA connection factories

domains 36

DUPS_OK_ACKNOWLEDGE mode 68

durable subscribers, *See* durable subscriptions

durable subscriptions
- about 37
- ClientID, and 39

E

editions, product
- about 27
- enterprise 28
- platform 27

enterprise edition 28

environment variables, *See* directory variables

F

flow control, *See* message flow control

H

hashtable for destination-list metrics 96, 105
heap space, JVM 114
HTTPS connections 114

I

IMQ_HOME directory variable 20
IMQ_JAVAHOME directory variable 21
IMQ_VARHOME directory variable 20
imqAckOnAcknowledge attribute 80, 163
imqAckOnProduce attribute 80, 163
imqAckTimeout attribute 80, 163
imqAddressList attribute 75, 109, 111, 163
imqAddressListBehavior attribute 75, 113, 163
imqAddressListIterations attribute 75, 109, 163
imqBrokerHostName attribute (Message Queue 3.0) 76, 163
imqBrokerHostPort attribute (Message Queue 3.0) 76, 163
imqBrokerServicetPort attribute (Message Queue 3.0) 76, 164
imqConfiguredClientID attribute 78, 164
imqConnectionFlowCount attribute 81, 164
imqConnectionFlowLimit attribute 81, 164
imqConnectionFlowLimitEnabled attribute 81, 164
imqConnectionType attribute (Message Queue 3.0) 76, 164
imqConnectionURL attribute (Message Queue 3.0) 77, 164
imqConsumerFlowLimit attribute 82, 164
imqConsumerFlowThreshold attribute 82, 164
imqDefaultPassword attribute 78, 164
imqDefaultUsername attribute 78, 164
imqDestinationDescription attribute 165
imqDestinationName attribute 165
imqDisableSetClientID attribute 78, 164
imqEndpointDescription attribute 166
imqEndpointName attribute 166
imqFlowControlLimit attribute 82, 164
imqJMSDeliveryMode attribute 79, 164
imqJMSExpiration attribute 79, 164
imqJMSPriority attribute 79, 164
imqLoadMaxToServerSession attribute 83, 164
imqOverrideJMSDeliveryMode attribute 79, 164
imqOverrideJMSExpiration attribute 79, 164
imqOverrideJMSHeadersToTemporaryDestinations attribute 79, 164
imqOverrideJMSPriority attribute 79, 164
imqQueueBrowserMax MessagesPerRetrieve attribute 82, 164
imqQueueBrowserRetrieveTimeout attribute 83, 164
imqReconnectAttempts attribute 75, 109, 164
imqReconnectDelay attribute 164
imqReconnectEnabled attribute 75, 109, 164
imqReconnectInterval attribute 75, 109, 112, 164
imqReconnectListBehavior attribute 109
imqSetJMSXAppID attribute 83, 165
imqSetJMSXConsumerTXID attribute 83, 165
imqSetJMSXProducerTXID attribute 83, 165
imqSetJMSXRcvTimestamp attribute 84, 165
imqSetJMSXUserID attribute 83, 165
imqSOAPEndpointList attribute 166
imqSSLIsHostTrusted attribute (Message Queue 3.0) 76, 165

J

J2EE applications
  EJB specification 43
  JMS, and 43
  message-driven beans, *See* message-driven beans
JAF API 141

JAXM API
  about 127
  client code 142
  exception handling 142
  fault handling 142, 148
  javax.xml.messaging package 127
  javax.xml.soap package 127
  JAXM servlet 145
  programming model for SOAP 120, 127, 138
  service code 145
JAXM servlet 145
JAXM specification 17, 23
JMS API 30
JMS clients
  about 30
  client runtime, and 29
  compiling 54
  deployment checklist 118
  development steps 51
  performance, *See* performance
  programming model 30
  provider independence 38
  requirements for deployment 118
  running 54
  setup summary 35
  system properties, and 64
JMS specification 17, 23, 25
JMSCorrelationID message header field 32
JMSDeliveryMode message header field 31, 78
JMSDestination message header field 31
JMSExpiration message header field 31, 78
JMSMessageID message header field 31
JMSPriority message header field 31, 78
JMSRedelivered message header field 32
JMSReplyTo message header field 32
JMSTimestamp message header field 32
JMSType message header field 32
JNDI
  administered objects, and 35, 38, 100
  connection factory lookup 59
  destination lookup 60
  Message Queue support of 26
  message-driven beans, and 45
JVM heap space 114
JVM metrics 92

L

LDAP object store 59
licenses for Message Queue editions 27
listeners 44
listeners, message
  about 34
  asynchronous consumption, and 68

M

managing
  client threads 115
  memory 114
master broker 113
MDB *See* message-driven beans
message consumers 34
message consumption
  about 67
  asynchronous 42
  synchronous 42
message delivery models 36
message flow control
  attributes 79
  limits 86
  metering 85
message headers
  fields 31
  overrides 78
message listeners, *See* listeners
message producers 34
Message Queue server 28
message server address 71
message-based monitoring 92
message-driven beans
  about 44
  application server support 46
  deployment descriptor 45
  MDB container 45
MessageFactory object 146

messages
  about 30
  acknowledgement, *See* client acknowledgements or broker acknowledgements
  body 32
  consumption of, *See* message consumption
  delivery models 36
  delivery modes, *See* delivery modes
  delivery of 65
  duplicate sends 69
  flow control, *See* message flow control
  headers, *See* message headers
  listeners for, *See* listeners, message
  ordering of 43
  persistent 39
  persistent storage 41
  point-to-point delivery 36
  prioritizing 43
  production of 66
  properties of 32, 94
  publish/subscribe delivery 37
  reliable delivery of 40, 79
  selection and filtering of 43
  sequencing 114
  size 114
  SOAP 26
  SOAP payloads, with 152
messaging system, architecture 28
metrics messages
  type 92
metrics, *See* message-based monitoring
MimeHeaders object 146
mq.metrics.broker topic 92
mq.metrics.destination.queue.*monitoredDestName* topic 92
mq.metrics.destination.topic.*monitoredDestName* topic 92
mq.metrics.destination_list topic 92
mq.metrics.jvm topic 92

N

namespaces, in SOAP 130

O

object stores
  administered objects, and 57
  file system 59
  LDAP server 59
OnewayListener object 140
onMessage() method 146
OutOfMemory error 114

P

passwords, default 78, 164
performance
  effect of delivery mode 84
  effect of sessions and connections on 87
  factors affecting 84
  message flow control 85
  message service resources 79
persistence
  about 41
  delivery modes, *See* delivery modes
  persistent messages 39
platform edition 27
point-to-point delivery 36
producers 34
programming domains 36
properties, client runtime *See* client runtime
provider independence
  about 38
  administered objects, and 58
publish/subscribe delivery 37

Q

queue destinations 36

R

reconnect, automatic *See* auto-reconnect
reliability, factors affecting 84
reliable delivery 39, 79
ReqRespListener object 139
resource adapter 46
running JMS clients 54

S

selection, of messages 43
sequencing partial messages 114
server failure and secure connections 115
ServletConfig object 146
sessions
  about 33
  acknowledgement options for 40
  transacted 40
Simple Object Access Protocol *See* SOAP
SOAP 26
SOAP message
  attachments to 141
  disassembling 147
  envelope 124
  header 124
  MIME envelope for 126
  models of 125
  Name object 133
  payload to JMS message, as 152
  SOAPMessage object 127
  structure of 124
SOAP messaging
  attachments, using 141
  client code 142
  connections 135
  endpoints 134
  exception handling 142
  fault codes 149
  fault handling 142, 148
  layers of 120
  message factories 135
  namespaces 130
  point-to-point connections 138
  programming models 138
  protocol for 120
  provider connections 140
  service code 145
  SOAPMessageFromJMSMessage method 152
  SOAPMessageIntoJMSMessage utility 152
synchronous consumers 117
system properties, setting 64

T

threads, *See* client threads
topic destinations 37
transactions
  about 40
  and custom client acknowledgement 90
  distributed, *See* distributed transactions
tunnel servlet connection 115

U

URLEndpoint object 144
user names 78, 164

W

web services 120

X

XA connection factories
  about 41
  See also connection factory administered objects
XA resource manager, *See* distributed transactions