Developer's Guide

Sun™ ONE Message Queue

Version 3.0

June, 2002

Contents

List of Figures

List of Tables

List of Procedures

List of Code Examples

Preface

This book provides information about the concepts and procedures needed by a developer of messaging applications in a Sun™ ONE Message Queue (MQ) environment.

This preface contains the following sections:

Audience for This Guide

This guide is meant principally for developers of applications that exchange messages using an MQ 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. The JMS and JAXM specifications are open standards.

This *Developer's Guide* assumes that you are familiar with the JMS API's 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 an MQ messaging system.

This *Developer's Guide* assumes no familiarity, however, with the JAXM APIs or with JAXM programming guidelines. This material is described in Chapter 5, "Working With SOAP Messages", which 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 Sun ONE Message Queue and of JMS concepts and programming issues.
Chapter 2, "Quick Start Tutorial"	A tutorial that acquaints you with the MQ development environment using a simple example client application.
Chapter 3, "Using Administered Objects"	Describes how to use MQ administered objects in both a provider- independent and provider-specific way.
Chapter 4, "Optimizing Clients"	Explains features of the MQ client runtime and how they can be used to optimize a client application.
Chapter 5, "Working With SOAP Messages"	Explains how you send and receive SOAP messages with and without MQ support.
Appendix A, "Administered Object Attributes"	Summarizes and documents administered object attributes.

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 URL's.
`[]`	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 acronyms (MQ, 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.

Environment Variable Conventions

MQ makes use of three environment variables—but how they are used varies from platform to platform. Table 3 describes these environment variables and summarizes how they are used on the Solaris, Windows, and Linux platforms.

Table 3 MQ Environment Variables

Environment Variable	Description
IMQ_HOME	This is generally the root MQ installation directory in which all installed files are placed: • On Solaris, there is no root MQ installation directory. `IMQ_HOME` is not used by MQ software and is not used in MQ documentation to refer to file locations on Solaris. • On Solaris, for Sun ONE Application Server, Evaluation Edition, `IMQ_HOME` is not used by MQ software, but is used in MQ documentation to refer to the root MQ installation directory (an `imq` subdirectory under the Application Server installation root directory). • On Windows, `IMQ_HOME` is used by MQ software and is also used in MQ documentation to refer to the root MQ installation directory. The value of `IMQ_HOME` is set by the MQ installer (by default, as `C:\Program Files\Sun Microsystems\Message Queue 3.0`). • On Linux, `IMQ_HOME` is not used by MQ software, but is used in MQ documentation to refer to the root MQ installation directory (by default, an `imq` subdirectory under `/opt`).
IMQ_VARHOME	This refers to the `/var` directory in which MQ temporary or dynamically-created configuration and data files are stored: • On Solaris, `IMQ_VARHOME` defaults to the `/var/imq` directory, but a user can optionally set the value to any directory. • On Solaris, for Sun ONE Application Server, Evaluation Edition, `IMQ_VARHOME` defaults to `IMQ_HOME/var`, but a user can optionally set the value to any directory. • On Windows `IMQ_VARHOME` defaults to `IMQ_HOME/var`, but a user can optionally set the value to any directory. • On Linux, `IMQ_VARHOME` defaults to `IMQ_HOME/var`, but a user can optionally set the value to any directory.

Table 3 MQ Environment Variables *(Continued)*

Environment Variable	Description
IMQ_JAVAHOME	This refers to the location of the Java runtime (JRE 1.4) required by MQ executables: • On Solaris, IMQ_JAVAHOME defaults to the /usr/j2se/jre directory, but a user can optionally set the value to wherever JRE 1.4 resides. • On Windows, IMQ_JAVAHOME defaults to IMQ_HOME/jre, but a user can optionally set the value to wherever JRE 1.4 resides. • On Linux, IMQ_JAVAHOME defaults to the /usr/java/j2sdk1.0/jre directory, but a user can optionally set the value to wherever JRE 1.4 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). All path names use UNIX file separator notation (/).

Other Documentation Resources

In addition to this guide, MQ provides additional documentation resources.

The MQ Documentation Set

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

Table 4 MQ Documentation Set

Document	Audience	Description
MQ Installation Guide	Developers and administrators	Explains how to install MQ software on Solaris, Linux, and Windows platforms.
Release Notes	Developers and administrators	Includes descriptions of new features, limitations, and known bugs, as well as technical notes.
MQ Developer's Guide	Developers	Provides a quick-start tutorial and programming information relevant to the MQ implementation of JMS.
MQ Administrator's Guide	Administrators, also recommended for developers	Provides background and information needed to perform administration tasks using MQ administration tools.

JavaDoc

JMS and MQ API documentation in JavaDoc format, is provided at the following location:

`IMQ_HOME/javadoc/index.html`
(`/usr/share/javadoc/imq/index.html` on Solaris)

This documentation can be viewed in any HTML browser such as Netscape or Internet Explorer. It includes standard JMS API documentation as well as MQ-specific API's for MQ administered objects (see Chapter 3, "Using Administered Objects"), which are of value to developers of messaging applications.

Example Client Applications

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

`IMQ_HOME/demo` (`/usr/demo/imq` on Solaris)

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.htm`

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 Programming* 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 ONE Message Queue (MQ) and to JMS concepts and programming issues of interest to developers.

What Is Sun ONE Message Queue?

The MQ product is a standards-based solution to the problem of inter-application communication and reliable message delivery. MQ is an enterprise messaging system that implements the Java Message Service (JMS) open standard: it is a JMS provider.

With Sun ONE Message Queue software, processes running on different platforms and operating systems can connect to a common MQ message 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 communicate across a network.

MQ has features which 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 an MQ message service and managing application-specific aspects of messaging, such as destinations and security.

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

Tunable performance Lets you increase performance of the MQ message service when less reliability of delivery is acceptable.

Multiple transports Supports the ability of JMS clients to communicate with each other over a number of different transports, including TCP and HTTP, and using secure (SSL) connections.

JNDI support Supports both file-based and LDAP directory services 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 Chapter 5, "Working With SOAP Messages" on page 83 for more information.

See the MQ 3.0 *Release Notes* for documentation of JMS compliance-related issues.

Product Editions

The Sun ONE Message Queue product is available in two editions: Platform and Enterprise—each corresponding to a different licensed capacity, as described below. (To upgrade MQ from one edition to another, see the instructions in the MQ *Installation Guide*.)

Platform Edition This edition can be downloaded free from the Sun website and is also bundled with the latest Sun ONE Application Server platform. The Platform Edition 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, and multiple queue delivery policies. 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, and multiple queue delivery policies) not included in the basic license. However, the license has a limited 90-day duration enforced by the software, and is therefore best suited for evaluating the enterprise features which are available in the Enterprise Edition of the product (see "Enterprise Edition" on page 25).

The Platform Edition places no limits on the number of JMS client connections supported by each MQ message service. (For information on how to switch from the basic license to the enterprise license, see the `license` command line option described in the MQ *Administrator's Guide*.)

Enterprise Edition This edition is for deploying and running messaging applications in a production environment. 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, nor on the number of client connections supported by each broker. However the license specifies the number of CPU's that it will support.

NOTE For all editions of MQ, 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 JMS client (one which can be connected to an MQ message server) that they can sell to a third party without incurring any MQ licensing fees. The third party will either need to purchase their own version of MQ to access an MQ message server or make a connection to yet another party that has an MQ message server installed and running.

MQ Messaging System Architecture

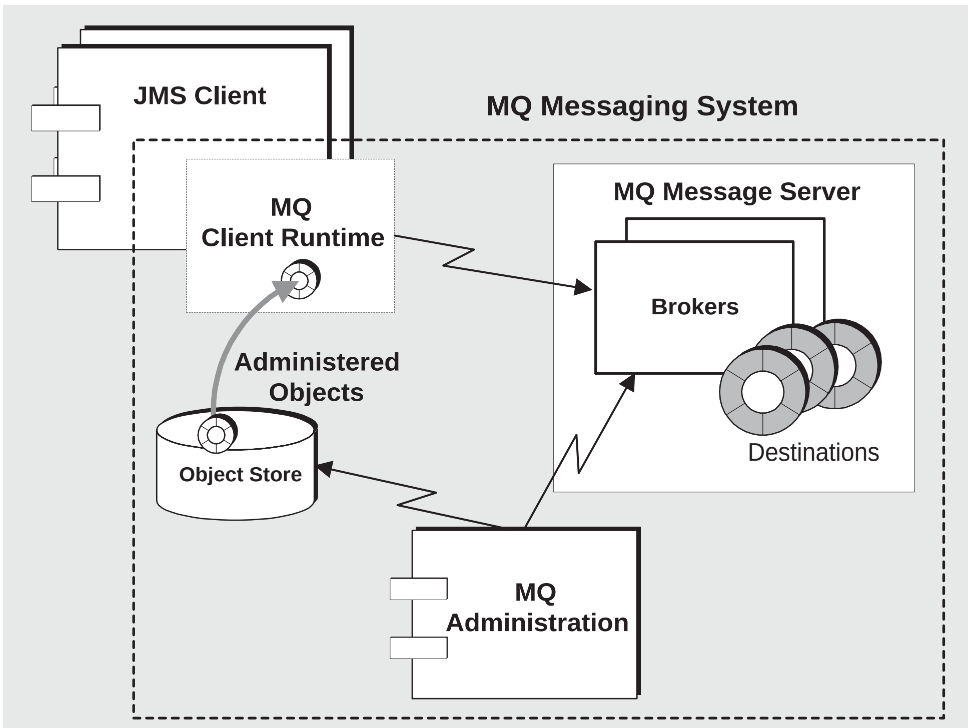
This section briefly describes the main parts of an MQ messaging system. While as a developer, you do not need to be familiar with the details of all of these parts or how they interact, a high-level understanding of the basic architecture will help you understand features of the system that impact JMS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

The main parts of an MQ messaging system, shown in Figure 1-1, are the following:

MQ message server The MQ message 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 JMS 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 MQ message server is described in detail in the MQ *Administrator's Guide*.

MQ client runtime The MQ client runtime provides JMS clients with an interface to the MQ message server—it supplies clients with all the JMS programming objects introduced in "The JMS Programming Model" on page 27. It supports all operations needed for clients to send messages to destinations and to receive messages from such destinations. The MQ client runtime is described in detail in Chapter 4, "Optimizing Clients."

Figure 1-1 MQ System Architecture



MQ administered objects Administered Objects encapsulate provider-specific implementation and configuration information in objects that are used by JMS 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 MQ client runtime is performed through administered object attributes, as described in Chapter 4, "Optimizing Clients."

MQ administration MQ provides a number of administration tools for managing an MQ messaging system. 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 MQ administrators and are described in the MQ *Administrator's 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

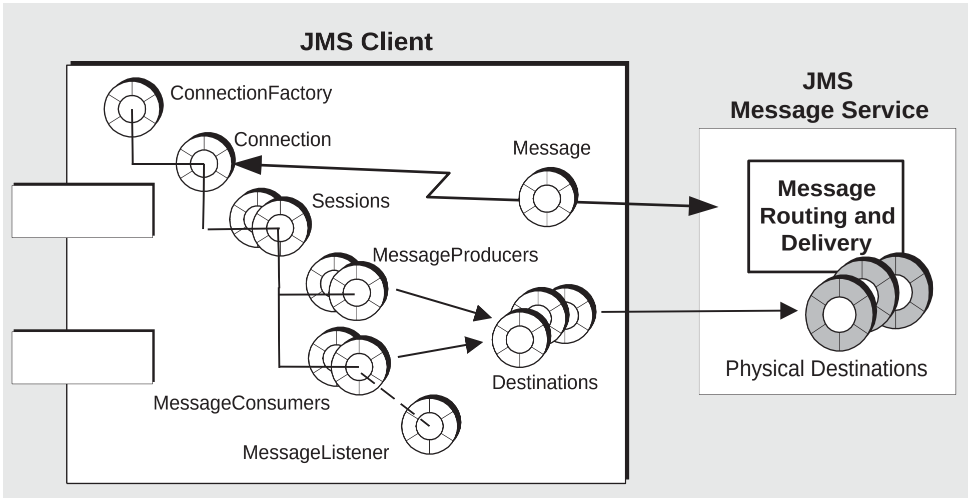
In the JMS programming model, JMS clients (components or applications) interact using a JMS application programming interface (API) to send and receive messages. This section introduces the objects that implement the JMS API and that are used to set up a JMS client for delivery of messages (see "JMS Client Setup Operations" on page 32). The main interface objects are shown in Figure 1-2 and described in the following paragraphs.

Message

In the MQ 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 MQ 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 API's. 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 a JMS client.

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.
ObjectMessage	a message whose body contains a serialized Java object.

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

Type	Description
BytesMessage	a message whose body contains a stream of uninterpreted bytes.

Destination

A `Destination` is a JMS administered object (see "Administered Objects" on page 31) 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 JMS clients be provider independent.

ConnectionFactory

A `ConnectionFactory` is a JMS administered object (see "Administered Objects" on page 31) 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 JMS 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 that 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 to the message consumer 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 39).

Message Listener

A JMS 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

Two of the objects described in the "The JMS Programming Model" on page 27 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 JMS client code dependent on a specific JMS implementation. To make JMS client code provider-independent, however, the JMS specification requires that provider-specific implementation and configuration information be encapsulated in what are called *administered objects.* These objects 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 JMS clients through standard Java Naming and Directory Service (JNDI) lookup code. Using administered objects in this way makes JMS client code provider-independent.

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

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 JMS 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 JMS 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 *MQ Administrator's 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)*	Queue	Topic
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber

* 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 JMS 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 JMS 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 MQ 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 ONE Application Server environment are limited to using the JMS 1.0.2 API. This is because the Sun ONE Application Server complies with the J2EE 1.3 specification, which supports only JMS 1.0.2. This means that any JMS messaging performed in servlets and EJBs—including message-driven beans (see “Message-driven Beans” on page 41)—must be based on the domain-specific JMS APIs.

JMS Provider Independence

JMS specifies the use of administered objects (see “Administered Objects” on page 31) to support the development of JMS 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 client code 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 MQ system objects created and configured by an MQ administrator. These objects are placed in a JNDI directory service, and a JMS client accesses them using a JNDI lookup.

MQ 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 API’s. It also undermines the ability of an MQ administrator to successfully control and manage an MQ message server.

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 33) 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 31). If hard wired into a connection factory, however, each user would then need an individual connection factory to have a unique identity.

MQ 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 73). 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 an MQ 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 client code 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

MQ 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 MQ 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.

MQ 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 27). 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 MQ-specific persistence and acknowledgement properties (see "Performance Factors" on page 80).

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 MQ 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 MQ *Administrator's Guide*), and message service availability.

JMS/J2EE Programming: Message-driven Beans

In addition to the general JMS client programming model introduced in "The JMS Programming Model" on page 27, 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.

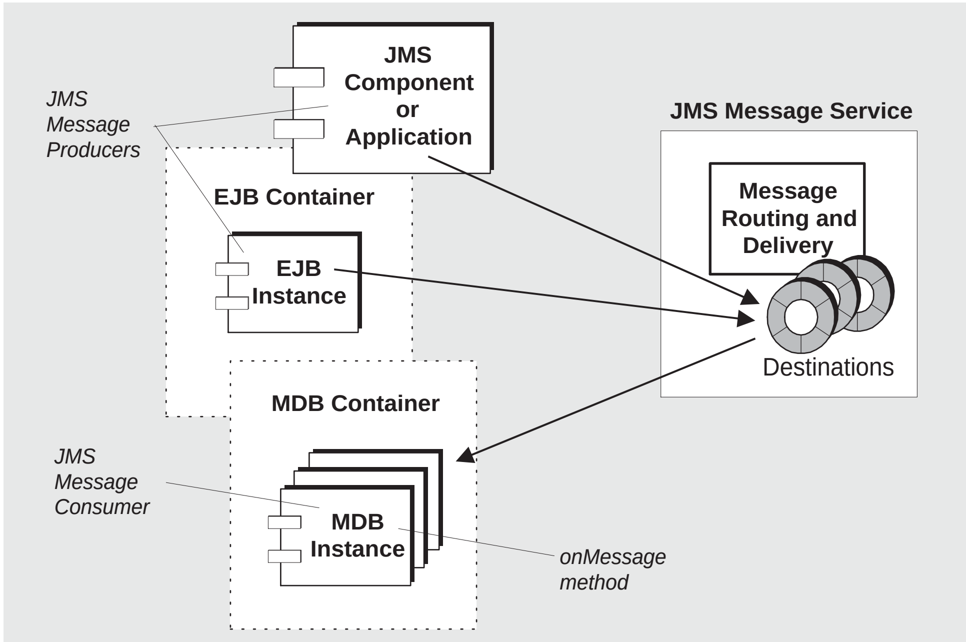
However, asynchronous messaging is a requirement of many enterprise applications. Most such applications require that server-side components be able to communicate and respond to each other without tying up server resources. Hence, the need for an EJB component that can receive messages and consume them 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 MDB's—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 on page 42.

Figure 1-3 Messaging with MDBs



MDB Container The MDB container is 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 MDB container. Each MDB container supports instances of only a single MDB.

Application Server Support

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

In the case of the Sun ONE Application Server, messaging resources are provided by Sun ONE Message Queue. This means that an MQ messaging system (see "MQ Messaging System Architecture" on page 25) is integrated into the Sun ONE Application Server, providing the support needed to send JMS messages to MDB's and other JMS messaging components running in the application server environment.

Chapter 2

Quick Start Tutorial

This chapter provides a quick introduction to JMS client programming in an MQ 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 procedures:

- setting up your environment
- starting and testing a broker
- developing a simple client application
- compiling and running a client application

For the purpose of this tutorial it is sufficient to run the MQ message server in a default configuration. For instructions on configuring an MQ message server, please refer to the MQ *Administrator's Guide*.

Setting Up Your Environment

A number of environment variables are used when compiling and running a JMS client. This section describes how to set them.

➤ **To set MQ-related environment variables**

1. If needed, set the `IMQ_HOME` environment variable (Windows platform only).

   The `IMQ_HOME` environment variable is used when compiling and running a JMS client. On the Windows platform, the MQ installer sets `IMQ_HOME`.

   The directories and environment variables that are set up at installation time are listed below.

The directories and environment variables that are set up at installation time are listed below.

Platform	Settings
Solaris	There is no single installation directory. The IMQ_HOME environment variable is not used.
Solaris: Sun ONE Application Server, EE	For Sun ONE Application Server, Evaluation Edition (EE): the default installation directory is the *Application Server installation directory*/imq The IMQ_HOME environment variable is not used by the software, but is used in MQ documentation to refer to the root installation directory
Windows	The default installation directory is C:\Program Files\Sun Microsystems\Message Queue 3.0\ The IMQ_HOME environment variable is set by the installer to the MQ installation directory
Linux	The default installation directory is /opt/imq The IMQ_HOME environment variable is not used by the software, but is used in MQ documentation to refer to the root installation directory

2. Set the JAVA_HOME environment variable.

   Set JAVA_HOME to the directory where you installed the J2SE SDK (Java2 Standard Edition Software Development Kit).

3. Change to the directory in which the example applications are located:

Platform	Settings
Solaris	/usr/demo/imq/jms
Solaris: Sun ONE Application Server, EE	IMQ_HOME/demo/jms
Windows	IMQ_HOME/demo/jms
Linux	IMQ_HOME/demo/jms

In this directory you will find the application used as an example in this chapter.

4. Set the `CLASSPATH` environment variable.

   Set `CLASSPATH` to include the current directory, as well as the jar files (`jms.jar`, `imq.jar`, `jndi.jar`) in the `IMQ_HOME/lib` directory (`/usr/share/lib/imq` on Solaris). These are required for compiling and running clients.

NOTE	JDK 1.4 includes the JNDI jar files and the JSSE jar files (needed for secure connections). So you don't have to include these jar files in the CLASSPATH, unless you are using JDK 1.3 or 1.2 for application development.

Platform	Settings and Details

Solaris (csh)	`% cd /usr/demo/imq/jms`
	`% setenv CLASSPATH .:`
	`/usr/share/lib/imq/jms.jar:`
	`/usr/share/lib/imq/imq.jar:`
	`/usr/share/lib/imq/jndi.jar`

Solaris: Sun ONE Application Server, EE	`% cd $IMQ_HOME/demo/jms`
	`% setenv CLASSPATH .:`
	`$IMQ_HOME/lib/jms.jar:`
	`$IMQ_HOME/lib/imq.jar:`
	`$IMQ_HOME/lib/jndi.jar`

Windows	`C:\>cd %IMQ_HOME%\demo\jms`
	`C:\Program Files\Sun Microsystems\`
	`Message Queue 3.0\demo> set CLASSPATH=.;`
	`%IMQ_HOME%\lib\jms.jar;`
	`%IMQ_HOME%\lib\imq.jar;`
	`%IMQ_HOME%\lib\jndi.jar`

Linux (csh)	`% cd $IMQ_HOME/demo/jms`
	`% setenv CLASSPATH .:`
	`$IMQ_HOME/lib/jms.jar:`
	`$IMQ_HOME/lib/imq.jar:`
	`$IMQ_HOME/lib/jndi.jar`

Starting and Testing the MQ Message Server

This tutorial assumes that you do not have an MQ message 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 directory to `IMQ_HOME/bin` (`/usr/bin` on Solaris).

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

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

Platform	Startup Command
Solaris	`% usr/bin/imqbrokerd -tty`
Solaris: Sun ONE Application Server, EE	`% IMQ_HOME/bin/imqbrokerd -tty`
Windows	`C:\Program Files\Sun Microsystems\` `Message Queue 3.0\bin> imqbrokerd -tty`
Linux	`% IMQ_HOME/bin/imqbrokerd -tty`

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 MQ Command (`imqcmd`) utility to display information about the broker.

1. In a separate terminal window, change directory to `IMQ_HOME/bin` (`/usr/bin` on Solaris).

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

Platform	Settings and Details
Solaris	`% /usr/bin/imqcmd query bkr -u admin -p admin`
Solaris: Sun ONE Application Server, EE	`% IMQ_HOME/bin/imqcmd query bkr -u admin -p admin`

Windows	`C:\Program Files\Sun Microsystems\`
	`Sun One Message Queue 3.0\bin>`
	`    imqcmd query bkr -u admin -p admin`

Linux	`% IMQ_HOME/bin/imqcmd query bkr -u admin -p admin`

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

```
Querying the broker specified by:
-------------------------
Host           Primary Port
-------------------------
localhost       7676


Auto Create Queues                         true
Auto Create Topics                         true
Auto Create Queue Delivery Policy          Single
Cluster Broker List (active)               myhost/111.222.333.444:7676 imqbroker
Cluster Broker List (configured)
Cluster Master Broker
Cluster URL
Current Number of Messages in System       0
Current Size of Messages in System         0
Instance Name                              imqbroker
Log Level                                  INFO
Log Rollover Interval (seconds)            604800
Log Rollover Size (bytes)                  0 (unlimited)
Max Message Size (bytes)                   70m
Max Number of Messages in System           0 (unlimited)
Max Size of Messages in System             0 (unlimited)
Primary Port                               7676
Version                                    3.0

Successfully queried the broker.
```

Developing a Simple Client Application

This section leads you through the steps used to create a simple "Hello World" application that sends a message to a queue destination and then retrieves the same message from the queue. You can find this HelloWorldMessage application at `IMQ_HOME/demo/jms` (`/usr/demo/imq/jms` on Solaris).

The following steps highlight Java programming language code that you use to set up a client to send and receive messages:

➤ To program the HelloWorldMessage example application

1. Import the interfaces and MQ 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 an MQ `QueueConnectionFactory` administered object.

   A `QueueConnectionFactory` object encapsulates all the MQ-specific configuration properties for creating `QueueConnection` connections to an MQ message 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 MQ message server.

   A `QueueConnection` object is the active connection to the MQ 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 an MQ `queue` administered object corresponding to a queue destination in the MQ 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 39). For samples of asynchronous consumption see Table 2-1 on page 54.

    ```
    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 JMS clients in an MQ 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 to point to the jms.jar and imq.jar files, as described in Step 4 on page 47, before attempting to compile or run a client application.

The following instructions are based on the HelloWorldMessage application created in "Developing a Simple Client Application" on page 50, and also located in the MQ 3.0 example applications directory:

```
IMQ_HOME/demo/jms (/usr/demo/imq/jms on Solaris)
```

➤ To compile and run the HelloWorldMessage application

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

   The MQ 3.0 example applications directory on Solaris is not writable by users, so copy the HelloWrodMessage application to a writable directory and make that directory your current directory.

2. Compile the HelloWorldMessage application as shown below.

Platform	Settings and Details
On Solaris (csh)	`% $JAVA_HOME/bin/javac HelloWorldMessage.java`
On Windows	`C:\Program Files\Sun Microsystems\Message Queue 3.0\demo\jms>%JAVA_HOME%\bin\javac HelloWorldMessage.java`
On Linux (csh)	`% $JAVA_HOME/bin/javac HelloWorldMessage.java`

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

3. Run the HelloWorldMessage application as shown below.

   The output shown is what displays when you run the HelloWorldMessage example in the examples/jms directory.

Platform	Settings and Details
On Solaris (csh)	`% $JAVA_HOME\bin\java HelloWorldMessage`
On Windows	`C:\Program Files\Sun Microsystems\Message Queue 3.0\ \demo\jms> %JAVA_HOME%\bin\java HelloWorldMessage`
On Linux (csh)	`% $JAVA_HOME\bin\java HelloWorldMessage`

The output displayed when you run the HelloWorldMessage is as follows:

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

Example Application Code

The example applications provided by MQ 3.0 consist of both JMS messaging applications as well as JAXM messaging examples (see "Working With SOAP Messages" on page 83 for more information).

JMS Examples

A listing of the code in the HelloWorldMessage tutorial example can be found, along with code from a number of other example applications, at the following location:

`IMQ_HOME/demo/jms` (`/usr/demo/imq/jms` on Solaris)

The directory includes a `README` file that describes each example application and how to run it. The examples include standard JMS sample programs as well as MQ-supplied example applications. They are summarized in the following two tables.

Table 2-1 is a listing and brief description of the JMS sample programs.

Table 2-1 JMS Sample Programs

Name of Example Application	Description
SenderToQueue	Sends a text message using a queue.
SynchQueueReceiver	Synchronously receives a text message using a queue.
SynchTopicExample	Publishes and synchronously receives a text message using a topic.
AsynchQueueReceiver	Asynchronously receives a number of text messages using a message listener.
AsynchTopicExample	Publishes five text messages to a topic and asynchronously gets them using a message listener.
MessageFormats	Writes and reads messages in five supported message formats.

Table 2-1 JMS Sample Programs *(Continued)*

Name of Example Application	Description
MessageConversion	Shows that for some message formats, you can write a message using one data type and read it using another.
ObjectMessages	Shows that objects are copied into messages, not passed by reference.
BytesMessages	Shows how to write, then read, a Bytes Message of indeterminate length.
MessageHeadersTopic	Illustrates the use of the JMS message header fields.
TopicSelectors	Shows how to use message properties as message selectors.
DurableSubscriberExample	Shows how you can create a durable subscriber that retains messages published to a topic while the subscriber is inactive.
AckEquivExample	Shows how to ensure that a message will not be acknowledged until processing is complete.
TransactedExample	Demonstrates the use of transactions in a simulated e-commerce application.
RequestReplyQueue	Demonstrates use of the JMS request/reply facility.

Table 2-2 is a listing and brief description of the MQ-supplied example applications.

Table 2-2 MQ-supplied Example Applications

Name of Example Application	Description
HelloWorldMessage	Sends and receives a “Hello World” message.
XMLMessageExample	Reads an XML document from a file, sends it to a queue, processes the message from the queue as an XML document, and converts it to a DOM object.
SimpleChat	Illustrates how MQ can be used to create a simple GUI chat application.

Table 2-2 MQ-supplied Example Applications *(Continued)*

Name of Example Application	Description
SimpleJNDIClient	Illustrates how a client would use JNDI lookups to access administered objects created by an administrator and placed in an object store (see the Administration Console tutorial in the MQ *Administrator's Guide*).

JAXM Examples

A number of examples illustrating how to send and receive SOAP messages are provided at the following location:

`IMQ_HOME/demo/jaxm` (`/usr/demo/imq/jaxm` on Solaris)

The directory includes a `README` file that describes each example application and how to run it. These example applications are summarized in Table 2-3.

Table 2-3 SOAP Messaging Example Applications

Name of Example Application	Description
SendSOAPMessage	A standalone client that sends a SOAP message.
SOAPEchoServlet	A servlet that echoes a SOAP message.
SendSOAPMessageWithJMS	A standalone client that constructs a SOAP message, wraps it as a JMS message, and then publishes this message to a topic.
ReceiveSOAPMessageWithJMS	A JMS message listener that subscribes to a topic where it receives a JMS-wrapped SOAP message, which it then converts to a SOAP message.
SOAPtoJMSServlet	A servlet that receives a SOAP message, wraps it as a JMS message and publishes it to a topic.

Chapter 3

Using Administered Objects

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

MQ 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 MQ message server, while destination objects (which represent physical destinations) are used to create JMS message consumers and producers (see "Developing a Simple Client Application" on page 50).

The JAXM endpoint administered object is used to send SOAP messages (see Chapter 5, "Working With SOAP Messages").

There are two approaches to the use of administered objects:

- They can be created and configured by an administrator, stored in a name service, accessed by clients through standard JNDI lookup code, and then used in a provider-independent manner.
- They can be instantiated and configured by a developer when writing client 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 MQ-specific configuration details. This chapter describes these two approaches and explains how to code your JMS 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 MQ 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 using MQ administration tools, as described in the MQ *Administrator's Guide*. When creating an administered object, the administrator can specify that it be read only—that is, clients cannot change MQ-specific configuration values specified when the object was created. In other words, client 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 store or an LDAP server for your MQ administered objects. The code below assumes a file-system store. For information about the corresponding LDAP server properties, see the MQ *Administrator's 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);
   ```

   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` subdirectory of the example applications directory:

   `IMQ_HOME/demo/jms` (`/usr/demo/imq/jms` on Solaris)

   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(“cn=MyQueueConnectionFactory”);
   ```

   It is recommended that you use this connection factory as originally configured. For a discussion of `ConnectionFactory` and `XAConnectionFactory` object configuration properties, see “MQ Client Runtime Configurable Properties” on page 69 and for a complete list of properties, see “ConnectionFactory Administered Object” on page 127.

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.

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("cn=MyQueueConnectionFactory");
QueueConnection myQConn =
    myQConnFactory.createQueueConnection();
```

Looking Up Destination Objects

➤ To perform a JNDI lookup of a Destination object

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

```
Queue myQ =
(Queue) ctx.lookup("cn=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 client 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 client 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 client 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 MQ `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 "MQ Client Runtime Configurable Properties" on page 69 and for a complete list of properties, see "ConnectionFactory Administered Object" on page 127.

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 MQ `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 be used, as well, to override attribute values of MQ administered objects used in client code. You can override the configuration of MQ administered objects accessed through a JNDI lookup as well as MQ administered objects instantiated and configured using `setProperty()` methods in client 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 "MQ 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 client 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 client code using the `System.setProperty()` method. This method will override attribute values of MQ 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

Optimizing Clients

The performance of JMS clients depends both on the inherent design of these applications and on the features and capabilities of the MQ client runtime.

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

The chapter covers the following topics:

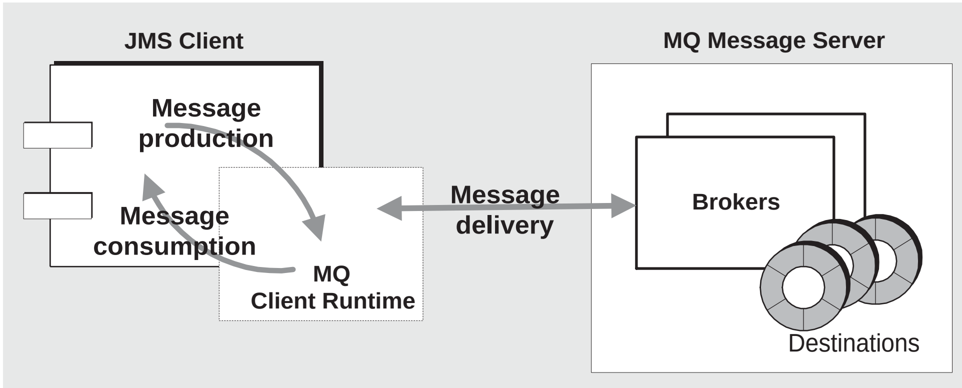
- message production and consumption
- configurable properties of the MQ client runtime
- factors that affect performance

Message Production and Consumption

The MQ client runtime provides JMS clients with an interface to the MQ message server—it supplies these clients with all the JMS programming objects introduced in "The JMS Programming Model" on page 27. 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 MQ 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 MQ client runtime, while message delivery involves an interaction between the MQ client runtime and the MQ 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 4-7 on page 78), but this eliminates the guarantee that persistent messages are reliably delivered.

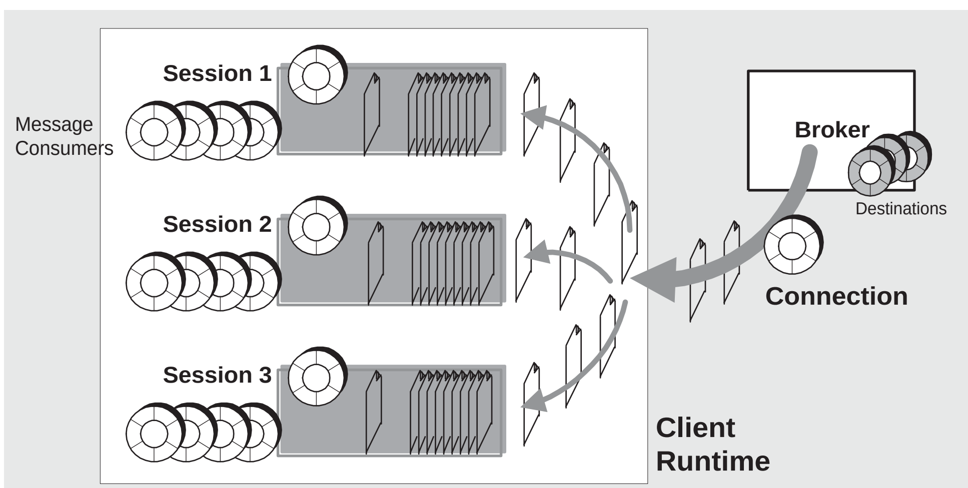
Message Consumption

Message consumption is more complex than production. Messages arriving at a destination on a broker are delivered over a connection to the MQ 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 MQ sessions where they are queued up to be consumed by the appropriate MessageConsumer objects, as shown in Figure 4-2. 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).

Figure 4-2 Message Delivery to MQ Client Runtime



When a broker delivers messages to the client runtime, it marks the messages accordingly, but does not really know if they have been received or consumed. Therefore, the broker waits for the client to acknowledge receipt of a message before deleting the message from the broker's destination.

In accordance with the JMS specification, there are three acknowledgment options that a client developer can set 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 option 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 up to that point in time. (This could include messages consumed asynchronously by many different message listeners in the session, independent of the *order* in which they were consumed.)
- `DUPS_OK_ACKNOWLEDGE`: the session acknowledges after a configurable number of messages have been consumed 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 options requires a different level of processing and bandwidth overhead. Automatic 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` options, 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 4-5 on page 76), but this eliminates the guarantee that persistent messages are reliably delivered.

NOTE In the `DUPS_OK_ACKNOWLEDGE` mode, the session does not wait for broker acknowledgements. This option is used in JMS 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.

MQ Client Runtime Configurable Properties

The MQ 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 JMS client and an MQ message server.

A `ConnectionFactory` object has no physical representation in a broker—it is used simply to enable the client to establish connections with a broker. The `ConnectionFactory` object is also used to specify behaviors of the connection and of the client runtime using the connection to access a broker. The `ConnectionFactory` object can also be used to manage MQ message server resources by overriding message header values set by clients.

If you wish to support distributed transactions (see "Local Transactions" on page 37), you need to use a special `XAConnectionFactory` object that supports distributed transactions.

`ConnectionFactory` administered objects are created by adminstrators or instantiated in client code, as described in Chapter 3, "Using Administered Objects."

By configuring a `ConnectionFactory` 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 specification
- Auto-reconnect behavior
- Client identification
- Message header overrides
- Reliability and flow control
- Queue browser behavior
- Application server support
- 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 MQ administration tools, as described in the MQ *Administrator's Guide*.

Connection Specification

Connections are specified by a broker's host name, the port number at which its Port Mapper resides (or at which a specific connection service resides), and the kind of connection service it supports. The behavior of a connection might require setting additional attribute values, depending on the connection type (the protocol used by the connection service).

The attributes that affect connection behavior are described in Table 4-1.

Table 4-1 Connection Factory Attributes: Connection Specification

Attribute/property name	Description
`imqConnectionType`	Specifies transport protocol of the connection service used by the client. Supported types are `TCP`, `TLS`, `HTTP`. Default: `TCP`
`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. 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.
`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`

Table 4-1 Connection Factory Attributes: Connection Specification *(Continued)*

Attribute/property name	Description
`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 MQ *Administrator's Guide*). Default: `0 (not used)`
`imqSSLIsHostTrusted`	Specifies whether the host is trusted (if `imqConnectionType` is `TLS`). Default: `true`
`imqConnectionURL`	Specifies the URL that will be used to connect to the MQ message server (if `imqConnectionType` is `HTTP`). A typical value (HTTPS connection) might be `https://`*hostName:port*`/imq/tunnel` Default: `http://localhost/imq/tunnel`

Auto-reconnect Behavior

MQ provides an automatic reconnect capability. If a connection fails, MQ maintains objects provided by the client runtime (sessions, message consumers, message producers, and so forth) while attempting to re-establish the connection. For transacted sessions, the impact of reconnect is indeterminate, and the behavior is not predictable. For non-transacted sessions, however, the effect of reconnection is different for message production and message consumption.

Message Production

During reconnect, producers cannot send messages. An exception is thrown and the producer must continue to retry until the connection is re-established.

Temporary destinations exist only for the duration of a connection and are therefore not recovered when a connection is re-established.

Message Consumption

For consumers, messages that have been delivered to the client runtime will continue to be available for consumption while auto-reconnect is being attempted. However, behavior upon reconnect depends on several factors:

- For sessions using CLIENT_ACKNOWLEDGE, the unacknowledged messages will be redelivered to a durable subscriber or queue receiver.
- For sessions using AUTO_ACKNOWLEDGE, there is a possibility that the last delivered message will be redelivered to the durable subscriber or queue receiver (because it might not yet have been acknowledged).
- For non-durable subscribers during connection recovery, messages will not be delivered under the following conditions.
    - if messages are produced during connection recovery
    - if messages have been produced but have not yet been delivered during connection recovery

The attributes that affect auto-connect behavior are described in Table 4-2.

Table 4-2 Connection Factory Attributes: Auto-reconnect Behavior

Attribute/property name	Description
`imqReconnect`	Specifies whether the client runtime will attempt to reconnect to the broker if the connection is lost. Default: `false`
`imqReconnectDelay`	Specifies the time between successive attempts of the client runtime to reconnect to the MQ message server (if `imqReconnect=true`). Default: `30000 milliseconds`
`imqReconnectRetries`	Specifies the number of attempts the client runtime will make to reconnect to the broker (if `imqReconnect=true`). A value of 0 indicates that the number of retries is not limited. Default: `0`

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 36).

For authentication purposes MQ 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 MQ *Administrator's Guide*) to perform application testing.

To keep track of durable subscriptions, MQ 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, client 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 client code, however, is not optimal. Each user needs a unique identification: this implies some centralized coordination. MQ 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 MQ 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 4-3.

Table 4-3 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

An MQ 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 21):

- 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 MQ administrator more control over the resources of an MQ 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.

MQ 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 4-4.

Table 4-4 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)
`imqOverrideJMSHeadersToTemporaryDestinations`	Specifies whether overrides apply to temporary destinations. Default: `false`

Reliability And Flow Control

A number of attributes determine the use and flow of MQ 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 4-5.

Table 4-5 Connection Factory Attributes: Reliability and Flow Control

Attribute/property name	Description
`imqAckOnProduce`	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`	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 4-5 Connection Factory Attributes: Reliability and Flow Control *(Continued)*

Attribute/property name	Description
imqFlowControlCount	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 MQ control messages are not blocked by heavy JMS message delivery. Default: 100
imqFlowControlIsLimited	Specifies if imqFlowControlLimit is enabled (only active if imqFlowControlCount is a non-zero value). Default: false
imqFlowControlLimit	Specifies a limit on the number of unconsumed messages that can be delivered to a client runtime (only used if imqFlowControlIsLimited=true). When the number of JMS messages delivered to the client runtime (in accordance with the flow metering governed by imqFlowControlCount) exceeds the 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

Queue Browser Behavior

The attributes that affect queue browsing for the client runtime are described in Table 4-6.

Table 4-6 Connection Factory Attributes: Queue Browser Behavior

Attribute/property name	Description
`imqQueueBrowserMaxMessagesPerRetrieve`	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`
`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`.

Application Server Support

The behavior of sessions running in an application server environment is affected by the attribute described in Table 4-7. For background see the JMS specification.

Table 4-7 Connection Factory Attributes: Application Server Support

Attribute/property name	Description
`imqLoadMaxToServerSession`	Used only for JMS application server facilities. Specifies whether an MQ 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 21). These properties enhance client programming capabilities.

The JMS-defined properties supported by MQ are described in Table 4-8.

Table 4-8 Connection Factory Attributes: JMS-defined Properties Support

Attribute/property name	Description
`imqSetJMSXUserID`	Specifies whether MQ should set the JMS-defined property, `JMSXUserID` (identity of user sending the message), on produced messages. Default: `false`
`imqSetJMSXAppID`	Specifies whether MQ should set the JMS-defined property, `JMSXAppID` (identity of application sending the message), on produced messages. Default: `false`
`imqSetJMSXProducerTXID`	Specifies whether MQ 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 MQ should set the JMS-defined property, `JMSXConsumerTXID` (transaction identifier of the transaction within which this message was consumed), on consumed messages. Default: `false`
`imqSetJMSXRcvTimestamp`	Specifies whether MQ should set the JMS-defined property, `JMSXRcvTimestamp` (the time the message is delivered to the consumer), on consumed messages. Default: `false`

Performance Factors

Because of the mechanisms by which messages are delivered to and from a broker, and because of the various MQ control messages used to assure reliable delivery, there are a number of factors that affect message flow and consumption.

The factors that affect message throughput and performance are the following: delivery mode, message flow metering, message flow limits, acknowledgement mode, and number of sessions. These factors are quite distinct, yet interact in a way that can make it difficult to determine which of them might be impacting performance in any given client.

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, especially for guaranteeing that persistent messages are not lost or delivered twice (as can happen in the case of a broker failure, a client failure, or a lost connection). For example, the number of client and broker control messages flowing across a connection, and the handling of client and broker acknowledgements, have a strong impact on message throughput and performance.

Message flow metering Messages sent and received by JMS clients (JMS messages) and MQ control messages pass over the same connection between client and broker. This can result in a bottleneck in the flow of control messages. For example, suppose a broker is flooding a connection with JMS messages being delivered to a client. That could create a delay in the delivery of control messages, such as broker acknowledgements. Hence, if you are experiencing delays, the cause might be that control messages are being held up by heavy delivery of JMS messages. To prevent this type of congestion, MQ meters the flow of JMS messages across connections: JMS messages are batched so that only a set number can be delivered before delivery is temporarily suspended. Before JMS message delivery resumes, control messages that had previously been held up are delivered to the client. You can specify the number of messages allowed in such metered batches of JMS messages (see `imqFlowControlCount` property, Table 4-5 on page 76). In cases of heavy JMS message delivery, decreasing the count should allow control messages to flow across a connection with less delay.

Message flow limits MQ client runtime code can handle only a limited number of delivered JMS messages before encountering local resource limitations, such as memory. When this limit is approached, performance suffers. Hence, MQ lets you limit the number of messages queued up in sessions awaiting consumption by controlling the flow of JMS messages to the client. You do this by specifying a threshold value (see `FlowControlLimit` property, Table 4-5 on page 76). If this threshold value is exceeded by delivery of a batch of JMS messages (see "Message flow metering" on page 80), the client runtime will wait until the number of un-consumed messages drops below this threshold before requesting that the delivery of JMS messages be resumed. Message delivery then continues until the threshold value is again exceeded.

Client acknowledgement mode The different client acknowledgement modes affect the number of client and broker acknowledgement messages passing over a connection:

- In the `AUTO_ACKNOWLEDGE` mode, a client acknowledgement and broker acknowledgement are required for each consumed message, and the session thread blocks waiting for the broker acknowledgement.
- 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 wait time for broker acknowledgements.
- 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.

Number of sessions and connections The number of messages queued up in a session, and the time it takes to process them, is a function of the number of message consumers in the session and the message load for each consumer. If a client is exhibiting delays in producing or consuming messages, performance might be improved 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

Working With SOAP Messages

Using Sun ONE MQ, 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 MQ
- 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 91.

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.

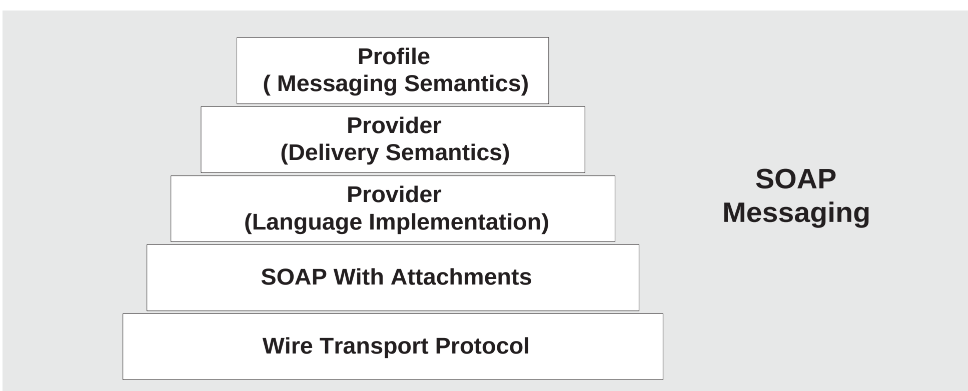
The SOAP protocol is fully described at `http://www.w3org/TR/SOAP.` This section restricts itself to discussing the reasons why you would use SOAP and to describing some basic concepts that will make it easier to work with the JAXM API.

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

Figure 5-1 SOAP Messaging Layers



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

The Transport Layer

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

The SOAP Layer

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

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

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

The Provider Layer

In Figure 5-1 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.

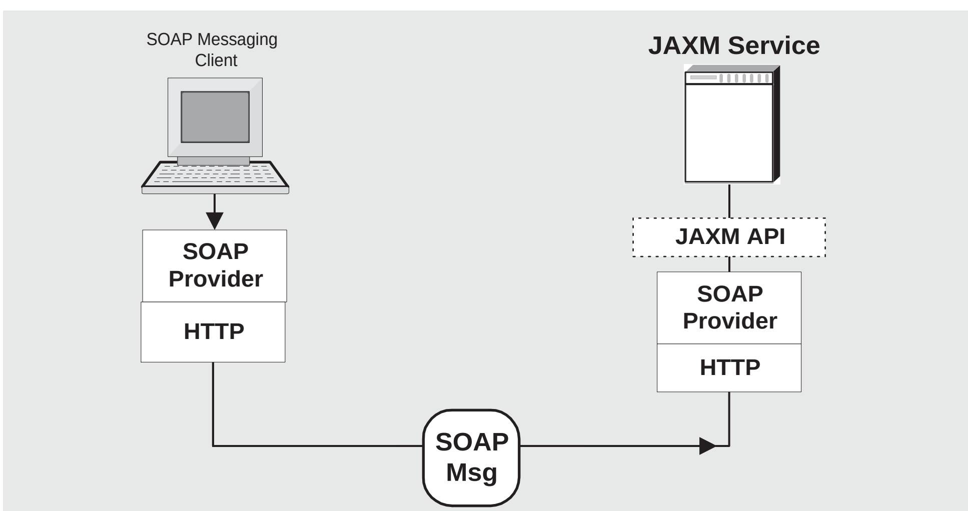
Messaging Semantics

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 MQ in future releases.

Interoperability

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

Figure 5-2 SOAP Interoperability



In order for a 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 MQ.

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.

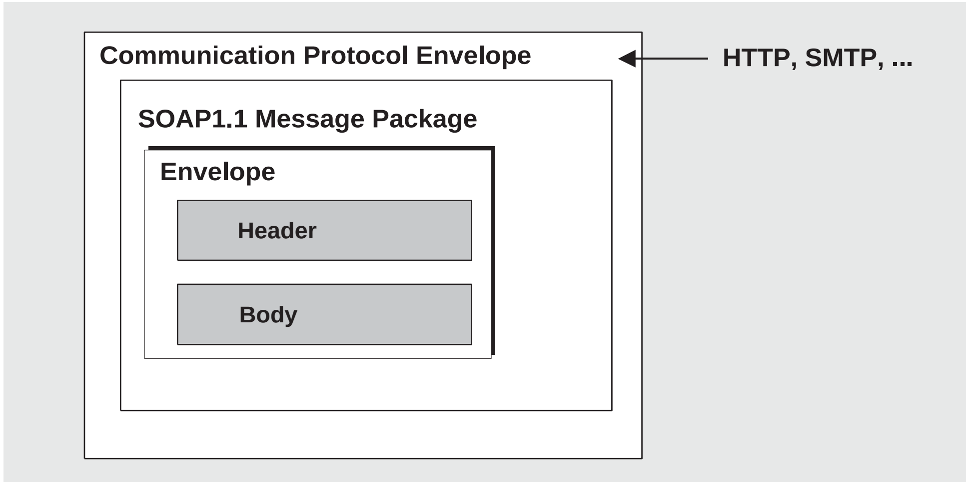
A SOAP message is constructed like a nested matrioshka doll. When you use JAXM to assemble or disassemble a message, you need to make the API calls in the appropriate order to get to the message part that interests you. For example, in order to add content to the message, you need to get to the body 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 91.

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

Figure 5-3 SOAP Message Without Attachments

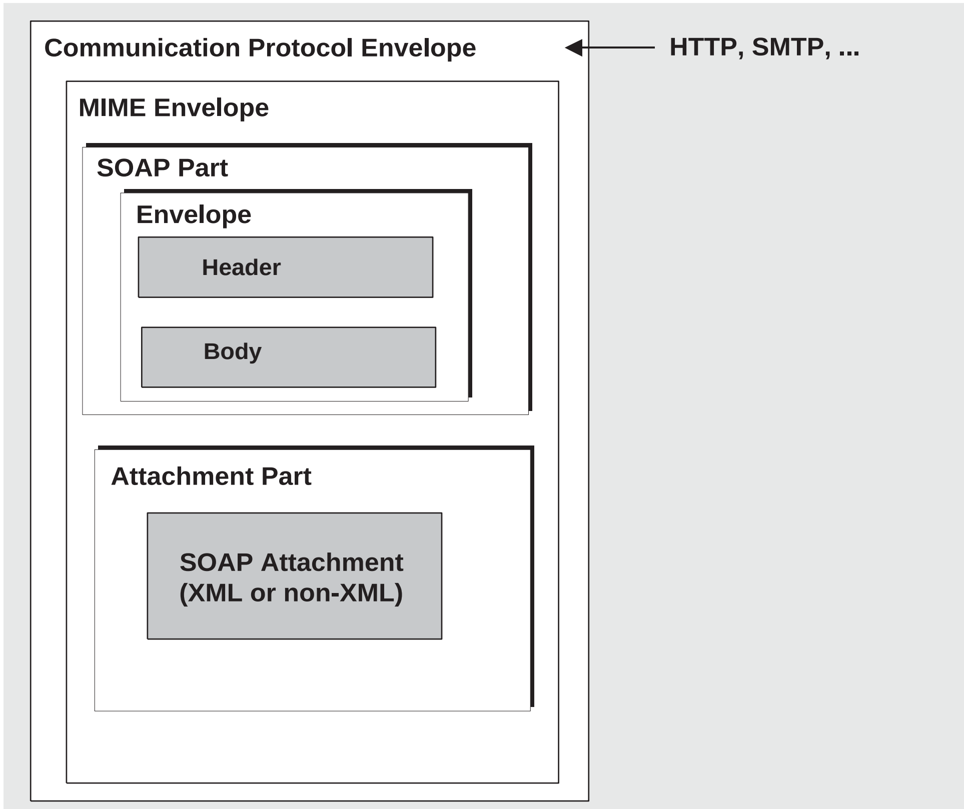


When you construct a SOAP message using JAXM, you do not have to specify which model you're following. If you add an attachment, a message like that shown in Figure 5-4 is constructed; if you don't, a message like that shown in Figure 5-3 is constructed.

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

A message that contains one or more attachments is enclosed in a MIME envelope that contains all the parts of the message. In 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 5-4 SOAP Message with Attachments



SOAP Messaging in JAVA

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

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

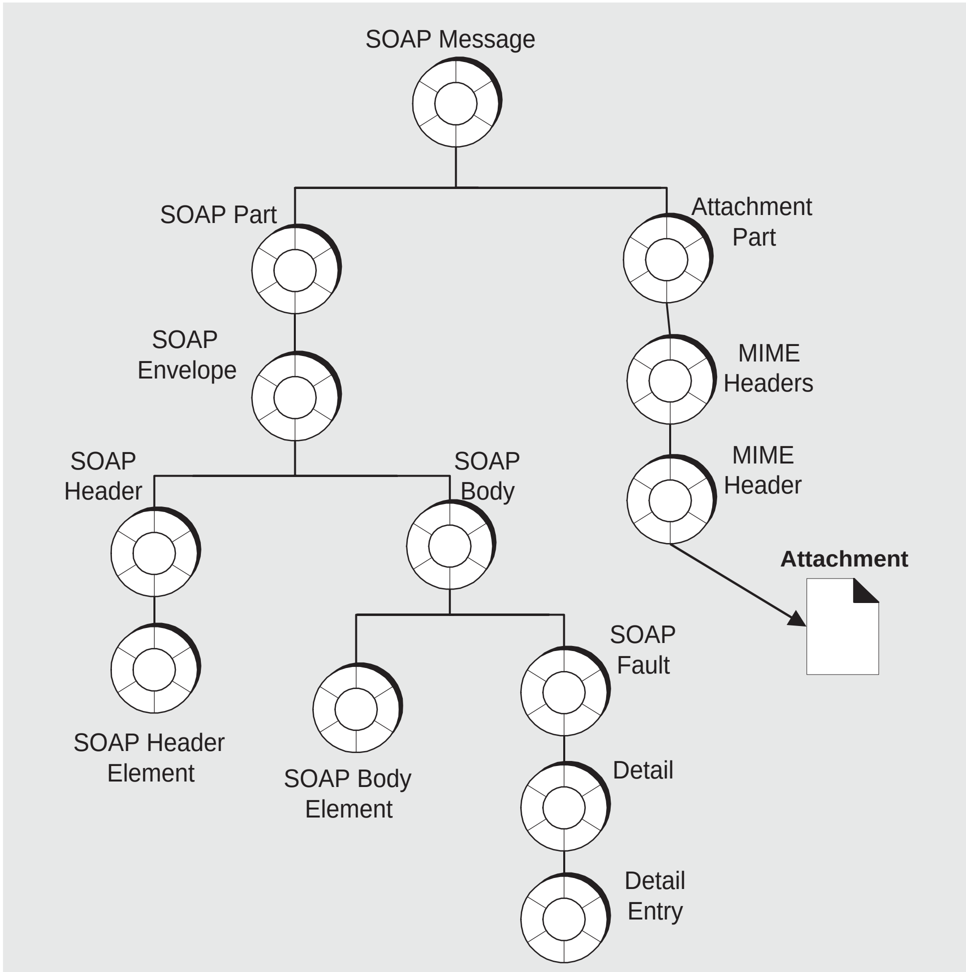
- to assemble and disassemble SOAP messages
- to send and receive these messages.

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

The SOAP Message Object

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

Figure 5-5 SOAP Message Object



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

The SOAP part of the message contains an envelope that contains a body (which can contain data or fault information) and an optional header. When you use 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 we have created the SOAPMessage `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 95) and add the body element to the SOAPMessage.

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

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

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

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

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

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

Inherited Methods

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

Table 5-1 Inherited Methods

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

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

	`addNameSpaceDeclaration (String, String)`	Add a namespace declaration with the specified prefix and URI.

Table 5-1 Inherited Methods *(Continued)*

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

Namespaces

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

An explicit XML namespace declaration takes the following form

<*prefix*:*myElement*

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

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

The following is an example of a namespace declaration:

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

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

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

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

Code Example 5-1 Explicit Namespace Declarations

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

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

Pre-defined SOAP Namespaces

SOAP defines two namespaces:

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

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

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

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

When you use 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 100.

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 URL's--each URL, is capable of processing incoming SOAP messages. The code sample below associates the endpoint whose lookup name is `myEndpoint` with two URL's: `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 100.

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 MQ; for more information, see "Integrating SOAP and MQ" on page 116.

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. (MQ 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 URL's 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 MQ: you use the Object Manager (`imqobjmgr`) utility to specify the lookup name of the object, its attributes, and its type.

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

Table 5-2 SOAP Administered Object Information

Option	Description
-o "*attribute=val*"	Use this option to specify three possible attributes for an endpoint administered object: • **A URL list** -o "`imqSOAPEndpointList` = "*url1 url2 ....urln*" The list may contain one or more space-separated url's.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 "`imqSOAPEndpointName`=*SomeName*" If you don't specify a name, the name `Untitled_Endpoint_Object` is used by default. • **A description** `-o "imqSOAPEndpointDescription=my endpoints for broadcast"` If you don't specify a description, the default value `"A description for the endpoint object"` is supplied by default.

Table 5-2 SOAP Administered Object Information *(Continued)*

Option	Description
-l "cn=*lookupName*"	Use this option to specify the lookup name of the endpoint.
-t *type*	Use this option to specify the object's type. This is always `e` for an endpoint.
-i *filename*	Use this option to specify the name of an input file containing `imqobjmgr` commands. Such an input file is typically used to specify object store attributes.
-j "*attribute=val*"	Use this option to specify object store attributes. You can also specify these in an input file. Use the -i option to specify the input file.

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

Code Example 5-2 Adding an Endpoint Administered Object

```
imgobjmgr add
  -t ep
  -l "cn=myEndpoint"
  -o "imqSOAPEndpointList=http://www.myServlet/
                   http://www.myServlet2/"
  -o "imqSOAPEndpointName=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 5-3, you first create an initial context for the JNDI lookup and then you look up the desired object.

Code Example 5-3 Looking up an Endpoint Administered Object

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

You can also list, delete, and update administered objects. For additional information, please see MQ *Administrator's 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 MQ" on page 116.

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

SOAP Messaging Programming Models

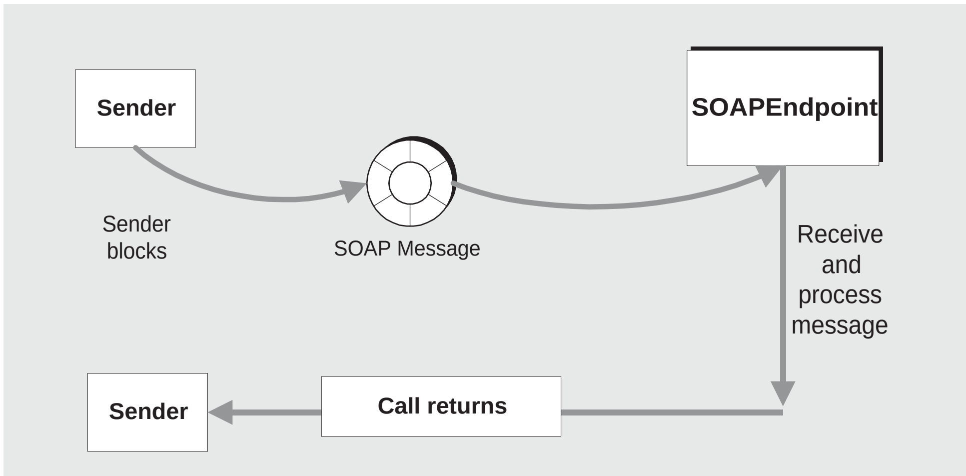
This section provides a brief summary of the programming models used in SOAP messaging using JAXM.

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 5-6.

Figure 5-6 Request-Reply Messaging



Using this model, the client does the following:

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

  See "Endpoint" on page 98 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.

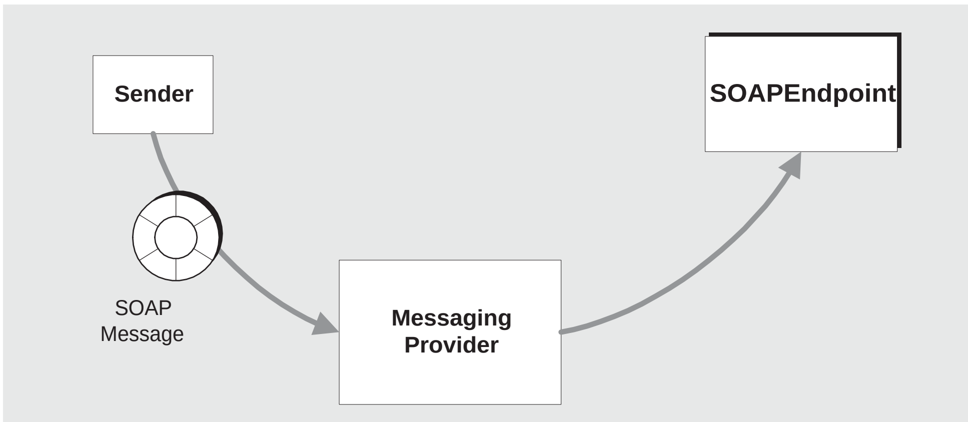
The JAXM service listening for a request-reply message uses a `ReqRespListener` object to receive messages.

For a detailed example of a client that does point-to-point messaging, see "Writing a SOAP Client" on page 106.

Provider Connections

You use a provider connection to implement one-way messaging. Figure 5-7 illustrates the one-way messaging model.

Figure 5-7 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 client 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 of topic using MQ.

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 112.

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.

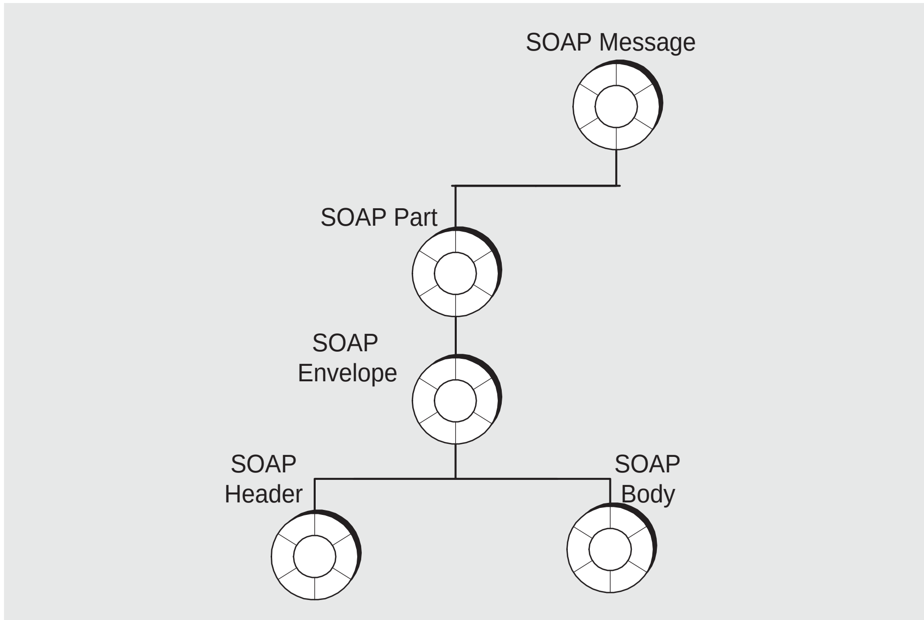
3. Get a `MessageFactory` object to create a message:

```
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 the next figure.



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, we'll create another body element to add to the `gltp` element:

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

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

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

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

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

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

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

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

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

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

11. Now, you can send the message:

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

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

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

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

Writing a SOAP Service

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

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

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

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

Code Example 5-4 Skeleton Message Consumer

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

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

Code Example 5-5 A Simple Ping Message Service

```
public class SOAPEchoServlet extends JAXMServlet
                  implements ReqRespListener{

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

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

Table 5-3 JAXMServlet Methods

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

Table 5-3 JAXMServlet Methods *(Continued)*

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

Disassembling Messages

The `onMessage` method needs to disassemble the SOAP message that is passed to it by the servlet and process its contents in an appropriate manner. If there are problems in the processing of the message, the service needs to create a SOAP fault object and send it back to the client as described in "Handling SOAP Faults" on page 112.

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

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

Code Example 5-6 Processing a SOAP Message

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

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

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

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

Code Example 5-6 Processing a SOAP Message *(Continued)*

```
{http://xml.coverpages.org/dom.html
  SOAPEnvelope env = reply.getSOAPPart().getEnvelope();
  //Get the value for XElement
  MyValue =   sbe.getValue();
}
```

Handling Attachments

A SOAP message may have attachments. For sample code that shows you how to create and add an attachment, see Code Example 5-7 on page 122. For sample code that shows you how to receive and process an attachment, see Code Example 5-8 on page 124.

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

Replying to Messages

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

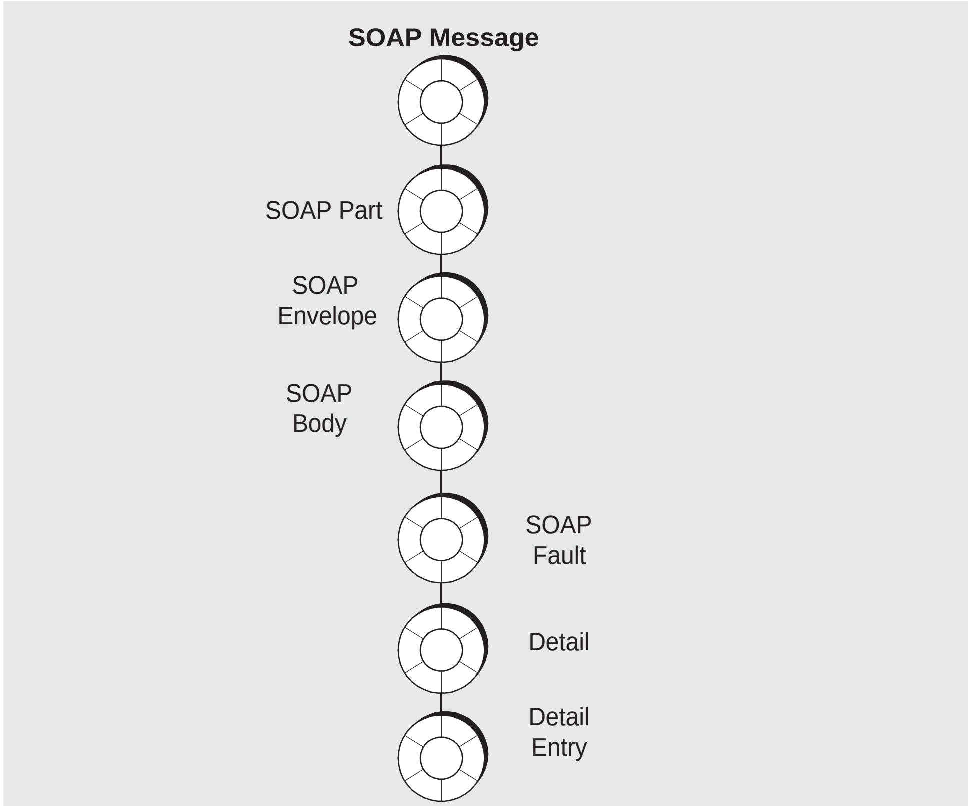
Handling SOAP Faults

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

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

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

Figure 5-8 SOAP Fault Element



The SOAP Fault element defines the following four subelements:

- `faultcode`

  A code (qualified name) that identifies the error. The code is intended for use by software to provide an algorithmic mechanism for identifying the fault. Predefined fault codes are listed in Table 5-4 on page 114. This element is required.

- `faultstring`

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

- `faultactor`

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

- `detail`

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

Predefined Fault Codes

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

Table 5-4 SOAP Faultcode Values

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

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

Faultcode Name	Meaning
`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 MQ

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

MQ 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 in order to take advantage of MQ's reliable messaging service, and then convert it back into a SOAP message on the receiving side and process it as such using the JAXM API.

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`.

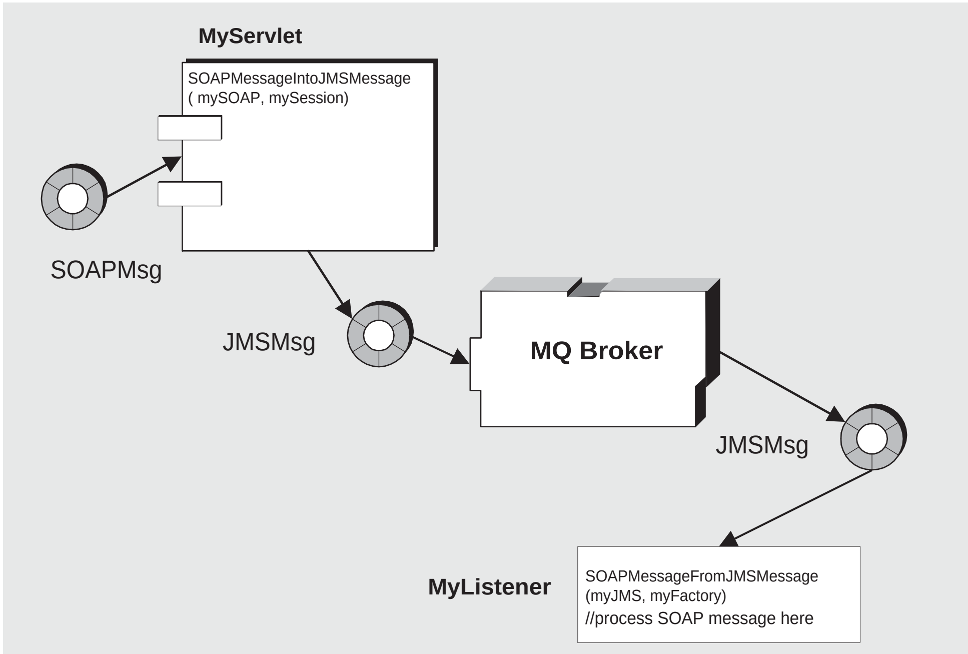
The following sections offer several use cases and code examples to illustrate this process.

Example 1: Deferring SOAP Processing

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

For information on how the servlet receives the SOAP message, see “Writing a SOAP Service” on page 109.

Figure 5-9 Deferring SOAP Processing



➤ To transform the SOAP message into a JMS message and send the JMS message

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

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

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

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

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

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

4. Instantiate an MQ Destination administered object corresponding to a physical destination in the MQ 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);
```

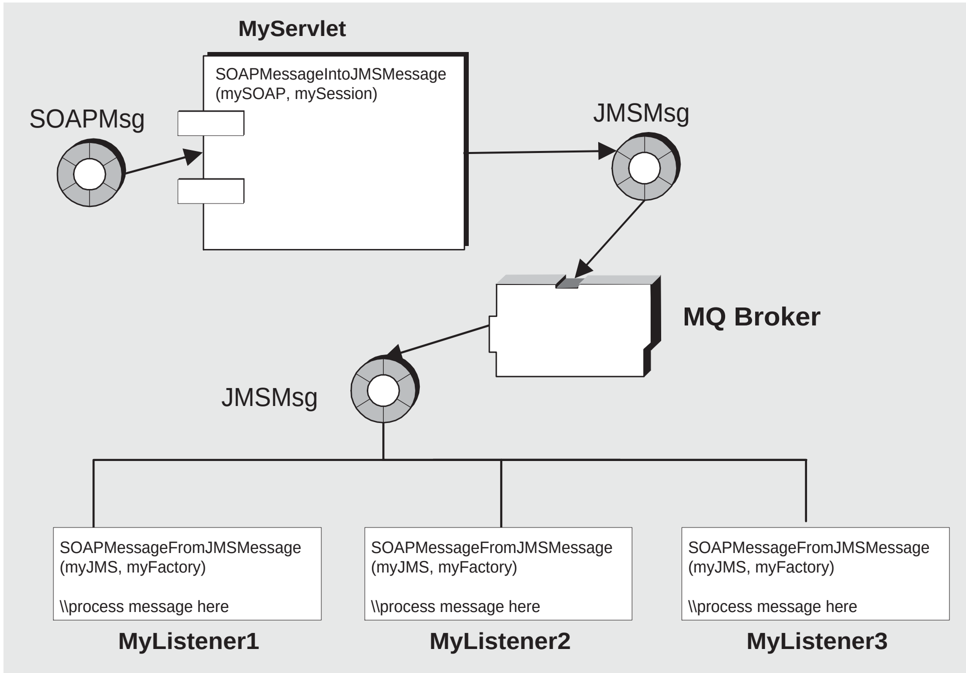
If you specify null for the `MessageFactory` argument, the default Message Factory is used to construct the SOAP Message.

10. Disassemble the SOAP message in preparation for further processing. See "The SOAP Message Object" on page 91 for information.

Example 2: Publishing SOAP Messages

In the next example, illustrated in Figure 5-10, an incoming SOAP message is received by a servlet. The servlet packages the SOAP message as a JMS message and (reliably) forwards it to a topic. Each application that subscribes to this topic, receives the JMS message, turns it back into a SOAP message, and processes its contents.

Figure 5-10 Publishing a SOAP Message



The code that accomplishes this is exactly the same as in the previous example, except that instead of sending the JMS message to a queue, you send it to a topic. For an example of publishing a SOAP message using MQ, see Code Example 5-7 on page 122.

Code Samples

This section includes and describes two code samples: one that sends a JMS message with a SOAP payload, and another that receives the JMS/SOAP message and processes the SOAP message.

Code Example 5-7 illustrates the use of the JMS API, the 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 5-7 Sending a JMS Message with a SOAP Payload

```
//Libraries needed to build SOAP message
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPPart;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.AttachmentPart;
import javax.xml.soap.Name

//Libraries needed to work with attachments (Java Activation Framework API)
import java.net.URL;
import javax.activation.DataHandler;

//Libraries needed to convert the SOAP message to a JMS message and to send it
import com.sun.messaging.xml.MessageTransformer;
import com.sun.messaging.BasicConnectionFactory;

//Libraries needed to set up a JMS connection and to send a message
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicConnection;
import javax.jms.JMSException;
import javax.jms.Session;
import javax.jms.Message;
import javax.jms.TopicSession;
import javax.jms.Topic;
import javax.jms.TopicPublisher;

//Define class that sends JMS message with SOAP payload
public class SendSOAPMessageWithJMS{

  TopicConnectionFactory tcf = null;
  TopicConnection tc = null;
  TopicSession session = null;
  Topic topic = null;
  TopicPublisher publisher = null;

//default constructor method
public SendSOAPMessageWithJMS(String topicName){
  init(topicName);
  }

//Method to nitialize JMS Connection, Session, Topic, and Publisher
public void init(String topicName) {
  try {
```

Code Example 5-7 Sending a JMS Message with a SOAP Payload *(Continued)*

```
    tcf = new com.sun.messaging.TopicConnectionFactory();
    tc = tcf.createTopicConnection();
    session = tc.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
    topic = session.createTopic(topicName);
    publisher = session.createPublisher(topic);
    }

//Method to create and send the SOAP/JMS message
public void send() throws Exception{
  MessageFactory mf = MessageFactory.newInstance(); //create default factory
  SOAPMessage soapMessage=mfcreateMessage(); //create SOAP message object
  SOAPPart soapPart = soapMessage.getSOAPPart();//start to drill down to body
  SOAPEnvelope soapEnvelope = soapPart.getEnvelope(); //first the envelope
  SOAPBody soapBody = soapEnvelope.getBody();
  Name myName = soapEnvelope.createName("HelloWorld", "hw",
                  http;//www.sun.com/imq’); //name for body element
  SOAPElement element = soapBody.addChildElement(myName); //add body element
  element.addTextNode("Welcome to SUnOne Web Services."); //add text value

  //Create an attachment with the Java Framework Activation API
  URL url = new URL("http://java.sun.com/webservices/");
  DataHandler dh = new DataHnadler (url);
  AttachmentPart ap = soapMessage.createAttachmentPart(dh);

  //Set content type and ID
  ap.setContentType("text/html");
  ap.setContentID(’cid-001");

  //Add attachment to the SOAP message
  soapMessage.addAttachmentPart(ap);
  soapMessage.saveChanges();

  //Convert SOAP to JMS message.
  Message m = MessageTransformer.SOAPMessageIntoJMSMessage(soapMessage,
                  session);

//Publish JMS message
  publisher.publish(m);

//Close JMS connection
  public void close() throws JMSException {
    tc.close();
  }

//Main program to send SOAP message with JMS
public static void main (String[] args) {
  try {
    String topicName = System.getProperty("TopicName");
    if(topicName == null) {
      topicName = "test";
    }

    SendSOAPMEssageWithJMS ssm = new SendSOAPMEssageWithJMS(topicName);
```

Code Example 5-7 Sending a JMS Message with a SOAP Payload *(Continued)*

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

Code Example 5-8 illustrates the use of the JMS API, 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 5-8 Receiving a JMS Message with a SOAP Payload

```
//Libraries that support SOAP processing
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.AttachmentPart

//Library containing the JMS to SOAP transformer
import com.sun.messaging.xml.MessageTransformer;

//Libraries for JMS messaging support
import com.sun.messaging.TopicConnectionFactory

//Interfaces for JMS messaging
import javax.jms.MessageListener;
import javax.jms.TopicConnection;
import javax.jms.TopicSession;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.Topic;
import javax.jms.JMSException;
```

Code Example 5-8 Receiving a JMS Message with a SOAP Payload *(Continued)*

```
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();

    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()) {
```

Code Example 5-8 Receiving a JMS Message with a SOAP Payload *(Continued)*

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

Administered Object Attributes

This appendix provides reference tables for the attributes of the ConnectionFactory, XAConnectionFactory, and destination 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 "MQ Client Runtime Configurable Properties" on page 69.

Table A-1 Connection Factory Attributes

Attribute/property name	Type	Default Value	Reference
imqAckOnAcknowledge	String	not specified	Table 4-6 on page 78
imqAckOnProduce	String	not specified	Table 4-6 on page 78
imqAckTimeout	String	0 milliseconds	Table 4-6 on page 78
imqBrokerHostName	String	localhost	Table 4-1 on page 70
imqBrokerHostPort	String	7676	Table 4-1 on page 70
imqBrokerServicePort	String	0	Table 4-1 on page 70
imqConfiguredClientID	String	not specified	Table 4-3 on page 74
imqConnectionType	String	TCP	Table 4-1 on page 70
imqConnectionURL	String	http://localhost/imq/tunnel	Table 4-1 on page 70

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

Attribute/property name	Type	Default Value	Reference
imqDefaultPassword	String	guest	Table 4-3 on page 74
imqDefaultUsername	String	guest	Table 4-3 on page 74
imqDisableSetClientID	String	false	Table 4-3 on page 74
imqFlowControlCount	String	100	Table 4-5 on page 76
imqFlowControlIsLimited	String	false	Table 4-5 on page 76
imqFlowControlLimit	String	1000	Table 4-5 on page 76
imqJMSDeliveryMode	String	2 (persistent)	Table 4-4 on page 75
imqJMSExpiration	Integer	0 (does not expire)	Table 4-4 on page 75
imqJMSPriority	Integer	4 (normal)	Table 4-4 on page 75
imqLoadMaxToServerSession	String	true	Table 4-8 on page 79
imqOverrideJMSDeliveryMode	Boolean	false	Table 4-4 on page 75
imqOverrideJMSExpiration	Boolean	false	Table 4-4 on page 75
imqOverrideJMSPriority	Boolean	false	Table 4-4 on page 75
imqOverrideJMSHeadersTo TemporaryDestinations	Boolean	false	Table 4-4 on page 75
imqQueueBrowserMaxMessages PerRetrieve	String	1000	Table 4-7 on page 78
imqQueueBrowserRetrieveTimeout	String	60,000 milliseconds	Table 4-7 on page 78
imqReconnect	String	false	Table 4-2 on page 72
imqReconnectDelay	String	30,000 milliseconds	Table 4-2 on page 72
imqReconnectRetries	String	0	Table 4-2 on page 72
imqSetJMSXAppID	String	false	Table 4-8 on page 79
imqSetJMSXConsumerTXID	String	false	Table 4-8 on page 79
imqSetJMSXProducerTXID	String	false	Table 4-8 on page 79
imqSetJMSXRcvTimestamp	String	false	Table 4-8 on page 79
imqSetJMSXUserID	String	false	Table 4-8 on page 79
imqSSLIsHostTrusted	String	true	Table 4-1 on page 70

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*	`Untitled_Destination_Object`

* 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."

Index

D

E

F

I

J

L

M

N

O

P

Q

R

S

T

U

W

X