Java™ Message Queue QuickStart Guide

V1.1

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131

Part No.: 806-4796-10
Revision A May 2000

Copyright

Please Recycle

May 2000

Contents

Chapter 1:

Overview

The Java Message Queue product is a standards-based solution to the problem of inter-application communication and reliable data transmission across networks. With Java Message Queue software, processes running in different architectures and operating systems simply connect to the same virtual network to send and receive information.

Java Message Queue handles all data translation between application processes. Application developers are free to focus on the key business logic of their applications, rather than low-level details of how their applications will be physically deployed on the network. Based on the Java Messaging Service (JMS) open standard, Java Message Queue applications are designed to be easily portable across different computer architectures and operating systems.

The following sections describe the product in greater detail:

- "Product Features" on page 1 provides a listing of key elements of the Java Message Queue product.
- "Product Releases" on page 3 explains the available versions of the product and outlines what they contain.

Product Features

The Java Message Queue 1.1 product is a powerful yet flexible messaging solution. An implementation of the Java Message Service specification, it represents an architecture designed and adopted by major messaging vendors. Support for Java Naming and Directory Interface (JNDI) enables the creation of provider-independent client applications. Simple installation and easy-to-use administrative utilities round out the offering—making Java Message Queue a solid choice for many messaging applications. Additional details on some of these features are provided below.

Cross-platform

Designed with the heterogeneous environment in mind, the Java Message Queue product is supported on both the Windows (NT and 2000) and Solaris SPARC platforms. You can run the product with either JDK™ 1.1.8 or Java 2, Standard Edition SDK—choose the release-level that fits your environment.

JMS 1.0.2 API Conformance

Java Message Queue now supports Java Message Service (JMS) 1.0.2—the previous release supported JMS 1.0.1. The JMS architecture includes:

- Publish-and-subscribe Message Model. Broadcasts messages to multiple recipients. The *publish-and-subscribe* model provides for synchronous and asynchronous delivery of messages to *topics* with multiple subscribers.
- Point-to-point Message Model. Limits a message to a single recipient. The *point-to-point* model provides for synchronous and asynchronous delivery of a message to the *queue* of a given recipient.
- Certified Message Delivery. Enables the tracking of messages receipt. *Client acknowledgment* offers the highest certainty that messages are properly formed, reliably delivered, and accurately processed.

Access to Objects Through JNDI

Java Message Queue now provides a utility for the management of administered objects, such as Topics and Queues, through the Java Naming and Directory Interface (JNDI). The utility, called `jmqconfig`, provides the following functionality:

- Configuring and storing an administered object into a service provider.
- Removing an administered object from a service provider.
- Displaying the configuration of an administered object.
- Listing administered objects of a particular type.

Supported service providers are LDAP servers and file systems. The Java Message Queue product includes Sun Microsystems' LDAP and File System service providers for JNDI. (The included File System service provider is at Beta level and therefore not supported for deployment.)

Multi-router Networks

Fully connected multi-router configurations are supported. By a "fully connected network" it is meant a network in which every router is connected to every other router by no more than one "hop".

Administration

With the Java Message Queue product, you can manage multiple distributed applications as if they were a single local application. The Java Message Queue software includes several utilities that simplify and centralize the critical tasks of debugging, tuning, and monitoring the Java Message Queue network.

Java 2 Policy Files

Three policy files are provided with Java Message Queue for use with the Java 2 security manager: `jmqadmin.policy`, `jmqclient.policy`, and `jmqexamples.policy`.

- The `jmqadmin.policy` file provides the permissions needed by the `jmqconfig` and `jmqadmin` administration utilities to create log files, delete log files, and open socket connections (for instance, for starting the Client Authentication Server or connecting to the Router). In most cases, the `jmqadmin.policy` file will not require modification.
- The `jmqclient.policy` file provides the permissions necessary for your client application to access the router and network resources. The file should be modified to reflect the security requirements of your particular application. Guidelines for possible modifications are provided in the `jmqclient.policy` file itself.
- The `jmqexamples.policy` files provides the permissions necessary for running the Java Message Queue examples. If you modify the examples, you might need to modify the policy file to reflect those changes. Guidelines for making modifications are provided in the file itself.

Product Releases

The Java Message Queue 1.1 product is available in two forms: a free Developer Edition (development-only) and a for-purchase Business Edition (development and deployment). The Business Edition is available in either a single-license or 10-license pack. The nature and contents of each edition are detailed below.

Developer Edition

The Developer Edition of the Java Message Queue product provides all the tools you need to start creating Java Message Queue (JMS-compliant) client applications. It includes:

- Class libraries that implement the JMS API.
- Redistributable client runtime.
- 5-connection router.
- Single router instance, development-only license.
- Command-line utilities.
- Client Authentication Server for development-only.
- Sample client applications.
- Online documentation, consisting of:
  - Java Message Queue QuickStart Guide. Includes a high-level overview of the product, an introduction to Java Message Queue concepts, and an outline of the minimum steps for creating a simple client application.

- Java Message Queue Deployment Guide. Describes creating a router network, running a Java Message Queue router and applications, monitoring the network, and troubleshooting network problems.
- Java Message Queue API Guide. Provides complete class and member descriptions for Java Message Queue and JMS APIs.
- Java Message Service 1.0.2 Specification.

The product license allows you to run one router instance. To run multiple router instances, you must purchase additional licenses. For example, running one router on one subnet and another router on another subnet requires you to have two licenses (regardless of whether you are running the routers on the same host or different hosts).

Business Edition

In addition to the capabilities of the Developer Edition, the Business Edition provides an unlimited-connection router licensed for development or deployment on a single host.

To run multiple router instances, you must purchase additional licenses (see Development Edition for details).

Business 10-pack Edition

Identical to the standard Business Edition, the Business 10-pack Edition provides a 10-license pack at a cost savings over purchasing individual licenses.

This allows you to run up to 10 router instances on the same host, or up to 10 router instances (total) across several hosts.

Chapter 2:

Concepts

The Java Message Queue product enables the transmission of messages (sharing of data) between application processes in a distributed environment. This document describes the basic concepts underlying the Java Message Queue messaging system. The document is divided into the following sections:

- "Messaging Systems" on page 5 introduces the purpose and components of a messaging system.
- "Data Distribution Architecture" on page 7 discusses the differences between fully-connected and virtual fully-connected networks.
- "Communication Models" on page 8 describes the point-to-point and publish-and-subscribe models of communication.
- "Synchronous and Asynchronous Communications" on page 9 provides insight into the ways the sender and receiver function in the transmission of messages.
- "Administered Objects" on page 9 outlines the benefits of using administered objects to encapsulate provider-specific information.
- "Messages" on page 10 defines the structure of messages as defined by the Java Message Service Specification.
- "Transactions" on page 12 explains the role of transacted sessions in a messaging system.

Messaging Systems

Message-oriented middleware (MOM) refers to data transmission between front-end applications and back-end applications. Middleware can be composed of many layers, each addressing the specific requirements of interapplication communication. In the past, proprietary messaging solutions placed significant demands on vendors and restrictions on customers. To address this situation, an open standard—Java Message Service (JMS)— was developed with input from major MOM vendors.

A *JMS provider* is a messaging product that implements the JMS specification while providing the additional features needed by enterprise messaging systems. Examples of features which might be valued by enterprises are: ease and completeness of administration, security, load balancing, and business process analysis.

As indicated, the purpose of a messaging system is to accept messages (data) from one client for delivery to another. Typically, the system is implemented in a manner that promises some degree of performance, reliability, and security. A *messaging system* is composed of client applications, messages, and the messaging service.

A client application refers to one or more processes which coordinate to implement some functionality. These processes may be located on a single machine or distributed across different machines in different locations. As indicated in FIGURE 2-1, a client application might itself be part of a larger application.

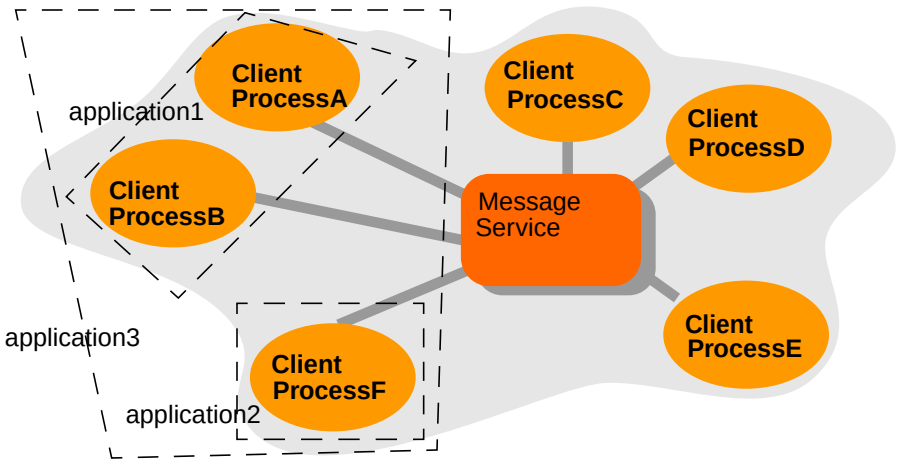
The messages contain not only data but information about the data as well. Messages are sent in a pre-defined format understood by the sending client process and the receiving client process.

The message service includes:

- A message server (sometimes called ”message router“ or a ”message broker“).
- Administration tools to manage the message server and messages in the system.
- An API for creating messages and interacting with the service.

In the case of the Java Message Queue product, client applications are written in conformance with the Java Message Service specification. The message formats are defined by that specification. The message service is comprised of the Java Message Queue router and its utilities. Together, these components form the Java Message Queue messaging system.

FIGURE 2-1 Messaging System



Data Distribution Architecture

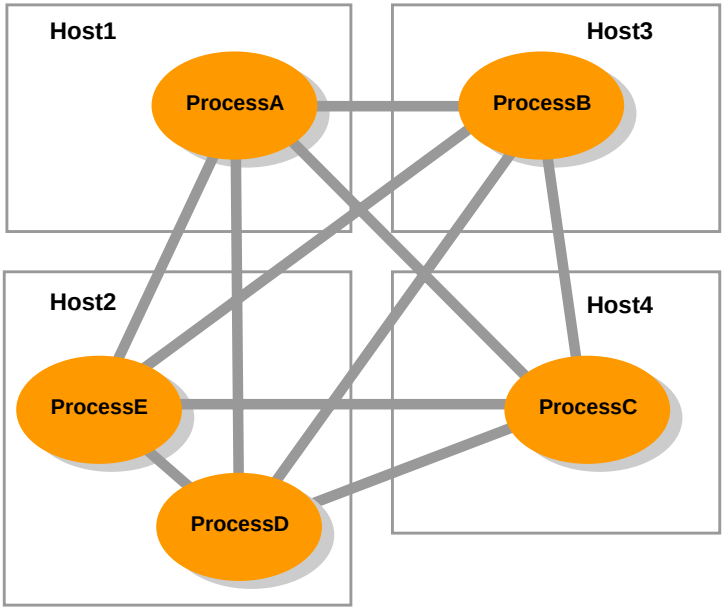
The design of the messaging network has important implications for the efficiency of the messaging delivery. Efficiency can be thought of both in terms of the speed and certainty with which messages are delivered, as well as programming effort to create and maintain and extend the system. Two common data distribution architectures are the fully-connected network and the virtual fully-connected network. Although they are similarly named, there are differences.

Fully Connected Network

In a fully connected network, all processes are directly connected to all other processes. FIGURE 2-2 shows an example of such a system.

Any process in this system can read or write information to or from any other process. In most implementations of such a network, each process would have an open file descriptor for every other process and would have to keep track of its connections with those processes. This results in significant overhead from both a system and programming perspective.

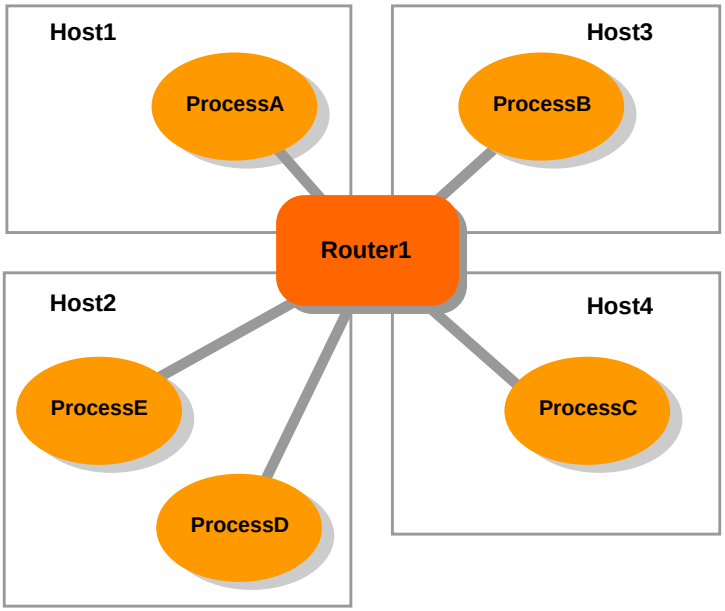
FIGURE 2-2 Traditional Fully Connected Network



Virtual Fully Connected Network

The Java Message Queue product implements a “virtual” fully connected network, as shown in FIGURE 2-3. The arrangement removes the burden of managing connections from the user program by making a separate process, called the *router*, responsible for maintaining all necessary connections. The user program has only a single connection—with its local Java Message Queue router. This simplification not only reduces the complexity of the user code, but also improves performance by optimizing the data distribution. All intraprocess (within router) and interprocess communications are handled by the router (or routers) automatically.

FIGURE 2-3 Virtual Fully Connected Network



Communication Models

There are two prevalent communication models used in middleware messaging: point-to-point (queues) and publish-and-subscribe (topics).

Point-to-point Model

In point-to-point communication, a message has at most one recipient. A sending client addresses the message to the *queue* that holds the messages for the intended (receiving) client. You can think of the queue as a mailbox. Many clients might send

messages to the queue, but a message is taken out by only one client. And, like a mailbox, messages remain in the queue until they are removed. Thus the availability of the recipient client does not affect the ability to deliver a message.

In a point-to-point system, a client can be a sender (message producer), a receiver (message consumer), or both.

Publish-and-subscribe Model

In contrast to the point-to-point model of communication, the publish-and-subscribe model enables the delivery of a message to multiple recipients. A sending client addresses (publishes) the message to a *topic* to which multiple clients can be subscribed. There can be multiple publishers, as well as subscribers, to a topic.

A *durable subscription (*or *interest)* exists across client shutdowns and restarts. While a client is down, all objects that would have been delivered to the topic are stored and then sent to the client when it renews the subscription.

In a publish-and-subscribe system, a client can be a publisher (message producer), a subscriber (message consumer), or both.

Synchronous and Asynchronous Communications

In the JMS specification which Java Message Queue implements, the sending client is either synchronous or asynchronous in nature. When asynchronous, sending to the message server (router) completes the sending client's end of the communication.

The receiving client can be either synchronous or asynchronous in nature. Receiving from the router defines the receiving client's end of the communication. For *synchronous receipt*, the receiving client chooses the appropriate `receive` method to either wait for an incoming message or periodically poll the connection. For *asynchronous receipt*, the receiving client sets up a `MessageListener` object. Incoming messages are automatically delivered by the router calling the `onMessage` method of the `MessageListener` class.

In all cases, the router is responsible for ensuring delivery according to the terms set by the sending client. These terms include the delivery guarantee and how long the message should live if undeliverable. They also tell the router how to handle a message whose delivery has been unsuccessful thus far.

Administered Objects

Administered objects are provider-specific implementations of the JMS `Destination` and `ConnectionFactory` interfaces. These objects are created by the administrator of the messaging system for use by the client applications. The objects generally

contain provider-specific configuration information. Abstracting provider-specific information in this way has two advantages: it enables the client to ignore (remain independent of) the details of a specific provider's configuration, and eases the organization and administration from a common management console.

Administrators create the objects in a namespace where they can be looked up by client applications through the use of the Java Naming and Directory Interface™ (JNDI). Although it is technically possible to directly instantiate such objects from the provider's implementation classes, doing so can result in the loss of portability from one JMS provider to another.

There are two administered object types defined by the Java Message Service API—`ConnectionFactory` and `Destination`. The administered objects contain configuration parameters for connecting to the provider's messaging system. From the `ConnectionFactory` and `Destination` interfaces, the Java Message Queue software derives and implements the following four administered objects:

- `QueueConnectionFactory`
- `TopicConnectionFactory`
- `Topic`
- `Queue`

The Java Message Queue product provides the `jmqconfig` utility to aid in configuring administered objects. For more information on this utility and its use, see the Java Message Queue *Deployment Guide*.

Messages

In the Java Message Queue product, data is exchanged in the form of messages. All standard messages types defined by the JMS specification are supported.

Structure

As defined in the JMS specification, a message is composed of a header, properties (which can be thought of as an extension of the header), and a body.

A header is required—-its fields contain values used for routing and identification purposes. Properties are optional—they provide values upon which clients can filter messages. A body is also optional—-it contains the actual data to be exchanged.

Header Fields

Some header field values are set by the send method, others by the client, and others by the JMS provider. The following table lists the header fields defined (and required) by JMS, as well as the entity responsible for setting them.

TABLE 2-1 JMS-defined Message Header

Header Field	Set By	Default
`JMSCorrelationID`	Client.	
`JMSDeliveryMode`	Send method	Persistent
`JMSDestination`	Send method	
`JMSExpiration`	Send method	timeToLive is 0 (no expiry)
`JMSMessageID`	Send method	
`JMSPriority`	Send method	4 (normal)
`JMSRedelivered`	Provider	
`JMSReplyTo`	Client	
`JMSTimestamp`	Send method	
`JMSType`	Client	

Set By Client means ”set by the JMS client application using JMS APIs“.
Set by Send method means ”set by the JMS provider during the execution of the send method“.

Properties

When a piece of data is sent between two processes, other information besides the 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 consist of *property name: property value* pairs.

Having registered an interest in a particular queue or topic, clients can fine-tune their selection by specifying certain property values as selection criteria. For instance, a client might use properties to 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.

Transactions

A session is a sequence of messages sent and received on a connection between a client and the messaging system. When a session is created, it is created as either nontransacted (default) or transacted. A *transacted* session ensures that a group of messages is sent and received on an all-or-none basis. A *nontransacted* session means that messages are sent and received individually.

An example of the use of a transacted session would be online shopping. The customer opens an order (beginning the transaction). Each item selection the customer makes is a message to add an item to the order. The customer closes the order (ending the transaction). If the sender commits the transaction, all messages in the group are sent. If the sender *rolls back* the transaction, none of the items are ordered.

Java Message Queue Implementation

The messaging middleware concepts discussed in this chapter provide insight into the design and functionality of the Java Message Queue product. The Java Message Queue product is comprised of the following functional components:

- JMQ Router, `irouter`, which is responsible for routing messages between clients written to the JMS specification. The router supports the use of both Queues (point-to-point) and Topic (publish-and-subscribe) message distribution models. The architecture of a Java Message Queue router network is that of a virtual fully-connected network with any given client application requiring only a connect to a single router, not to every potential peer.
- JMQ Administration Console, `jmqadmin`, whose primary function is to enable the centralized management of persistent messages. This utility also serves as an interface to the simple, unencrypted client authentication server. Persistent messages provide the reliability required in enterprise scenarios in the face of hardware and system failures.
- JMQ Configuration Utility, `jmqconfig`, which simplifies the configuration of administered objects such as Topics and Queues for use with JNDI—allowing for the development of provider-independent JMS client applications.
- JMQ Command Interface, `ircmd`, for querying and performing administrative tasks on the router network.
- JMQ Monitor, `irmon`, to display debugging information and network output.
- Java class libraries for development of JMS applications and interaction with the Java Message Queue message system. These libraries can be redistributed with your Java Message Queue application.

For more information on the router and utilities, see the *Deployment Guide*.

Chapter 3:

Basic Steps

This document provides a quick introduction to setting up and running the Java™ Message Queue product after installation. For each topic discussed, pointers are provided to more extensive information located elsewhere in the documentation set.

This document is divided into the following sections:

- "Setting Up Your Environment" on page 13 discusses the default environment settings.
- "Configuring and Testing the Router" on page 14 provides instructions for starting the message router and verifying its correct operation.
- "Developing Client Applications" on page 16 highlights the Java Message Service API which the Java Message Queue product implements.
- "Running Client Applications" on page 22 details the steps for connecting a client application into the messaging system.

Setting Up Your Environment

The Java Message Queue software uses the following environment variables:

- `JMQ_HOME` This is the directory of the installed Java Message Queue product. This environment variable is used by the router and utilities as well as client applications. The location of necessary `.jar` files is given relative to `JMQ_HOME`.

  On Windows NT, the default is:
  C:\>: `Program Files\JavaMessageQueue1.1\`

  In Solaris environments, the default is:
  `/opt/SUNWjmq/`

- `JAVA_HOME` This is the location of your full Java Development Kit (JDK). Note that the JDK is a prerequisite and is not installed as part of the Java Message Queue product. This environment variable is used by client applications.

  On Windows NT, `JAVA_HOME` must be set—there is no default for this variable.

  In Solaris environments, if `JAVA_HOME` is not set by the user, it defaults to `/usr/java1.1`.

Configuring and Testing the Router

In the default configuration, the client application and single router are on the same machine. The following sections describe the simplest start method for each platform, assuming this configuration.

Starting in the Windows Environment

To start the router manually from the Windows Start menu:

1. From the Start menu, choose Programs.
2. From the listing of programs, choose Java Message Queue 1.1 and then Router.

   An MS-DOS command window opens. It displays copyright information, the number of connections available (reflects license agreement), and a startup message.

   You can minimize, but not close, the command window. If you close the window, the router process will terminate. Shut down the router before logging off the system. If you do not, the router attempts to shut down but might leave some processes running. You must end these orphan processes using the Windows NT Task Manager.

To start the router manually from the Windows Service Manager (applies only to routers installed as a Windows Service):

1. From the Start menu, choose Settings and then Control Panel.
2. On the Control Panel, open Services.
3. On the Services panel, choose the JMQ Router and then click Start.

   The Status column now shows the Router as Started. Closing the Services panel does not affect the running router.

For a discussion of other ways to start and stop the router, and a listing of options that can be passed, see the Java Message Queue *Deployment Guide*.

Testing the Router in the Windows Environment

After you've started the router, one of the simplest ways to check your setup is to run a small client application. The Java Message Queue product provides example client applications that are suitable for this purpose. By default, they are located in:

C:\>: `Program Files\JavaMessageQueue1.1\examples\`

Each of the examples is in its own subdirectory. Source files, compiled files, and a a README file are provided. Choose an example (such as `selectors` or `durability`) and follow the instructions in its README.

Administrators might prefer a more straightforward test using the router command environment rather than examples. To do this:

1. To start the command environment, at a command prompt, type:

   `ircmd`

2. To connect to the router, type:

   `open` *hostname*

   The system responds either with a ”Connected“ message (the router is set up correctly) or with an ”Unable to Connect“ message. For help with router problems, see the Java Message Queue *Deployment Guide*.

Starting in the Solaris Environment

To start the router:

1. Change to the executables directory. From a terminal window, type:

   `cd /opt/SUNWjmq/bin/`

2. Start the router by typing:

   `irouter &`

For a discussion of alternative ways to start the router, and a listing of options that can be passed, see the Java Message Queue *Deployment Guide*.

Testing the router in the Solaris Environment

One of the simplest ways to check the router setup is to run a small client application. The Java Message Queue product provides example client applications that are suitable for this purpose. By default, they are located in:

`/opt/SUNWjmq/examples/`

Each of the examples is in its own subdirectory. Source files, compiled files, and a a README file are provided. Choose an example (such as `selectors` or `durability`) and follow the instructions contained in the example’s README.

Administrators might prefer a more straightforward test using the router command shell rather than examples.

1. To start the command environment, in a terminal window, type:

   `ircmd`

2. To connect to the router, type:

   open *hostname*

   The system responds either with a ”Connected“ message (the router set up correctly) or with an ”Unable to Connect“ message. For help with router problems, see the Java Message Queue *Deployment Guide*.

Developing Client Applications

The Java Message Queue product implements both messaging models defined in the Java Message Service (JMS) specification: point-to-point (queue) and publish-subscribe (topic).

- Point-to-point.

  Embodies the idea of a single receiver per message. Each sender (there can be many) addresses a message to the queue that holds the messages for the single intended receiver. The receiver then extracts the message from the receiver's designated queue.

- Publish-subscribe.

  Embodies the idea of many (one or more) receivers per message. Each publisher (there can be many) publishes the message to a topic in the content hierarchy. One or more subscribers who have registered an interest in the topic can retrieve a copy of the message.

The Java Message Service specification defines the model and interfaces that clients use to produce and consume messages in both models.

The Java Message Queue product implements both the point-to-point and publish-subscribe models. As a result, you can use both models within the same client code. FIGURE 3-1 outlines the steps for writing each type of client. As you can see, the basic approach to writing a client program is almost identical in the two models.

The following steps show what is involved in writing a client that uses the publish-subscribe model. The point-to-point explanation would be similar. (Be sure to examine the source code of the Java Message Queue examples to see these steps in working code.) Additional information on the use of JNDI for storing and retrieving administered objects, such as topics and queues, can be found in the QuickStart Appendix, “JNDI and Java Message Queue” and the Deployment Guide Chapter ”Configuring JMQ Administered Objects“.

1. Import the necessary classes or packages.

   For creating a Java Message Service client:

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

For connecting to the Java Message Queue messaging system:

```
import com.sun.messaging.TopicConnectionFactory;
```

Typically, the administrator instantiates and stores administered objects using the `jmqconfig` utility.

2. Get a `TopicConnectionFactory`.

The JMS specification provides a `TopicConnectionFactory` interface for connecting to a JMS provider's messaging system. It is an interface because it is implemented in a provider-specific way by the messaging system vendor.

- Provider-independent approach. The administrator stores a `TopicConnectionFactory` object in JNDI using some agreed-upon name (here, the object is named `MyTopicConnectionFactory`). The Java Message Queue administrator must specify the name so that the client can lookup object using that name in the JNDI namespace.

  (To see this approach in working code, see the `admin-jndi` example located in the *$JMQ_HOME*`/examples/java/admin-jndi/` directory—where *$JMQ_HOME* is the directory of the installed Java Message Queue product.)

  To retrieve a stored object:

a. Set up an environment.
JNDI requires environment settings to tell it what service provider it should access and how. The necessary environment variables are described in the JNDI specification. If the service provider and initial context you use require authentication for the lookup operation, see "Handling JNDI Service Provider Authentication" on page 19 for the additional environment settings required and the values supported by Java Message Queue.

Setting up an environment can be as simple as storing the environment settings in a properties (flat text) file or in a hashtable.

b. Create an initial context.
Use the environment you set up in the previous step (for example, `env`) to create an initial context. A context is a name-object binding and the initial context provides a root for those naming operations.

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

c. Finally, locate the object by performing a lookup operation on the (agreed upon) name under which the object was stored.

```
TopicConnectionFactory factory=(TopicConnectionFactory)
    ctx.lookup("MyTopicConnectionFactory");
```

- Provider-dependent approach. Clients can instantiate the class directly. However, doing so will impact the portability of the client application to the messaging system of another JMS-provider.

```
TopicConnectionFactory myjmqTopicConnectionFactory;

myjmqTopicConnectionFactory = new

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

3. Get, or create, a message topic.

   - Provider-independent approach. As was the case for the `TopicConnectionFactory` above, `Topic` is an interface implemented by the JMS provider. The messaging system administrator creates the object and stores it in the JNDI namespace with the name expected by the client. The client is then able to look the object up. For example, using the same context (`ctx`) created to retrieve the previous object:

     ```
     Topic topicA;
     topicA=(Topic) ctx.lookup("MyTopic");
     ```

   - Provider-dependent approach. Clients can instantiate the class directly if portability between JMS providers is not an issue.

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

4. Use the `TopicConnectionFactory` interface to create a `TopicConnection` object. For example:

   ```
   TopicConnection connection;
   connection=factory.createTopicConnection();
   ```

   To require client authentication, use the following method signature instead:

   ```
   TopicConnection connection;
   connection=factory.createTopicConnection(java.lang.String
   userName, java.lang.String password);
   ```

   For information on using client authentication in the Java Message Queue product, see the Deployment Guide's JMQ Administration Console chapter.

5. Use the `TopicConnection` object to create a `TopicSession` object, providing a Boolean value to indicate if the session is transacted and the mode of acknowledgment. For example:

   ```
   TopicSession session;

   session=connection.createTopicSession(false,

      Session.DUPS_OK_ACKNOWLEDGE);
   ```

6. Use the `TopicSession` object to create a `TopicPublisher` object, a `TopicSubscriber` object, or both. You must indicate the topic being subscribed to.

```
TopicSubscriber subscriber;
subscriber=session.createSubscriber(topicA);
```

Synchronous message delivery is the default. If you want to set up asynchronous message delivery, you must create a `MessageListener` object for the topic and register it with the subscriber. For an illustration of how this is done, see the `connections` example.

7. Once you are finished with the session, be sure to close it. This frees up system resources. To close the session, include a code block similar to the following:

```
try {
  session.close();
  connection.close();

catch (JMSException jmse)  {
  jmse.printStackTrace();
}
```

Strictly speaking, the code to explicitly close the session is not needed here because closing the connection also closes the session.

Ensure that `JMQ_HOME` and `JAVA_HOME` environment variables are set and that your classpath contains `JMQ_HOME/lib/jms.jar`, `JMQ_HOME/lib/jmq.jar`, and `JMQ_HOME/lib/jndi.jar`. (The `jndi.jar` file is needed even if you are not using JNDI directly.) Then compile as you would any Java program.

Handling JNDI Service Provider Authentication

An application may need to authenticate itself with the service provider before it is able to lookup and retrieve an object. This will be the case if the service provider disallows anonymous access to "read" objects within the context provided, or if an access control list for the object or context has been set up.

The client application has two ways to provide the information required by JNDI for service provider authentication. The authentication information can either be placed directly in the `jndi.properties` file or inserted in the "environment" used to create the initial context for the lookup operation.

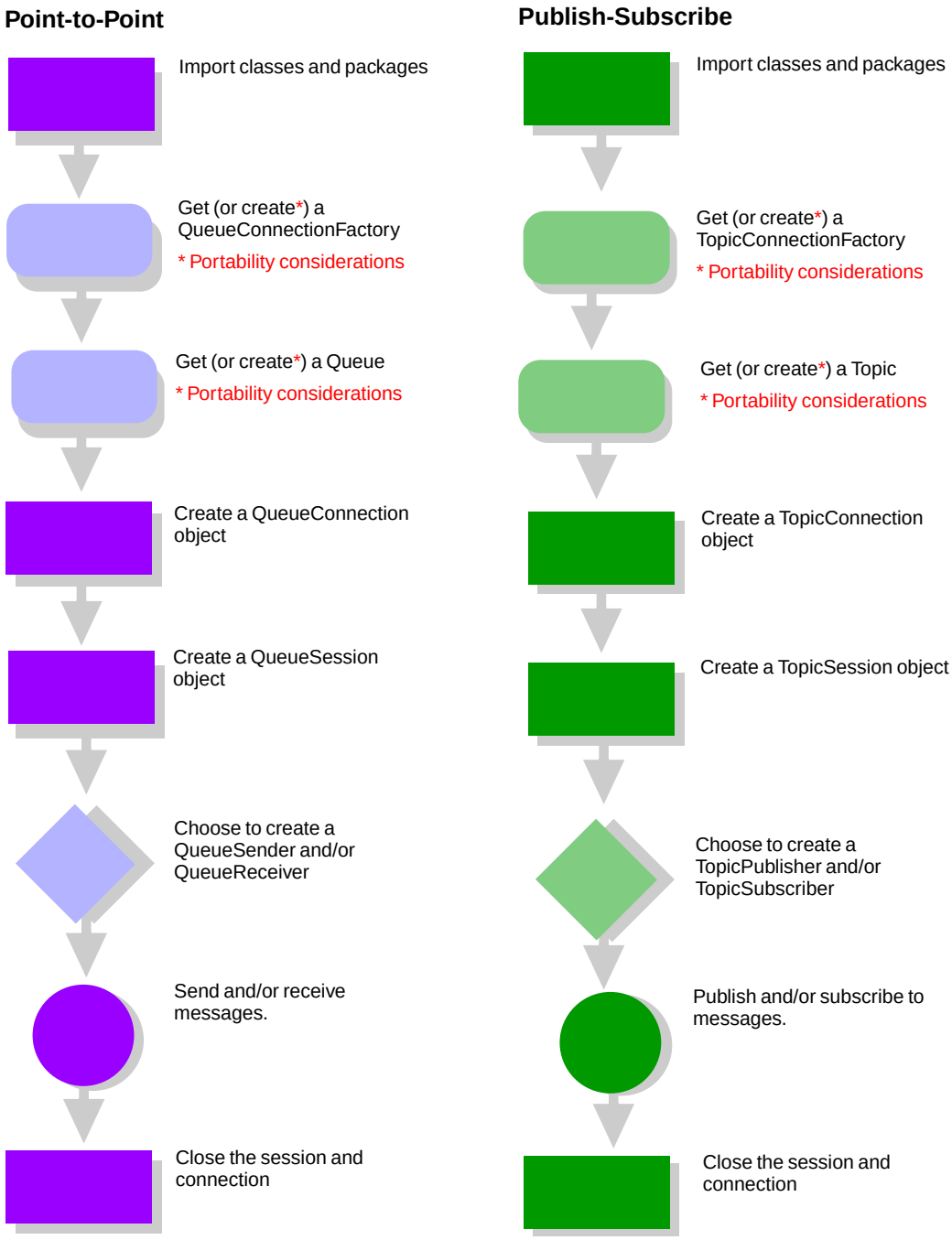
Using a Customized Environment to Create an Initial Context

In addition to the `INITIAL_CONTEXT_FACTORY` and the `PROVIDER_URL` properties, the client application will need to set three security-related properties in the environment with which the initial context is created. These security-related properties are: `SECURITY_AUTHENTICATION`, `SECURITY_PRINCIPAL`,

and SECURITY_CREDENTIALS. The values provided for the properties in the example code below need to be replaced with values appropriate to your client application.

```
Hashtable env = new Hashtable(10);

//Specify the initial context factory class and provider url
//to locate the service provider

env.put(Context.INITIAL_CONTEXT_FACTORY,
  "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL,
  "ldap://mydomain.com:389/ou=JMQACL,o=mydomain.com");


//Specify the authentication scheme, principal (username)
//and credentials (password) to authenticate with the
//service provider in the supplied initial context

env.put(Context.SECURITY_AUTHENTICATION, "simple");
env.put(Context.SECURITY_PRINCIPAL,
  "uid=foo,ou=JMQACL,o=mydomain.com");
env.put(Context.SECURITY_CREDENTIALS, "foo");
```

FIGURE 3-1 Basic Client Program Development



Running Client Applications

The follow instructions assume the default configuration of router and client on the same machine. For all other configurations, and for a listing of command options, see the Java Message Queue *Deployment Guide*.

1. Ensure your `JMQ_HOME` and `JAVA_HOME` environment variables are set and that `jmq.jar`, `jms.jar`, and `jndi.jar` are in your `CLASSPATH`—they are located in the `lib` directory of the installed Java Message Queue product. (Even if you are not using JNDI, `jndi.jar` must appear in your `CLASSPATH` because it is referenced through the `Destination` and `ConnectionFactory` classes.)

   If you are using JNDI, in addition to the above, you must also include `providerutil.jar` plus either `fscontext.jar` (if you are using the filesystem context) or `ldap.jar` (if you are using the LDAP context).

2. If the router is not already running, start it.
3. At the command prompt, change to the directory containing the application. (Alternatively, ensure the application directory is in your CLASSPATH.)
4. Start the Java runtime environment.

   `java` *applicationname*

Chapter 4:

Java 2 Security

When operating under the Java™ 2 environment, subsets of the Java™ 2 security architecture can be used to protect critical Java Message Queue system resources. This document focuses specifically on the policy file portion of Java™ 2 security.

The document is divided into the following sections:

- "Understanding Security Policies" on page 23 provides an introduction to the concept of security policies and an explanation of the relationship between Java 2 and Java Message Queue policy files.
- "Using the Default Policy Files" on page 25 describes the policy files provided with the product and their proper use.
- "Modifying a Policy File" on page 28 outlines the basic steps for editing a policy file using the Java 2 `policytool` utility.
- "Finding More Information" on page 29 lists some additional sources of information on the topics discussed in this document.

Note – The information is this chapter is relevant only if you are running either the Java Message Queue Administration utilities or your Java Message Queue client applications under a Java 2 environment. If you are running both the utilities and your applications under a JDK 1.1.x environment, you can safely ignore the information contained here—it has no impact and requires no action on your part.

Understanding Security Policies

Under the Java 2 security architecture, system resources (such as files or sockets) are protected by a *security manager*. When a piece of code attempts to access a system resource, the security manager allows or disallows the access in accordance with the *Security Policy* in effect. The Security Policy addresses the question of which system resources may be accessed, by which applications, and to what degree.

A Security Policy is defined by one or more *policy configuration files*. A policy configuration file contains a series of entries. Each entry details, for a specific or general entity, a system resource that the entity may access and the level of access

allowed. For example, a policy file entry might indicate that code in the `examples` directory is permitted to access a socket on port `9312` and that the access is allowed for connection and resolution only.

The Java 2 platform uses the following mechanism for identifying relevant policy files:

- A security properties file called `java.security` which it loads on system startup. The security properties file is located in:

  `$JAVA_HOME/jre/lib/security/java.security` (Solaris)
  `%JAVA_HOME%\jre\lib\security\java.security` (Windows)

  From the security properties file, it reads the name and location of the policy configuration files to be used. These can include default or user-provided policy files. Each property entry is of the form:

  `policy.url.`*n*`=`*URL*

  For example, typically the default policy files (one intended to be system-wide, the other user-specific) would appear in the `java.security` file as follows:

  ```
  policy.url.1=file:${java.home}/lib/security/java.policy
  policy.url.2=file:${user.home}/.java.policy
  ```

  where `${`*propertyName*`}` will be replaced at runtime with the actual value of the property.

- Policy files passed through the commandline when a Java application is invoked.

The Security Policy in effect at any given time, then, is the cumulative effect of permissions provided in each policy file listed in security properties file and in any additional policy file passed on the commandline. Because permissions are strictly positive in nature, you can grant a permission but not deny a permission. That is, granting only `read` access is not the same as disallowing `write` access. If another policy files provides `write` access for the same resource by the same agent, the additive effect allows both `read` and `write` access.

The Java Message Queue product requires access to certain system resources. When running under the Java 2 environment and using the Security Manager, a policy must be provided that can grant access to those required resources. Different applications need different kinds of policy permissions. Three policy files are provided with the Java Message Queue product:

- `jmqadmin.policy` - used only by the `jmqadmin` and `jmqconfig` utilities.
- `jmqclient.policy` - a generic client policy file which can be modified for your purpose.
- `jmqexamples.policy` - a policy file required by the examples.

The `jmqadmin.policy` file is automatically loaded as part of the startup script for the Java Message Queue administration and configuration utilities. However, policy files for client applications *must* be passed through the commandline (either directly or through a startup script) when the application is started. For example, the `jmqexamples.policy` file is automatically loaded as part of the `runnit` script for each example client application.

In Java Message Queue only one policy file is invoked per application. For example, the `jmqadmin` and `jmqconfig` applications are each started with the `jmqadmin.policy` file. Similarly, each client example is started with the `jmqexamples.policy` file.

What may not be obvious from the foregoing discussion is the fact that the Java 2 environment provides the base level of access permissions on which the Java Message Queue permissions are then layered.

Using the Default Policy Files

As mentioned earlier, the Java Message Queue product provides both an administration (`jmqadmin.policy`)and a client policy file (`jmqclient.policy`)for use when running in a Java 2 environment. (It also provides `jmqexamples.policy`, a specific example of a client policy file.) Java Message Queue administrators will be interested in both administration and client policy files because these files control access to critical Java Message Queue system resources. Application developers will be interested only in client policy files which grant access for their client applications to Java Message Queue system resources.

The policy files are located in:

$JMQ_HOME`/security/jmqadmin.policy`
$JMQ_HOME`/security/jmqclient.policy`
$JMQ_HOME`/security/jmqexamples.policy`

where $*JMQ_HOME* is the location of the installed Java Message Queue product.

The Java Message Queue policy files follow the same format as the default Java 2 policy files. A policy file contains one or more grant entries of the following form:

```
grant [codeBase "code_base"]{
    permission permission_file ["target"] ["permission_type"];
};
```

For example, the following entry allows the code located in the "*$JMQ_HOME*/mydir" directory to connect to the router using subnet 0 (that is, default port 9312):

```
grant codeBase "file:${jmq.home}/mydir/" {
    permission java.net.SocketPermission \
    "localhost:9312", "connect, resolve";
};
```

The next two sections provide additional detail on each type of policy file: its purpose, instructions on its invocation, and guidance regarding modification.

jmqadmin.policy

The `jmqadmin.policy` *file is required if you want to run the Java Message Queue administration utility (*`jmqadmin`*) in a Java 2 environment.* The file provides the Administration Tool with the necessary permissions to access critical system resources in the course of its operation. For example, the Administration Tool requires permissions related to socket connections (connect, accept, resolve, and listen) in order to start the Security Server, connect to the router, or create Queues and Topics.

As mentioned earlier, when starting the Administration Tool, you don't have to worry about explicitly activating the security manager or passing the `jmqadmin.policy` file. The startup script for the utility handles that for you.

When you type:

```
jmqadmin
```

The script sets `JAVA_HOME`, `JMQ_ADMIN_HM`, and `JMQ_HOME` then calls:

```
$JAVA_HOME/bin/java -Djmq.home=$JMQ_HOME\
    -Djava.security.manager\
    -Djava.security.policy=$JMQ_HOME/security/jmqadmin.policy\
    com.sun.messaging.ia.admin.iajmsAdmin
```

Caution – In most cases, the `jmqadmin.policy` file should not require changes. However, if you do need to edit it, review your changes carefully because incorrect changes can adversely impact the behavior of the system.

---

jmqclient.policy

The decision whether or not to run your client application under a Java 2 environment is independent of the choice made by the Administrator regarding the Administration Console. The Administration Console can be run under a Java 1.1.x environment and the client application (including examples) run under Java 2 environment, or vice versa, with no adverse effect.

Further, even if you do decide to run your client application under a Java 2 environment, you can still decide not to use the security manager. If you do not use the security manager, the `jmqclient.policy` file is not needed.

However, if you do decide to run your application under a Java 2 environment and you do decide to use the security manager, you must call the security manager explicitly using the `java.security.manager` property and pass the policy file using the `java.security.policy` property. For example:

```
$JAVA_HOME/bin/java -Djava.security.manager \
    -Djava.security.policy=$JMQ_HOME/security/jmqclient.policy
    MyApp
```

(This example utilizes the built-in default security manager provided with the Java 2 SDK. For more details on setting security managers, see the Java 2 Security Documentation. A listing of security documentation and resources is provided in "Finding More Information" on page 29.)

The Java Message Queue 1.1 product is designed to take only one policy file per client application. The default client policy file is called `jmqclient.policy` file but you can call yours anything you like because it is passed by name on the command line. The `jmqclient.policy` currently contains a generic set of the permissions which might be useful when running client applications.

Here are some examples of the types of entries that might be made for client policy files:

- Permission to connect to the router.
- Permission to connect to the Client Authentication Server.
- Permission to read a system property, such as `user.name`
- Permission to read from or write to a log file.

The `jmqexamples.policy` file contains the specific set of permissions needed by the example applications shipped with the product. Take a look at the file for an idea of typical permissions.

Modifying a Policy File

As mentioned earlier, a policy configuration file contains a series of entries—each entry details for a given entity, the resources it is allowed to access and the level of that access. You can modify a Java 2 policy file using either your favorite text editor or the `policytool` graphical utility provided with the Java 2 Standard Development Kit (SDK). Using the `policytool` utility is recommended because it does not require that you know or remember the necessary format, thus making the editing process simpler and more accurate.

The basic steps for modifying a policy file using the `policytool` utility are outlined below.

1. Start the policy editing tool.

   The policytool is located at *$JAVA_HOME*`/bin/policytool`, where *$JAVA_HOME* is the location where you installed Java 2 SDK.

2. Open the policy file.

   From the policytool's menu, select File->Open. Use the file browser to locate and load a policy file from the *$JMQ_HOME*`/security` directory, where *$JMQ_HOME* is the location of the installed Java Message Queue product.

   When you load the policy file, the entries of the file will be visible in the policytool screen.

3. Modify the policy file.

   As you modify the policy file, the following definitions should be kept in mind:

   - *Codebase* refers to the .class or .jar file, or directory of such files, whose access to system resources you wish to control.
   - *Permission* refers to the set of permissions available for a particular type of system resource (java.io.FilePermission contains the set of permissions for file objects, for example).
   - *Targe*t refers to the specific protected system resource (a log file, for example)
   - *Action* refers to the specific actions you are allowing the named codebase to perform on the specified target (write access to a target file, for example).

   To create a new permission, click Add Permission and fill in the resulting New Permission dialogue box.

   Similarly, to delete or modify an existing entry, click the appropriate button and fill in the resulting Permission dialog box.

4. When finished, save the policy file and exit.

Finding More Information

This document provides a starting point for understanding Java 2 security as it relates to the Java Message Queue product. If you would like more information on the Java 2 security architecture and tools, check the following sources:

- Policy files.

  http://java.sun.com/products/jdk/1.2/docs/guide/security/PolicyFiles.html

- Policy file tool.

  http://java.sun.com/products/jdk/1.2/docs/tooldocs/solaris/policytool.html

  http://java.sun.com/products/jdk/1.2/docs/tooldocs/win32/policytool.html

- Java 2 security documentation.

  http://java.sun.com/products/jdk/1.2/docs/guide/security/index.html

Chapter 5:

Examples

The Java Message Queue distribution includes examples that demonstrate features of the product. This document briefly describes those examples:

- "Administration-JNDI" demonstrates retrieving administered objects stored in a namespace.
- "Connections" shows establishing connections from authenticated clients.
- "Durability" deals with handling messages for inactive clients through durable subscriptions.
- "Persistence" presents storing messages across router restarts.
- "Producer-Consumer" explains enabling properties in the Java Message Queue product.
- "Selectors" illustrates filtering messages on the basis of properties.
- "Simple Chat" demonstates publishing messages to a Topic and subscribing to that Topic's messages.
- "Transactions" focuses on grouping messages into a single unit that can be committed or rolled back as a whole.

For more detailed information related to building and executing a particular example, see the `README` file in the directory for that example. The examples can be found in the `examples/java` directory of the installed Java Message Queue product.

Administration-JNDI

The `admin-jndi` example is composed of three parts. The first one is a script that shows the use of the `jmqconfig` utility. The remaining parts show the use of JNDI within a Java Message Queue publish-and-subscribe application to retrieve the objects configured with `jmqconfig`.

runjmqconfig

This script calls the `jmqconfig` utility to create and store two JMS configured objects: a Topic (`myTopicBN`) and TopicConnectionFactory (`myTCFBN`).

Pub.java

This example application uses JNDI and the file system service provider to retrieve the two Topic and TopicConnectionFactory administered objects. Once the instances of Topic and TopicConnectionFactory objects are obtained, they are used to publish messages.

Sub.java

This example application uses JNDI and the file system service provider to retrieve the two Topic and TopicConnectionFactory administered objects.

Once the instances of Topic and TopicConnectionFactory objects are obtained, they are used to subscribe to messages.

Connections

The Connections example shows how to establish a connection then send or receive messages using Topics and Queues.

he Connections example first creates a Topic and a `TopicConnection` object. (A `TopicConnection` is an active connection to a JMS publish and subscribe provider.) The client uses the `TopicConnection` to create one or more `TopicSession` objects for publishing and subscribing to messages. Finally, theTopic connection and session are closed.

In a parallel fashion, a Queue is created, messages are sent and received, and then the Queue connection and session are closed.

The Connections example also demonstrates the use of the Java Message Queue Administration Console and the Client Authentication Server. The example creates a connection for the user name, `user`, having a password `blah`. The Client Authentication Server validates the user name before allowing the connection. (You must use the Java Message Queue Administration Console to start the Client Authentication Server on the appropriate port and add the username. This is described in the `README` file in the `connections` directory.)

For more information on client authentication and the Administration Console, see the Deployment Guide “JMQ Administration Console“ chapter.

Durability

Clients publish messages to and subscribe to messages from a topic node, which is essentially a message broker. Subscribers to Topics or Queues can be made durable; that is, Java Message Queue can “remember” the existence of subscribers while the subscribers are inactive.

The Durability example shows subscriber durability in the context of a Java Message Queue application that:

- Starts a router.
- Creates topics, publishers, subscribers, and messages.
- Publishes the messages.
- Uses a durable subscriber to subscribe to the messages.
- Disconnects from the router.
- Reconnects.
- Has the subscriber continue to receive messages after reconnection.

Persistence

This example demonstrates the persistence of outgoing JMS messages.

By default, the Java Message Queue router stores JMS messages on disk to maintain them between router invocations until the router can deliver the messages to subscribers, or until the messages’ Time To Live, as defined by the Java Message Service Specification, expires. In this example, the example prompts the user to stop and restart the router, Java Message Queue still reliably delivers the messages to the subscriber.

Producer-Consumer

The Producer-Consumer example demonstrates Java Message Queue support for message properties and the role of properties in message selection. It shows Java Message Queue enabling properties and using them in message selection. The example uses a point-to-point (queue) distribution model, although properties can also be used with the publish-subscribe (topic) model.

Note – Java Message Queue supports JMS message properties defined in the Java Message Service Specification. However, the specification does not require that all vendors support JMS properties.

The point-to-point model consists of the following:

- Senders that produce messages.
- Receivers that consume messages.
- Queues that act as mailboxes for receivers.
- Listeners that notify a receiver of messages arriving at the receiver's queue.

In this example, the deployer of the receiver creates a listener by implementing the `MessageListener` interface. The deployer of the receiver can stipulate whether the receiver is notified of all incoming messages to its queue or is notified only of selected messages (messages that meet certain criteria).

The listener notifies the receiver when a message of interest to the receiver (as determined by the selection criteria) is on the queue. The receiver's `onMessage(Message)` method retrieves the message.

The example creates a single sender (called `QProducer`) and a single receiver (called `QConsumer`) in two separate threads. The selection criteria are values found either in the header or in the properties transmitted with the message body.

Selectors

A Java Message Queue application might be interested in only some of the messages that are routed to it based on its subscriptions. The application uses a `TopicSubscriber` to receive messages that have been published to a Topic. The Selectors example shows how property selectors can be used with TopicSubscriber objects to filter the messages an application receives. In this way, a Java Message Queue application can specify to receive only those messages in which it is interested.

Simple Chat

This example illustrates how you can use a JMS application to create a simple chat application.

Specifically, Simplechat:

- Uses JMS topics. Each instance of the chat application is both a topic publisher and subscriber.
- Uses the `javax.jms.MessageListener` interface (the `onMessage(Message)` method).
- Uses `ObjectMessage` to transmit chat messages. The application object that is passed to JMS's `ObjectMessage` and is used to broadcast messages in the chat is `SimpleChatMessage`. `SimpleChatMessage` is defined in this example and is not a JMS class.

- Uses the `SimpleChatMessage` class to broadcast messages.

Most of the code in this example supplies the graphical user interface; the JMS-related code is relatively simple and is similar to that used in the other examples.

Transactions

The Transactions example demonstrates the use of JMS transactions within the Java Message Queue framework. Transactions combine work that spans a session, and group sets of produced and consumed messages into an atomic unit of work.

Transactions complete when a session commit or rollback occurs:

- A commit indicates that the unit of input from the transaction is acknowledged within the Java Message Queue framework and its corresponding output is sent.
- A rollback indicates that the produced messages are destroyed within the Java Message Queue framework and JMS recovers the consumed messages.

In this example, one JMS transaction is committed and another is rolled back. The example uses a `MessageListener` to receive all messages.

Appendix A:

JNDI and Java Message Queue

This document discusses the use of the Java Naming and Directory Interface™ (JNDI) for the storage and retrieval of Java Message Queue administered objects.

The document is divided into the following sections:

- "What is JNDI?" discusses access to objects in a location-independent fashion.
- "Binding and Contexts" describes the nature of the name-object mapping.
- "Lookup Operations" explains how objects are looked up by name.
- "Directories and Service Providers" mentions the use of directory services such as LDAP.

What is JNDI?

The Java Naming and Directory Interface™ (JNDI) provides names and contexts for Java Message Queue administered objects you store with a Directory Service. Objects can be stored either on an Lightweight Directory Access Protocol (LDAP) Directory Service or within a Solaris or Windows filesystem. The two Java Message Queue administered objects you can store are:

- `Destination`
- `ConnectionFactory`

JNDI provides the means for you to associate an administered object with a name when you store the object. Once stored on the directory service, your application code can retrieve the administered object through its name and then use the object.

The JNDI tutorial, available at `http://java.sun.com/products/jndi`, contains complete information on JNDI. This section contains some capsule information designed to enable you to understand your use of JNDI with the Java Message Queue product.

The Deployment Guide specifies the use of the `jmqconfig` utility to manage and configure Java Message Queue administered objects.

Note – The Sun Microsystem implementation of the Filesystem service provider for JNDI that is shipped with Java Message Queue 1.1 is at Beta level. It is not supported for deployment.

Binding and Contexts

When you associate a name with an object, you provide it with a binding. For example, a file name is bound to a file in a filesystem, and an LDAP name is bound to an LDAP entry. Objects that are not stored directly in a given service are stored as JNDI References. When you use a file system service provider, Referenceable objects are bound as a set of properties in a file called `.bindings`. The parent directory containing these `.bindings` is the parent context of these bindings.

A context is a set of name-to-object bindings, and each context has an associated naming convention. In JNDI, all naming and directory operations are performed relative to a context. Because of this, JNDI defines an initial context, which provides a starting point for naming and directory operations. Once you have an initial context, you can use it to look up other contexts and objects.

A naming system is a connected set of contexts having the same type of naming convention and providing a common set of operations. The naming system provides a naming service and the set of names in a naming system is a namespace.

Lookup Operations

A context provides a lookup (resolution) operation that returns a Java Message Queue administered object. The lookup operation might additionally provide other operations, such as binding or unbinding objects.

Within your application code, you can supply to the JNDI `lookup()` method the name of the object you want to look up. The method returns the instance of the object bound to that name.

Directories and Service Providers

A directory object represents an object in a computing environment and can have attributes associated with it. Java applications use the directory to store and retrieve Java objects.

Service providers are an implementation of a context or initial context that can be plugged in dynamically to the JNDI architecture for use by the JNDI client—in the case of Java Message Queue, your Java Message Queue application. Service providers thus provide the means by which naming and directory services are integrated into the JNDI framework.

Java Message Queue supplies Sun Microsystems' Lightweight Directory Access Protocol (LDAP) Service Provider and File System Service Provider for JNDI. The LDAP Service provider is supported for deployment. However, the File System Service Provider for JNDI is at Beta level and is therefore *not* supported for deployment.

To use other service providers, simply download the appropriate service provider jar files and add them to your CLASSPATH. You can then use the `jmqconfig` utility to configure and store Java Message Queue Administered Objects using your command line.

Glossary

asynchronous connection An exchange of information between clients in which the client generating the event can continue immediately to its next communication without waiting for the other client to receive, process, and respond to the message.

connection A channel through which one JMS client or router sends data to another JMS client or router.

connection topology The specification of how the routers in a network are connected to one another.

connection type The type of channel through which a router communicates to another process. In Java Message Queue, there are three types of connections: local/private, remote/private, and remote/public. Local/private means that the connection is internal to the router, and the data is internal to the application. Remote/private means that the connection is from another application, and the data is internal to the application. Remote/public means that the connection is from another application, and the data is available to a non-router application.

distributed system A system in which multiple processes—often residing on several different computers—exchange various types of information.

Domain Name Service (DNS) A distributed service used to look up an IP address, given a host name.

durable subscription An subscription that exists across client shutdowns and restarts. While a subscriber is inactive, all messages that would have been delivered to the subscriber are stored. The subscriber receives the objects when it registers the subscription again.

firewall A system designed to prevent unauthorized access to or from a private network. All network packets entering or leaving the intranet can pass through a firewall, which examines each packet and blocks those that do not meet the specified security criteria.

hosts file A file used to specify host names and their addresses.

`ircmd` A Java Message Queue utility that enables you to perform simple monitoring and administrative tasks on the router. (Also called the JMQ command interface.)

`irmon` A Java Message Queue utility that you can use to monitor and display debugging and network flow information.(Also called the JMQ monitor.)

JMQ Administration Console (jmqadmin) The Java Message Queue utility used to administer Java Message Queue client application authentication and Queues on a Java Message Queue router.

JMQ administered objects The Java Message Queue implementations of JMS Administered Objects that encapsulate Java Message Queue-specific behavior and enable JMS provider-independent client application development and deployment. Java Message Queue client applications typically lookup a Java Message Queue Administered Object using the Java Naming and Directory Interface (JNDI) API from a Service Provider.

JMQ Configuration Tool (jmqconfig) The Java Message Queue utility that configures and manages Java Message Queue administered objects.

JMQ message service The sub-system formed by a network of one or more connected routers that provide the messaging function between Java Message Queue client applications.

jmqsvcadmin The Java Message Queue utility (for Windows NT and 2000 only) used to install or remove the JMQ Router as a Windows Service.

Java Message Queue system The system comprised of one or more Java Message Queue client applications together with the Java Message Queue Message Service, in which client applications perform distributed tasks using JMS messages.

JMQ utilities The Java Message Queue executables named `ircmd`, `irmon`, `jmqadmin`, `jmqconfig`, and `jmqsvcadmin`.

Java Message Queue client application A program written in the Java programming language that uses the Java Message Service (JMS) API and runs in the Java Message Queue System. These applications are also sometimes referred to as JMS client applications or JMS clients.

JMS provider An entity that implements JMS for a messaging product.

Java Naming and Directory Interface (JNDI) An API that provides naming and directory functionality to applications written in the Java programming language. The JNDI is defined to be independent of any specific directory service implementation.

medium A physical transport. Java Message Queue has the built-in capability of using TCP as a medium.

message A method of exchanging data between applications. A message consists of a header and a body. The header includes routing information. The body contains the application's defined content.

message selector See **selector**.

Message Time-To-Live A value that defines a message expiration time, which is the sum of the message's time-to-live and the time that it was sent.

messaging Communication between enterprise applications.

monitor application The command-line utility `irmon`, which shows debugging messages from routers running on hosts that are connected to the same router network and are using the `-dn` flag.

point-to-point communication model A communication model in which an application sends information to a specific destination that is processed by a single client. See also **publish-and-subscribe communication model**.

property A characteristic of a message that a user can set. Developers of Java Message Queue applications can set such properties as the process that created the message, the time it was created, and how to handle its delivery.

publish-and-subscribe communication model A communication model in which an application publishes information needed by one or more client processes that subscribes to the information. See also **point-to-point communication model**.

remote router A router on a different machine than the one on which you are running. When a program needs to use a router on a remote machine, it must know the name of the host to which to connect. This can be specified using the -s option. See also **router**.

resource definition file A file that stores records of information including start-up defaults, connection configurations, and data logging.

resource index file An ASCII file named `res.ndx` that contains the names of all the installed binary resource files (`.rfb`) in the resource directory.

resources Data that pertains to a system or an application. Resources include persistent data files and messages.

`.rfb` file See **resource definition file**.

router A router forwards (or routes) messages between Java Message Queue client applications.The router maintains Queues and Topics, and a list of subscribers to those Topics.

Refers also to the executable named `irouter` responsible for implementing the router functionality.

router network A group of routers that are connected to one another. In a direct connection, two routers are linked directly to one another.

selector A set of conditions, specified by a message consumer, that a message must satisfy so that it can be delivered to that consumer.

subnet A communications channel (for example, a port) used by all routers and applications for all Java Message Queue communications. Routers and Java Message Queue applications must be on the same subnet to communicate with each other.

synchronous connection The exchange of information between clients in which one client must wait for a response to a specific event before sending another message.

transaction The act of grouping a set of produced messages and a set of consumed messages into an atomic unit and marking them as either acknowledged or not.

Index