Fuse Message Broker

Exploring JMS

Version 5.5
Febuary 2012

Exploring JMS

Version 5.5

Updated: 27 Mar 2014

Trademark Disclaimer

Third Party Acknowledgements

One or more products in the Red Hat JBoss Fuse release includes third party components covered by licenses that require that the following documentation notices be provided:

- JLine (http://jline.sourceforge.net) jline:jline:jar:1.0

  License: BSD (LICENSE.txt) - Copyright (c) 2002-2006, Marc Prud'hommeaux `<mwp1@cornell.edu>`

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JLine nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR

SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Stax2 API (http://woodstox.codehaus.org/StAX2) org.codehaus.woodstox:stax2-api:jar:3.1.1

  License: The BSD License (http://www.opensource.org/licenses/bsd-license.php)

  Copyright (c) <YEAR>, <OWNER> All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- jibx-run - JiBX runtime (http://www.jibx.org/main-reactor/jibx-run) org.jibx:jibx-run:bundle:1.2.3

  License: BSD (http://jibx.sourceforge.net/jibx-license.html) Copyright (c) 2003-2010, Dennis M. Sosnoski.

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JiBX nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON

ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- JavaAssist (http://www.jboss.org/javassist) org.jboss.javassist:com.springsource.javassist:jar:3.9.0.GA:compile

  License: MPL (http://www.mozilla.org/MPL/MPL-1.1.html)

- HAPI-OSGI-Base Module (http://hl7api.sourceforge.net/hapi-osgi-base/) ca.uhn.hapi:hapi-osgi-base:bundle:1.2

  License: Mozilla Public License 1.1 (http://www.mozilla.org/MPL/MPL-1.1.txt)

Table of Contents

List of Figures

Chapter 1. Introduction

JMS, and messaging in general, can be hard to grasp without concrete examples. This guide explores the basics of JMS using a combination of general knowledge and runnable samples.

Overview

This book is intended for developers that are unfamiliar with messaging and JMS. It describes the basic concepts involved in building messaging applications with JMS. It then walks you through running and deploying a number of basic application samples.

The included samples can be deconstructed to further explore the JMS APIs.

Using the book

The book is broken into three sections:

1. "Concepts in JMS Messaging" on page 11 describes the basic concepts used when developing JMS applications.

2. "Setting Up the Guided Tour" on page 29 through "Queue Test Loop Sample" on page 59 are samples that demonstrate the basic JMS concepts.

3. "Changing Parameters and Modifying Source Code" on page 61 describes different ways you can modify the samples to further explore JMS.

What you will see

In these samples, the standard input and standard output displayed in the console represents data flows to and from applications and Internet-enabled devices such as:

- **Application software** for accounting, auditing, reservations, online ordering, credit verification, medical records, and supply chains
- **Real-time devices** with embedded controls such as monitor cameras, cell phones, medical delivery systems, and climate control systems, and machinery
- **Distributed knowledge bases** such as collaborative designs, service histories, medical histories, and workflow monitors

What is demonstrated

The samples demonstrate the basic JMS features, as follows:

- publish and subscribe messaging—Messages are published to a destination and multiple consumers can subscribe to receive the messages.
- point-to-point messaging—Messages are published to a destination and a single consumer can receive the message.
- request and reply—The initiator of the transaction expects a reply to its message.
- test loop—This sample shows how quickly messages can be sent and received in a test loop.

Chapter 2. Concepts in JMS Messaging

This chapter provides an overview of basic JMS messaging concepts.

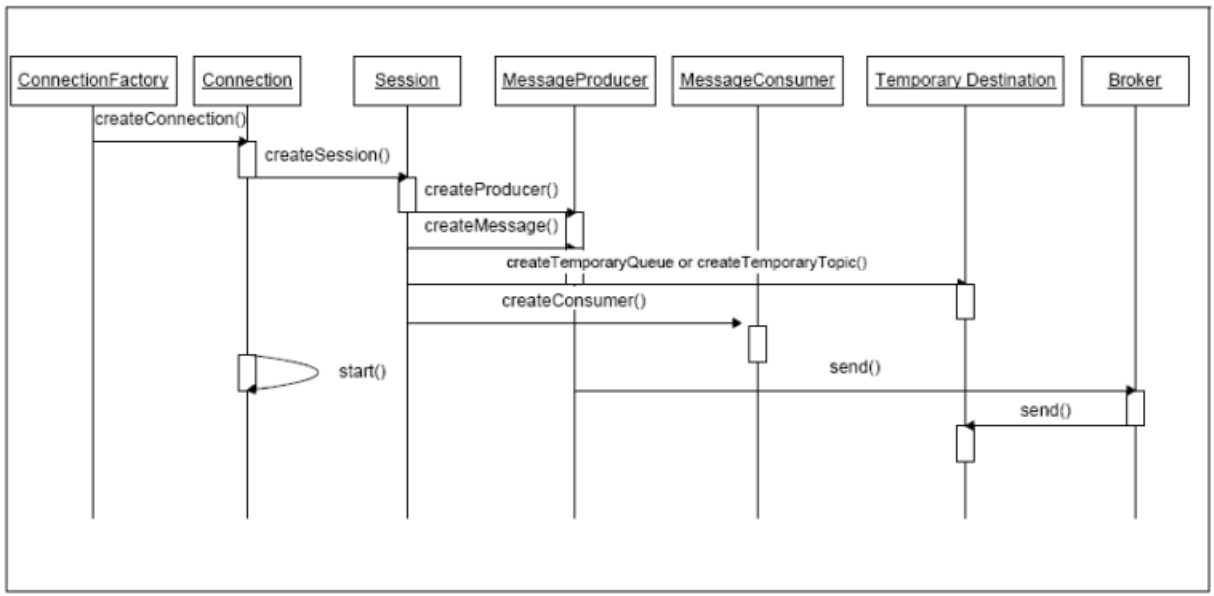
Overview

The *Java Message Service* (JMS) is message-oriented middleware that enables asynchronous, distributed messaging. The JMS 1.1 specification is available on the Sun Developers Network at http://java.sun.com/products/jms.

In JMS messaging, an application creates a connection, then establishes one or more sessions on the connection. Message producers send messages to destinations. Message consumers receive messages from destinations.

The flow of objects is shown in this sequence diagram:

Figure 2.1. JMS Sequence Diagram



As illustrated, a `ConnectionFactory` creates a `Connection` and then calls the `start` method. A `Session` is created in the context of a `Connection`. `MessageProducer` and `MessageConsumer` functions act on messages in the context of a `Session`.

Messaging Models

There are two messaging models that are used in JMS:

- **Publish/Subscribe (Pub/Sub)** — One-to-many communication. A producer publishes a message to a topic, where all consumers subscribing to the topic receive the message.
- **Point-to-point (PTP)** — One-to-one communication. A producer sends a message to a queue, where it is received by a single consumer, no matter how many consumers are listening to the queue.

These messaging models are substantially the same in terms of connection and session management, message structure, delivery mode options, and delivery methods. In both messaging models, a client application can be a message producer and a message consumer.

JMS provides a set of common interfaces that can be used for creating sessions that include both Point-to-point and Publish/Subscribe producers and consumers. This feature enables what the specification calls “unified domains.”

Connections

Connection Factories

Connection factories encapsulate connection information used by the application. They are either abstracted into administered stores where they are looked up or created by constructors in a client application. The preferred technique is to look up JMS administered `ConnectionFactory` objects stored in a JNDI store.

Creating Connections

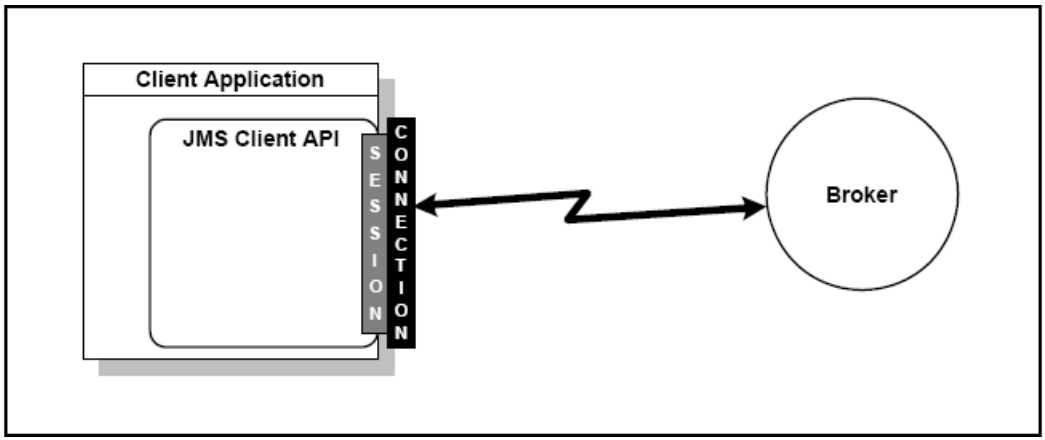
A Fuse Message Broker application starts by accessing a connection factory to create a connection that binds the client to the broker. Connection factories are administered objects so the details of connection operations can be abstracted from the application.

Creating Sessions on the Connection

Overview

A client can create multiple sessions within a connection to the broker, each independently sending and receiving messages. Sessions execute in parallel meaning that multiple threads are active concurrently, thereby optimizing throughput. The following illustration shows a client application where one connection is created, and then one session is created.

Figure 2.2. JMS Session on a Connection



Transacted Sessions: Commit or Rollback

In creating a session, you specify whether the session is **transacted** or ***non transacted***. If a session is transacted, batch acknowledgement is sent when the `commit()` method is called. When a session is non transacted, you set the acknowledgement mode.

Transaction processing significantly reduces the effort required to build applications by allowing applications to combine a group of one or more messages into a logical unit.

When a transaction **commits** , the message producer sends all the messages since the last transaction and the message consumer acknowledges the messages it received. If a transaction ***rolls back***, the message producer drops any produced messages for the transaction and, for PTP, messages received by the consumer in the scope of the transaction are automatically recovered into their queues.

Non transacted Sessions: Set the Acknowledgement Mode

When an application creates a non transacted session, the acknowledgement mode is set. The acknowledgement mode options are:

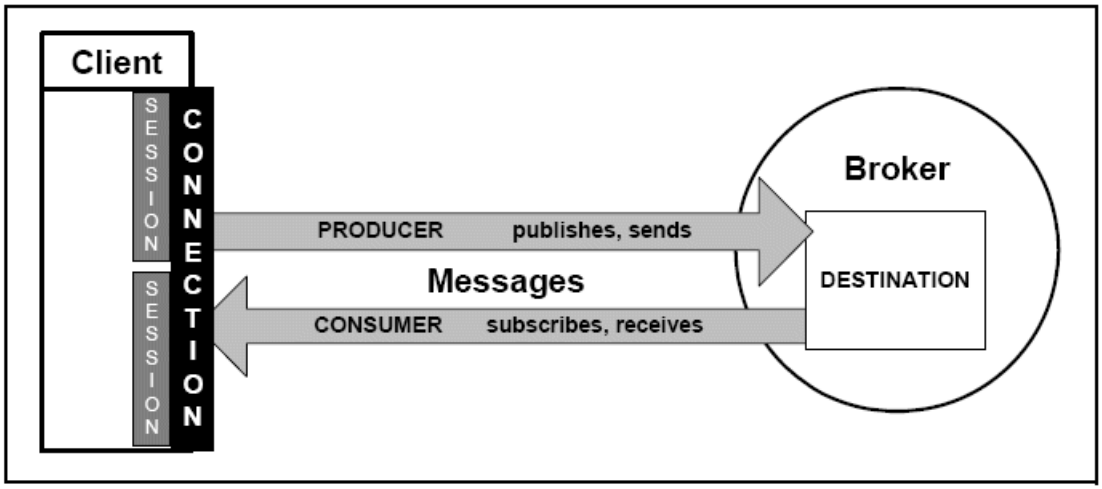
- *AUTO_ACKNOWLEDGE* — The client application session acknowledges receipt of a message when the session has successfully returned from a call to `receive`, or when the `MessageListener` (called to process messages) successfully returns a message to the client application.
- *CLIENT_ACKNOWLEDGE* — The client application calls a message's `acknowledge` method to acknowledge to the broker that all messages are not yet acknowledged.
- *DUPS_OK_ACKNOWLEDGE* — When the client session acknowledges the delivery of messages to consumers, the broker does not confirm the acknowledgement.
- *SINGLE_MESSAGE_ACKNOWLEDGE* — The client session explicitly acknowledges one specific message.

While acknowledgement sets standards for message delivery, there is no reply to the producer. The acknowledgment is always sent to, and consumed by, the broker.

Creating Producers and Consumers in a Session

Within a connection and session, a client application produces or consumes messages. You could create multiple sessions and produce messages in one session and consume messages in another session. Creating one session for sending and another for receiving messages prevents bottlenecks and enhances performance by preventing one activity from blocking another activity. As illustrated, a client application can be both a producer and a consumer:

Figure 2.3. Producer and consumer sessions on a connection



Message Producers

A message producer packages and encrypts the message body, sets the service level and protection for the outbound message, and then sends the message to its destination. If the delivery mode is **`PERSISTENT`**, the message will be placed in the broker's log or message store before starting delivery.

Message producers can set a time to live for each message. The calculated expiration time keeps a message in the queue until it expires. Expired messages are discarded.

Request/Reply

An application can use a request/reply mechanism to send a message and then wait for a response. The client application creates a temporary destination to receive the reply.

Quality of Service

Quality of Service refers to message delivery configuration that a producer specifies. Generally, Quality of Service affects the availability, reliability, scalability, and performance of message delivery. Loose Quality of Service uses a minimum number of features to maximize performance. A tighter Quality of Service uses multiple features, such as fault tolerant client connections, acknowledgements, and duplication elimination to maximize reliability, availability and transactional integrity.

For example, a loose Quality of Service might be used by a stock ticker application that wants to publish messages to a large number of subscribers to be consumed immediately. Since there are no security concerns and the information is time-sensitive, performance is at a premium. In contrast, a tight Quality of Service might be used for an online stock purchasing system, in which the company hosting the system wants to ensure transactional integrity, confidentiality, and reliability for all stock purchase transactions.

Quality of Service is supported by the following session options and message producer parameters:

- **Acknowledgement Mode** — The session determines whether the acknowledgment of communications between the client and the broker are controlled by the client, the broker, or are simply performed with reasonable efforts. Adding acknowledgements to messaging creates an additional demand on a client application, but also assures greater reliability.
- **Message Expiration** — Messages can be sent with a specific life span to make sure clients do not receive out-of-date information. When a message expires, it is dropped from the queue or from the messages stored for disconnected subscriptions.
- **Delivery Mode** — A persistent message is stored in the broker's logs and repository for later delivery to potentially disconnected users. This action provides a higher Quality of Service yet results in a corresponding decrease in performance. A persistent message survives system disconnection, or unexpected restarts. Persistence is maintained as a message is routed across brokers.

- **Guaranteed Persistence** — A message can be sent to a system ***dead letter queue*** (DLQ) if it expires or is undeliverable. On the DLQ, a message never expires and can only be discarded by explicit administrative action.
- **Priority** — Messages can be sent with a priority value that encourages the broker to position that message ahead of other messages in the same queue or topic for delivery.
- **Redelivery** — The broker can make repeated attempts to redeliver messages to each client that has not acknowledged receipt.

Message Consumers

A message consumer binds to a destination on a broker to receive messages. When messages are received, a message consumer acknowledges receipt to the broker according to the parameters set on the session.

Synchronous Message Consumption

A synchronous consumer uses a pull technique to receive messages from a destination. There are three variations of the `receive` method, one which never blocks, one which blocks until it times out, and one which blocks indefinitely (meaning that the connection could be blocked indefinitely too).

Asynchronous Message Consumption

An asynchronous consumer registers a message listener on a destination. As messages arrive, the application calls the listener's `onMessage` method. When the connection is broken, messages that are guaranteed to be delivered wait on the broker for reconnection.

Message Selectors

Message consumers can filter the messages they receive. When a client requests a session to create a consumer on a topic or queue, a message selector string in SQL-92 syntax can be used to indicate qualifications on messages. For example:

```
“Priority > 7 AND Form = ’Bid’ AND Amount is NOT NULL”.
```

Creating Messages

A message is comprised of:

- **Header** — Contains name-value pairs used by applications to identify and route messages, such as a time stamp, the expiration time, the priority, the delivery mode, the destination, an identifier, and settings used for correlation, redelivery, reply destination, and an arbitrary type (not the JMS message type.)
- **Properties** — Can be any of several data types: `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, or `String`. Custom-defined properties provide name-value pairs that can be named, typed, populated, sent, and then coerced by the consumer into other acceptable data types.
- **Body** — Set of bytes interpreted as the designated message type. The message body is optional; you can send a message where the header and property values contain all intended information. FUSE Message Broker provides the six message types defined by JMS—`Message`, `TextMessage`, `ObjectMessage`, `StreamMessage`, `MapMessage`, and `BytesMessage`.

Creating a Destination (Topic or Queue)

The unification of JMS messaging models, introduced in the JMS 1.1 specification, provides generic connection, session, message, message producer, and message consumer interfaces. The advantage is that you can have queue-based messages and topic-based messages in a single transaction.

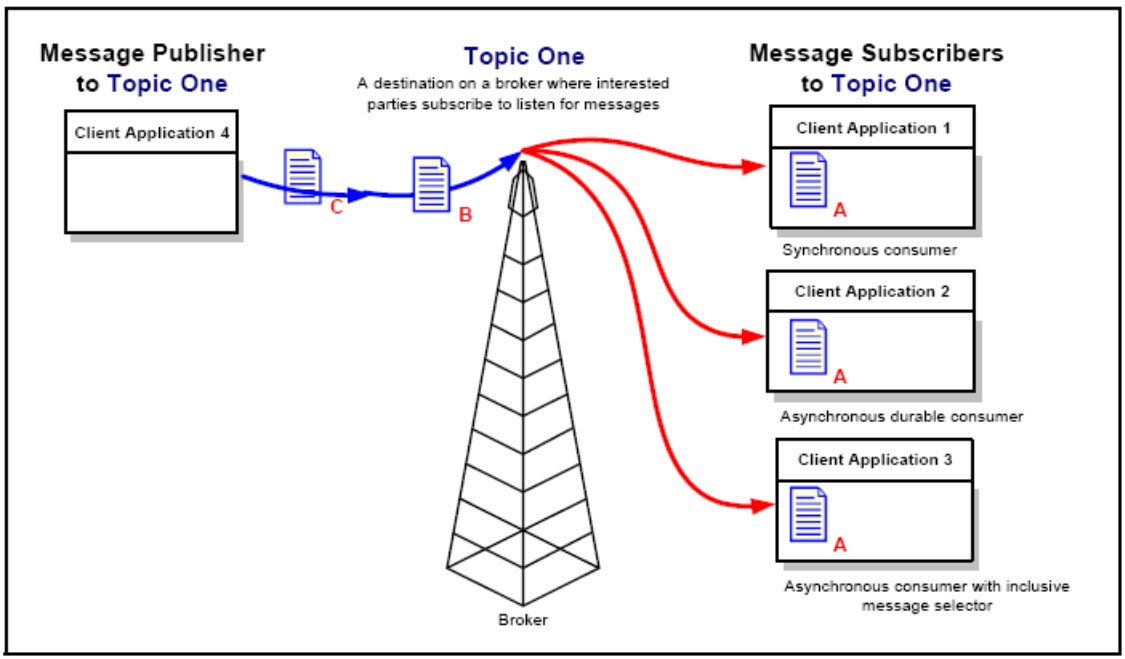
You can declare messaging model specific methods at any point, but you are then bound to that messaging model. For example, you can use a generic `ConnectionFactory` to create a `QueueConnection`. Then, you would have to create a `QueueSession`, `QueueSender`, and `QueueReceiver`.

Eventually you have to declare what set of behaviors to apply to a message. This occurs when you specify the destination. A destination is either a **topic** or a ***queue***. You can send to a statically defined queue or publish to a dynamically defined topic, both with the same name. The broker considers these different types of destinations—one is a queue and one is topic. They have different authorization rules and require different handling. The next two sections describe and differentiate between each of the messaging models.

Publish and Subscribe Messaging: Broadcast the Message

In the Publish and Subscribe (Pub/Sub) messaging model, one client application can send a message to many other client applications. In the Pub/Sub model, a message producer is a **publisher** and a consumer is a ***subscriber*** .

Figure 2.4. Concept of Publish and Subscribe Messaging Topics



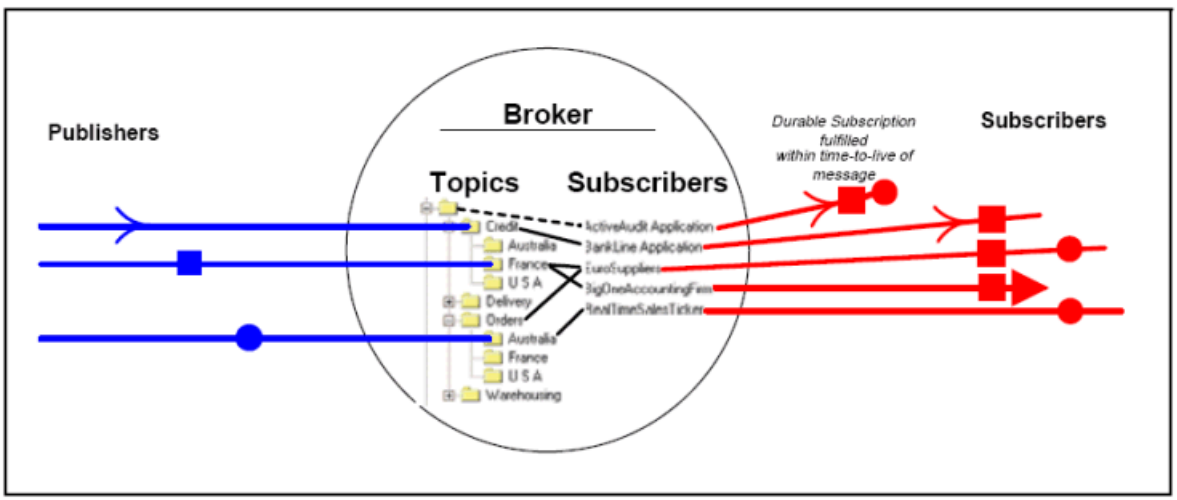
This illustration shows three subscribers who have each received message *A* and are about to receive message ***B***, and then message ***C***:

- *Client application 1*, a synchronous subscriber, waits for a message, for a specified time or forever, and then blocks to receive again after processing a message.
- *Client application 2*, a durable subscriber specified an interest in receiving messages from the broker on the selected topic, even when disconnected. Messages are saved for durable subscribers, although a saved message can expire while waiting for the durable subscriber to reconnect.

- *Client application 3*, an asynchronous subscriber has set up a message listener. When a message arrives, the `onMessage` method in the client is called. Although it is not shown in the illustration, this subscriber has a message selector. The subscriber provided a string in SQL syntax as a parameter when the subscription was created. If the selection criteria are not met, the subscriber does not take delivery of that message.

The following illustration shows the distribution of message delivery. The shapes on the publisher lines indicate messages published to specific topics over time. The crossovers within the broker represent the subscriptions that could have wildcard patterns so a single subscription delivers messages from many topics. The shapes on the subscriber lines indicate that everyone subscribed to a topic receives the same message at the same time. The exception is the durable subscription that, when not active, can receive its messages later, provided the messages do not expire.

Figure 2.5. Publishing Messages to Topics for Subscribers



Durable Subscribers

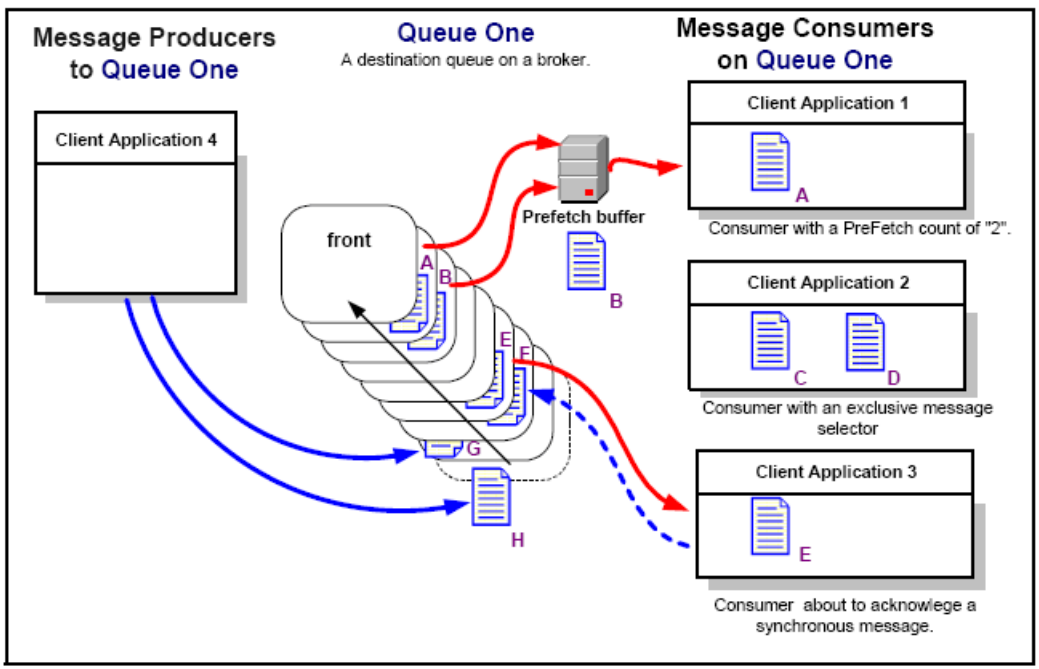
There are several special techniques for registering a durable interest and to distribute the load of subscribed messages but they all involve the notion of a subscription: if your consumer (or registered interest) is there when the message arrives on the topic, you and everyone else there gets that message. If you missed it (and did not register a durable interest), it is gone.

Any messages that are not acknowledged by durable subscribers when they are not connected to the broker, are preserved by the broker and redelivered when the durable subscriber reconnects. When a durable subscriber reconnects, messages are delivered in the correct order, although some messages might have expired. If a durable subscriber does not acknowledge delivery of a message, the message is retained until it expires.

Point-to-point Messaging: Each Message Has One Consumer

The Point-to-point (PTP) messaging model ensures that a message is delivered only once to a single consumer. The following illustration shows three message producers sending messages to three different queues. Queues are defined with a maximum size and a threshold that indicates at what point to persist messages on the queue in the data store. If messages are taken off the queue at the same rate they are placed on the queue, then no messages are persisted.

Figure 2.6. Sending Messages to Queues for Receivers



In the diagram, *Producer 2* sends a message to ***QueueB***, which has three registered listeners. Because only one receiver will receive this message, the broker decides to let ***Consumer B1*** receive the message. The other consumers do not receive that message.

Each message producer sends new messages to a queue, a destination on the broker. The broker, unless advised that there is a request for priority treatment, places new messages at the back of the queue. *Consumer A* takes the frontmost message off ***Queue A***.

On *Queue **B***, multiple consumers are listening to the queue, but only one consumer receives the message, ***Consumer B1.*** The ***QueueBrowser*** is browsing the queue without taking messages off the queue.

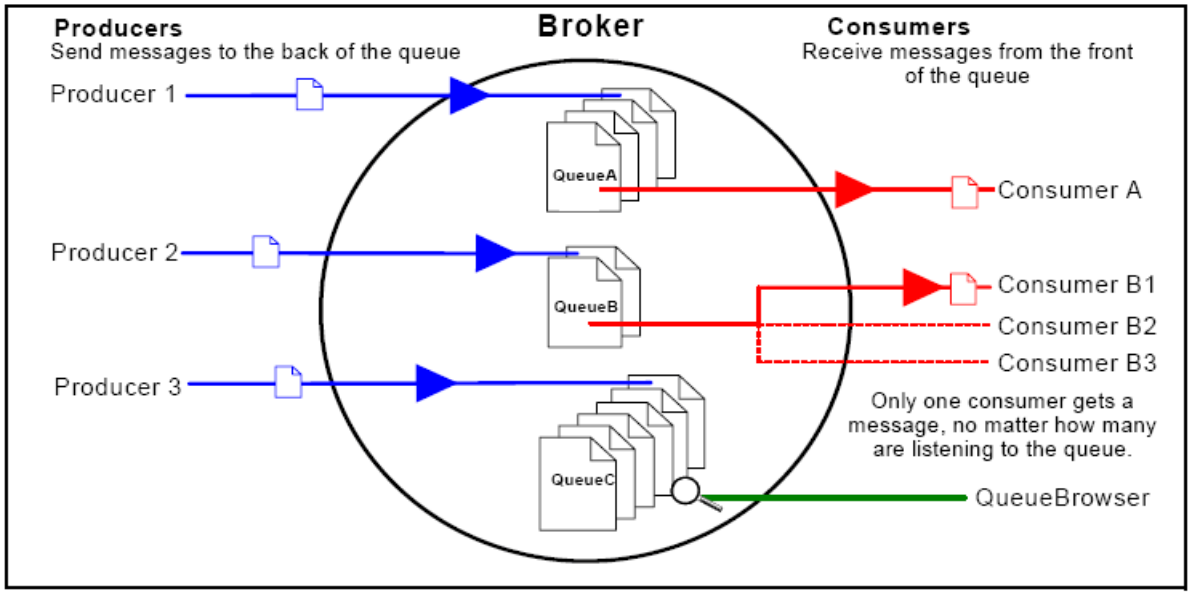
Queues are useful for load-balancing when many diverse systems share processing operations because additional receivers can be used to keep queue processing current. For example, applications for trading activities, credit card charges, online shopping carts, auctions, reservations, and ticketing often use queues.

Characteristics of the Point-to-point messaging model include:

- The first message received by the broker is the first message delivered. This “First In, First Out” (FIFO) technique makes the second through *n*th messages endure until that first message is consumed. (Note that mixed priority settings on messages affect FIFO.)
- Even when no clients specify an interest in receiving messages from a queue, messages wait for a consumer until the message expires.
- When a message’s delivery mode is set to `PERSISTENT`, the message is stored so that even a broker shutdown will not put the message at risk.
- There is only one message consumer for a given message. Many consumers can listen or receive on a queue, but only one takes delivery of a specific message.
- When the message is acknowledged as delivered by the consumer, it is removed from the queue permanently. No one else sees it and no one else receives it.

The following illustration shows other features in Fuse Message Broker Point-to-point messaging.

Figure 2.7. Features of Point-to-point messaging



The messages in this illustration are being received from the queue by three consumers:

- *Client Application 1*,with a prefetch count of ***2,*** took messages ***A*** and ***B*** off the queue. Message ***B*** is held in the prefetch buffer while message ***A*** is processed. When message ***B*** enters processing, the threshold trigger causes the consumer to draw off two more messages. The broker keeps track of unacknowledged messages and, if acknowledgement is not received before the session closes, the unacknowledged messages are reinstated in the queue.
- *Client Application 2*, consuming with a message selector, reviews qualified messages on the queue to filter the messages that it wants to process. In this example, the consumer selected and acknowledged messages ***C*** and ***D***. Assuming message ***F*** does not meet its criteria, this consumer perceives a momentarily empty queue.
- *Client Application 3*, a ***synchronous*** consumer, receives a message, acknowledges its receipt, processes the message, and then waits to receive another message. The queue state indicates that message ***E*** was the frontmost message because messages ***A*** and ***B***, while still in the queue, are awaiting acknowledgement from another consumer.
- *Client Application 4*, a message producer, has sent message ***G*** to the queue and is in the process of sending message ***H***.

The sample applications in the next chapter provide exercises that let you observe JMS messaging behaviors in both JMS messaging models.

Chapter 3. Setting Up the Guided Tour

Before you can run the samples, you need to configure your environment.

Overview

This guide uses a collection of samples to demonstrate basic JMS concepts. Before you can run them you need to set up your environment properly.

Prerequisites

T run the samples used in this guide you need:

- Fuse Message Broker
- Apache Ant
- Java 5 JDK

Accessing the samples

The sample applications used in this book are located in the `exploring-jms` folder of the Fuse Message Broker installation.

Setting up the environment and sample windows

The following procedures describe the steps on various platforms to set the required `HOME`, `PATH`, and `CLASSPATH` environment variables.

Windows

To set the environment and set up sample windows on Windows:

1. Open a console window, and then enter:

```
c:\> set ANT_HOME=ant_install
c:\> set JAVA_HOME=jvm_install
c:\> set PATH=%JAVA_HOME%\bin;%ANT_HOME%\bin;%PATH%
```

2. Spawn the windows that will run the sample applications:

```
c:\> start "Window 1" cmd
c:\> start "Window 2" cmd
c:\> start "Window 3" cmd
```

3. Launch the broker:

```
c:\> set FUSE_MB_ROOT=InstallDir
c:\> %FUSE_MB_ROOT%\bin\activemq
```

Linux/UNIX

To set the environment and set up sample windows on Linux/UNIX:

1. Open a terminal window, and then enter:

```
$ ANT_HOME=ant_install
$ export ANT_HOME
$ JAVA_HOME=jvm_install
$ export JAVA_HOME
$ PATH=$JAVA_HOME/bin:$ANT_HOME/bin:$PATH
$ export PATH
$ FUSE_MB_ROOT=InstallDir
$ export FUSE_MB_ROOT
```

2. Spawn the windows that will run the sample applications:

```
$ bg
$ xterm -title "Window 1" &
$ xterm -title "Window 2" &
$ xterm -title "Window 3" &
```

3. Launch the broker:

```
$ cd $FUSE_MB_ROOT/bin
$ ./activemq
```

OS X

To set the environment and set up sample windows on OS X:

1. Open a terminal window, and then enter:

```
$ ANT_HOME=ant_install
$ JAVA_HOME=jvm_install
$ PATH=$JAVA_HOME/bin:$ANT_HOME/bin:$PATH
$ FUSE_MB_ROOT=InstallDir
```

2. Launch the broker:

```
$ cd $FUSE_MB_ROOT/bin
$ ./activemq
```

3. Open three terminal windows to run the sample applications. Repeat Step 1 on page 30 in each of these windows.

Chapter 4. Publish and Subscribe Messaging Samples

This series of samples explores basic subscriptions, durable subscriptions, wildcards in topic hierarchies, filtered subscriptions, and batching in transacted sessions.

Chat Application

What the sample does

In the `Chat` application, whenever anyone sends a text message to a given topic, all active applications running `Chat` receive that message as subscribers to that topic. This is the most basic form of publish and subscribe activity.

Running the sample

To run the chat sample do the following:

1. In window 1, enter: **`ant chat1`**, then type **`Hello`**, and press **Enter**.

   Window 1 displays:

   ```
   Chatter_1: Hello
   ```

2. In window 2, enter: **`ant chat2`**, then type **`Pronto`**, and press **Enter**.

   Both subscribers get the message so both windows display:

   ```
   Chatter_2: Pronto
   ```

3. In window 3, enter: **`ant chat3`**, then type **`Bonjour`**, and press **Enter**.

   All three subscribers get the message, so each window displays:

   ```
   Chatter_3: Bonjour
   ```

4. In window 3, stop `chat3` by pressing **Ctrl**+**C**.
5. Send some messages in the `chat1` and `chat2` windows.
6. In window 3, run: **`ant chat3`** again.
7. Send some messages in the `chat1` and `chat2` windows.

   All three subscribers get the message. But `Chatter_3` gets only the messages since it reconnected, and gets none of the messages that were sent while it was disconnected.

   If subscribers miss some of the messages, they pick up just the latest messages whenever they reconnect to the broker. Nothing is retained and nothing is guaranteed to be delivered, so throughput is fast.

Stopping the sample

To stop the Chat sessions press **Ctrl**+**C** in each of the windows.

DurableChat Application

What the sample does

In Pub/Sub messaging, when messages are produced, they are sent to all active consumers who subscribe to a topic. Some subscribers register an enduring interest in receiving messages that were sent while they were inactive. These **durable subscriptions** are permanent records in the broker's persistent storage mechanism.

Running the sample

To run the DurableChatting sample do the following:

1. In window 1, enter **`ant durable1`**.
2. In window 2, enter **`ant durable2`**.
3. Type text in each window and press **Enter**. Each window displays all the messages. Connected durable subscribers are the same as connected nondurable subscribers.
4. In window 2, press **Ctrl**+**C**.
5. In window 1, type text, and press **Enter**.
6. In window 2, enter **`ant durable2`** again.

   When the window opens, it first displays all the messages that were stored for its subscription. (If you waited a while, messages sent more than 30 minutes ago were dropped.)
7. In window 3, enter **`ant durable3`**.

   In the console window that opens, no messages are displayed. While it is a durable subscriber, it had not yet established its durable subscription on this broker topic.

Important

If this sample has run against this broker before this run, the durable interest has been established so you **do** get the stored messages. You could unsubscribe the user from the durable subscription, but in the scope of this JMS exploration, you might find it as easy to close all the open windows, delete the /data directory, and then restart the exploration.

Important

While some applications tolerate multiple instances of the same user producing or consuming on the same destination, running a second instance of one of these `DurableChatter` scripts fails (as it should) with the error (in the broker's console window):

```
InvalidClientIDException:
Broker:localhost, Client:DurableChatter_n already connected
```

Stopping the sample

To stop the DurableChat sessions press **Ctrl**+**C** in each of the windows.

HierarchicalChat Application

What the sample does

Fuse Message Broker supports a hierarchical topic structure that allows wildcard subscriptions.

Each application instance specifies a publish topic and a subscribe topic, as follows:

- *Chat*:
  - Publish to `jms.samples.chat`
  - Subscribe to `jms.samples.chat`
- *DurableChat*:
  - Publish to `jms.samples.durablechat`
  - Subscribe to `jms.samples.durablechat`
- *HierarchicalChat*:
  - Publish to `jms.samples.hierarchicalchat`
  - Subscribe to `jms.samples.*`

You can see that each of the applications is publishing to a different topic. However, the `HierarchicalChat` application is subscribing to a topic that ends in asterisk (**`*`**), a wildcard that accepts any topic with the root `jms.samples`. It is important to note that this is referred to a hierarchical wildcard, as it must be adjacent to dot delimiters.

Running the sample

To run the HierarchicalChatting sample do the following:

1. In window 1, enter **`ant chat1`**
2. In window 2, enter **`ant durable1`**
3. In window 3, enter **`ant wildcard`**
4. In the **wildcard** window, enter some text and then press **Enter**.

   The message is displayed in only that window.

5. In the **chat1** window, enter some text and then press **Enter**.

   The message is displayed in that window and in the **wildcard** window.

6. In the **durable1** window, enter some text and then press **Enter**.

   The message is displayed in that window and in the **wildcard** window.

Stopping the sample

To stop the applications, press **Ctrl**+**C** in each of the windows.

MessageMonitor Application

What the sample does

The `MessageMonitor` sample application provides an example of a supervisory application with a graphical interface. By subscribing with the wildcard syntax used in the `HierarchicalChat`, the monitor gets messages on all topics in the topic hierarchy. The application listens for any message activity, and then displays each message in its window.

Running the sample

To start the MessageMonitor sample, enter **`ant tmonitor`** in window 3.

The **MessageMonitor** Java window opens.

To send messages to the MessageMonitor do the following:

1. In window 1, enter **`ant chat1`**, then type **`Hello`**, and press **Enter**.

   The message displays in the **MessageMonitor** window, noting that the message was received on the `jms.samples.chat` topic

2. In window 2, enter **`ant durable1`**, then type **`Hello`**, and press **Enter**.

   The message displays in the **MessageMonitor** window, noting that the message was received on the `jms.samples.durablechat` topic.

   Because the *MessageMonitor* subscribes to the topic `jms.samples.*`, messages are received from both publishers. The `Chat` and `DurableChat` applications subscribe only to their respective topics.

3. Click the **Clear** button to empty the listed messages.

Stopping the sample

To stop the applications, press **Ctrl**+**C** in each of the windows.

SelectorChat Application

What the sample shows

While specific queues and topics provide focused content nodes for messages that are of interest to an application, there are circumstances where the application developer might want to qualify the scope of interest a consumer has in messages using a syntax similar to an SQL `WHERE` clause.

In the `SelectorChat` samples, each of the `SelectorChat` samples publishes to the `jms.samples.chat` topic with messages that have a property set to a different value, and then sets the subscriber to select only messages set to the appropriate property value.

Running the sample

To run the SelectorChatting samples do the following:

1. In window 1, enter **`ant filterchat1`**.
2. In window 2, enter **`ant filterchat2`**.
3. In window 3, enter **`ant chat1`**.
4. In the **SelectiveChatter_1** window, enter some text and then press **Enter**.

   The message is displayed in that window and the **Chatter_1** window.
5. In the **SelectiveChatter_2** window, enter some text and then press **Enter**.

   The message is displayed in that window and the **Chatter_1** window.
6. In the **Chatter_1** window, enter some text and then press **Enter**.

   The message is displayed only in that window.

Stopping the sample

To stop all the applications, press **Ctrl+C** in each of the windows.

TransactedChat Application

What that sample shows

Transacted messages are a group of messages that form a single unit of work. Much like an accounting transaction made up of a set of balancing entries, a messaging example might be a set of financial statistics where each entry is a completely formed message and the full set of data comprises the update.

A session is declared as *transacted* when the session is created. While producers—PTP Senders and Pub/Sub Publishers—produce messages as usual, the messages are stored at the broker until the broker is notified to act on the transaction by delivering or deleting the messages. To determine when the transaction is complete, the programmer must:

- Call the method to *commit* the set of messages. The session's `commit()` method tells the broker to sequentially release each of the messages that have been cached since the last transaction. In this sample, the commit case is set for the string `COMMIT`.
- Call the method to *roll back* the set of messages. The session's `rollback()` method tells the broker to flush all the messages that have been cached since the last transaction ended. In this sample, the rollback case is set for the string `CANCEL`.

Running the sample

To run the Pub/Sub TransactedChat samples do the following:

1. In window 1, enter **`ant chat1`**.
2. In window 2, enter **`ant xnchat`**.
3. In the `TransactedChatter` window, type some text and press **Enter**.
4. Type **`COMMIT`** in the `TransactedChatter` window and press **Enter**.

   The message is delivered to the `TransactedChatter` window and to the `Chatter_1` window.
5. Type text in the `TransactedChatter` window and press **Enter**.
6. Repeat to produce a few messages.
7. Type **`COMMIT`** in the `TransactedChatter` window and press **Enter**.

   The batch of messages is delivered as a series of individual messages to the `TransactedChatter` window and to the `Chatter_1` window.

8. Type text in the `TransactedChatter` window and press **Enter**.
9. Repeat to produce a few messages.
10. Type **`CANCEL`** in the `TransactedChatter` window and press **Enter**.

    The batch of messages is dropped.

Subsequent entries will form a new transaction to either commit or rollback.

Stopping the sample

To stop all the applications, press **Ctrl**+**C** in each of the windows.

Chapter 5. Point-to-Point Messaging Samples

The following samples demonstrates how point-to-point messaging differs from publish and subscribe messaging.

Talk Application

What the sample does

In the `Talk` application, whenever a text message is sent to a given queue, all active `Talk` applications that are waiting to receive messages on that queue take turns receiving the message at the front of the queue.

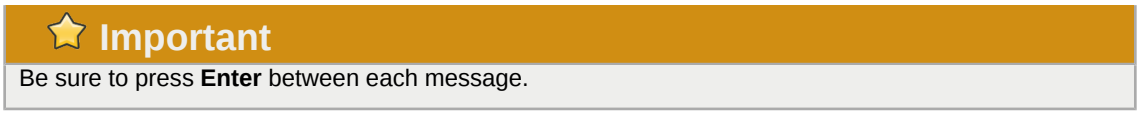
Running the sample

To run the Talking sample do the following:

1. In window 1, enter **`ant talk1`**.
2. In window 2, enter **`ant talk2`**.
3. In window 3, enter **`ant talk3`**.
4. In the **Talker1** window, type **`1`**, and then press **Enter**.

   The text is displayed in only one of the other `Q1` receiver windows. A point-to-point message has only one receiver.
5. Again, in the **Talker1** window, type **`2`**, and then press **Enter**.

   The text is displayed in the other `Q1` receiver window. Multiple receivers on a queue take turns receiving messages.
6. In the **Talker1** window, create several messages, such as **`3`** ,**`4`**, ..., **`9`**.



   One of the `Q1` receivers gets messages `1, 3, 5, 7, 9` while the other gets `2, 4, 6, 8`.

   If you opened another **Talker_2** or **Talker_3** window, the distribution to the `Q1` receivers would be `1, 4, 7` for the first, `2, 5, 8` for the next and `3, 6, 9` for the third.

Stopping the sample

To stop all the sessions, press **Ctrl**+**C** in each of the windows.

The QueueMonitor Application

What the sample does

The QueueMonitor moves through a specified set of queues, listing the active messages it finds as it examines each queue. In the examples to this point, the Talk samples left no messages in any queue.

Comparing MessageMonitor and QueueMonitor

The monitor samples each open GUI windows that provide a scrolling array of its contents. The nature of the two monitors underscores fundamental differences between the Publish and Subscribe messaging model and the Point-to-point messaging model. The differences between MessageMonitor and QueueMonitor are as follows:

What messages are displayed?

- *MessageMonitor:* Delivered.
- *QueueMonitor:* Undelivered.

When does the display update?

- *MessageMonitor:* When a message is published to a subscribed topic, it is added to the displayed list.
- *QueueMonitor:* When you click the ***Browse Queues*** button, the list is refreshed.

When does the message go away?

- *MessageMonitor:* When the display is cleared for any reason.
- *QueueMonitor:* When the message is delivered (or when it expires).

What happens when the broker and monitor are restarted?

- *MessageMonitor:* As messages are listed at the moment they are delivered, there are no messages in the **`MessageMonitor`** until new deliveries occur.
- *QueueMonitor:* Listed messages marked `PERSISTENT` are stored in the broker persistent storage mechanism. They are redisplayed when the broker and the ***QueueMonitor*** restart and then choose to browse queues.

Starting the sample

To run the QueueMonitor sample do the following:

1. In window 1, enter **`ant talk1`**.
2. In window 3, enter **`ant qmonitor`**.

   The `QueueMonitors` console window lists the queues that have been specified for it to browse.

   The **QueueMonitor** Java window opens.
3. Click **Browse Queues**.

   The messages in the queue at the moment it is browsed are listed. If you are following along carefully, there should be no messages in any queue.

Putting messages into a queue

To put messages into a queue do the following:

1. In window 1 (where `Talker1` is running), type **`1`** and then press **Enter**.
2. Repeat step 1 to create a few messages, such as **`2`** **Enter**, **`3`** **Enter**, **`4`** **Enter**.
3. In the **QueueBrowser** window, click **Browse Queues**.

   The messages are in the queue. They will continue to be there until a receiver receives them on that queue, or they expire (set to 30 minutes by the sender).

The messages that are waiting on the queue will get delivered to the next receiver that chooses to receive from that queue.

Receiving messages from a queue

To receive the queued messages do the following:

1. In window 2, enter **`ant talk2`**.
2. When the **Talker_2** application opens, it shows that it consumes the messages in the queue in sequence.
3. In the **QueueBrowser** window, click **Browse Queues**.

   The queues are all empty. As long as you have receivers on the sample queues, no messages will display in the **QueueMonitor** window.

Stopping the sample

To stop all the applications, press **Ctrl**+**C** in each of the windows.

SelectorTalk Application

What the sample does

The `SelectorTalk` sample applications are similar yet consistent with the behavior of the messaging model. The **`SelectiveTalkers`** both send and receive on **`Q1`**. The **`Talkers`** do not specify selection parameters.

Running the sample

To run the SelectorTalk sample do the following:

1. In window 1, enter **`ant filtertalk1`**.
2. In window 2, enter **`ant filtertalk2`**.
3. In window 3, enter **`ant talk2`**.

   `Talker_2` sends to `Q2` and receives on `Q1`.
4. In the **SelectiveTalker_1** window, enter some text and then press **Enter**. Send a few messages in this window.

   The messages are displayed in *either* that window or the **Talker_2** window (usually alternately.)
5. In the **SelectiveTalker_2** window, enter some text and then press **Enter**. Send a few messages in this window.

   The messages are displayed in *either* that window or the **Talker_2** window (usually alternately.)
6. In the **Talker_2** window, enter some text and then press **Enter**.

   None of the windows receives the message. **Talker_2** is receiving on `Q2`. The selective talkers are receiving on `Q1`, but they are qualifying their selection as only messages that have the specified property, and, at that, set to their preferred value. So the message will be stored in `Q1` even though there are receivers on `Q1`.
7. In window 3, press **Ctrl+C** to stop `Talker_2`.
8. In window 3, enter **`ant talk1`**.

   `Talker_1` sends to `Q2` and receives on `Q1`. When it starts, it immediately receives the messages stored on `Q1` (unless they have expired.)

Stopping the sample

To stop the applications press **Ctrl**+**C** in each window.

TransactedTalk Application

What the sample does

The transaction samples show that the transaction scope is between the client in the JMS session and the broker. When the broker receives commitment, the messages are placed onto queues or topics in the order in which they were buffered as standard messages. The following message delivery is normal:

- **Pub/Sub Messages** Messages are delivered in the order entered in the transaction yet influenced by the priority setting of these and other messages, the use of additional receiving sessions, and the use of additional or alternate topics. The messages are not delivered as a group.
- **PTP Messages** The order of messages in the queue is maintained with adjustments for priority differences but there is no guarantee thatwhen multiple consumers are active on the queuea `MessageConsumer` will receive one or more of the `MessageProducers` transacted messages.

Running the sample

To run PTP TransactedTalk sample do the following:

1. In window 1, enter **`ant xntalk`**.
2. In the console window that opens, type text in the window and press **Enter**.
3. Repeat to produce a few messages.
4. Type **`COMMIT`** in the window and press **Enter**.
5. In window 2, enter **`ant qmonitor`**.
6. In the **QueueBrowser** window, click **Browse Queues**.

   The messages are listed.
7. In window 3, enter **`ant talk2`**.

   The window opens with the series of messages in `Q1` from the **`TransactedTalker`**.
8. In window 1, type some text, and press **Enter**.
9. Repeat to produce a few messages.
10. Type **`CANCEL`** in the window and press **Enter**.

The batch of messages is dropped:

- In window 3the **talk2** window, no messages are received on `Q1`.
- In the **QueueBrowser** window, **Browse Queues** lists no messages in `Q1`.

11. In window 1, do the following:
    a. Type some text.
    b. Press **Enter**.
    c. Type **`COMMIT`**.
    d. Press **Enter.**

    The newly-committed transaction batch is received in window 3.

Stopping the sample

To stop the applications press **Ctrl**+**C** in each window.

Chapter 6. Request and Reply Samples

Implementing the request-reply messaging pattern using JMS requires some special tricks.

Loosely coupled applications require special techniques when it is important for the publisher to certify that a message was delivered in either messaging domain:

- **Publish and Subscribe** — While the publisher can send long-lived messages to durable receivers and get acknowledgement from the broker, neither of these techniques confirms that a message was actually delivered or how many, if any, subscribers received the message.
- **Point-to-point** — While a sender can see if a message was removed from a queue, implying that it was delivered, there is no indication where it went.

A message producer can request a reply when a message is sent. A common way to do this is to set up a **temporary destination** and header information that the consumer can use to create a reply to the sender of the original message.

In both request and reply samples, the replier's task is a simple data processing exercise: standardize the case of the text sent/receive text and send back the same text as either all uppercase characters or all lowercase characters—then publish the modified message to the temporary destination that was set up for the reply.

While request-and-reply provides proof of delivery, it is a blocking transaction—the requestor waits until the reply arrives. While this situation might be appropriate for a system that, for example, issues lottery tickets, it might be preferable in other situations to have a formally established return destination that echoes the original message and a **correlation identifier**—a designated identifier that certifies that each reply is referred to its original requestor.

The sample applications use JMS sample classes `TopicRequestor` and `QueueRequestor`. You should create the Request/Reply helper classes that are appropriate for your application.

These request and reply samples show that request/reply mechanisms are very similar across messaging models, and that, while there might be zero or many subscriber replies, there will be, at most, one PTP reply.

Request and Reply (Pub/Sub)

What the sample does

In this example in the Pub/Sub domain, the replier application must be started before sending messages from the requestor so that the Pub/Sub replier's message listener can receive the message and release the blocked requestor.

Running the sample

To run the Pub/Sub Request and Reply sample do the following:

1. In window 1, enter **`ant trequest`**.
2. In window 2, enter **`ant treply`**.
3. In the **Requestor** window, type **`AaBbCc`** then press **Enter**.

   The **Replier** window reflects the activity, displaying:

   ```
   [Request] RequestingChatter:
                               AaBbCc
   ```

   The replier completes its operation (converts text to uppercase) and sends the result in a message to the requestor. The requestor gets the reply from the replier:

   ```
   [Reply] Transformed RequestingChatter: AaBbCc to all uppercase: REQUESTINGCHATTER: AABBCC
   ```

Stopping the sample

To stop the applications press **Ctrl**+**C** in each window.

Request and Reply (PTP)

Overview

In the PTP domain, the requestor application can be started and even send a message before the replier application is started. The queue holds the message until the replier is available. The requestor is still blocked, but when the replier's message listener receives the message, it releases the blocked requestor. The sample code includes an option (`-m`) to switch the mode between uppercase and lowercase.

Running the sample

To run the PTP Request and Reply sessions do the following:

1. In window 1, enter: **`ant qrequest`**.
2. In window 2, enter: **`ant qreply`**.
3. In the `Requestor` window, type **`AaBbCc`** then press **Enter**.

   The **Replier** window reflects the activity, displaying:

   ```
   [Request] RequestingTalker:
                               AaBbCc
   ```

   The replier completes its operation (converts text to uppercase) and sends the result in a message to the requestor. The requestor gets the reply from the replier:

   ```
   [Reply] Transformed RequestingTalker: AaBbCc to all uppercase: REQUESTINGTALKER: AABBCC
   ```

Stopping the sample

To stop the applications press **Ctrl**+**C** in each window.

Chapter 7. Queue Test Loop Sample

A simple loop test lets you experiment with messaging performance.

Overview

A simple loop test lets you experiment with messaging performance.

The `RoundTrip` sample application sends a brief message to a sample queue and then uses a temporary queue to receive the message back. A counter is incremented and the message is sent for another trip. After completing the number of cycles you entered when you started the test, the run completes by displaying summary and average statistics.

Note

This sample is not intended as a performance tool.

Running the sample

To run the QueueRoundTrip sample enter **`ant roundtrip`** in Window 1.

The `QueueRoundTrip` window produces and consumes queue messages in a loop, producing the next message after the prior one has been consumed. When it has completed the specified number of cycles (set to 10,000 for the sample run), it reports how long it took to complete all the cycles.

Chapter 8. Changing Parameters and Modifying Source Code

Modifying the parameters of the scripted samples makes it easy to see the samples behave differently. Then, modifying and compiling the Java source files for the samples shows how you can modify these applications in ways that can prepare you for weaving the JMS patterns into your applications.

The JMS samples are compiled and run using the Apache Ant build tool, where the Ant build file, `build.xml`, defines the rules for building the samples in a platform-independent manner. We now explore some ways of modifying and customizing the samples, as follows:

- **Modify and extend application parameters in the Ant build file**—You will change a few of the parameters and reset some that were allowed to use their default values. Then, you will run the same samples to observe the changed behaviors.
- **Analyze and modify the JMS methods and patterns in the Java source files** —You will examine some of the sample application source files to see the JMS methods in the samples. The you will change some sample applications to revise the quality of service and the functionality in the scope of the original application. Then, compiling and running the application will let us test out the changes.

Revising Parameters in the Build File

Basics

The Ant `build.xml` file was revised to add the Fuse Message Broker version and installation location. Now you will change and add parameters in the `target` section for each sample run. Each parameter's name-value pair is presented as two sequential arguments. For example:

```
<target name="chat1">
  <java classname="Chat" ... >
    ...
    <arg value="-u"/>
    <arg value="Chatter1"/>
  </java>
</target>
```

That listing shows that the `-u` parameter, the user name, is assigned the value `"Chatter1"`.

Changing the User Name

Change the user name value from `"Chatter1"` to `"Fred"` as shown:

```
<target name="chat1">
  <java classname="Chat" ... >
    ...
    <arg value="-u"/>
    <arg value="Fred"/>
  </java>
</target>
```

Save the file, and then enter **`ant chat1`** in one of the sample windows, type **`Hello`**, and then press **Enter**. The response line is:

```
[java] Fred: Hello
```

Note

The *`password`* parameter requires implementation of security. For these samples, that would require all the user names and their respective passwords be defined in the security store. For information about securing JMS applications see *Security Guide*.

Changing destination names

The queue and topic names used in the sample are arbitrary and can be created dynamically. Some pub/sub application applications let you specify the publish topic and the subscribe topic as parameters. For `wildcard`, the `HierarchicalChat` exposes the topic to which the application publishes as the `-t` parameter, and the topic to which it subscribes as the `-s` parameter. You could change these topics to demonstrate a Fuse Message Broker feature by extending the publish topic to a fourth level, and changing the wildcard on the subscriber topic from `*` to `>`. That expands the wildcard from all topics at the current hierarchical level to all topics at that level or deeper.

```
<target name="wildcard">
  <java classname="HierarchicalChat" fork="true">
    ...
    <arg value="-u"/>
    <arg value="HierarchicalChatter"/>
  </java>
</target>
```

Other topic and queues could be changed in the build file so you can observe how they interact with other sample applications. For the `MessageMonitor` application, change its depth of topic hierarchies in its properties file, `MessageMonitor.properties`, to **subscriptionTopics jms.samples.>**

Changing the queue round trips

The QueueRoundTrip application is set in the build file to iterate 10,000 times through sending a message to a queue, receiving it off the queue, and taking that as the cue to send another message. You can change that number to a larger or smaller value to see start and stop operations are impactful. Try 1000 (one thousand), save the build file, and run **ant roundtrip**. Try 100000 (one hundred thousand), save the build file, and run **ant roundtrip**. For example:

```
<target name="chat1">
  <java classname="QueueRoundTrip" ... >
    ...
    <arg value="-n"/>
    <arg value="1000"/>
  </java>
</target>
```

Distributing the client and the broker

At this point in exploring JMS, the broker used by the samples is always on the computer where the sample applications run. While you might use a local or an embedded broker, JMS messaging is designed so that the sample applications can run on a computer that has the appropriate libraries, yet can connect to a broker on a different system to produce and consume messages. The standalone broker system would typically be in a

location where it can be monitored and provided resources that ensure optimal availability to any applications that use it.

The sample applications provide a `-b` parameter to specify the protocol, host, and port of the preferred broker. As the default broker configuration specifies the TCP protocol on `localhost`, listening on port 61616, you can use another installation of the Java, Fuse Message Broker, Ant, and the Exploring JMS files (as described in the previous chapter) on another computer to experience distributed connection. On one host set up and start the broker. On the other host (the remote host), do the same.

Stop the broker on the remote host, then modify the `build.xml` file on the remote host to add the connection parameter and specify the host name where the broker is running. For example:

```
<target name="chat1">
  <java classname="Chat" ... >
    ...
    <arg value="-u"/>
    <arg value="Fred"/>
    <arg value="-b"/>
    <arg value="tcp://remoteHostName:61616"/>
  </java>
</target>
```

Save the build file and then run `chat1` in a sample window on each of the computers. When you enter messages in either `Chatter_1` window, the subscribers both get the message from the same broker.

Analyzing and Modifying the Java Source Files

Overview

Now that you are familiar with JMS behavior, let's look inside the application source files to examine some of the patterns that are used. You will see how you can make and compile some changes to the source files that expose Fuse Message Broker client's JMS features you can then test. (There are features that require setting broker configurations; those will be discussed in a forthcoming chapter.)

Basics

The application source files and the class files that they create are located in application-specific folders that are subfolders of the two messaging models, `QueuePTPSamples`, and `TopicPubSubSamples`. The build file enabled you to enter ant commands at the root of the samples such that **ant chat1** did the same as navigating to the `TopicPubSubSamples/Chat` directory to run the `Chat.class` file at that location as:

```
java Chat -u Chatter1
```

The `Chat` folder contains the source file `Chat.java`.

When you make changes to that or any other `.java` file, compile the source file to an updated class file and then run the modified file. To do this, you need to have a Java Development Kit (JDK) specified as `JAVA_HOME`, and the `CLASSPATH` needs to specify the JDK's tools JAR, the Fuse Message Broker's activemq core JAR, and the Geronimo JMS JAR. For example, on a Windows system, you might do the following:

```
set JAVA_HOME=C:\jdk1.5.0_11
set FUSE_MB_=C:\progress\-5.5.1-fuse-00-xx
set CLASSPATH=.;%JAVA_HOME%\lib\tools.jar;   \
                %FUSE_MB%\lib\activemq-core-5.5.1-fuse-00-xx-fuse.jar;   \
                %FUSE_MB%\lib\geronimo-jms_1.1_spec-1.1.1.jar
```

Then, with the command line located at the `Chat` directory, enter:

```
javac Chat.java
```

Setting noLocal in chat

The TopicPubSub samples show that the message is received by every subscriber to the topic and its hierarchy -- including the publisher's connection. A feature of JMS is the ability to set a `noLocal` Boolean on the subscription that inhibits echoing messages sent in the publisher's connection. In `Chat.java` you can set the `noLocal` value to `true` but there is a catch: the method signature that sets `noLocal` requires a message selector string also. In `Chat.java`, the message consumer is created as follows:

```
javax.jms.MessageConsumer subscriber = subSession.createConsumer(topic);
```

Edit the line to set the message selector to a zero-length String to satisfy the method signature, and then add the boolean value `true`, as follows:

```
javax.jms.MessageConsumer subscriber = subSession.createConsumer(topic, "", true);
```

When you save and compile this sample, see the effect in your samples. Start `chat1` and `chat2`. Send some messages from each of the chatters. The messages are not echoed in that sender's window.

Steal this code!

Use the patterns in these sample applications to meld them and transform them from user interactive examples to applications that pass your data to the message producer, and that take messages received by message consumers to move them into your data stores and application logic. As always, provide proper copyrights and licenses in your source files and application packages.

Index