The Java EE 6 Tutorial, Volume II

Advanced Topics Beta

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

Part No: 820–7628
December 2009

091204@23031

Contents

Preface

This tutorial is a guide to developing enterprise applications for the Java™ Platform, Enterprise Edition 6 (Java EE 6).

This preface contains information about and conventions for the entire Sun GlassFish™ Enterprise Server (Enterprise Server) documentation set.

Enterprise Server v3 is developed through the GlassFish project open-source community at `https://glassfish.dev.java.net/`. The GlassFish project provides a structured process for developing the Enterprise Server platform that makes the new features of the Java EE platform available faster, while maintaining the most important feature of Java EE: compatibility. It enables Java developers to access the Enterprise Server source code and to contribute to the development of the Enterprise Server. The GlassFish project is designed to encourage communication between Sun engineers and the community.

Before You Read This Book

Before proceeding with this tutorial, you should have a good knowledge of the Java programming language. A good way to get to that point is to work through *The Java Tutorial, Fourth Edition*, Sharon Zakhour et al. (Addison-Wesley, 2006). You should also be familiar with the Java DataBase Connectivity (JDBC™) and relational database features described in *JDBC API Tutorial and Reference, Third Edition*, Maydene Fisher et al. (Addison-Wesley, 2003).

Enterprise Server Documentation Set

The Enterprise Server documentation set describes deployment planning and system installation. The Uniform Resource Locator (URL) for Enterprise Server documentation is `http://docs.sun.com/coll/1343.9`. For an introduction to Enterprise Server, refer to the books in the order in which they are listed in the following table.

TABLE P–1 Books in the Enterprise Server Documentation Set

Book Title	Description
Release Notes	Provides late-breaking information about the software and the documentation. Includes a comprehensive, table-based summary of the supported hardware, operating system, Java Development Kit (JDK™), and database drivers.
Quick Start Guide	Explains how to get started with the Enterprise Server product.
Installation Guide	Explains how to install the software and its components.
Administration Guide	Explains how to configure, monitor, and manage Enterprise Server subsystems and components from the command line by using the `asadmin(1M)` utility. Instructions for performing these tasks from the Administration Console are provided in the Administration Console online help.
Application Deployment Guide	Explains how to assemble and deploy applications to the Enterprise Server and provides information about deployment descriptors.
Your First Cup: An Introduction to the Java EE Platform	Provides a short tutorial for beginning Java EE programmers that explains the entire process for developing a simple enterprise application. The sample application is a web application that consists of a component that is based on the Enterprise JavaBeans™ specification, a JAX-RS web service, and a JavaServer™ Faces component for the web front end.
Application Development Guide	Explains how to create and implement Java Platform, Enterprise Edition (Java EE platform) applications that are intended to run on the Enterprise Server. These applications follow the open Java standards model for Java EE components and APIs. This guide provides information about developer tools, security, and debugging.
Add-On Component Development Guide	Explains how to use published interfaces of Enterprise Server to develop add-on components for Enterprise Server. This document explains how to perform *only* those tasks that ensure that the add-on component is suitable for Enterprise Server.
Scripting Framework Guide	Explains how to develop scripting applications in languages such as Ruby on Rails and Groovy on Grails for deployment to Enterprise Server.
Troubleshooting Guide	Describes common problems that you might encounter when using Enterprise Server and how to solve them.
Reference Manual	Provides reference information in man page format for Enterprise Server administration commands, utility commands, and related concepts.
Java EE 6 Tutorial, Volume I	Explains how to use Java EE 6 platform technologies and APIs to develop Java EE applications.
Message Queue Release Notes	Describes new features, compatibility issues, and existing bugs for Sun GlassFish Message Queue.

TABLE P–1 Books in the Enterprise Server Documentation Set *(Continued)*

Book Title	Description
Message Queue Developer's Guide for JMX Clients	Describes the application programming interface in Sun GlassFish Message Queue for programmatically configuring and monitoring Message Queue resources in conformance with the Java Management Extensions (JMX).
System Virtualization Support in Sun Java System Products	Summarizes Sun support for Sun Java System products when used in conjunction with system virtualization products and features.

Related Documentation

A Javadoc™ tool reference for packages that are provided with the Enterprise Server is located at `http://java.sun.com/javaee/6/docs/api/`.

Additionally, the following resources might be useful:

- The Java EE Specifications (`http://java.sun.com/javaee/technologies/index.jsp`)
- The Java EE Blueprints (`http://java.sun.com/reference/blueprints/index.html`)

For information about creating enterprise applications in the NetBeans™ Integrated Development Environment (IDE), see `http://www.netbeans.org/kb/60/index.html`.

For information about the Java DB for use with the Enterprise Server, see `http://developers.sun.com/javadb/`.

The sample applications demonstrate a broad range of Java EE technologies. The samples are bundled with the Java EE Software Development Kit (SDK).

Default Paths and File Names

The following table describes the default paths and file names that are used in this book.

TABLE P–2 Default Paths and File Names

Placeholder	Description	Default Value
as-install	Represents the base installation directory for the Enterprise Server or the Software Development Kit (SDK) of which the Enterprise Server is a part.	Solaris™ and Linux installations: *user's-home-directory*`/glassfishv3` Windows installations: *SystemDrive*`:\glassfishv3`

TABLE P–2 Default Paths and File Names *(Continued)*

Placeholder	Description	Default Value
tut-install	Represents the base installation directory for the Java EE Tutorial after you install the Java EE 6 SDK Preview and run the Update Tool.	*as-install*`/glassfish/docs/javaee-tutorial`
domain-root-dir	Represents the directory containing all Enterprise Server domains.	All installations: *as-install*`/glassfish/domains/`
domain-dir	Represents the directory for a domain. In configuration files, you might see *domain-dir* represented as follows: `${com.sun.aas.instanceRoot}`	*domain-root-dir*`/`*domain-dir*

Typographic Conventions

The following table describes the typographic changes that are used in this book.

TABLE P–3 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	A placeholder to be replaced with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized (note that some emphasized items appear bold online)	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file.

Symbol Conventions

The following table explains symbols that might be used in this book.

TABLE P–4 Symbol Conventions

Symbol	Description	Example	Meaning
`[ ]`	Contains optional arguments and command options.	`ls [-l]`	The `-l` option is not required.
`{ | }`	Contains a set of choices for a required command option.	`-d {y|n}`	The `-d` option requires that you use either the `y` argument or the `n` argument.
`${ }`	Indicates a variable reference.	`${com.sun.javaRoot}`	References the value of the `com.sun.javaRoot` variable.
-	Joins simultaneous multiple keystrokes.	Control-A	Press the Control key while you press the A key.
+	Joins consecutive multiple keystrokes.	Ctrl+A+N	Press the Control key, release it, and then press the subsequent keys.
→	Indicates menu item selection in a graphical user interface.	File → New → Templates	From the File menu, choose New. From the New submenu, choose Templates.

Documentation, Support, and Training

The Sun web site provides information about the following additional resources:

- Documentation (`http://www.sun.com/documentation/`)
- Support (`http://www.sun.com/support/`)
- Training (`http://www.sun.com/training/`)

PART I

Introduction

Part One introduces the platform, the tutorial, and the examples.

PART II

The Web Tier

Part Two explores the technologies in the web tier.

PART III

Web Services

Part Three explores web services.

PART IV

Enterprise Beans

Part Four explores Enterprise JavaBeans.

PART V

Contexts and Dependency Injection for the Java EE Platform

Part Five explores Contexts and Dependency Injection for the Java EE Platform.

PART VI

Persistence

Part Six explores the Java Persistence API.

PART VII

Security

Part Seven introduces basic security concepts and examples.

PART VIII

Java EE Supporting Technologies

Part Eight explores several technologies that support the Java EE platform.

◆ ◆ ◆ CHAPTER 32

Java Message Service Examples

This chapter provides examples that show how to use the JMS API in various kinds of Java EE applications. It covers the following topics:

The examples are in the following directory:

tut-install`/examples/jms/`

To build and run the examples, you will do the following:

1. Use NetBeans IDE or the Ant tool to compile and package the example.
2. Use the Ant tool to create resources.
3. Use NetBeans IDE or the Ant tool to deploy the example.
4. Use NetBeans IDE or the Ant tool to run the client.

Each example has a `build.xml` file that refers to files in the following directory:

tut-install`/examples/bp-project/`

See Chapter 17, "A Message-Driven Bean Example," for a simpler example of a Java EE application that uses the JMS API.

Writing Simple JMS Applications

This section shows how to create, package, and run simple JMS clients that are packaged as application clients and deployed to a Java EE server. The clients demonstrate the basic tasks that a JMS application must perform:

- Creating a connection and a session
- Creating message producers and consumers
- Sending and receiving messages

In a Java EE application, some of these tasks are performed, in whole or in part, by the container. If you learn about these tasks, you will have a good basis for understanding how a JMS application works on the Java EE platform.

This section covers the following topics:

Each example uses two clients: one that sends messages and one that receives them. You can run the clients in NetBeans IDE or in two terminal windows.

When you write a JMS client to run in a anterprise bean application, you use many of the same methods in much the same sequence as you do for an application client. However, there are some significant differences. "Using the JMS API in Java EE Applications" on page describes these differences, and this chapter provides examples that illustrate them.

The examples for this section are in the following directory:

tut-install`/examples/jms/simple/`

The examples are in the following four subdirectories:

```
producer
synchconsumer
asynchconsumer
messagebrowser
```

A Simple Example of Synchronous Message Receives

This section describes the sending and receiving clients in an example that uses the `receive` method to consume messages synchronously. This section then explains how to compile, package, and run the clients using the Enterprise Server.

The following sections describe the steps in creating and running the example:

Writing the Clients for the Synchronous Receive Example

The sending client, `producer/src/java/Producer.java`, performs the following steps:

1. Injects resources for a connection factory, queue, and topic:

```
@Resource(mappedName="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
@Resource(mappedName="jms/Queue")private static Queue queue;
@Resource(mappedName="jms/Topic")private static Topic topic;
```

2. Retrieves and verifies command-line arguments that specify the destination type and the number of arguments:

```
final int NUM_MSGS;
String destType = args[0];
System.out.println("Destination type is " + destType);
if ( ! ( destType.equals("queue") || destType.equals("topic") ) ) {
    System.err.println("Argument must be \"queue\" or " + "\"topic\"");
    System.exit(1);
}
if (args.length == 2){
    NUM_MSGS = (new Integer(args[1])).intValue();
}
else {
    NUM_MSGS = 1;
}
```

3. Assigns either the queue or topic to a destination object, based on the specified destination type:

```
Destination dest = null;
try {
    if (destType.equals("queue")) {
        dest = (Destination) queue;
    } else {
        dest = (Destination) topic;
    }
}
catch (Exception e) {
    System.err.println("Error setting destination: " + e.toString());
    e.printStackTrace();
```

```
    System.exit(1);
}
```

4. Creates a `Connection` and a `Session`:

```
Connection connection = connectionFactory.createConnection();
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

5. Creates a `MessageProducer` and a `TextMessage`:

```
MessageProducer producer = session.createProducer(dest);
TextMessage message = session.createTextMessage();
```

6. Sends one or more messages to the destination:

```
for (int i = 0; i < NUM_MSGS; i++) {
    message.setText("This is message " + (i + 1) + " from producer");
    System.out.println("Sending message: " + message.getText());
    producer.send(message);
}
```

7. Sends an empty control message to indicate the end of the message stream:

```
producer.send(session.createMessage());
```

   Sending an empty message of no specified type is a convenient way to indicate to the consumer that the final message has arrived.

8. Closes the connection in a `finally` block, automatically closing the session and `MessageProducer`:

```
} finally {
    if (connection != null) {
        try { connection.close(); }
        catch (JMSException e) { }
    }
}
```

The receiving client, `synchconsumer/src/java/SynchConsumer.java`, performs the following steps:

1. Injects resources for a connection factory, queue, and topic.
2. Assigns either the queue or topic to a destination object, based on the specified destination type.
3. Creates a `Connection` and a `Session`.
4. Creates a `MessageConsumer`:

```
consumer = session.createConsumer(dest);
```

5. Starts the connection, causing message delivery to begin:

```
connection.start();
```

6. Receives the messages sent to the destination until the end-of-message-stream control message is received:

```
while (true) {
    Message m = consumer.receive(1);
    if (m != null) {
        if (m instanceof TextMessage) {
            message = (TextMessage) m;
            System.out.println("Reading message: " + message.getText());
        } else {
            break;
        }
    }
}
```

   Because the control message is not a `TextMessage`, the receiving client terminates the `while` loop and stops receiving messages after the control message arrives.

7. Closes the connection in a `finally` block, automatically closing the session and `MessageConsumer`.

The `receive` method can be used in several ways to perform a synchronous receive. If you specify no arguments or an argument of `0`, the method blocks indefinitely until a message arrives:

```
Message m = consumer.receive();
Message m = consumer.receive(0);
```

For a simple client, this may not matter. But if you do not want your application to consume system resources unnecessarily, use a timed synchronous receive. Do one of the following:

- Call the `receive` method with a timeout argument greater than `0`:

  ```
  Message m = consumer.receive(1); // 1 millisecond
  ```

- Call the `receiveNoWait` method, which receives a message only if one is available:

  ```
  Message m = consumer.receiveNoWait();
  ```

The `SynchConsumer` client uses an indefinite `while` loop to receive messages, calling `receive` with a timeout argument. Calling `receiveNoWait` would have the same effect.

Starting the JMS Provider

When you use the Enterprise Server, your JMS provider is the Enterprise Server. Start the server as described in "Starting and Stopping the Enterprise Server" on page .

Creating JMS Administered Objects for the Synchronous Receive Example

Creating the JMS administered objects for this section involves the following:

- Creating a connection factory
- Creating two destination resources

If you built and ran the `SimpleMessage` example in Chapter 17, "A Message-Driven Bean Example," and did not delete the resources afterward, you need to create only the topic resource.

You can create these objects using the Ant tool. To create all the resources, do the following:

1. In a terminal window, go to the `producer` directory:

   ```
   cd producer
   ```

2. To create all the resources, type the following command:

   ```
   ant create-resources
   ```

   To create only the topic resource, type the following command:

   ```
   ant create-topic
   ```

These Ant targets use the `asadmin create-jms-resource` command to create the connection factory and the destination resources.

To verify that the resources have been created, use the following command:

```
asadmin list-jms-resources
```

The output looks like this:

```
jms/Queue
jms/Topic
jms/ConnectionFactory
Command list-jms-resources executed successfully.
```

Compiling and Packaging the Clients for the Synchronous Receive Example

To run these examples using the Enterprise Server, package each one in an application client JAR file. The application client JAR file requires a manifest file, located in the `src/conf` directory for each example, along with the `.class` file.

The `build.xml` file for each example contains Ant targets that compile and package the example. The targets place the `.class` file for the example in the `build/jar` directory. Then the targets use the `jar` command to package the class file and the manifest file in an application client JAR file.

To compile and package the `Producer` and `SynchConsumer` examples using NetBeans IDE, follow these steps:

1. In NetBeans IDE, choose Open Project from the File menu.
2. In the Open Project dialog, navigate to *tut-install*`/examples/jms/simple/`.
3. Select the `producer` folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. Right-click the project and choose Build.
7. In NetBeans IDE, choose Open Project from the File menu.
8. In the Open Project dialog, navigate to *tut-install*`/examples/jms/simple/`.
9. Select the `synchconsumer` folder.
10. Select the Open as Main Project check box.
11. Click Open Project.
12. Right-click the project and choose Build.

To compile and package the `Producer` and `SynchConsumer` examples using Ant, follow these steps:

1. In a terminal window, go to the `producer` directory:

   ```
   cd producer
   ```

2. Type the following command:

   ```
   ant
   ```

3. In a terminal window, go to the `synchconsumer` directory:

   ```
   cd ../synchconsumer
   ```

4. Type the following command:

   ```
   ant
   ```

The targets place the application client JAR file in the `dist` directory for each example.

Deploying and Running the Clients for the Synchronous Receive Example

To deploy and run the clients using NetBeans IDE, follow these steps.

1. Run the `Producer` example:
   a. Right-click the `producer` project and choose Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:

      ```
      queue 3
      ```

   d. Click OK.
   e. Right-click the project and choose Run.

The output of the program looks like this:

```
Destination type is queue
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
```

The messages are now in the queue, waiting to be received.

2. Now run the `SynchConsumer` example:
   a. Right-click the `synchconsumer` project and choose Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:

      ```
      queue
      ```

   d. Click OK.
   e. Right-click the project and choose Run.

      The output of the program looks like this:

      ```
      Destination type is queue
      Reading message: This is message 1 from producer
      Reading message: This is message 2 from producer
      Reading message: This is message 3 from producer
      ```

3. Now try running the programs in the opposite order. Right-click the `synchconsumer` project and choose Run.

   The Output pane displays the destination type and then appears to hang, waiting for messages.

4. Right-click the `producer` project and choose Run.

   The Output pane shows the output of both programs, in two different tabs.

5. Now run the `Producer` example using a topic instead of a queue.
   a. Right-click the `producer` project and choose Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:

      ```
      topic 3
      ```

   d. Click OK.
   e. Right-click the project and choose Run.

      The output looks like this:

      ```
      Destination type is topic
      Sending message: This is message 1 from producer
      Sending message: This is message 2 from producer
      Sending message: This is message 3 from producer
      ```

6. Now run the `SynchConsumer` example using the topic.
   a. Right-click the `synchconsumer` project and choose Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:

      ```
      topic
      ```

   d. Click OK.
   e. Right-click the project and choose Run.

      The result, however, is different. Because you are using a topic, messages that were sent before you started the consumer cannot be received. (See "Publish/Subscribe Messaging Domain" on page , for details.) Instead of receiving the messages, the program appears to hang.
7. Run the `Producer` example again. Right-click the `producer` project and choose Run.

   Now the `SynchConsumer` example receives the messages:

   ```
   Destination type is topic
   Reading message: This is message 1 from producer
   Reading message: This is message 2 from producer
   Reading message: This is message 3 from producer
   ```

You can also run the clients using the `appclient` command. The `build.xml` file for each project includes a target that deploys the client and then retrieves the client stubs that the `appclient` command uses. Each of the clients takes one or more command-line arguments: a destination type and, for `Producer`, a number of messages.

To run the clients using the `appclient` command, follow these steps:

1. In a terminal window, go to the `producer` directory:

   ```
   cd ../producer
   ```

2. Deploy the client JAR file to the Enterprise Server, then retrieve the client stubs:

   ```
   ant getclient
   ```

   Ignore the message that states that the application is deployed at a URL.
3. Run the `Producer` program, sending three messages to the queue:

   ```
   appclient -client client-jar/producerClient.jar queue 3
   ```

   The output of the program looks like this (along with some application client container output):

   ```
   Destination type is queue
   Sending message: This is message 1 from producer
   Sending message: This is message 2 from producer
   Sending message: This is message 3 from producer
   ```

The messages are now in the queue, waiting to be received.

4. In the same window, go to the `synchconsumer` directory:

   ```
   cd ../synchconsumer
   ```

5. Deploy the client JAR file to the Enterprise Server, then retrieve the client stubs:

   ```
   ant getclient
   ```

   Ignore the message that states that the application is deployed at a URL.

6. Run the `SynchConsumer` client, specifying the queue:

   ```
   appclient -client client-jar/synchconsumerClient.jar queue
   ```

   The output of the client looks like this (along with some application client container output):

   ```
   Destination type is queue
   Reading message: This is message 1 from producer
   Reading message: This is message 2 from producer
   Reading message: This is message 3 from producer
   ```

7. Now try running the clients in the opposite order. Run the `SynchConsumer` client. It displays the destination type and then appears to hang, waiting for messages.

   ```
   appclient -client client-jar/synchconsumerClient.jar queue
   ```

8. In a different terminal window, run the `Producer` client.

   ```
   cd tut-install/examples/jms/simple/producer
   appclient -client client-jar/producerClient.jar queue 3
   ```

   When the messages have been sent, the `SynchConsumer` client receives them and exits.

9. Now run the `Producer` client using a topic instead of a queue:

   ```
   appclient -client client-jar/producerClient.jar topic 3
   ```

   The output of the client looks like this (along with some application client container output):

   ```
   Destination type is topic
   Sending message: This is message 1 from producer
   Sending message: This is message 2 from producer
   Sending message: This is message 3 from producer
   ```

10. Now run the `SynchConsumer` client using the topic:

    ```
    appclient -client client-jar/synchconsumerClient.jar topic
    ```

    The result, however, is different. Because you are using a topic, messages that were sent before you started the consumer cannot be received. (See "Publish/Subscribe Messaging Domain" on page , for details.) Instead of receiving the messages, the client appears to hang.

11. Run the `Producer` client again. Now the `SynchConsumer` client receives the messages (along with some application client container output):

```
Destination type is topic
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
```

Because the examples use the common interfaces, you can run them using either a queue or a topic.

A Simple Example of Asynchronous Message Consumption

This section describes the receiving clients in an example that uses a message listener to consume messages asynchronously. This section then explains how to compile and run the clients using the Enterprise Server.

The following sections describe the steps in creating and running the example:

- "Writing the Clients for the Asynchronous Receive Example" on page 37
- "Compiling and Packaging the `AsynchConsumer` Client" on page 38
- "Running the Clients for the Asynchronous Receive Example" on page 39

Writing the Clients for the Asynchronous Receive Example

The sending client is `producer/src/java/Producer.java`, the same client used in the example in "A Simple Example of Synchronous Message Receives" on page 28.

An asynchronous consumer normally runs indefinitely. This one runs until the user types the letter `q` or `Q` to stop the client.

The receiving client, `asynchconsumer/src/java/AsynchConsumer.java`, performs the following steps:

1. Injects resources for a connection factory, queue, and topic.
2. Assigns either the queue or topic to a destination object, based on the specified destination type.
3. Creates a `Connection` and a `Session`.
4. Creates a `MessageConsumer`.
5. Creates an instance of the `TextListener` class and registers it as the message listener for the `MessageConsumer`:

```
listener = new TextListener();consumer.setMessageListener(listener);
```

6. Starts the connection, causing message delivery to begin.

7. Listens for the messages published to the destination, stopping when the user types the character q or Q:

```
System.out.println("To end program, type Q or q, " + "then <return>");
inputStreamReader = new InputStreamReader(System.in);
while (!((answer == 'q') || (answer == 'Q'))) {
    try {
        answer = (char) inputStreamReader.read();
    } catch (IOException e) {
        System.out.println("I/O exception: " + e.toString());
    }
}
```

8. Closes the connection, which automatically closes the session and `MessageConsumer`.

The message listener, `asynchconsumer/src/java/TextListener.java`, follows these steps:

1. When a message arrives, the `onMessage` method is called automatically.
2. The `onMessage` method converts the incoming message to a `TextMessage` and displays its content. If the message is not a text message, it reports this fact:

```
public void onMessage(Message message) {
    TextMessage msg = null;
    try {
        if (message instanceof TextMessage) {
            msg = (TextMessage) message;
             System.out.println("Reading message: " + msg.getText());
        } else {
             System.out.println("Message is not a " + "TextMessage");
        }
    } catch (JMSException e) {
        System.out.println("JMSException in onMessage(): " + e.toString());
    } catch (Throwable t) {
        System.out.println("Exception in onMessage():" + t.getMessage());
    }
}
```

You will use the connection factory and destinations you created in "Creating JMS Administered Objects for the Synchronous Receive Example" on page 32.

Compiling and Packaging the `AsynchConsumer` Client

To compile and package the `AsynchConsumer` example using NetBeans IDE, follow these steps:

1. In NetBeans IDE, choose Open Project from the File menu.
2. In the Open Project dialog, navigate to *tut-install*`/examples/jms/simple/`.
3. Select the `asynchconsumer` folder.
4. Select the Open as Main Project check box.
5. Click Open Project.

6. Right-click the project and choose Build.

To compile and package the `AsynchConsumer` example using Ant, follow these steps:

1. In a terminal window, go to the `asynchconsumer` directory:

   ```
   cd ../asynchconsumer
   ```

2. Type the following command:

   ```
   ant
   ```

The targets package both the main class and the message listener class in the JAR file and place the file in the `dist` directory for the example.

Running the Clients for the Asynchronous Receive Example

To run the clients using NetBeans IDE, follow these steps.

1. Run the `AsynchConsumer` example:
   a. Right-click the `asynchconsumer` project and choose Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:

      ```
      topic
      ```

   d. Click OK.
   e. Right-click the project and choose Run.

      The client displays the following lines and appears to hang:

      ```
      Destination type is topic
      To end program, type Q or q, then <return>
      ```

2. Now run the `Producer` example:
   a. Right-click the `producer` project and choose Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:

      ```
      topic 3
      ```

   d. Click OK.
   e. Right-click the project and choose Run.

      The output of the client looks like this:

      ```
      Destination type is topic
      Sending message: This is message 1 from producer
      Sending message: This is message 2 from producer
      Sending message: This is message 3 from producer
      ```

      In the other window, the `AsynchConsumer` client displays the following:

```
Destination type is topic
To end program, type Q or q, then <return>
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Message is not a TextMessage
```

The last line appears because the client has received the non-text control message sent by the `Producer` client.

3. Type `Q` or `q` in the Output window and press Return to stop the client.
4. Now run the clients using a queue. In this case, as with the synchronous example, you can run the `Producer` client first, because there is no timing dependency between the sender and receiver.
   a. Right-click the `producer` project and choose Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:

      ```
      queue 3
      ```

   d. Click OK.
   e. Right-click the project and choose Run.

      The output of the client looks like this:

      ```
      Destination type is queue
      Sending message: This is message 1 from producer
      Sending message: This is message 2 from producer
      Sending message: This is message 3 from producer
      ```

5. Run the `AsynchConsumer` client.
   a. Right-click the `asynchconsumer` project and choose Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:

      ```
      queue
      ```

   d. Click OK.
   e. Right-click the project and choose Run.

      The output of the client looks like this:

      ```
      Destination type is queue
      To end program, type Q or q, then <return>
      Reading message: This is message 1 from producer
      Reading message: This is message 2 from producer
      Reading message: This is message 3 from producer
      Message is not a TextMessage
      ```

6. Type `Q` or `q` in the Output window and press Return to stop the client.

To run the clients using the `appclient` command, follow these steps:

1. Deploy the client JAR file to the Enterprise Server, then retrieve the client stubs:

   ```
   ant getclient
   ```

   Ignore the message that states that the application is deployed at a URL.
2. Run the `AsynchConsumer` client, specifying the `topic` destination type.

   ```
   appclient -client client-jar/asynchconsumerClient.jar topic
   ```

   The client displays the following lines (along with some application client container output) and appears to hang:

   ```
   Destination type is topic
   To end program, type Q or q, then <return>
   ```
3. In the terminal window where you ran the `Producer` client previously, run the client again, sending three messages. The command looks like this:

   ```
   appclient -client client-jar/producerClient.jar topic 3
   ```

   The output of the client looks like this (along with some application client container output):

   ```
   Destination type is topic
   Sending message: This is message 1 from producer
   Sending message: This is message 2 from producer
   Sending message: This is message 3 from producer
   ```

   In the other window, the `AsynchConsumer` client displays the following (along with some application client container output):

   ```
   Destination type is topic
   To end program, type Q or q, then <return>
   Reading message: This is message 1 from producer
   Reading message: This is message 2 from producer
   Reading message: This is message 3 from producer
   Message is not a TextMessage
   ```

   The last line appears because the client has received the non-text control message sent by the `Producer` client.
4. Type `Q` or `q` and press Return to stop the client.
5. Now run the clients using a queue. In this case, as with the synchronous example, you can run the `Producer` client first, because there is no timing dependency between the sender and receiver:

   ```
   appclient -client client-jar/producerClient.jar queue 3
   ```

   The output of the client looks like this:

```
Destination type is queue
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
```

6. Run the `AsynchConsumer` client:

```
appclient -client client-jar/asynchconsumerClient.jar queue
```

   The output of the client looks like this (along with some application client container output):

```
Destination type is queue
To end program, type Q or q, then <return>
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Message is not a TextMessage
```

7. Type `Q` or `q` to stop the client.

A Simple Example of Browsing Messages in a Queue

This section describes an example that creates a `QueueBrowser` object to examine messages on a queue, as described in "JMS Queue Browsers" on page . This section then explains how to compile, package, and run the example using the Enterprise Server.

The following sections describe the steps in creating and running the example:

Writing the Client for the Queue Browser Example

To create a `QueueBrowser` for a queue, you call the `Session.createBrowser` method with the queue as the argument. You obtain the messages in the queue as an `Enumeration` object. You can then iterate through the `Enumeration` object and display the contents of each message.

The `messagebrowser/src/java/MessageBrowser.java` client performs the following steps:

1. Injects resources for a connection factory and a queue.
2. Creates a `Connection` and a `Session`.
3. Creates a `QueueBrowser`:

```
QueueBrowser browser = session.createBrowser(queue);
```

4. Retrieves the `Enumeration` that contains the messages:

```
Enumeration msgs = browser.getEnumeration();
```

5. Verifies that the `Enumeration` contains messages, then displays the contents of the messages:

```
if ( !msgs.hasMoreElements() ) {
    System.out.println("No messages in queue");
} else {
    while (msgs.hasMoreElements()) {
        Message tempMsg = (Message)msgs.nextElement();
        System.out.println("Message: " + tempMsg);
    }
}
```

6. Closes the connection, which automatically closes the session and `QueueBrowser`.

The format in which the message contents appear is implementation-specific. In the Enterprise Server, the message format looks like this:

```
Message contents:
Text:   This is message 3 from producer
Class:                  com.sun.messaging.jmq.jmsclient.TextMessageImpl
getJMSMessageID():      ID:14-129.148.71.199(f9:86:a2:d5:46:9b)-40814-1255980521747
getJMSTimestamp():      1129061034355
getJMSCorrelationID():  null
JMSReplyTo:             null
JMSDestination:         PhysicalQueue
getJMSDeliveryMode():   PERSISTENT
getJMSRedelivered():    false
getJMSType():           null
getJMSExpiration():     0
getJMSPriority():       4
Properties:             null
```

You will use the connection factory and queue you created in "Creating JMS Administered Objects for the Synchronous Receive Example" on page 32.

Compiling and Packaging the `MessageBrowser` Client

To compile and package the `MessageBrowser` example using NetBeans IDE, follow these steps:

1. In NetBeans IDE, choose Open Project from the File menu.
2. In the Open Project dialog, navigate to *tut-install*`/examples/jms/simple/`.
3. Select the `messagebrowser` folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. Right-click the project and choose Build.

To compile and package the `MessageBrowser` example using Ant, follow these steps:

1. In a terminal window, go to the `messagebrowser` directory.

```
cd ../messagebrowser
```

2. Type the following command:

   ```
   ant
   ```

The targets place the application client JAR file in the `dist` directory for the example.

You also need the `Producer` example to send the message to the queue, and one of the consumer clients to consume the messages after you inspect them. If you did not do so already, package these examples.

Running the Clients for the Queue Browser Example

To run the clients using NetBeans IDE, follow these steps.

1. Run the `Producer` client, sending one message to the queue:
   a. Right-click the `producer` project and choose Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:

      ```
      queue
      ```

   d. Click OK.
   e. Right-click the project and choose Run.

      The output of the client looks like this:

      ```
      Destination type is queue
      Sending message: This is message 1 from producer
      ```

2. Run the `MessageBrowser` client. Right-click the `messagebrowser` project and choose Run.

   The output of the client looks like this:

   ```
   Message:
   Text: This is message 1 from producer
   Class: com.sun.messaging.jmq.jmsclient.TextMessageImpl
   getJMSMessageID(): ID:12-129.148.71.199(8c:34:4a:1a:1b:b8)-40883-1255980521747
   getJMSTimestamp(): 1129062957611
   getJMSCorrelationID(): null
   JMSReplyTo: null
   JMSDestination: PhysicalQueue
   getJMSDeliveryMode(): PERSISTENT
   getJMSRedelivered(): false
   getJMSType(): null
   getJMSExpiration(): 0
   getJMSPriority(): 4
   Properties: null
   Message:
   Class: com.sun.messaging.jmq.jmsclient.MessageImpl
   getJMSMessageID(): ID:13-129.148.71.199(8c:34:4a:1a:1b:b8)-40883-1255980521747
   getJMSTimestamp(): 1129062957616
   ```

```
getJMSCorrelationID(): null
JMSReplyTo: null
JMSDestination: PhysicalQueue
getJMSDeliveryMode(): PERSISTENT
getJMSRedelivered(): false
getJMSType(): null
getJMSExpiration(): 0
getJMSPriority(): 4
Properties: null
```

3. The first message is the `TextMessage`, and the second is the non-text control message.
4. Run the `SynchConsumer` client to consume the messages.
   a. Right-click the `synchconsumer` project and choose Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:

      ```
      queue
      ```

   d. Click OK.
   e. Right-click the project and choose Run.

      The output of the client looks like this:

      ```
      Destination type is queue
      Reading message: This is message 1 from producer
      ```

To run the clients using the `appclient` command, follow these steps. You may want to use two terminal windows.

1. Go to the `producer` directory.
2. Run the `Producer` client, sending one message to the queue:

   ```
   appclient -client client-jar/producerClient.jar queue
   ```

   The output of the client looks like this (along with some application client container output):

   ```
   Destination type is queue
   Sending message: This is message 1 from producer
   ```

3. Go to the `messagebrowser` directory.
4. Deploy the client JAR file to the Enterprise Server, then retrieve the client stubs:

   ```
   ant getclient
   ```

   Ignore the message that states that the application is deployed at a URL.

5. Run the `MessageBrowser` client:

   ```
   appclient -client client-jar/messagebrowserClient.jar
   ```

The output of the client looks like this (along with some application client container output):

```
Message:
Text: This is message 1 from producer
Class: com.sun.messaging.jmq.jmsclient.TextMessageImpl
getJMSMessageID(): ID:12-129.148.71.199(8c:34:4a:1a:1b:b8)-40883-1255980521747
getJMSTimestamp(): 1255980521747
getJMSCorrelationID(): null
JMSReplyTo: null
JMSDestination: PhysicalQueue
getJMSDeliveryMode(): PERSISTENT
getJMSRedelivered(): false
getJMSType(): null
getJMSExpiration(): 0
getJMSPriority(): 4
Properties: null
Message:
Class: com.sun.messaging.jmq.jmsclient.MessageImpl
getJMSMessageID(): ID:13-129.148.71.199(8c:34:4a:1a:1b:b8)-40883-1255980521767
getJMSTimestamp(): 1255980521767
getJMSCorrelationID(): null
JMSReplyTo: null
JMSDestination: PhysicalQueue
getJMSDeliveryMode(): PERSISTENT
getJMSRedelivered(): false
getJMSType(): null
getJMSExpiration(): 0
getJMSPriority(): 4
Properties: null
```

The first message is the `TextMessage`, and the second is the non-text control message.

6. Go to the `synchconsumer` directory.
7. Run the `SynchConsumer` client to consume the messages:

   ```
   appclient -client client-jar/synchconsumerClient.jar queue
   ```

   The output of the client looks like this (along with some application client container output):

   ```
   Destination type is queue
   Reading message: This is message 1 from producer
   ```

Running JMS Clients on Multiple Systems

JMS clients that use the Enterprise Server can exchange messages with each other when they are running on different systems in a network. The systems must be visible to each other by name

(the UNIX host name or the Microsoft Windows computer name) and must both be running the Enterprise Server. You do not have to install the tutorial examples on both systems; you can use the examples installed on one system if you can access its file system from the other system.

Note – Any mechanism for exchanging messages between systems is specific to the Java EE server implementation. This tutorial describes how to use the Enterprise Server for this purpose.

Suppose that you want to run the `Producer` client on one system, `earth`, and the `SynchConsumer` client on another system, `jupiter`. Before you can do so, you need to perform these tasks:

- Create two new connection factories
- Edit the source code for the two examples
- Recompile and repackage the examples

Note – A limitation in the JMS provider in the Enterprise Server may cause a runtime failure to create a connection to systems that use the Dynamic Host Configuration Protocol (DHCP) to obtain an IP address. You can, however, create a connection *from* a system that uses DHCP *to* a system that does not use DHCP. In the examples in this tutorial, `earth` can be a system that uses DHCP, and `jupiter` can be a system that does not use DHCP.

Before you begin, start the server on both systems:

1. Start the Enterprise Server on `earth`.
2. Start the Enterprise Server on `jupiter`.

Creating Administered Objects for Multiple Systems

To run these clients, you must do the following:

- Create a new connection factory on both `earth` and `jupiter`
- Create a destination resource on both `earth` and `jupiter`

You do not have to install the tutorial examples on both systems, but you must be able to access the filesystem where it is installed. You may find it more convenient to install the tutorial examples on both systems if the two systems use different operating systems (for example, Windows and Solaris). Otherwise you will have to edit the file *tut-install*`/examples/bp-project/build.properties` and change the location of the `javaee.home` property each time you build or run a client on a different system.

To create a new connection factory on `jupiter`, perform these steps:

1. From a command shell on `jupiter`, go to the directory *tut-install*`/examples/jms/simple/producer/`.
2. Type the following command:

   ```
   ant create-local-factory
   ```

The `create-local-factory` target, defined in the `build.xml` file for the `Producer` example, creates a connection factory named `jms/JupiterConnectionFactory`.

To create a new connection factory on `earth` that points to the connection factory on `jupiter`, perform these steps:

1. From a command shell on `earth`, go to the directory *tut-install*`/examples/jms/simple/producer/`.
2. Type the following command:

   ```
   ant create-remote-factory -Dsys=remote-system-name
   ```

   Replace *remote-system-name* with the actual name of the remote system.

The `create-remote-factory` target, defined in the `build.xml` file for the `Producer` example, also creates a connection factory named `jms/JupiterConnectionFactory`. In addition, it sets the `AddressList` property for this factory to the name of the remote system.

If you have already been working on either `earth` or `jupiter`, you have the queue and topic on one system. On the system that does not have the queue and topic, type the following command:

```
ant create-resources
```

When you run the clients, they will work as shown in Figure 32–1. The client run on `earth` needs the queue on `earth` only in order that the resource injection will succeed. The connection, session, and message producer are all created on `jupiter` using the connection factory that points to `jupiter`. The messages sent from `earth` will be received on `jupiter`.

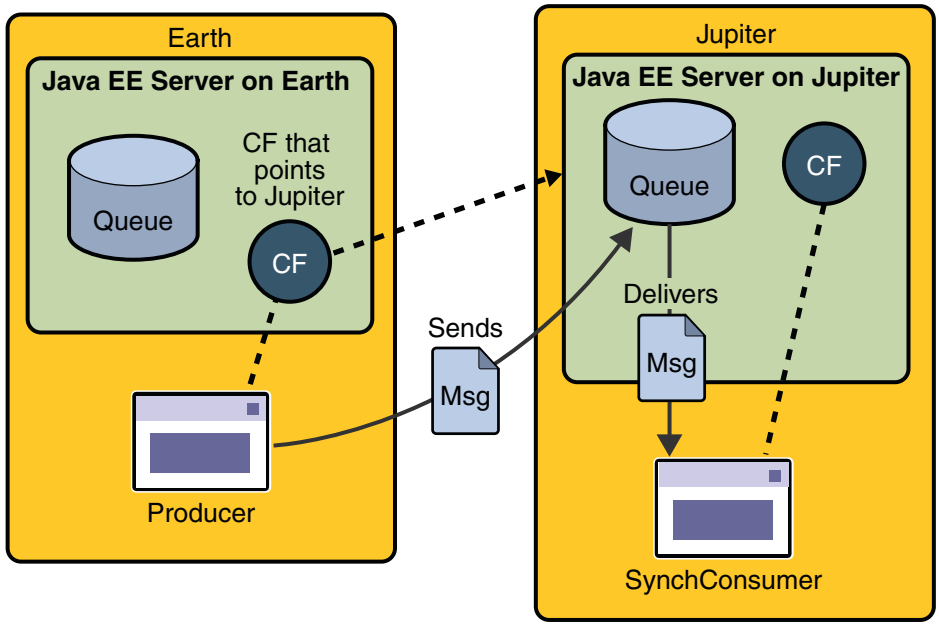

FIGURE 32–1 Sending Messages from One System to Another

Editing, Recompiling, Repackaging, and Running the Clients

These steps assume that you have the tutorial installed on only one of the two systems you are using and that you are able to access the file system of `jupiter` from `earth` or vice versa.

After you create the connection factories, edit the source files to specify the new connection factory. Then recompile, repackage, and run the clients. Perform the following steps:

1. Open the following file in a text editor or in NetBeans IDE:

   tut-install`/examples/jms/simple/producer/src/java/Producer.java`

2. Find the following line:

   ```
   @Resource(mappedName="jms/ConnectionFactory")
   ```

3. Change the line to the following:

   ```
   @Resource(mappedName="jms/JupiterConnectionFactory")
   ```

4. Recompile and repackage the `Producer` example on `earth`.

   If you are using NetBeans IDE, right-click the `producer` project and choose Clean and Build.

   If you are using Ant, type the following:

   ```
   ant
   ```

5. Open the following file in a text editor:

tut-install/examples/jms/simple/synchconsumer/src/java/SynchConsumer.java

6. Repeat steps 2 and 3.
7. Recompile and repackage the SynchConsumer example on jupiter.

   If you are using NetBeans IDE, right-click the synchconsumer project and choose Clean and Build.

   If you are using Ant, go to the synchconsumer directory and type:

   ```
   ant
   ```

8. On earth, deploy and run Producer. If you are using NetBeans IDE on earth, perform these steps:
   a. Right-click the producer project and choose Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:

      ```
      queue 3
      ```

   d. Click OK.
   e. Right-click the project and choose Run.

   If you are using appclient, perform these steps:

   a. On earth, from the producer directory, deploy the client JAR file to the Enterprise Server, then retrieve the client stubs:

      ```
      ant getclient
      ```

      Ignore the message that states that the application is deployed at a URL.

   b. Type the following:

      ```
      appclient -client client-jar/producerClient.jar queue 3
      ```

9. On jupiter, run SynchConsumer. If you are using NetBeans IDE on jupiter, perform these steps:
   a. Right-click the synchconsumer project and choose Properties.
   b. Select Run from the Categories tree.
   c. In the Arguments field, type the following:

      ```
      queue
      ```

   d. Click OK.
   e. Right-click the project and choose Run.

   If you are using appclient, perform these steps:

   a. From the synchconsumer directory, deploy the client JAR file to the Enterprise Server, then retrieve the client stubs:

      ```
      ant getclient
      ```

Ignore the message that states that the application is deployed at a URL.

b. Type the following:

```
appclient -client client-jar/synchconsumerClient.jar queue
```

For examples showing how to deploy applications on two different systems, see "An Application Example That Consumes Messages from a Remote Server" on page 76 and "An Application Example That Deploys a Message-Driven Bean on Two Servers" on page 82.

Deleting the Connection Factory and Stopping the Server

You will need the connection factory `jms/JupiterConnectionFactory` in "An Application Example That Consumes Messages from a Remote Server" on page 76 and "An Application Example That Deploys a Message-Driven Bean on Two Servers" on page 82. However, if you wish to delete it, go to the `producer` directory and type the following command:

```
ant delete-remote-factory
```

Remember to delete the connection factory on both systems.

You can also use Ant targets in the `producer/build.xml` file to delete the destinations and connection factories you created in "Creating JMS Administered Objects for the Synchronous Receive Example" on page 32. However, it is recommended that you keep them, because they will be used in most of the examples later in this chapter. After you have created them, they will be available whenever you restart the Enterprise Server.

To delete the class and JAR files for each client using NetBeans IDE, right-click each project and choose Clean.

To undeploy each client using Ant, go to the directory for the client and type the following:

```
ant undeploy
```

To delete the class and JAR files for each client using Ant, type the following:

```
ant clean
```

You can also stop the Enterprise Server, but you will need it to run the sample clients in the next section.

Writing Robust JMS Applications

The following examples show how to use some of the more advanced features of the JMS API.

A Message Acknowledgment Example

The `AckEquivExample.java` client shows how both of the following two scenarios ensure that a message will not be acknowledged until processing of it is complete:

- Using an asynchronous message consumer (a message listener) in an `AUTO_ACKNOWLEDGE` session
- Using a synchronous receiver in a `CLIENT_ACKNOWLEDGE` session

With a message listener, the automatic acknowledgment happens when the `onMessage` method returns (that is, after message processing has finished). With a synchronous receiver, the client acknowledges the message after processing is complete. If you use `AUTO_ACKNOWLEDGE` with a synchronous receive, the acknowledgment happens immediately after the `receive` call; if any subsequent processing steps fail, the message cannot be redelivered.

The example is in the following directory:

tut-install`/examples/jms/advanced/ackequivexample/src/java/`

The example contains an `AsynchSubscriber` class with a `TextListener` class, a `MultiplePublisher` class, a `SynchReceiver` class, a `SynchSender` class, a `main` method, and a method that runs the other classes' threads.

The example uses the following objects:

- `jms/ConnectionFactory`, `jms/Queue`, and `jms/Topic`: resources that you created in "Creating JMS Administered Objects for the Synchronous Receive Example" on page 32
- `jms/ControlQueue`: an additional queue
- `jms/DurableConnectionFactory`: a connection factory with a client ID (see "Creating Durable Subscriptions" on page , for more information)

To create the new queue and connection factory, you can use Ant targets defined in the file *tut-install*`/examples/jms/advanced/ackequivexample/build.xml`.

To run this example, follow these steps:

1. In a terminal window, go to the following directory:

   tut-install`/examples/jms/advanced/ackequivexample/`

2. To create the objects needed in this example, type the following commands:

   ```
   ant create-control-queue
   ant create-durable-cf
   ```

3. To compile and package the client using NetBeans IDE, follow these steps:
   a. In NetBeans IDE, choose Open Project from the File menu.
   b. In the Open Project dialog, navigate to *tut-install*`/examples/jms/advanced/`.
   c. Select the `ackequivexample` folder.
   d. Select the Open as Main Project check box.
   e. Click Open Project.
   f. Right-click the project and choose Build.

   To compile and package the client using Ant, type the following command:

   ```
   ant
   ```

4. To run the client using NetBeans IDE, right-click the `ackequivexample` project and choose Run.

   To run the client from the command line, follow these steps:

   a. Deploy the client JAR file to the Enterprise Server, then retrieve the client stubs:

      ```
      ant getclient
      ```

      Ignore the message that states that the application is deployed at a URL.

   b. Type the following command:

      ```
      appclient -client client-jar/ackequivexampleClient.jar
      ```

The client output looks something like this (along with some application client container output):

```
Queue name is jms/ControlQueue
Queue name is jms/Queue
Topic name is jms/Topic
Connection factory name is jms/DurableConnectionFactory
  SENDER: Created client-acknowledge session
  SENDER: Sending message: Here is a client-acknowledge message
  RECEIVER: Created client-acknowledge session
  RECEIVER: Processing message: Here is a client-acknowledge message
  RECEIVER: Now I'll acknowledge the message
SUBSCRIBER: Created auto-acknowledge session
SUBSCRIBER: Sending synchronize message to control queue
PUBLISHER: Created auto-acknowledge session
PUBLISHER: Receiving synchronize messages from control queue; count = 1
PUBLISHER: Received synchronize message;  expect 0 more
PUBLISHER: Publishing message: Here is an auto-acknowledge message 1
PUBLISHER: Publishing message: Here is an auto-acknowledge message 2
SUBSCRIBER: Processing message: Here is an auto-acknowledge message 1
PUBLISHER: Publishing message: Here is an auto-acknowledge message 3
SUBSCRIBER: Processing message: Here is an auto-acknowledge message 2
SUBSCRIBER: Processing message: Here is an auto-acknowledge message 3
```

After you run the client, you can delete the destination resource `jms/ControlQueue`. Go to the directory *tut-install*`/examples/jms/advanced/ackequivexample/` and type the following command:

```
ant delete-control-queue
```

You will need the other resources for other examples.

To delete the class and JAR files for the client using NetBeans IDE, right-click the project and choose Clean.

To undeploy the client using Ant, type the following:

```
ant undeploy
```

To delete the class and JAR files for the client using Ant, type the following:

```
ant clean
```

A Durable Subscription Example

The `DurableSubscriberExample.java` example shows how durable subscriptions work. It demonstrates that a durable subscription is active even when the subscriber is not active. The example contains a `DurableSubscriber` class, a `MultiplePublisher` class, a `main` method, and a method that instantiates the classes and calls their methods in sequence.

The example is in the following directory:

tut-install`/examples/jms/advanced/durablesubscriberexample/src/java/`

The example begins in the same way as any publish/subscribe client: The subscriber starts, the publisher publishes some messages, and the subscriber receives them. At this point, the subscriber closes itself. The publisher then publishes some messages while the subscriber is not active. The subscriber then restarts and receives the messages.

Before you run this example, compile and package the source file and create a connection factory that has a client ID. Perform the following steps:

1. To compile and package the client using NetBeans IDE, follow these steps:
    a. In NetBeans IDE, choose Open Project from the File menu.
    b. In the Open Project dialog, navigate to *tut-install*`/examples/jms/advanced/`.
    c. Select the `durablesubscriberexample` folder.
    d. Select the Open as Main Project check box.
    e. Click Open Project.
    f. Right-click the project and choose Build.

    To compile and package the client using Ant, follow these steps:

a. Go to the following directory:

   tut-install`/examples/jms/advanced/durablesubscriberexample/`

b. Type the following command:

   `ant`

2. If you did not do so for "A Message Acknowledgment Example" on page 52, create a connection factory named `jms/DurableConnectionFactory`:

   `ant create-durable-cf`

To run the client using NetBeans IDE, right-click the `durablesubscriberexample` project and choose Run.

To run the client from the command line, follow these steps:

1. Deploy the client JAR file to the Enterprise Server, then retrieve the client stubs:

   `ant getclient`

   Ignore the message that states that the application is deployed at a URL.

2. Type the following command:

   `appclient -client client-jar/durablesubscriberexampleClient.jar`

The output looks something like this (along with some application client container output):

```
Connection factory without client ID is jms/ConnectionFactory
Connection factory with client ID is jms/DurableConnectionFactory
Topic name is jms/Topic
Starting subscriber
PUBLISHER: Publishing message: Here is a message 1
SUBSCRIBER: Reading message: Here is a message 1
PUBLISHER: Publishing message: Here is a message 2
SUBSCRIBER: Reading message: Here is a message 2
PUBLISHER: Publishing message: Here is a message 3
SUBSCRIBER: Reading message: Here is a message 3
Closing subscriber
PUBLISHER: Publishing message: Here is a message 4
PUBLISHER: Publishing message: Here is a message 5
PUBLISHER: Publishing message: Here is a message 6
Starting subscriber
SUBSCRIBER: Reading message: Here is a message 4
SUBSCRIBER: Reading message: Here is a message 5
SUBSCRIBER: Reading message: Here is a message 6
Closing subscriber
Unsubscribing from durable subscription
```

After you run the client, you can delete the connection factory `jms/DurableConnectionFactory`. Go to the directory *tut-install*`/examples/jms/advanced/durablesubscriberexample/` and type the following command:

```
ant delete-durable-cf
```

To delete the class and JAR files for the client using NetBeans IDE, right-click the project and choose Clean.

To undeploy the client using Ant, type the following:

```
ant undeploy
```

To delete the class and JAR files for the client using Ant, type the following:

```
ant clean
```

A Local Transaction Example

The `TransactedExample.java` example demonstrates the use of transactions in a JMS client application. The example is in the following directory:

tut-install`/examples/jms/advanced/transactedexample/src/java/`

This example shows how to use a queue and a topic in a single transaction as well as how to pass a session to a message listener's constructor function. The example represents a highly simplified e-commerce application in which the following things happen.

1. A retailer sends a `MapMessage` to the vendor order queue, ordering a quantity of computers, and waits for the vendor's reply:

   ```
   producer = session.createProducer(vendorOrderQueue);
   outMessage = session.createMapMessage();
   outMessage.setString("Item", "Computer(s)");
   outMessage.setInt("Quantity", quantity);
   outMessage.setJMSReplyTo(retailerConfirmQueue);
   producer.send(outMessage);
   System.out.println("Retailer: ordered " + quantity + " computer(s)");
   orderConfirmReceiver = session.createConsumer(retailerConfirmQueue);
   connection.start();
   ```

2. The vendor receives the retailer's order message and sends an order message to the supplier order topic in one transaction. This JMS transaction uses a single session, so you can combine a receive from a queue with a send to a topic. Here is the code that uses the same session to create a consumer for a queue and a producer for a topic:

   ```
   vendorOrderReceiver = session.createConsumer(vendorOrderQueue);
   supplierOrderProducer = session.createProducer(supplierOrderTopic);
   ```

The following code receives the incoming message, sends an outgoing message, and commits the session. The message processing has been removed to keep the sequence simple:

```
inMessage = vendorOrderReceiver.receive();
// Process the incoming message and format the outgoing
// message
...
supplierOrderProducer.send(orderMessage);
...
session.commit();
```

3. Each supplier receives the order from the order topic, checks its inventory, and then sends the items ordered to the queue named in the order message's `JMSReplyTo` field. If it does not have enough in stock, the supplier sends what it has. The synchronous receive from the topic and the send to the queue take place in one JMS transaction.

```
receiver = session.createConsumer(orderTopic);
...
inMessage = receiver.receive();
if (inMessage instanceof MapMessage) {
    orderMessage = (MapMessage) inMessage;
}
// Process message
MessageProducer producer =
    session.createProducer((Queue) orderMessage.getJMSReplyTo());
outMessage = session.createMapMessage();
// Add content to message
producer.send(outMessage);
// Display message contentssession.commit();
```

4. The vendor receives the replies from the suppliers from its confirmation queue and updates the state of the order. Messages are processed by an asynchronous message listener; this step shows the use of JMS transactions with a message listener.

```
MapMessage component = (MapMessage) message;
...
orderNumber = component.getInt("VendorOrderNumber");
Order order = Order.getOrder(orderNumber).processSubOrder(component);
session.commit();
```

5. When all outstanding replies are processed for a given order, the vendor message listener sends a message notifying the retailer whether it can fulfill the order.

```
Queue replyQueue = (Queue) order.order.getJMSReplyTo();
MessageProducer producer = session.createProducer(replyQueue);
MapMessage retailerConfirmMessage = session.createMapMessage();
// Format the message
producer.send(retailerConfirmMessage);
session.commit();
```

6. The retailer receives the message from the vendor:

```
inMessage = (MapMessage) orderConfirmReceiver.receive();
```

Figure 32–2 illustrates these steps.

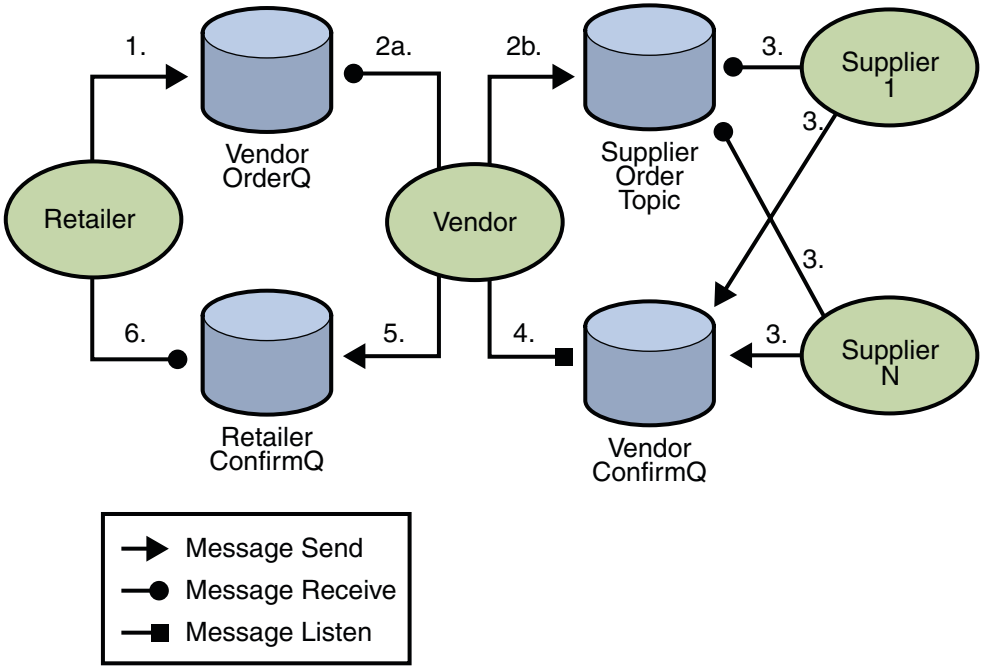

FIGURE 32–2 Transactions: JMS Client Example

The example contains five classes: `GenericSupplier`, `Order`, `Retailer`, `Vendor`, and `VendorMessageListener`. The example also contains a `main` method and a method that runs the threads of the `Retailer`, `Vendor`, and two supplier classes.

All the messages use the `MapMessage` message type. Synchronous receives are used for all message reception except for the case of the vendor processing the replies of the suppliers. These replies are processed asynchronously and demonstrate how to use transactions within a message listener.

At random intervals, the `Vendor` class throws an exception to simulate a database problem and cause a rollback.

All classes except `Retailer` use transacted sessions.

The example uses three queues named `jms/AQueue`, `jms/BQueue`, and `jms/CQueue`, and one topic named `jms/OTopic`.

Before you run the example, do the following:

1. In a terminal window, go to the following directory:

   tut-install`/examples/jms/advanced/transactedexample/`

2. Create the necessary resources using the following command:

   ```
   ant create-resources
   ```

   This command creates three destination resources with the names `jms/AQueue`, `jms/BQueue`, and `jms/CQueue`, all of type `javax.jms.Queue`, and one destination resource with the name `jms/OTopic`, of type `javax.jms.Topic`.

3. To compile and package the client using NetBeans IDE, follow these steps:

   a. In NetBeans IDE, choose Open Project from the File menu.
   b. In the Open Project dialog, navigate to *tut-install*`/examples/jms/advanced/`.
   c. Select the `transactedexample` folder.
   d. Select the Open as Main Project check box.
   e. Click Open Project.
   f. Right-click the project and choose Build.

   To compile and package the client using Ant, follow these steps:

   a. Go to the following directory:

      tut-install`/examples/jms/advanced/transactedexample/`

   b. Type the following command:

      ```
      ant
      ```

To run the client using NetBeans IDE, follow these steps:

1. Right-click the `transactedexample` project and choose Properties.
2. Select Run from the Categories tree.
3. In the Arguments field, type a number that specifies the number of computers to order:

   `3`

4. Click OK.
5. Right-click the project and choose Run.

To run the client from the command line, follow these steps:

1. Deploy the client JAR file to the Enterprise Server, then retrieve the client stubs:

   ```
   ant getclient
   ```

   Ignore the message that states that the application is deployed at a URL.

2. Use a command like the following to run the client. The argument specifies the number of computers to order:

```
appclient -client client-jar/transactedexampleClient.jar 3
```

The output looks something like this (along with some application client container output):

```
Quantity to be ordered is 3
Retailer: ordered 3 computer(s)
Vendor: Retailer ordered 3 Computer(s)
Vendor: ordered 3 monitor(s) and hard drive(s)
Monitor Supplier: Vendor ordered 3 Monitor(s)
Monitor Supplier: sent 3 Monitor(s)
  Monitor Supplier: committed transaction
  Vendor: committed transaction 1
Hard Drive Supplier: Vendor ordered 3 Hard Drive(s)
Hard Drive Supplier: sent 1 Hard Drive(s)
Vendor: Completed processing for order 1
  Hard Drive Supplier: committed transaction
Vendor: unable to send 3 computer(s)
  Vendor: committed transaction 2
Retailer: Order not filled
Retailer: placing another order
Retailer: ordered 6 computer(s)
Vendor: JMSException occurred: javax.jms.JMSException:
Simulated database concurrent access exception
javax.jms.JMSException: Simulated database concurrent access exception
        at TransactedExample$Vendor.run(Unknown Source)
  Vendor: rolled back transaction 1
Vendor: Retailer ordered 6 Computer(s)
Vendor: ordered 6 monitor(s) and hard drive(s)
Monitor Supplier: Vendor ordered 6 Monitor(s)
Hard Drive Supplier: Vendor ordered 6 Hard Drive(s)
Monitor Supplier: sent 6 Monitor(s)
  Monitor Supplier: committed transaction
Hard Drive Supplier: sent 6 Hard Drive(s)
  Hard Drive Supplier: committed transaction
  Vendor: committed transaction 1
Vendor: Completed processing for order 2
Vendor: sent 6 computer(s)
Retailer: Order filled
  Vendor: committed transaction 2
```

After you run the client, you can delete the physical destinations and the destination resources. Go to the directory *tut-install*`/examples/jms/advanced/transactedexample/` and type the following command:

```
ant delete-resources
```

To undeploy the client using Ant, type the following:

```
ant undeploy
```

To delete the class and JAR files for the client using Ant, type the following:

```
ant clean
```

An Application That Uses the JMS API with a Session Bean

This section explains how to write, compile, package, deploy, and run an application that uses the JMS API in conjunction with a session bean. The application contains the following components:

- An application client that invokes a session bean
- A session bean that publishes several messages to a topic
- A message-driven bean that receives and processes the messages using a durable topic subscriber and a message selector

The section covers the following topics:

You will find the source files for this section in the directory *tut-install*`/examples/jms/clientsessionmdb/`. Path names in this section are relative to this directory.

Writing the Application Components for the `clientsessionmdb` Example

This application demonstrates how to send messages from an enterprise bean (in this case, a session bean) rather than from an application client, as in the example in Chapter 17, "A Message-Driven Bean Example." Figure 32–3 illustrates the structure of this application.

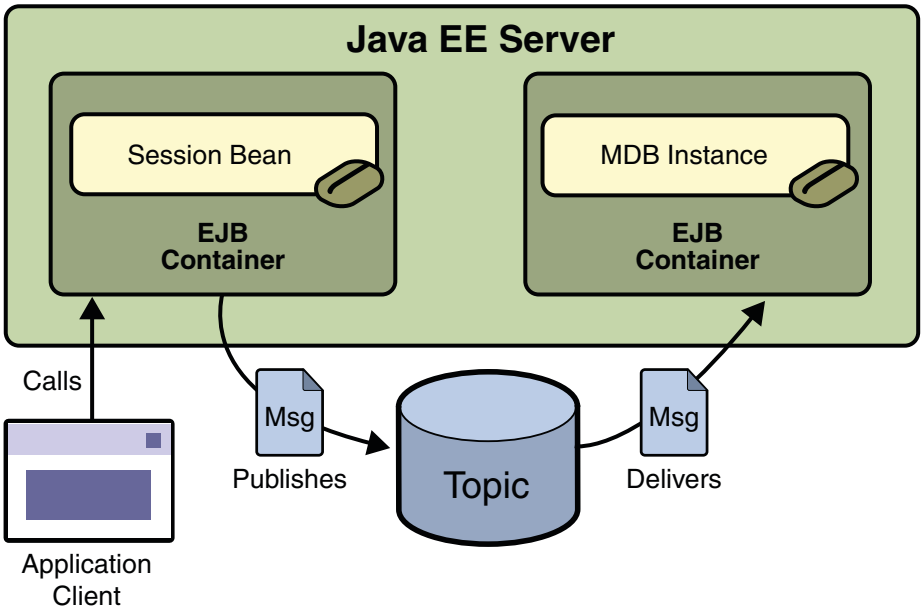

FIGURE 32–3 An Enterprise Bean Application: Client to Session Bean to Message-Driven Bean

The Publisher enterprise bean in this example is the enterprise-application equivalent of a wire-service news feed that categorizes news events into six news categories. The message-driven bean could represent a newsroom, where the sports desk, for example, would set up a subscription for all news events pertaining to sports.

The application client in the example injects the Publisher enterprise bean's remote home interface and then calls the bean's business method. The enterprise bean creates 18 text messages. For each message, it sets a `String` property randomly to one of six values representing the news categories and then publishes the message to a topic. The message-driven bean uses a message selector for the property to limit which of the published messages it receives.

Writing the components of the application involves the following:

- "Coding the Application Client: `MyAppClient.java`" on page 62
- "Coding the Publisher Session Bean" on page 63
- "Coding the Message-Driven Bean: `MessageBean.java`" on page 63

Coding the Application Client: `MyAppClient.java`

The application client, `clientsessionmdb-app-client/src/java/MyAppClient.java`, performs no JMS API operations and so is simpler than the client in Chapter 17, "A Message-Driven Bean Example." The client uses dependency injection to obtain the Publisher enterprise bean's business interface:

```
@EJB(name="PublisherRemote")
static private PublisherRemote publisher;
```

The client then calls the bean's business method twice.

Coding the Publisher Session Bean

The Publisher bean is a stateless session bean that has one business method. The Publisher bean uses a remote interface rather than a local interface because it is accessed from the application client.

The remote interface, `clientsessionmdb-ejb/src/java/sb/PublisherRemote.java`, declares a single business method, `publishNews`.

The bean class, `clientsessionmdb-ejb/src/java/sb/PublisherBean.java`, implements the `publishNews` method and its helper method `chooseType`. The bean class also injects `SessionContext`, `ConnectionFactory`, and `Topic` resources and implements `@PostConstruct` and `@PreDestroy` callback methods. The bean class begins as follows:

```
@Stateless
@Remote({PublisherRemote.class})
public class PublisherBean implements PublisherRemote {

    @Resource
    private SessionContext sc;

    @Resource(mappedName="jms/ConnectionFactory")
    private ConnectionFactory connectionFactory;

    @Resource(mappedName="jms/Topic")
    private Topic topic;
    ...
```

The `@PostConstruct` callback method of the bean class, `makeConnection`, creates the `Connection` used by the bean. The business method `publishNews` creates a `Session` and a `MessageProducer` and publishes the messages.

The `@PreDestroy` callback method, `endConnection`, deallocates the resources that were allocated by the `@PostConstruct` callback method. In this case, the method closes the `Connection`.

Coding the Message-Driven Bean: `MessageBean.java`

The message-driven bean class, `clientsessionmdb-ejb/src/java/mdb/MessageBean.java`, is almost identical to the one in Chapter 17, "A Message-Driven Bean Example." However, the `@MessageDriven` annotation is different, because instead of a queue the bean is using a topic

with a durable subscription, and it is also using a message selector. Therefore, the annotation sets the activation config properties `messageSelector`, `subscriptionDurability`, `clientId`, and `subscriptionName`, as follows:

```
@MessageDriven(mappedName="jms/Topic",
activationConfig=
{ @ActivationConfigProperty(propertyName="messageSelector",
    propertyValue="NewsType = 'Sports' OR NewsType = 'Opinion'"),
  @ActivationConfigProperty(
    propertyName="subscriptionDurability",
    propertyValue="Durable"),
  @ActivationConfigProperty(propertyName="clientId",
    propertyValue="MyID"),
  @ActivationConfigProperty(propertyName="subscriptionName",
    propertyValue="MySub")
 })
```

The JMS resource adapter uses these properties to create a connection factory for the message-driven bean that allows the bean to use a durable subscriber.

Creating Resources for the `clientsessionmdb` Example

This example uses the topic named `jms/Topic` and the connection factory `jms/ConnectionFactory`, which you created in "Creating JMS Administered Objects for the Synchronous Receive Example" on page 32. If you deleted the connection factory or topic, you can create them again using targets in the `build.xml` file for this example. Use the following commands to create the resources:

```
ant create-cf
ant create-topic
```

Building, Deploying, and Running the `clientsessionmdb` Example Using NetBeans IDE

To build, deploy, and run the application using NetBeans IDE, do the following:

1. In NetBeans IDE, choose Open Project from the File menu.
2. In the Open Project dialog, navigate to *tut-install*`/examples/jms/`.
3. Select the `clientsessionmdb` folder.
4. Select the Open as Main Project check box and the Open Required Projects check box.
5. Click Open Project.

6. Right-click the `clientsessionmdb` project and choose Build.

   This task creates the following:

   - An application client JAR file that contains the client class file and the session bean's remote interface, along with a manifest file that specifies the main class and places the EJB JAR file in its classpath
   - An EJB JAR file that contains both the session bean and the message-driven bean
   - An application EAR file that contains the two JAR files

7. Right-click the project and choose Run.

   This command deploys the project, returns a JAR file named `clientsessionmdbClient.jar`, and then executes it.

The output of the application client in the Output pane looks like this (preceded by application client container output):

```
To view the bean output,
 check <install_dir>/domains/domain1/logs/server.log.
```

The output from the enterprise beans appears in the server log (*domain-dir*`/logs/server.log`), wrapped in logging information. The Publisher session bean sends two sets of 18 messages numbered 0 through 17. Because of the message selector, the message-driven bean receives only the messages whose `NewsType` property is `Sports` or `Opinion`.

Undeploy the application after you finish running the client. To undeploy the application, follow these steps:

1. Click the Services tab.
2. Expand the Servers node.
3. Expand the GlassFish v3 Domain node.
4. Expand the Applications node.

   You may need to right-click the node and choose Refresh to see the contents of the Applications node.

5. Right-click `clientsessionmdb` and choose Undeploy.

To remove the generated files, click the Projects tab, then right-click the `clientsessionmdb` project and choose Clean.

Building, Deploying, and Running the `clientsessionmdb` Example Using Ant

To build the application using Ant, do the following:

1. Go to the following directory:

   tut-install`/examples/jms/clientsessionmdb/`

2. To compile the source files and package the application, use the following command:

   ```
   ant
   ```

The `ant` command creates the following:

- An application client JAR file that contains the client class file and the session bean's remote interface, along with a manifest file that specifies the main class and places the EJB JAR file in its classpath
- An EJB JAR file that contains both the session bean and the message-driven bean
- An application EAR file that contains the two JAR files

The `clientsessionmdb.ear` file is created in the `dist` directory.

To deploy the application and run the client, use the following command:

```
ant run
```

Ignore the message that states that the application is deployed at a URL.

The client displays these lines (preceded by application client container output):

```
To view the bean output,
 check <install_dir>/domains/domain1/logs/server.log.
```

The output from the enterprise beans appears in the server log file, wrapped in logging information. The Publisher session bean sends two sets of 18 messages numbered 0 through 17. Because of the message selector, the message-driven bean receives only the messages whose `NewsType` property is `Sports` or `Opinion`.

Undeploy the application after you finish running the client. Use the following command:

```
ant undeploy
```

To remove the generated files, use the following command in the `clientsessionmdb`, `clientsessionmdb-app-client`, and `clientsessionmdb-ejb` directories:

```
ant clean
```

An Application That Uses the JMS API with an Entity

This section explains how to write, compile, package, deploy, and run an application that uses the JMS API with an entity. The application uses the following components:

- An application client that both sends and receives messages
- Two message-driven beans
- An entity class

This section covers the following topics:

You will find the source files for this section in the directory *tut-install*`/examples/jms/clientmdbentity/`. Path names in this section are relative to this directory.

Overview of the `clientmdbentity` Example Application

This application simulates, in a simplified way, the work flow of a company's human resources (HR) department when it processes a new hire. This application also demonstrates how to use the Java EE platform to accomplish a task that many JMS applications need to perform.

A JMS client must often wait for several messages from various sources. It then uses the information in all these messages to assemble a message that it then sends to another destination. The common term for this process is *joining messages*. Such a task must be transactional, with all the receives and the send as a single transaction. If not all the messages are received successfully, the transaction can be rolled back. For an application client example that illustrates this task, see "A Local Transaction Example" on page 56.

A message-driven bean can process only one message at a time in a transaction. To provide the ability to join messages, an application can have the message-driven bean store the interim information in an entity. The entity can then determine whether all the information has been received; when it has, the entity can report this back to one of the message-driven beans, which then creates and sends the message to the other destination. After it has completed its task, the entity can be removed.

The basic steps of the application are as follows.

1. The HR department's application client generates an employee ID for each new hire and then publishes a message (M1) containing the new hire's name, employee ID, and position. The client then creates a temporary queue, `ReplyQueue`, with a message listener that waits for a reply to the message. (See "Creating Temporary Destinations" on page for more information.)
2. Two message-driven beans process each message: One bean, `OfficeMDB`, assigns the new hire's office number, and the other bean, `EquipmentMDB`, assigns the new hire's equipment. The first bean to process the message creates and persists an entity named `SetupOffice`, then calls a business method of the entity to store the information it has generated. The second bean locates the existing entity and calls another business method to add its information.
3. When both the office and the equipment have been assigned, the entity business method returns a value of `true` to the message-driven bean that called the method. The message-driven bean then sends to the reply queue a message (M2) describing the assignments. Then it removes the entity. The application client's message listener retrieves the information.

Figure 32–4 illustrates the structure of this application. Of course, an actual HR application would have more components; other beans could set up payroll and benefits records, schedule orientation, and so on.

Figure 32–4 assumes that `OfficeMDB` is the first message-driven bean to consume the message from the client. `OfficeMDB` then creates and persists the `SetupOffice` entity and stores the office information. `EquipmentMDB` then finds the entity, stores the equipment information, and learns that the entity has completed its work. `EquipmentMDB` then sends the message to the reply queue and removes the entity.

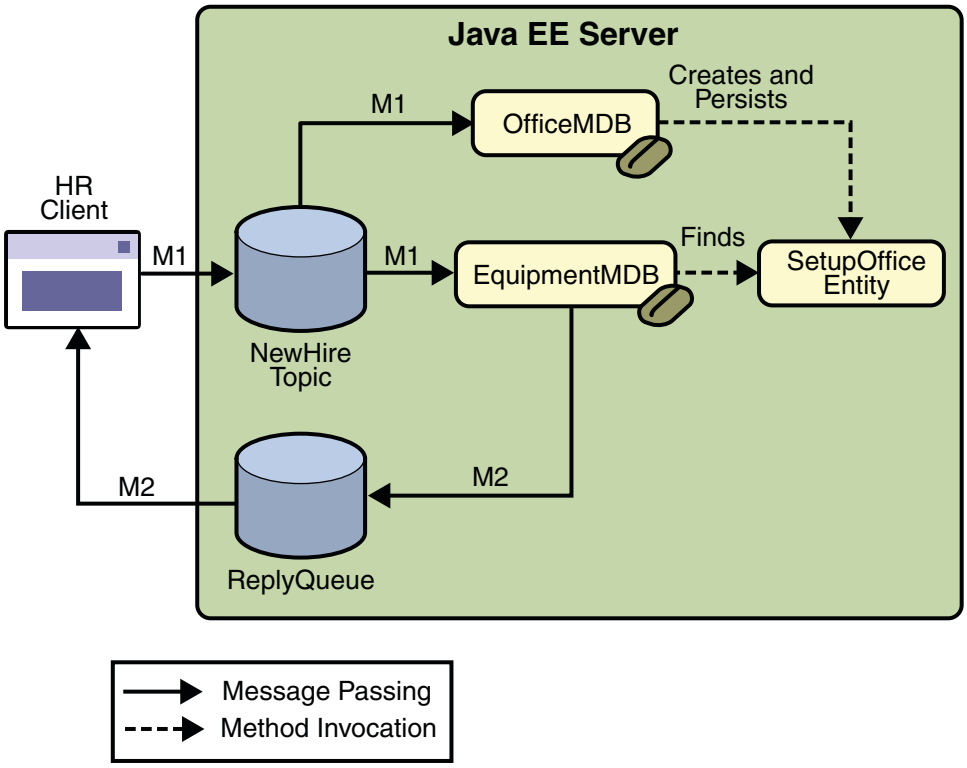

FIGURE 32–4 An Enterprise Bean Application: Client to Message-Driven Beans to Entity

Writing the Application Components for the `clientmdbentity` Example

Writing the components of the application involves the following:

- "Coding the Application Client: `HumanResourceClient.java`" on page 69
- "Coding the Message-Driven Beans for the `clientmdbentity` Example" on page 70
- "Coding the Entity Class for the `clientmdbentity` Example" on page 70

Coding the Application Client: `HumanResourceClient.java`

The application client, `clientmdbentity-app-client/src/java/HumanResourceClient.java`, performs the following steps:

1. Injects `ConnectionFactory` and `Topic` resources
2. Creates a `TemporaryQueue` to receive notification of processing that occurs, based on new-hire events it has published

3. Creates a `MessageConsumer` for the `TemporaryQueue`, sets the `MessageConsumer`'s message listener, and starts the connection
4. Creates a `MessageProducer` and a `MapMessage`
5. Creates five new employees with randomly generated names, positions, and ID numbers (in sequence) and publishes five messages containing this information

The message listener, `HRListener`, waits for messages that contain the assigned office and equipment for each employee. When a message arrives, the message listener displays the information received and determines whether all five messages have arrived. When they have, the message listener notifies the `main` method, which then exits.

Coding the Message-Driven Beans for the `clientmdbentity` Example

This example uses two message-driven beans:

- `clientmdbentity-ejb/src/java/EquipmentMDB.java`
- `clientmdbentity-ejb/src/java/OfficeMDB.java`

The beans take the following steps:

1. They inject `MessageDrivenContext` and `ConnectionFactory` resources.
2. The `onMessage` method retrieves the information in the message. The `EquipmentMDB`'s `onMessage` method chooses equipment, based on the new hire's position; the `OfficeMDB`'s `onMessage` method randomly generates an office number.
3. After a slight delay to simulate real world processing hitches, the `onMessage` method calls a helper method, `compose`.
4. The `compose` method takes the following steps:
    a. It either creates and persists the `SetupOffice` entity or finds it by primary key.
    b. It uses the entity to store the equipment or the office information in the database, calling either the `doEquipmentList` or the `doOfficeNumber` business method.
    c. If the business method returns `true`, meaning that all of the information has been stored, it creates a connection and a session, retrieves the reply destination information from the message, creates a `MessageProducer`, and sends a reply message that contains the information stored in the entity.
    d. It removes the entity.

Coding the Entity Class for the `clientmdbentity` Example

The `SetupOffice` class, `SetupOffice.java`, is an entity class. The entity and the message-driven beans are packaged together in an EJB JAR file. The entity class is declared as follows:

```
@Entity
public class SetupOffice implements Serializable {
```

The class contains a no-argument constructor and a constructor that takes two arguments, the employee ID and name. It also contains getter and setter methods for the employee ID, name, office number, and equipment list. The getter method for the employee ID has the `@Id` annotation to indicate that this field is the primary key:

```
@Id public String getEmployeeId() {
    return id;
}
```

The class also implements the two business methods, `doEquipmentList` and `doOfficeNumber`, and their helper method, `checkIfSetupComplete`.

The message-driven beans call the business methods and the getter methods.

The `persistence.xml` file for the entity specifies the most basic settings:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
             xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/per
  <persistence-unit name="clientmdbentity-ejbPU" transaction-type="JTA">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <jta-data-source>jdbc/__default</jta-data-source>
    <properties>
      <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
    </properties>
  </persistence-unit>
</persistence>
```

Creating Resources for the `clientmdbentity` Example

This example uses the connection factory `jms/ConnectionFactory` and the topic `jms/Topic`, both of which you used in "An Application That Uses the JMS API with a Session Bean" on page 61. It also uses the JDBC resource named `jdbc/__default`, which is enabled by default when you start the Enterprise Server.

If you deleted the connection factory or topic, you can create them again using targets in the `build.xml` file for this example. Use the following commands to create the resources:

```
ant create-cf
ant create-topic
```

Building, Deploying, and Running the `clientmdbentity` Example Using NetBeans IDE

To build, deploy, and run the application using NetBeans IDE, do the following:

1. In NetBeans IDE, choose Open Project from the File menu.
2. In the Open Project dialog, navigate to *tut-install*`/examples/jms/`.
3. Select the `clientmdbentity` folder.
4. Select the Open as Main Project check box and the Open Required Projects check box.
5. Click Open Project.
6. Right-click the `clientmdbentity` project and choose Build.

   This task creates the following:

   - An application client JAR file that contains the client class and listener class files, along with a manifest file that specifies the main class
   - An EJB JAR file that contains the message-driven beans and the entity class, along with the `persistence.xml` file
   - An application EAR file that contains the two JAR files along with an `application.xml` file
7. Right-click the project and choose Run.

   This command deploys the project, returns a client JAR file named `clientmdbentityClient.jar` and then executes it.

Note – Because NetBeans IDE starts the database at the same time it starts the Enterprise Server, you do not need to start the database explicitly.

The output of the application client in the Output pane looks something like this:

```
PUBLISHER: Setting hire ID to 25, name Gertrude Bourbon, position Senior Programmer
PUBLISHER: Setting hire ID to 26, name Jack Verdon, position Manager
PUBLISHER: Setting hire ID to 27, name Fred Tudor, position Manager
PUBLISHER: Setting hire ID to 28, name Fred Martin, position Programmer
PUBLISHER: Setting hire ID to 29, name Mary Stuart, position Manager
Waiting for 5 message(s)
New hire event processed:
  Employee ID: 25
  Name: Gertrude Bourbon
  Equipment: Laptop
  Office number: 183
Waiting for 4 message(s)
New hire event processed:
```

```
  Employee ID: 26
  Name: Jack Verdon
  Equipment: Pager
  Office number: 20
Waiting for 3 message(s)
New hire event processed:
  Employee ID: 27
  Name: Fred Tudor
  Equipment: Pager
  Office number: 51
Waiting for 2 message(s)
New hire event processed:
  Employee ID: 28
  Name: Fred Martin
  Equipment: Desktop System
  Office number: 141
Waiting for 1 message(s)
New hire event processed:
  Employee ID: 29
  Name: Mary Stuart
  Equipment: Pager
  Office number: 238
```

The output from the message-driven beans and the entity class appears in the server log, wrapped in logging information.

For each employee, the application first creates the entity and then finds it. You may see runtime errors in the server log, and transaction rollbacks may occur. The errors occur if both of the message-driven beans discover at the same time that the entity does not yet exist, so they both try to create it. The first attempt succeeds, but the second fails because the bean already exists. After the rollback, the second message-driven bean tries again and succeeds in finding the entity. Container-managed transactions allow the application to run correctly, in spite of these errors, with no special programming.

You can run the application client repeatedly.

Undeploy the application after you finish running the client. To undeploy the application, follow these steps:

1. Click the Services tab.
2. Expand the Servers node.
3. Expand the GlassFish v3 Domain node.
4. Expand the Applications node.

   You may need to right-click the node and choose Refresh to see the contents of the Applications node.
5. Right-click `clientmdbentity` and choose Undeploy.

To remove the generated files, right-click the `clientmdbentity`, `clientmdbentity-app-client`, and `clientmdbentity-ejb` projects in turn and choose Clean.

Building, Deploying, and Running the `clientmdbentity` Example Using Ant

To create and package the application using Ant, perform these steps:

1. Start the Enterprise Server, if it is not already running.
2. Go to the following directory:

   tut-install`/examples/jms/clientmdbentity/`
3. To compile the source files and package the application, use the following command:

   `ant`

The `ant` command creates the following:

- An application client JAR file that contains the client class and listener class files, along with a manifest file that specifies the main class
- An EJB JAR file that contains the message-driven beans and the entity class, along with the `persistence.xml` file
- An application EAR file that contains the two JAR files along with an `application.xml` file

To deploy the application and run the client, use the following command:

```
ant run
```

This command starts the database server if it is not already running, then deploys and runs the application.

Ignore the message that states that the application is deployed at a URL.

The output in the terminal window looks something like this (preceded by application client container output):

```
running application client container.
PUBLISHER: Setting hire ID to 25, name Gertrude Bourbon, position Senior Programmer
PUBLISHER: Setting hire ID to 26, name Jack Verdon, position Manager
PUBLISHER: Setting hire ID to 27, name Fred Tudor, position Manager
PUBLISHER: Setting hire ID to 28, name Fred Martin, position Programmer
PUBLISHER: Setting hire ID to 29, name Mary Stuart, position Manager
Waiting for 5 message(s)
New hire event processed:
  Employee ID: 25
  Name: Gertrude Bourbon
```

```
  Equipment: Laptop
  Office number: 183
Waiting for 4 message(s)
New hire event processed:
  Employee ID: 26
  Name: Jack Verdon
  Equipment: Pager
  Office number: 20
Waiting for 3 message(s)
New hire event processed:
  Employee ID: 27
  Name: Fred Tudor
  Equipment: Pager
  Office number: 51
Waiting for 2 message(s)
New hire event processed:
  Employee ID: 28
  Name: Fred Martin
  Equipment: Desktop System
  Office number: 141
Waiting for 1 message(s)
New hire event processed:
  Employee ID: 29
  Name: Mary Stuart
  Equipment: Pager
  Office number: 238
```

The output from the message-driven beans and the entity class appears in the server log, wrapped in logging information.

For each employee, the application first creates the entity and then finds it. You may see runtime errors in the server log, and transaction rollbacks may occur. The errors occur if both of the message-driven beans discover at the same time that the entity does not yet exist, so they both try to create it. The first attempt succeeds, but the second fails because the bean already exists. After the rollback, the second message-driven bean tries again and succeeds in finding the entity. Container-managed transactions allow the application to run correctly, in spite of these errors, with no special programming.

Undeploy the application after you finish running the client:

```
ant undeploy
```

To remove the generated files, use the following command in the `clientmdbentity`, `clientmdbentity-app-client`, and `clientmdbentity-ejb` directories:

```
ant clean
```

An Application Example That Consumes Messages from a Remote Server

This section and the following section explain how to write, compile, package, deploy, and run a pair of Java EE modules that run on two Java EE servers and that use the JMS API to interchange messages with each other. It is a common practice to deploy different components of an enterprise application on different systems within a company, and these examples illustrate on a small scale how to do this for an application that uses the JMS API.

However, the two examples work in slightly different ways. In this first example, the deployment information for a message-driven bean specifies the remote server from which it will *consume* messages. In the next example, the same message-driven bean is deployed on two different servers, so it is the client module that specifies the servers (one local, one remote) to which it is *sending* messages.

This first example divides the example in Chapter 17, "A Message-Driven Bean Example," into two modules: one containing the application client, and the other containing the message-driven bean.

This section covers the following topics:

You will find the source files for this section in *tut-install*`/examples/jms/consumeremote/`. Path names in this section are relative to this directory.

Overview of the `consumeremote` Example Modules

Except for the fact that it is packaged as two separate modules, this example is very similar to the one in Chapter 17, "A Message-Driven Bean Example":

- One module contains the application client, which runs on the remote system and sends three messages to a queue.
- The other module contains the message-driven bean, which is deployed on the local server and consumes the messages from the queue on the remote server.

The basic steps of the modules are as follows.

1. The administrator starts two Java EE servers, one on each system.

2. On the local server, the administrator deploys the message-driven bean module, which specifies the remote server where the client is deployed.
3. On the remote server, the administrator places the client JAR file.
4. The client module sends three messages to a queue.
5. The message-driven bean consumes the messages.

Figure 32–5 illustrates the structure of this application. You can see that it is almost identical to Figure 17–1 except that there are two Java EE servers. The queue used is the one on the remote server; the queue must also exist on the local server for resource injection to succeed.

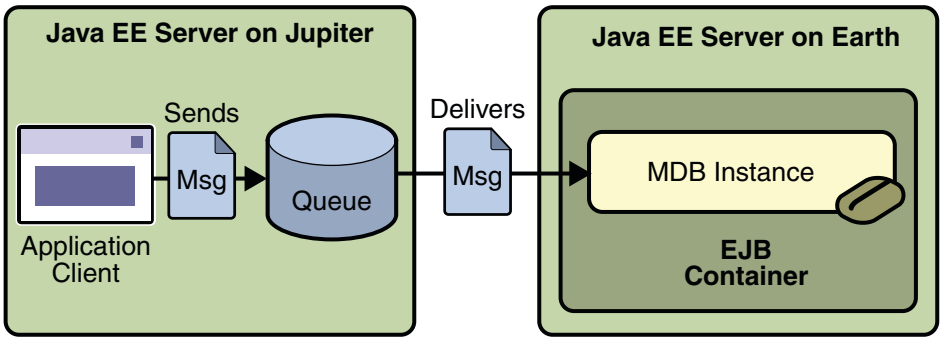

FIGURE 32–5 A Java EE Application That Consumes Messages from a Remote Server

Writing the Module Components for the `consumeremote` Example

Writing the components of the modules involves

- Coding the application client
- Coding the message-driven bean

The application client, `jupiterclient/src/java/SimpleClient.java`, is almost identical to the one in "The `simplemessage` Application Client" on page .

Similarly, the message-driven bean, `earthmdb/src/java/MessageBean.java`, is almost identical to the one in "The Message-Driven Bean Class" on page . The only significant difference is that the activation config properties include one property that specifies the name of the remote system. You need to edit the source file to specify the name of your system.

Creating Resources for the `consumeremote` Example

The application client can use any connection factory that exists on the remote server; it uses `jms/ConnectionFactory`. Both components use the queue named `jms/Queue`, which you created in "Creating JMS Administered Objects for the Synchronous Receive Example" on page 32. The message-driven bean does not need a previously created connection factory; the resource adapter creates one for it.

Using Two Application Servers for the `consumeremote` Example

As in "Running JMS Clients on Multiple Systems" on page 46, the two servers are named `earth` and `jupiter`.

The Enterprise Server must be running on both systems.

Which system you use to package and deploy the modules and which system you use to run the client depend on your network configuration (which file system you can access remotely). These instructions assume that you can access the file system of `jupiter` from `earth` but cannot access the file system of `earth` from `jupiter`. (You can use the same systems for `jupiter` and `earth` that you used in "Running JMS Clients on Multiple Systems" on page 46.)

You can package both modules on `earth` and deploy the message-driven bean there. The only action you perform on `jupiter` is running the client module.

Building, Deploying, and Running the `consumeremote`Modules Using NetBeans IDE

To edit the message-driven bean source file and package the modules using NetBeans IDE, perform these steps:

1. In NetBeans IDE, choose Open Project from the File menu.
2. In the Open Project dialog, navigate to *tut-install*`/examples/jms/consumeremote/`.
3. Select the `earthmdb` folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. Edit the `MessageBean.java` file as follows:
    a. Expand the `earthmdb`, Source Packages, and `mdb` nodes, then double-click `MessageBean.java`.
    b. Find the following line within the `@MessageBean` annotation:

```
@ActivationConfigProperty(propertyName = "addressList",
    propertyValue = "remotesystem"),
```

c. Replace `remotesystem` with the name of your remote system.

7. Right-click the `earthmdb` project and choose Build.

   This command creates a JAR file that contains the bean class file.

8. Choose Open Project from the File menu.
9. Select the `jupiterclient` folder.
10. Select the Open as Main Project check box.
11. Click Open Project.
12. Right-click the `jupiterclient` project and choose Build.

    This target creates a JAR file that contains the client class file and a manifest file.

To deploy the `earthmdb` module and run the application client, perform these steps:

1. Right-click the `earthmdb` project and choose Deploy.
2. Copy the `jupiterclient` module to the remote system (`jupiter`). The module is *tut-install*`/examples/jms/consumeremote/jupiterclient/dist/jupiterclient.jar`. You can use either the file system user interface on your system or the command line.
3. Go to the directory on the remote system where you copied the client JAR file.
4. Use the following command:

   ```
   asadmin deploy --retrieve . jupiterclient.jar
   ```

   This command deploys the client JAR file and retrieves the client stubs in a file named `jupiterclientClient.jar`

5. Use the following command:

   ```
   appclient -client jupiterclientClient.jar
   ```

On `jupiter`, the output of the `appclient` command looks like this (preceded by application client container output):

```
Sending message: This is message 1 from jupiter
Sending message: This is message 2 from jupiter
Sending message: This is message 3 from jupiter
```

On `earth`, the output in the server log looks something like this (preceded by logging information):

```
MESSAGE BEAN: Message received: This is message 1 from jupiter
MESSAGE BEAN: Message received: This is message 2 from jupiter
MESSAGE BEAN: Message received: This is message 3 from jupiter
```

Undeploy the message-driven bean after you finish running the client. To undeploy the `earthmdb` module, perform these steps on `earth`:

1. Click the Services tab.
2. Expand the Servers node.
3. Expand the GlassFish v3 Domain node.
4. Expand the Applications node.
5. Right-click `earthmdb` and choose Undeploy.

To undeploy the `jupiterclient` module, use the following command on `jupiter`:

```
asadmin undeploy jupiterclient
```

To remove the generated files, follow these steps:

1. On `earth`, right-click the `earthmdb` project and choose Clean.
2. Right-click the `jupiterclient` project and choose Clean.

You can also delete the `jupiterclient.jar` file from the remote filesystem, along with the `jupiterclientClient.jar` file and the directory `jupiterclientClient` that were created when you retrieved the client stubs.

Building, Deploying, and Running the `consumeremote` Modules Using Ant

To edit the message-driven bean source file and package and deploy the `earthmdb` module using Ant, perform these steps:

1. Open the file *tut-install*`/examples/jms/consumeremote/earthmdb/src/java/mdb/MessageBean.java` in an editor.
2. Find the following line within the `@MessageBean` annotation:

```
@ActivationConfigProperty(propertyName = "addressList",
    propertyValue = "remotesystem"),
```

3. Replace `remotesystem` with the name of your remote system, then save and close the file.
4. Go to the following directory:

   tut-install`/examples/jms/consumeremote/earthmdb/`

5. Type the following command:

   ```
   ant
   ```

   This command creates a JAR file that contains the bean class file.

6. Type the following command:

   ```
   ant deploy
   ```

7. Go to the `jupiterclient` directory:

   ```
   cd ../jupiterclient
   ```

8. Type the following command:

   ```
   ant
   ```

   This target creates a JAR file that contains the client class file and a manifest file.

To copy the `jupiterclient` module to the remote system, perform these steps:

1. Change to the directory `jupiterclient/dist`:

   ```
   cd ../jupiterclient/dist
   ```

2. Type a command like the following:

   ```
   cp jupiterclient.jar F:/
   ```

   That is, copy the client JAR file to a location on the remote filesystem.

To run the client, perform the following steps:

1. Go to the directory on the remote system (`jupiter`) where you copied the client JAR file.
2. Use the following command:

   ```
   asadmin deploy --retrieve . jupiterclient.jar
   ```

   This command deploys the client JAR file and retrieves the client stubs in a file named `jupiterclientClient.jar`

3. Use the following command:

   ```
   appclient -client jupiterclientClient.jar
   ```

On `jupiter`, the output of the `appclient` command looks like this:

```
Sending message: This is message 1 from jupiter
Sending message: This is message 2 from jupiter
Sending message: This is message 3 from jupiter
```

On `earth`, the output in the server log looks something like this (wrapped in logging information):

```
MESSAGE BEAN: Message received: This is message 1 from jupiter
MESSAGE BEAN: Message received: This is message 2 from jupiter
MESSAGE BEAN: Message received: This is message 3 from jupiter
```

Undeploy the message-driven bean after you finish running the client. To undeploy the `earthmdb` module, perform these steps:

1. Change to the directory `earthmdb`.
2. Type the following command:

```
ant undeploy
```

You can also delete the `jupiterclient.jar` file from the remote filesystem, along with the `jupiterclientClient.jar` file and the directory `jupiterclientClient` that were created when you retrieved the client stubs.

To remove the generated files, use the following command in both the `earthmdb` and `jupiterclient` directories:

```
ant clean
```

An Application Example That Deploys a Message-Driven Bean on Two Servers

This section, like the preceding one, explains how to write, compile, package, deploy, and run a pair of Java EE modules that use the JMS API and run on two Java EE servers. The modules are slightly more complex than the ones in the first example.

The modules use the following components:

- An application client that is deployed on the local server. It uses two connection factories, one ordinary one and one that is configured to communicate with the remote server, to create two publishers and two subscribers and to publish and to consume messages.
- A message-driven bean that is deployed twice: once on the local server, and once on the remote one. It processes the messages and sends replies.

In this section, the term *local server* means the server on which both the application client and the message-driven bean are deployed (`earth` in the preceding example). The term *remote server* means the server on which only the message-driven bean is deployed (`jupiter` in the preceding example).

The section covers the following topics:

You will find the source files for this section in *tut-install*`/examples/jms/sendremote/`. Path names in this section are relative to this directory.

Overview of the `sendremote` Example Modules

This pair of modules is somewhat similar to the modules in "An Application Example That Consumes Messages from a Remote Server" on page 76 in that the only components are a client and a message-driven bean. However, the modules here use these components in more complex ways. One module consists of the application client. The other module contains only the message-driven bean and is deployed twice, once on each server.

The basic steps of the modules are as follows.

1. You start two Java EE servers, one on each system.
2. On the local server (`earth`), you create two connection factories: one local and one that communicates with the remote server (`jupiter`). On the remote server, you create a connection factory that has the same name as the one that communicates with the remote server.
3. The application client looks up the two connection factories (the local one and the one that communicates with the remote server) to create two connections, sessions, publishers, and subscribers. The subscribers use a message listener.
4. Each publisher publishes five messages.
5. Each of the local and the remote message-driven beans receives five messages and sends replies.
6. The client's message listener consumes the replies.

Figure 32–6 illustrates the structure of this application. M1 represents the first message sent using the local connection factory, and RM1 represents the first reply message sent by the local MDB. M2 represents the first message sent using the remote connection factory, and RM2 represents the first reply message sent by the remote MDB.

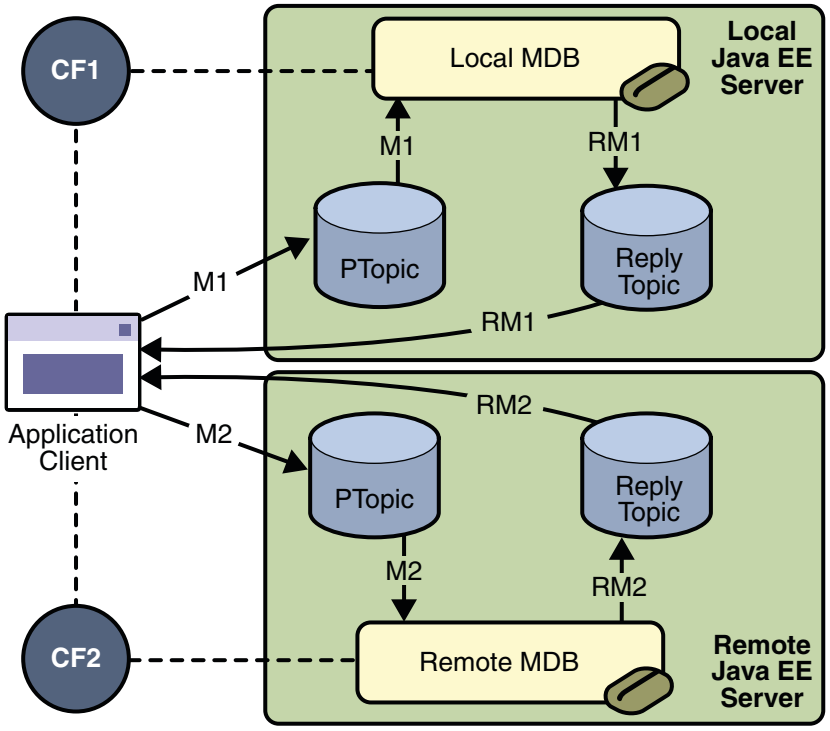

FIGURE 32–6 A Java EE Application That Sends Messages to Two Servers

Writing the Module Components for the `sendremote` Example

Writing the components of the modules involves two tasks:

- "Coding the Application Client: `MultiAppServerClient.java`" on page 84
- "Coding the Message-Driven Bean: `ReplyMsgBean.java`" on page 85

Coding the Application Client: `MultiAppServerClient.java`

The application client class, `multiclient/src/java/MultiAppServerClient.java`, does the following.

1. It injects resources for two connection factories and a topic.
2. For each connection factory, it creates a connection, a publisher session, a publisher, a subscriber session, a subscriber, and a temporary topic for replies.
3. Each subscriber sets its message listener, `ReplyListener`, and starts the connection.
4. Each publisher publishes five messages and creates a list of the messages the listener should expect.

5. When each reply arrives, the message listener displays its contents and removes it from the list of expected messages.
6. When all the messages have arrived, the client exits.

Coding the Message-Driven Bean: `ReplyMsgBean.java`

The message-driven bean class, `replybean/src/ReplyMsgBean.java`, does the following:

1. Uses the `@MessageDriven` annotation:

   ```
   @MessageDriven(mappedName="jms/Topic")
   ```

2. Injects resources for the `MessageDrivenContext` and for a connection factory. It does not need a destination resource because it uses the value of the incoming message's `JMSReplyTo` header as the destination.
3. Uses a `@PostConstruct` callback method to create the connection, and a `@PreDestroy` callback method to close the connection.

The `onMessage` method of the message-driven bean class does the following:

1. Casts the incoming message to a `TextMessage` and displays the text
2. Creates a connection, a session, and a publisher for the reply message
3. Publishes the message to the reply topic
4. Closes the connection

On both servers, the bean will consume messages from the topic `jms/Topic`.

Creating Resources for the `sendremote` Example

This example uses the connection factory named `jms/ConnectionFactory` and the topic named `jms/Topic`. These objects must exist on both the local and the remote servers.

This example uses an additional connection factory, `jms/JupiterConnectionFactory`, which communicates with the remote system; you created it in "Creating Administered Objects for Multiple Systems" on page 47. This connection factory must exist on the local server.

The `build.xml` file for the `multiclient` module contains targets that you can use to create these resources if you deleted them previously.

Using Two Application Servers for the `sendremote` Example

If you are using NetBeans IDE, you need to add the remote server in order to deploy the message-driven bean there. To do so, perform these steps:

1. In NetBeans IDE, click the Runtime tab.
2. Right-click the Servers node and choose Add Server. In the Add Server Instance dialog, perform these steps:
   a. Select GlassFish v3 Domain (the default) from the Server list.
   b. In the Name field, specify a name slightly different from that of the local server, such as `GlassFish v3 Domain (1)`.
   c. Click Next.
   d. For the Platform Folder location, you can either browse to the location of the Enterprise Server on the remote system or, if that location is not visible from the local system, use the default location on the local system.
   e. Select the Register Remote Domain radio button.
   f. Click Next.
   g. Type the system name of the host in the Host field.
   h. Click Next.
   i. Type the administrative username and password for the remote system in the Admin Username and Admin Password fields.
   j. Click Finish.

      There may be a delay while NetBeans IDE registers the remote domain.

Building, Deploying, and Running the `sendremote` Modules Using NetBeans IDE

To package the modules using NetBeans IDE, perform these steps:

1. In NetBeans IDE, choose Open Project from the File menu.
2. In the Open Project dialog, navigate to *tut-install*`/examples/jms/sendremote/`.
3. Select the `replybean` folder.
4. Select the Open as Main Project check box.
5. Click Open Project.
6. Right-click the `replybean` project and choose Build.

   This command creates a JAR file that contains the bean class file.

7. Choose Open Project from the File menu.
8. Select the `multiclient` folder.
9. Select the Open as Main Project check box.
10. Click Open Project.
11. Right-click the `multiclient` project and choose Build.

    This command creates a JAR file that contains the client class file and a manifest file.

To deploy the `multiclient` module on the local server, perform these steps:

1. Right-click the `multiclient` project and choose Properties.
2. Select Run from the Categories tree.
3. From the Server list, select GlassFish v3 Domain (the local server).
4. Click OK.
5. Right-click the `multiclient` project and choose Undeploy and Deploy.

To deploy the `replybean` module on the local and remote servers, perform these steps:

1. Right-click the `replybean` project and choose Properties.
2. Select Run from the Categories tree.
3. From the Server list, select GlassFish v3 Domain (the local server).
4. Click OK.
5. Right-click the `replybean` project and choose Undeploy and Deploy.
6. Right-click the `replybean` project again and choose Properties.
7. Select Run from the Categories tree.
8. From the Server list, select GlassFish v3 Domain (1) (the remote server).
9. Click OK.
10. Right-click the `replybean` project and choose Undeploy and Deploy.

You can use the Services tab to verify that `multiclient` is deployed as an App Client Module on the local server and that `replybean` is deployed as an EJB Module on both servers.

To run the application client, right-click the `multiclient` project and choose Run Project.

This command returns a JAR file named `multiclientClient.jar` and then executes it.

On the local system, the output of the `appclient` command looks something like this:

```
running application client container.
Sent message: text: id=1 to local app server
Sent message: text: id=2 to remote app server
ReplyListener: Received message: id=1, text=ReplyMsgBean processed message: text: id=1 to local
 app server
Sent message: text: id=3 to local app server
ReplyListener: Received message: id=3, text=ReplyMsgBean processed message: text: id=3 to local
 app server
ReplyListener: Received message: id=2, text=ReplyMsgBean processed message: text: id=2 to remote
```

```
 app server
Sent message: text: id=4 to remote app server
ReplyListener: Received message: id=4, text=ReplyMsgBean processed message: text: id=4 to remote
 app server
Sent message: text: id=5 to local app server
ReplyListener: Received message: id=5, text=ReplyMsgBean processed message: text: id=5 to local
 app server
Sent message: text: id=6 to remote app server
ReplyListener: Received message: id=6, text=ReplyMsgBean processed message: text: id=6 to remote
 app server
Sent message: text: id=7 to local app server
ReplyListener: Received message: id=7, text=ReplyMsgBean processed message: text: id=7 to local
 app server
Sent message: text: id=8 to remote app server
ReplyListener: Received message: id=8, text=ReplyMsgBean processed message: text: id=8 to remote
 app server
Sent message: text: id=9 to local app server
ReplyListener: Received message: id=9, text=ReplyMsgBean processed message: text: id=9 to local
 app server
Sent message: text: id=10 to remote app server
ReplyListener: Received message: id=10, text=ReplyMsgBean processed message: text: id=10 to remote
 app server
Waiting for 0 message(s) from local app server
Waiting for 0 message(s) from remote app server
Finished
Closing connection 1
Closing connection 2
```

On the local system, where the message-driven bean receives the odd-numbered messages, the output in the server log looks like this (wrapped in logging information):

```
ReplyMsgBean: Received message: text: id=1 to local app server
ReplyMsgBean: Received message: text: id=3 to local app server
ReplyMsgBean: Received message: text: id=5 to local app server
ReplyMsgBean: Received message: text: id=7 to local app server
ReplyMsgBean: Received message: text: id=9 to local app server
```

On the remote system, where the bean receives the even-numbered messages, the output in the server log looks like this (wrapped in logging information):

```
ReplyMsgBean: Received message: text: id=2 to remote app server
ReplyMsgBean: Received message: text: id=4 to remote app server
ReplyMsgBean: Received message: text: id=6 to remote app server
ReplyMsgBean: Received message: text: id=8 to remote app server
ReplyMsgBean: Received message: text: id=10 to remote app server
```

Undeploy the modules after you finish running the client. To undeploy the modules, perform these steps:

1. Click the Services tab.
2. Expand the Servers node.
3. Expand the GlassFish v3 Domain node (the local system).
4. Expand the Applications node.
5. Expand the EJB Modules node.
6. Right-click `replybean` and choose Undeploy.
7. Expand the App Client Modules node.
8. Right-click `multiclient` and choose Undeploy.
9. Expand the GlassFish v3 Domain (1) node (the remote system).
10. Expand the Applications node.
11. Expand the EJB Modules node.
12. Right-click `replybean` and choose Undeploy.

To remove the generated files, follow these steps:

1. Right-click the `replybean` project and choose Clean.
2. Right-click the `multiclient` project and choose Clean.

Building, Deploying, and Running the `sendremote` Modules Using Ant

To package the modules, perform these steps:

1. Go to the following directory:

   tut-install`/examples/jms/sendremote/multiclient/`

2. Type the following command:

   ```
   ant
   ```

   This command creates a JAR file that contains the client class file and a manifest file.

3. Change to the directory `replybean`:

   ```
   cd ../replybean
   ```

4. Type the following command:

   ```
   ant
   ```

   This command creates a JAR file that contains the bean class file.

To deploy the `replybean` module on the local and remote servers, perform the following steps:

1. Verify that you are still in the directory `replybean`.
2. Type the following command:

```
ant deploy
```

Ignore the message that states that the application is deployed at a URL.

3. Type the following command:

```
ant deploy-remote -Dsys=remote-system-name
```

Replace *remote-system-name* with the actual name of the remote system.

To deploy and run the client, perform these steps:

1. Change to the directory `multiclient`:

```
cd ../multiclient
```

2. Type the following command:

```
ant run
```

On the local system, the output looks something like this:

```
running application client container.
Sent message: text: id=1 to local app server
Sent message: text: id=2 to remote app server
ReplyListener: Received message: id=1, text=ReplyMsgBean processed message: text: id=1 to local
 app server
Sent message: text: id=3 to local app server
ReplyListener: Received message: id=3, text=ReplyMsgBean processed message: text: id=3 to local
 app server
ReplyListener: Received message: id=2, text=ReplyMsgBean processed message: text: id=2 to remote
 app server
Sent message: text: id=4 to remote app server
ReplyListener: Received message: id=4, text=ReplyMsgBean processed message: text: id=4 to remote
 app server
Sent message: text: id=5 to local app server
ReplyListener: Received message: id=5, text=ReplyMsgBean processed message: text: id=5 to local
 app server
Sent message: text: id=6 to remote app server
ReplyListener: Received message: id=6, text=ReplyMsgBean processed message: text: id=6 to remote
 app server
Sent message: text: id=7 to local app server
ReplyListener: Received message: id=7, text=ReplyMsgBean processed message: text: id=7 to local
 app server
Sent message: text: id=8 to remote app server
ReplyListener: Received message: id=8, text=ReplyMsgBean processed message: text: id=8 to remote
 app server
Sent message: text: id=9 to local app server
ReplyListener: Received message: id=9, text=ReplyMsgBean processed message: text: id=9 to local
 app server
Sent message: text: id=10 to remote app server
```

```
ReplyListener: Received message: id=10, text=ReplyMsgBean processed message: text: id=10 to remote
 app server
Waiting for 0 message(s) from local app server
Waiting for 0 message(s) from remote app server
Finished
Closing connection 1
Closing connection 2
```

On the local system, where the message-driven bean receives the odd-numbered messages, the output in the server log looks like this (wrapped in logging information):

```
ReplyMsgBean: Received message: text: id=1 to local app server
ReplyMsgBean: Received message: text: id=3 to local app server
ReplyMsgBean: Received message: text: id=5 to local app server
ReplyMsgBean: Received message: text: id=7 to local app server
ReplyMsgBean: Received message: text: id=9 to local app server
```

On the remote system, where the bean receives the even-numbered messages, the output in the server log looks like this (wrapped in logging information):

```
ReplyMsgBean: Received message: text: id=2 to remote app server
ReplyMsgBean: Received message: text: id=4 to remote app server
ReplyMsgBean: Received message: text: id=6 to remote app server
ReplyMsgBean: Received message: text: id=8 to remote app server
ReplyMsgBean: Received message: text: id=10 to remote app server
```

Undeploy the modules after you finish running the client. To undeploy the `multiclient` module, perform these steps:

1. Verify that you are still in the directory `multiclient`.
2. Type the following command:

   `ant undeploy`

To undeploy the `replybean` module, perform these steps:

1. Change to the directory `replybean`:

   `cd ../replybean`

2. Type the following command:

   `ant undeploy`

3. Type the following command:

   `ant undeploy-remote -Dsys=`*`remote-system-name`*

   Replace *remote-system-name* with the actual name of the remote system.

To remove the generated files, use the following command in both the `replybean` and `multiclient` directories:

```
ant clean
```

Index

P

Q

S

T