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:

- Writing Simple JMS Applications
- Writing Robust JMS Applications
- An Application That Uses the JMS API with a Session Bean
- An Application That Uses the JMS API with an Entity
- An Application Example That Consumes Messages from a Remote Server
- An Application Example That Deploys a Message-Driven Bean on Two Servers

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 NetBeans IDE or the Ant tool to deploy the example and create resources for it.

3. Use NetBeans IDE, the `appclient` command, 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/`

Each example has a `setup/glassfish-resources.xml` file that is used to create resources for the example.

See Chapter 25, 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.

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 enterprise 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 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 GlassFish 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
(lookup ="jms/ConnectionFactory")
private static ConnectionFactory
connectionFactory;
@Resource
(lookup = "jms/Queue")
private static Queue queue;
@Resource
(lookup = "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:

```java
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 GlassFish Server, your JMS provider is the GlassFish Server.

Start the server as described in Starting and Stopping the GlassFish Server.

JMS Administered Objects for the Synchronous Receive Example

This example uses the following JMS administered objects:

- **A connection factory**
- **Two destination resources, a topic and a queue**

NetBeans IDE and the Ant tasks for the JMS examples create needed JMS resources when you deploy the applications, using a file named `setup/glassfish-resources.xml`.

This file is most easily created using NetBeans IDE, although you can create it by hand.

You can also use the

`asadmin create-jms-resource` command

to create resources, and the

`asadmin delete-jms-resource` command

to remove them.

To Create JMS Resources Using NetBeans IDE

Follow these steps to create a JMS resource in GlassFish Server using NetBeans IDE.

Repeat these steps for each resource you need.

The example applications in Chapter 46, Java Message Service Examples already have the resources, so you will need to follow these steps only when you create your own applications.

1. Right-click the project for which you want to create resources and choose **New**, then choose Other.

The **New** File wizard opens.

2. Under Categories, select GlassFish.

3. Under File Types, select JMS Resource.

The General Attributes - JMS Resource page opens.

4. In the JNDI Name field, type the name of the resource.

By convention, JMS resource names begin with `jms/`.

5. Select the radio button for the resource type.

   Normally, this is either `javax.jms.Queue`, `javax.jms.Topic`, or `javax.jms.ConnectionFactory`.

6. Click Next.

The JMS Properties page opens.

7. **For a queue or topic, type a name for a physical queue in the Value field for the Name property.**

You can type any value for this required field.

Connection factories have no required properties.

In a few situations, discussed in later sections, you may need to specify a property.

8. Click Finish.

A file named `glassfish-resources.xml` is created in your project, in a directory named `setup`.

In the project pane, you can find it under the Server Resources node.

If this file exists, resources are created automatically by NetBeans IDE when you deploy the project.

To Delete JMS Resources Using NetBeans IDE

1. In the Services pane, expand the Servers node, then expand the GlassFish Server 3.1 node.

2. Expand the Resources node, then expand the Connector Resources node.

3. Expand the Admin Object Resources node.

4. Right-click any destination you want to remove and select Unregister.

5. Expand the Connector Connection Pools node.

6. Right-click any connection factory you want to remove and select Unregister.

Every connection factory has both a connector connection pool and an associated connector resource.

When you remove the connector connection pool, the resource is removed automatically.

You can verify the removal by expanding the Connector Resources node.

Building, Packaging, Deploying, and Running the Clients for the Synchronous Receive Example

To run these examples using the GlassFish 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, package, and deploy 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.

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

To Build and Package the Clients for the Synchronous Receive Example Using NetBeans IDE

1. From the File menu, choose Open Project.

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. In the Projects tab, right-click the project and select Build.

7. Select the `synchconsumer` folder.

8. Select the Open as Main Project check box.

9. Click Open Project.

10. In the Projects tab, right-click the project and select Build.

To Deploy and Run the Clients for the Synchronous Receive Example Using NetBeans IDE

1. **Deploy and run the `Producer` example:**

   a. **Right-click the `producer` project and select 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 select Run.

The output of the program looks like this (along with some additional 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.

Note - When you run an application client, the command often takes a long time to complete.

2. Now deploy and run the `SynchConsumer` example:

a. Right-click the `synchconsumer` project and select 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 select Run.

The output of the program looks like this (along with some additional 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
```

3. Now try running the programs in the opposite order.

Right-click the `synchconsumer` project and select Run.

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

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

The output looks like this (along with some additional 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
```

6. Now run the `SynchConsumer` example using the topic.

   a. Right-click the `synchconsumer` project and select 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 select 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 for details.) Instead of receiving the messages, the program appears to hang.

7. Run the `Producer` example again.

Right-click the `producer` project and select 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
```

To Build and Package the Clients for the Synchronous Receive Example Using Ant

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.

To Deploy and Run the Clients for the Synchronous Receive Example Using Ant and the `appclient` Command

You can run the clients using the `appclient` command.

The `build.xml` file for each project includes a target that creates resources, 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 build, deploy, and run the `Producer` and `SynchConsumer` examples using Ant and the `appclient` command, follow these steps.

To run the clients, you need two terminal windows.

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

```
cd ../producer
```

2. Create any needed resources, deploy the client JAR file to the GlassFish 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 additional 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.

Note - When you run an application client, the command often takes a long time to complete.

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

```
cd ../synchconsumer
```

5. Deploy the client JAR file to the GlassFish 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 additional 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:

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

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

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

9. `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.

10. 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 additional 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
```

11. 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, for details.) Instead of receiving the messages, the client appears to hang.

12. Run the `Producer` client again.

Now the `SynchConsumer` client receives the messages (along with some additional 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
```

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 GlassFish Server.

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.

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:

3.

```
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 for A Simple Example of Synchronous Message Receives.

To Build and Package the `AsynchConsumer` Client Using NetBeans IDE

1. From the File menu, choose Open Project.

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. In the Projects tab, right-click the project and select Build.

To Deploy and Run the Clients for the Asynchronous Receive Example Using NetBeans IDE

1. Run the `AsynchConsumer` example:
   a. Right-click the `asynchconsumer` project and select 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 select 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 select 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 select 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 `Producer` client 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 select 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 select 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 select 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 select 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 Build and Package the `AsynchConsumer` Client Using Ant

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.

To Deploy and Run the Clients for the Asynchronous Receive Example Using Ant and the `appclient` Command

1. **Deploy the client JAR file to the GlassFish 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 additional 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.

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

The output of the client looks like this (along with some additional 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 additional 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 additional 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.

This section then explains how to compile, package, and run the example using the GlassFish Server.

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 GlassFish 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:14-128.149.71.199(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 for A Simple Example of Synchronous Message Receives.

To Build, Package, Deploy, and Run the `MessageBrowser` Client Using NetBeans IDE

To build, package, deploy, and run the `MessageBrowser` example using NetBeans IDE, follow these steps.

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.

1. From the File menu, choose Open Project.

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. In the Projects tab, right-click the project and select Build.

7. Run the `Producer` client, sending one message to the queue:
   a. Right-click the `producer` project and select 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 select Run.

The output of the client looks like this:

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

8. Run the `MessageBrowser` client.

Right-click the `messagebrowser` project and select Run.

The output of the client looks something like this:

```
Message:
Text: This is message 1 from producer
Class: com.sun.messaging.jmq.jmsclient.TextMessageImpl
getJMSMessageID(): ID:12-
128.149.71.199(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-
128.149.71.199(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
```

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

9. Run the `SynchConsumer` client to consume the messages.
   a. Right-click the `synchconsumer` project and select 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 select Run.

The output of the client looks like this:

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

To Build, Package, Deploy, and Run the `MessageBrowser` Client Using Ant and the `appclient` Command

To build, package, deploy, and run the `MessageBrowser` example using Ant, follow these steps.

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.

To run the clients, you need two terminal windows.

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.

3. Go to the `producer` directory.

4. 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 additional output):

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

5. Go to the `messagebrowser` directory.

6. Deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:

```
ant getclient
```

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

7. Because this example takes no command-line arguments, you can run the `MessageBrowser` client using the following command:

```
ant run
```

Alternatively, you can type the following command:

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

The output of the client looks something like this (along with some additional output):

```
Message:
Text: This is message 1 from producer
Class: com.sun.messaging.jmq.jmsclient.TextMessageImpl
```

```
getJMSMessageID(): ID:12-
128.149.71.199(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-
128.149.71.199(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.

8. Go to the `synchconsumer` directory.

9. 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 additional output):

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

Running JMS Clients on Multiple Systems

JMS clients that use the GlassFish 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 GlassFish Server.

Note - Any mechanism for exchanging messages between systems is specific to the Java EE server implementation.

This tutorial describes how to use the GlassFish 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:

1. Create two new connection factories

2. Change the name of the default JMS host on one system

3. Edit the source code for the two examples

4. Recompile and repackage the examples

Note - A limitation in the JMS provider in the GlassFish 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.

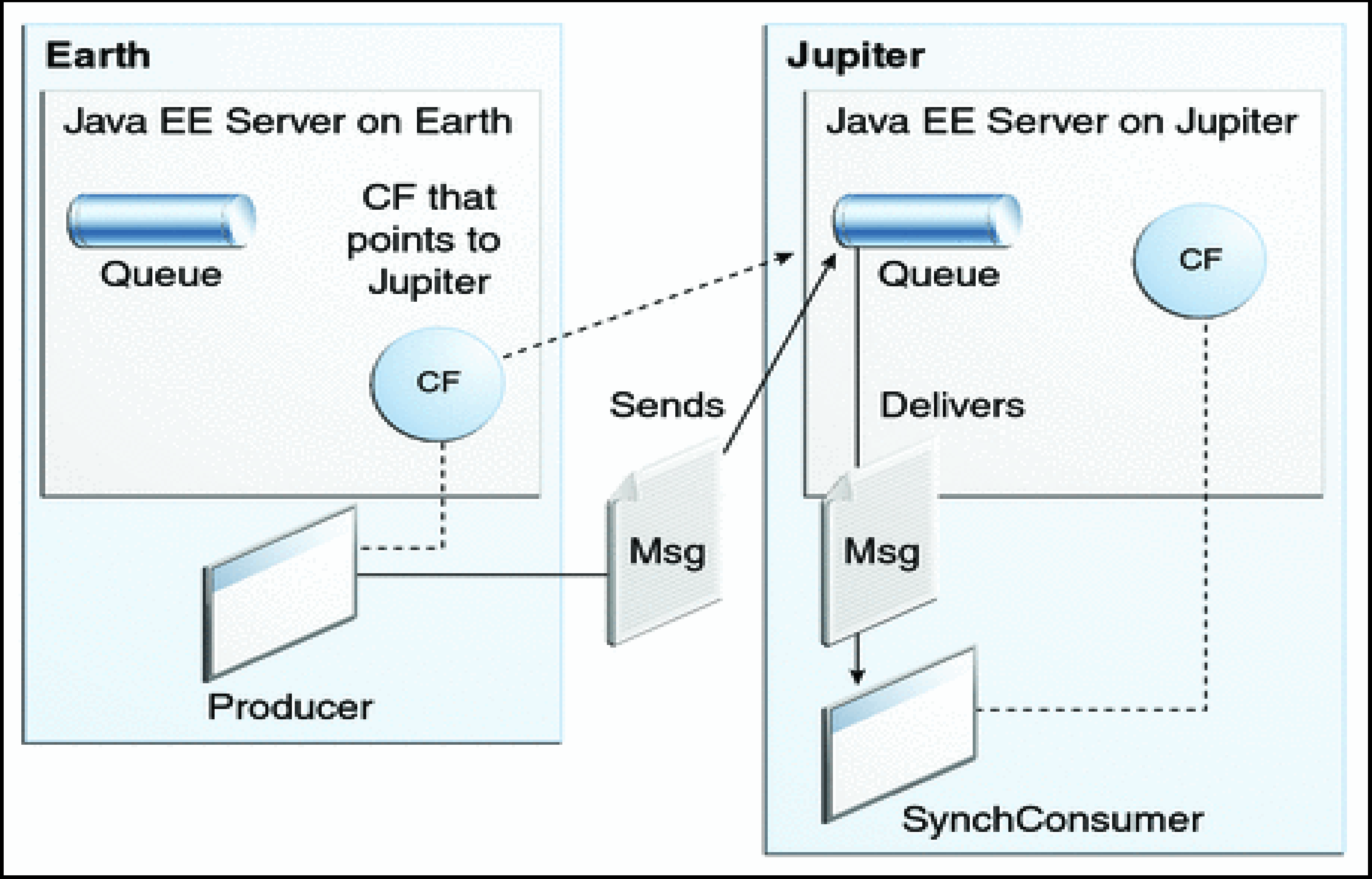
When you run the clients, they will work as shown in Figure 46-1.

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 46-1 Sending Messages from One System to Another



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

To Create 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.**

1. **Start the GlassFish Server on** `earth`**.**

2. Start the GlassFish Server on `jupiter`.

3. To create a new connection factory on `jupiter`, follow these steps:

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

b. 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`.

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

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

b. 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.

5. Additional resources will be created when you deploy the application, if they have not been created before.

6. The reason
the `glassfish-resources.xml` file does not specify `jms/JupiterConnectionFactory` is that on `earth` the connection factory requires the `AddressList` property setting, whereas on `jupiter` it does not.

You can examine the targets in the `build.xml` file for details.

Changing the Default Host Name

By default, the default host name for the JMS service on the GlassFish Server is `localhost`.

To access the JMS service from another system, however, you must change the host name.

You can change it to either the actual host name or to `0.0.0.0`.

You can change the default host name using either the Administration Console or the `asadmin` command.

To Change the Default Host Name Using the Administration Console

1. On `jupiter`, start the Administration Console by opening a browser at `http://localhost:4848/`.

2. In the navigation tree, expand the Configurations node, then expand the server-config node.

3. Under the server-config node, expand the Java Message Service node.

4. Under the Java Message Service node, expand the JMS Hosts node.

5. Under the JMS Hosts node, select `default_JMS_host`.

   The Edit JMS Host page opens.

6. In the Host field, type the name of the system, or type `0.0.0.0`.

7. Click Save.

8. Restart the GlassFish Server.

To Change the Default Host Name Using the `asadmin` Command

1. Specify a command like one of the following:

```
asadmin set server-config.jms-service.jms-host.default_JMS_host.host="0.0.0.0"
asadmin set server-config.jms-service.jms-host.default_JMS_host.host="hostname"
```

2. Restart the GlassFish Server.

To Edit, Build, Package, Deploy, and Run the Clients Using NetBeans IDE

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

You will edit the source files to specify the new connection factory.

Then you will rebuild and run the clients.

Follow these steps.

1. To edit the source files, follow these steps:

a. On `earth`,, open the following file in NetBeans IDE:

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

b. Find the following line:

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

c. Change the line to the following:

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

d. Save the file.

e. On `jupiter`, open the following file inNetBeans IDE:

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

f. Repeat Step b and Step c, then save the file.

2. To recompile and repackage the `Producer` example on `earth`, right-click the `producer` project and select Clean and Build.

3. To recompile and repackage the `SynchConsumer` example on `jupiter`, right-click the `synchconsumer` project and select Clean and Build.

4. On `earth`, deploy and run `Producer`.

5. Follow these steps:

a. Right-click the `producer` project and select 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 select Run.

The output looks like this (along with some additional 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
```

6. On `jupiter`, run `SynchConsumer`.

Follow these steps:

a. Right-click the `synchconsumer` project and select 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 select Run.

The output of the program looks like this (along with some additional 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
```

To Edit, Build, Package, Deploy, and Run the Clients Using Ant and the `appclient` Command

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

You will edit the source files to specify the new connection factory.

Then you will rebuild and run the clients.

1. To edit the source files, follow these steps:

   a. On `earth`,, open the following file in a text editor:

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

b. Find the following line:

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

c. Change the line to the following:

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

d. Save and close the file.

e. On `jupiter`, open the following file in a text editor:

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

f. Repeat Step b and Step c, then save and close the file.

2. To recompile and repackage the `Producer` example on `earth`, type the following:

```
ant
```

3. To recompile and repackage the `SynchConsumer` example on `jupiter`, go to the `synchconsumer` directory and type the following:

```
ant
```

4. On `earth`, deploy and run `Producer`.

Follow these steps:

a. On `earth`, from the `producer` directory, create any needed resources, deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:

```
ant getclient
```

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

b. To run the client, type the following:

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

The output looks like this (along with some additional 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
```

5. On `jupiter`, run `SynchConsumer`.

Follow these steps:

a. From the `synchconsumer` directory, create any needed resources, deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:

```
ant getclient
```

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

b. To run the client, type the following:

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

The output of the program looks like this (along with some additional 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
```

Undeploying and Cleaning the Simple JMS Examples

After you finish running the examples, you can undeploy them and remove the build artifacts.

You can also use the `asadmin delete-jms-resource` command to delete the destinations and connection factories you created.

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 GlassFish Server.

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 for A Simple Example of Synchronous Message Receives.

- `jms/ControlQueue`: an additional queue

- `jms/DurableConnectionFactory`: a connection factory with a client ID (see Creating Durable Subscriptions, for more information)

The new queue and connection factory are created at deployment time.

To Build, Package, Deploy, and Run the `ackequivexample` Using NetBeans IDE

1. To build and package the client, follow these steps.

   a. From the File menu, choose Open Project.

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. In the Projects tab, right-click the project and select Build.

2. To run the client, right-click the `ackequivexample` project and select Run.

The client output looks something like this (along with some additional 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
```

3. After you run the client, you can delete the destination resource `jms/ControlQueue` by using the following command:

```
asadmin delete-jms-resource
jms/ControlQueue
```

You will need the other resources for other examples.

To Build, Package, Deploy, and Run `ackequivexample` *Using Ant*

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

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

2. To compile and package the client, type the following command:

```
ant
```

3. To create needed resources, deploy the client JAR file to the GlassFish Server, then retrieve the client stubs, type the following command:

```
ant getclient
```

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

4. Because this example takes no command-line arguments, you can run the client using the following command:

```
ant run
```

Alternatively, you can type the following command:

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

The client output looks something like this (along with some additional 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
```

5. After you run the client, you can delete the destination resource `jms/ControlQueue` by using the following command:

```
asadmin delete-jms-resource
jms/ControlQueue
```

You will need the other resources for other examples.

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.

To Build, Package, Deploy, and Run `durablesubscriberexample` Using NetBeans IDE

1. **To compile and package the client, follow these steps:**

   a. **From the File menu, choose Open Project.**

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. In the Projects tab, right-click the project and select Build.

2. To run the client, right-click the `durablesubscriberexample` project and select Run.

The output looks something like this (along with some additional 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
```

3. After you run the client, you can delete the connection factory `jms/DurableConnectionFactory` by using the following command:

```
asadmin delete-jms-resource
jms/DurableConnectionFactory
```

To Build, Package, Deploy, and Run `durablesubscriberexample` Using Ant

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

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

2. To compile and package the client, type the following command:

```
ant
```

3. To create any needed resources, deploy the client JAR file to the GlassFish Server, then retrieve the client stubs, type the following command:

```
ant getclient
```

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

4. **Because this example takes no command-line arguments, you can run the client using the following command:**

```
ant run
```

Alternatively, you can type the following command:

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

5. After you run the client, you can delete the connection factory `jms/DurableConnectionFactory` by using the following command:

```
asadmin delete-jms-resource
jms/DurableConnectionFactory
```

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.

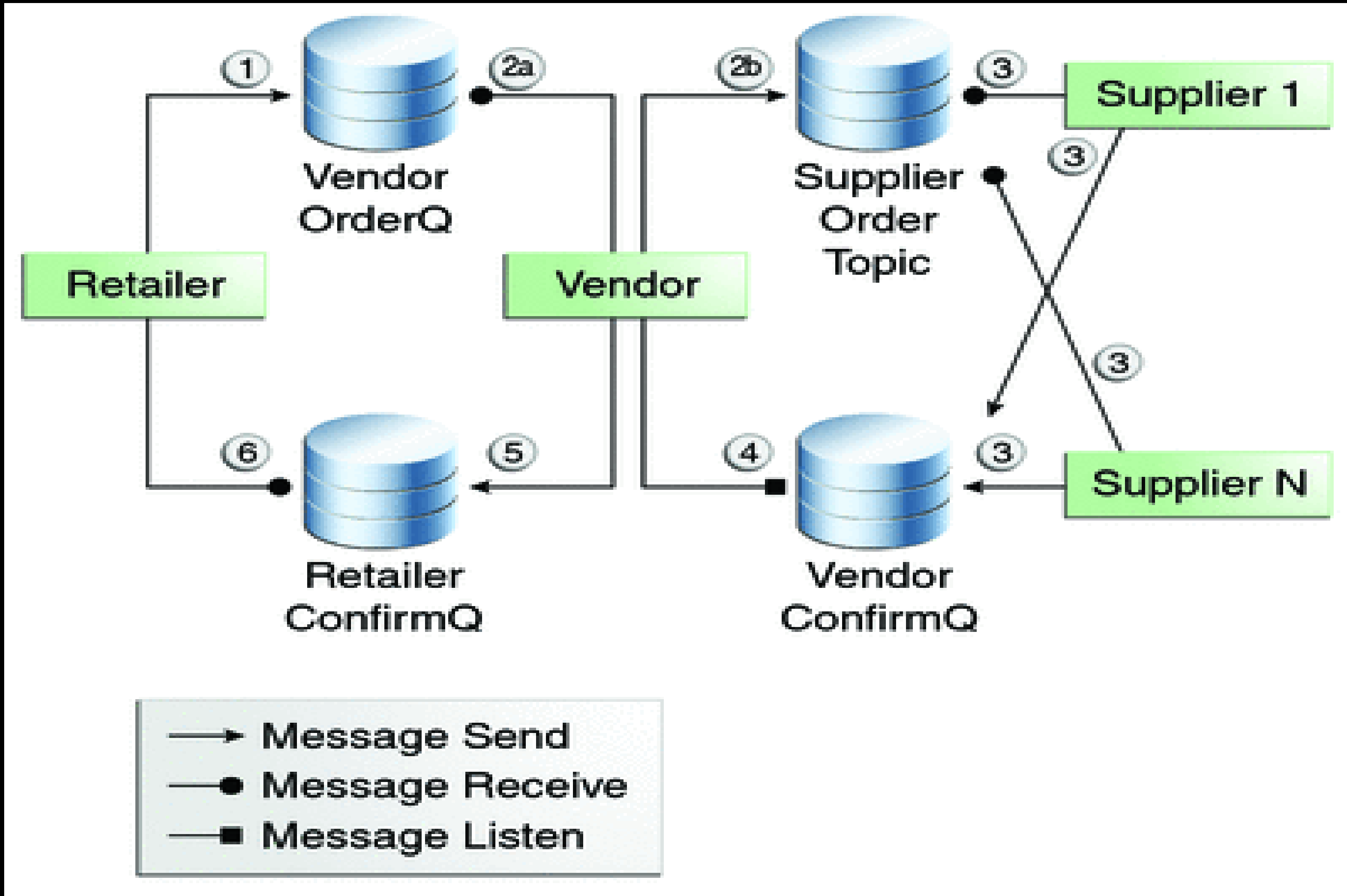
```
Queue replyQueue = (Queue)
order.order.getJMSReplyTo();
MessageProducer producer =
session.createProducer(replyQueue);
MapMessage retailerConfirmMessage =
session.createMapMessage();
// Format the message
producer.send
(retailerConfirmMessage);
session.commit();
```

6. The retailer receives the message from the vendor:

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

Figure 46-2 illustrates these steps.

Figure 46-2 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`.

To Build, Package, Deploy, and Run ***`transactedexample`*** *Using NetBeans IDE*

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

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

2. To compile and package the client, follow these steps:

   a. From the File menu, choose Open Project.

   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. In the Projects tab, right-click the project and select Build.

3. To deploy and run the client, follow these steps:

a. Right-click the `transactedexample` project and select Properties.

b. Select Run from the Categories tree.

c. In the Arguments field, type a number that specifies the number of computers to order:

```
3
```

d. Click OK.

e. Right-click the project and select Run.

The output looks something like this (along with some additional 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(Unkn
own 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
```

4. After you run the client, you can delete the destination resources from the IDE or by using the following commands:

```
asadmin
delete-jms-resource jms/AQueue
```

```
asadmin
delete-jms-resource jms/BQueue
asadmin
delete-jms-resource jms/CQueue
asadmin
delete-jms-resource jms/OTopic
```

To Build, Package, Deploy, and Run `transactedexample` Using Ant and the `appclient` Command

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

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

2. To build and package the client, type the following command:

```
ant
```

3. Create needed resources, deploy the client JAR file to the GlassFish Server, then retrieve the client stubs:

```
ant getclient
```

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

4. 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 additional 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(Unkn
own 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
```

5. After you run the client, you can delete the destination resources by using the following command:

```
asadmin
delete-jms-resource jms/AQueue
asadmin
delete-jms-resource jms/BQueue
asadmin
delete-jms-resource jms/CQueue
asadmin
delete-jms-resource jms/OTopic
```

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**

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

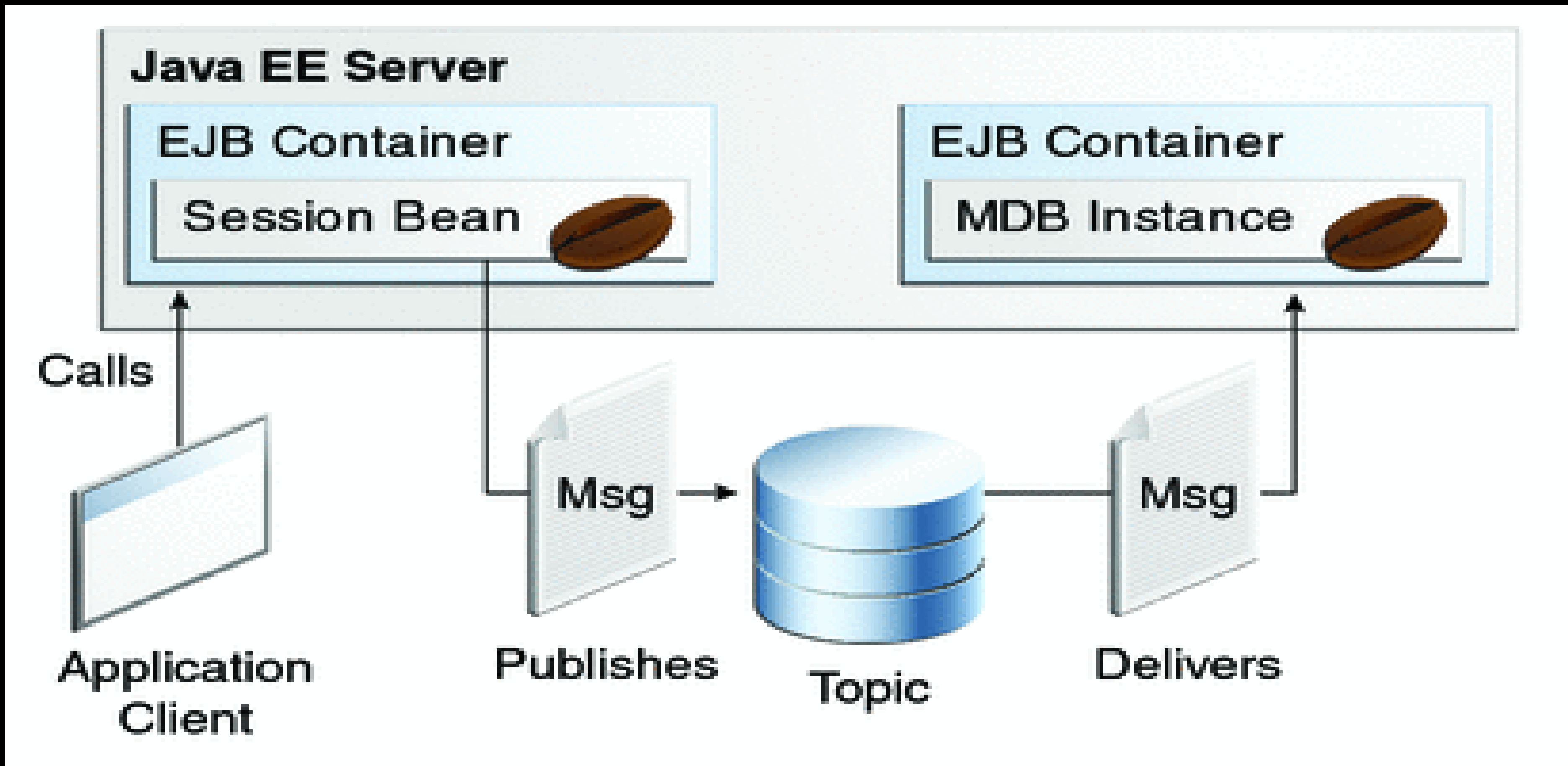
Path names in this section are relative to this directory.

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 25, A Message-Driven Bean Example.

Figure 46-3 illustrates the structure of this application.

Figure 46-3 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.

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 25, 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
(lookup = "jms/ConnectionFactory")
private ConnectionFactory
connectionFactory;
@Resource(lookup = "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 25, 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")
})
```

Note - For a message-driven bean, the destination is specified with the `mappedName` element instead of the `lookup` element.

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 are used in previous examples..

If you deleted the connection factory or topic, they will be recreated when you deploy the example.

To Build, Package, Deploy, and Run the `clientsessionmdb` Example Using NetBeans IDE

1. To compile and package the project, follow these steps:

   a. From the File menu, choose Open Project.

b. In the Open Project dialog, navigate to:

`tut-install/examples/jms/`

c. Select the `clientsessionmdb` folder.

d. Select the Open as Main Project check box and the Open Required Projects check box.

e. Click Open Project.

f. In the Projects tab, right-click the `clientsessionmdb` project and select 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

2. Right-click the project and select Run.

This command creates any needed resources, 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`.

To Build, Package, Deploy, and Run the `clientsessionmdb` Example Using Ant

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.

3. To create any needed resources, 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`.

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

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.

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 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 46-4 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.

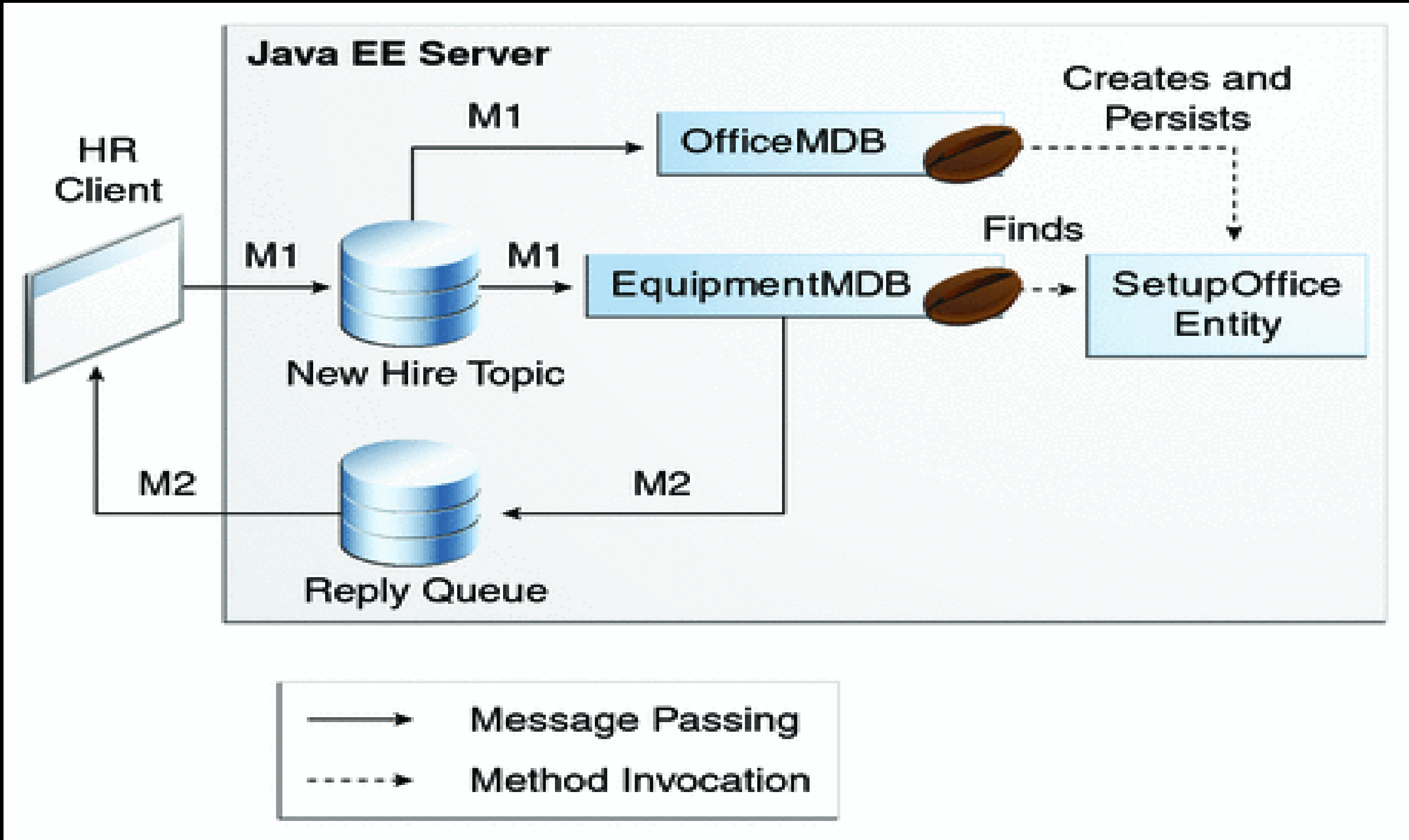
Figure 46-4 assumes that `OfficeMDB` is the first message-driven bean to consume the message from the client.

`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 46-4 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 coding the application client, the message-driven beans, and the entity class.

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/eb/EquipmentMDB.java`
- `clientmdbentity-ejb/src/java/eb/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, `clientmdbentity-ejb/src/java/eb/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:

```java
@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/persistence_2_0.xsd">
<persistence-unit
name="clientmdbentity-ejbPU"
transaction-type="JTA">
```

```
<provider>
org.eclipse.persistence.jpa.
PersistenceProvider
</provider>
<jta-data-source>
jdbc/__default
</jta-data-source>
<class>eb.SetupOffice</class>
<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.

It also uses the JDBC resource named `jdbc/__default`, which is enabled by default when you start the GlassFish Server.

If you deleted the connection factory or topic, they will be created when you deploy the example.

To Build, Package, Deploy, and Run the `clientmdbentity` Example Using NetBeans IDE

1. From the File menu, choose Open Project.

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. In the Projects tab, right-click the `clientmdbentity` project and select 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. If the Java DB database is not already running, follow these steps:
   a. Click the Services tab.
   b. Expand the Databases node.
   c. Right-click the Java DB node and select Start Server.

8. In the Projects tab, right-click the project and select Run.

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

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

```
PUBLISHER: Setting hire ID to 50,
name Bill Tudor, position
Programmer
PUBLISHER: Setting hire ID to 51,
name Carol Jones, position Senior
Programmer
PUBLISHER: Setting hire ID to 52,
name Mark Wilson, position
Manager
```

```
PUBLISHER: Setting hire ID to 53,
name Polly Wren, position Senior
Programmer
PUBLISHER: Setting hire ID to 54,
name Joe Lawrence, position
Director
Waiting for 5 message(s)
New hire event processed:
 Employee ID: 52
 Name: Mark Wilson
 Equipment: PDA
```

```
 Office number: 294
Waiting for 4 message(s)
New hire event processed:
 Employee ID: 53
 Name: Polly Wren
 Equipment: Laptop
 Office number: 186
Waiting for 3 message(s)
New hire event processed:
 Employee ID: 54
 Name: Joe Lawrence
```

```
  Equipment: Java Phone
  Office number: 135
Waiting for 2 message(s)
New hire event processed:
  Employee ID: 50
  Name: Bill Tudor
  Equipment: Desktop System
  Office number: 200
Waiting for 1 message(s)
New hire event processed:
  Employee ID: 51
```

```
Name: Carol Jones
Equipment: Laptop
Office number: 262
```

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.

To Build, Package, Deploy, and Run the `clientmdbentity` Example Using Ant

1. Go to the following directory:

   `tut-install``/examples/jms/clientmdbentity/`

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 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

3. To create any needed resources, 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 50,
name Bill Tudor, position
Programmer
PUBLISHER: Setting hire ID to 51,
name Carol Jones, position Senior
Programmer
PUBLISHER: Setting hire ID to 52,
name Mark Wilson, position
Manager
```

```
PUBLISHER: Setting hire ID to 53,
name Polly Wren, position Senior
Programmer
PUBLISHER: Setting hire ID to 54,
name Joe Lawrence, position
Director
Waiting for 5 message(s)
New hire event processed:
 Employee ID: 52
 Name: Mark Wilson
 Equipment: PDA
```

```
  Office number: 294
 Waiting for 4 message(s)
 New hire event processed:
  Employee ID: 53
  Name: Polly Wren
  Equipment: Laptop
  Office number: 186
 Waiting for 3 message(s)
 New hire event processed:
  Employee ID: 54
  Name: Joe Lawrence
```

```
  Equipment: Java Phone
  Office number: 135
Waiting for 2 message(s)
New hire event processed:
  Employee ID: 50
  Name: Bill Tudor
  Equipment: Desktop System
  Office number: 200
Waiting for 1 message(s)
New hire event processed:
  Employee ID: 51
```

```
Name: Carol Jones
Equipment: Laptop
Office number: 262
```

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.

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 25, A Message-Driven Bean Example into two modules: one containing the application client, and the other containing the message-driven bean.

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 25, 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.

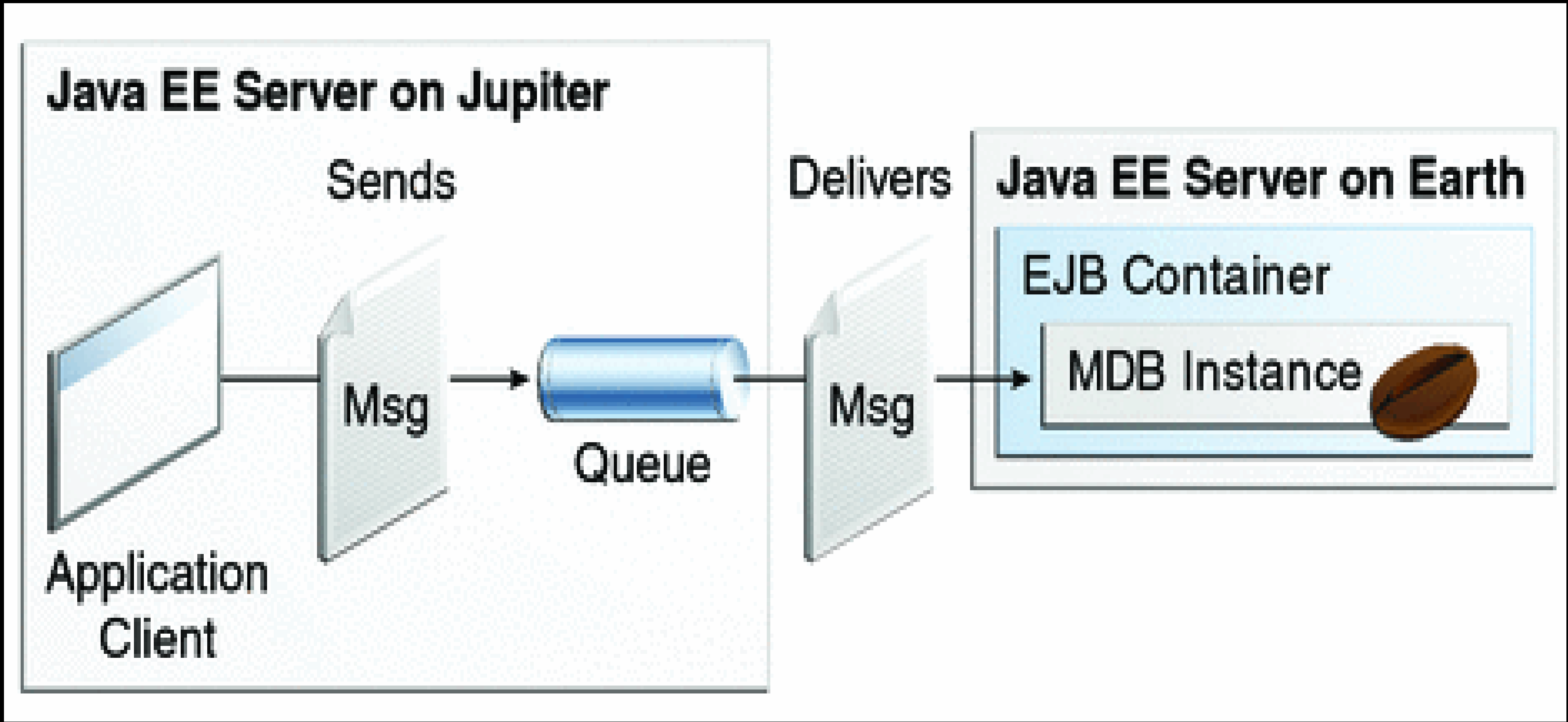
5. The message-driven bean consumes the messages.

Figure 46-5 illustrates the structure of this application.

You can see that it is almost identical to Figure 25-1 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 46-5 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.

Similarly, the message-driven bean, `earthmdb/src/java/MessageBean.java`, is almost identical to the one in The Message-Driven Bean Class.

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 for A Simple Example of Synchronous Message Receives.

The message-driven bean does not need a previously created connection factory; the resource adapter creates one for it.

Any missing resources will be created when you deploy the example.

Using Two Application Servers for the `consumeremote` Example

As in Running JMS Clients on Multiple Systems, the two servers are referred to as `earth` and `jupiter`.

The GlassFish Server must be running on both systems.

Before you can run the example, you must change the default name of the JMS host on `jupiter`, as described in To Change the Default Host Name Using the Administration Console.

If you have already performed this task, you do not have to repeat it.

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.)

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.

To Build, Package, Deploy, and Run the `consumeremote` Modules Using NetBeans IDE

To edit the message-driven bean source file and then package, deploy, and run the modules using NetBeans IDE, follow these steps.

1. From the File menu, choose Open Project.

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. In the Projects tab, expand the `earthmdb`, Source Packages, and `mdb` nodes, then double-click `MessageBean.java`.

b. Find the following line within the `@MessageDriven` annotation:

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

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

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

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

8. From the File menu, choose Open Project.

9. Select the `jupiterclient` folder.

10. Select the Open as Main Project check box.

11. Click Open Project.

12. In the Projects tab, right-click the `jupiterclient` project and select Build.

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

13. Right-click the `earthmdb` project and select Deploy.

14. To copy the `jupiterclient` module to the remote system, follow these steps:

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

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

b. Type a command like the following:

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

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

You can use the file system graphical user interface on your system instead of the command line.

15. To run the application client, follow these steps:

a. If you did not previously create the queue and connection factory on the remote system (`jupiter`), go to *tut-install*`/examples/jms/consumeremote/jupiterclient` on the remote system and type the following command: `ant add-resources`

b. Go to the directory on the remote system (`jupiter`) where you copied the client JAR file.

c. To deploy the client module and retrieve the client stubs, 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`

d. To run the client, 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 jupiterclient
Sending message: This is
message 2 from jupiterclient
Sending message: This is
message 3 from jupiterclient
```

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
jupiterclient
MESSAGE BEAN: Message received:
This is message 2 from
jupiterclient
```

```
MESSAGE BEAN: Message received:
This is message 3 from
jupiterclient
```

To Build, Package, Deploy, and Run the `consumeremote` Modules Using Ant

To edit the message-driven bean source file and then package, deploy, and run the modules using Ant, follow 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 `@MessageDriven` 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.

9. To copy the `jupiterclient` module to the remote system, follow these steps:

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

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

b. Type a command like the following:

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

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

10. To run the application client, follow these steps:

a. If you did not previously create the queue and connection factory on the remote system (`jupiter`), go to *tut-*

install/`examples/jms/consumeremote/jupiterclient` on the remote system and type the following command:

```
ant add-resources
```

b. Go to the directory on the remote system (`jupiter`) where you copied the client JAR file.

c. To deploy the client module and retrieve the client stubs, 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`

d. To run the client, 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 jupiterclient
Sending message: This is
message 2 from jupiterclient
Sending message: This is
message 3 from jupiterclient
```

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

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).

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 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.

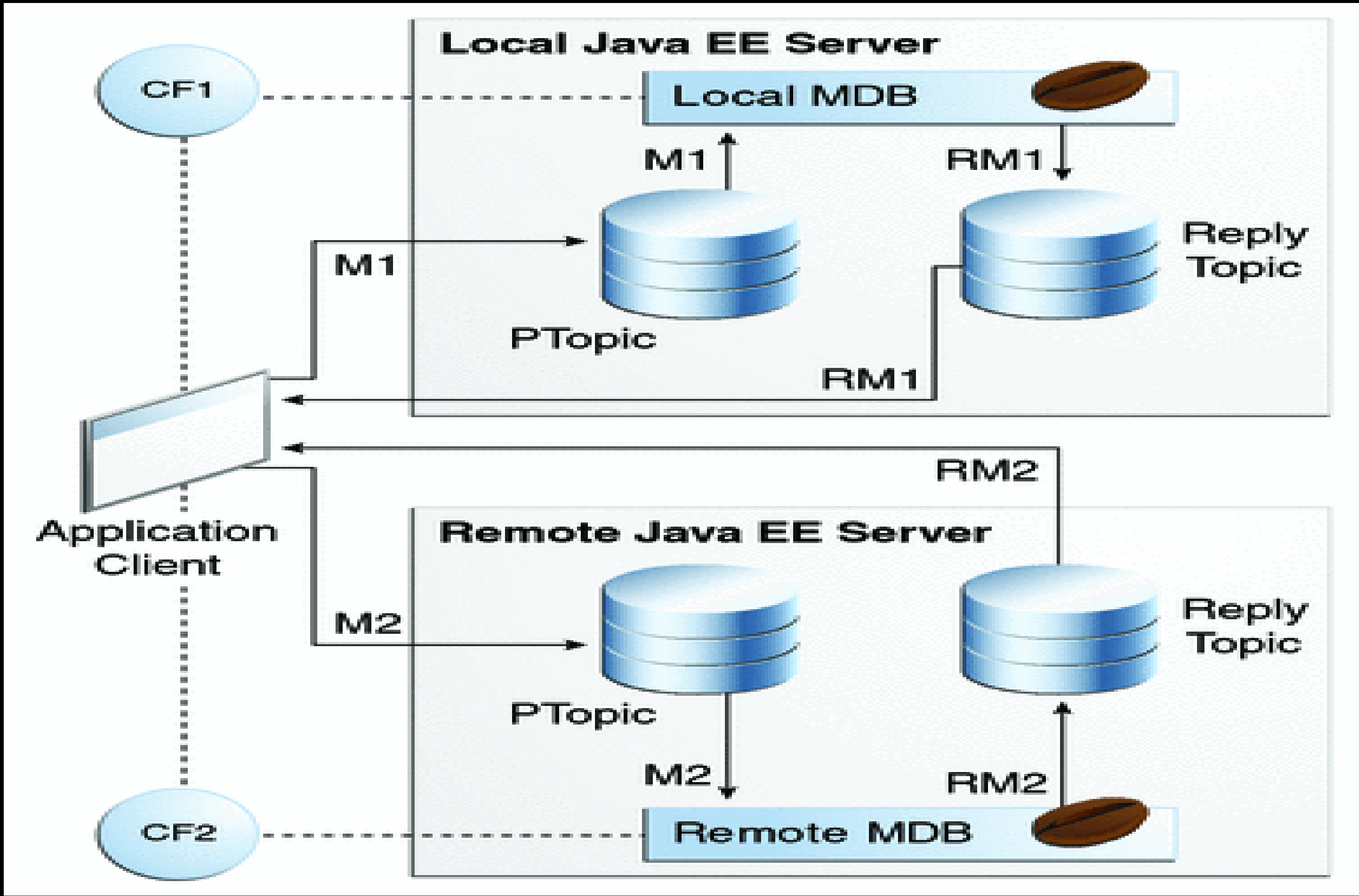
6. The client's message listener consumes the replies.

Figure 46-6 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 46-6 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 coding the application client and the message-driven bean.

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 To Create Administered Objects for Multiple Systems.

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.

To create the resource needed only on the local system, use the following command:

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

The other resources will be created when you deploy the application.

To Enable Deployment on the Remote System

GlassFish Server by default does not allow deployment from a remote system.

You must execute an `asadmin` command on the remote system to enable deployment of the message-driven bean on that system.

1. From a command prompt on the remote system (`jupiter`), run the following command:

```
asadmin enable-secure-admin
```

2. Stop and restart the server on `jupiter`.

To Use 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, follow these steps.

1. In NetBeans IDE, click the Runtime tab.

2. Right-click the Servers node and select Add Server.

   In the Add Server Instance dialog, follow these steps:

   a. Select GlassFish Server 3.1 from the Server list.

b. In the Name field, specify a name slightly different from that of the local server, such as `GlassFish Server 3.1 (2)`.

c. Click Next.

d. For the Server Location, browse to the location of the GlassFish Server on the remote system.

This location must be visible from the local system.

e. Click Next.

f. Select the Register Remote Domain radio button.

g. In the Host Name field, type the name of the remote system.

h. Click Finish.

Next Steps

Before you can run the example, you must change the default name of the JMS host on `jupiter`, as described in To Change the Default Host Name Using the Administration Console.

If you have already performed this task, you do not have to repeat it.

To Build, Package, Deploy, and Run the `sendremote` Modules Using NetBeans IDE

1. To build the `replybean` module, follow these steps:

   a. From the File menu, choose Open Project.

b. In the Open Project dialog, navigate to:

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

c. Select the `replybean` folder.

d. Select the Open as Main Project check box.

e. Click Open Project.

f. In the Projects tab, right-click the `replybean` project and select Build.

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

2. To build the `multiclient` module, follow these steps:
   a. From the File menu, choose Open Project.
   b. Select the `multiclient` folder.
   c. Select the Open as Main Project check box.
   d. Click Open Project.
   e. In the Projects tab, right-click the `multiclient` project and select Build.

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

3. To create any needed resources and deploy the `multiclient` module on the local server, follow these steps:

   a. Right-click the `multiclient` project and select Properties.

b. Select Run from the Categories tree.

c. From the Server list, select GlassFish Server 3.1 (the local server).

d. Click OK.

e. Right-click the `multiclient` project and select Deploy.

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

4. To deploy the `replybean` module on the local and remote servers, follow these steps:

   a. Right-click the `replybean` project and select Properties.

b. Select Run from the Categories tree.

c. From the Server list, select GlassFish Server 3.1 (the local server).

d. Click OK.

e. Right-click the `replybean` project and select Deploy.

f. Right-click the `replybean` project again and select Properties.

g. Select Run from the Categories tree.

h. From the Server list, select GlassFish Server 3.1 (2) (the remote server).

i. Click OK.

j. Right-click the `replybean` project and select Deploy.

You can use the Services tab to verify that `replybean` is deployed as an EJB Module on both servers.

5. To run the application client, right-click the `multiclient` project and select 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
```

To Build, Package, Deploy, and Run the `sendremote` Modules Using Ant

1. To package the modules, follow these steps:

   a. Go to the following directory:

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

b. Type the following command:

```
ant
```

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

c. Change to the directory `replybean`:

```
cd ../replybean
```

d. Type the following command:

```
ant
```

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

2. To deploy the `replybean` module on the local and remote servers, follow these steps:

a. Verify that you are still in the directory `replybean`.

b. Type the following command:

```
ant deploy
```

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

c. Type the following command:

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

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

3. To deploy and run the client, follow these steps:

a. Change to the directory `multiclient`:

```
cd ../multiclient
```

b. Type the following command:

```
ant getclient
```

c. 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
```