Steal This Book!

Yes, you read that right. Steal this book. For free.

Nothing. Zero. Zilch. Nadda. Zip.

Undoubtedly you're asking yourself, "Why would he give away a book he probably spent six grueling months writing? Is he crazy?"

The answer…is yes. Sort of. I know that every day you're faced with hundreds of computer titles, and a lot of them don't deliver the value or the information you need. So here's the deal: I'll give you this book (or this chapter, if you've only downloaded part of the book) for free provided you do me a couple favors:

1. **Send this book to your friends:** No, not your manager. Your "with it" computer friends who are looking for the next Big Thing. JXTA is it. Trust me. They want to know about it.

2. **Send a link to the book's web site:** Maybe the book is too big to send. After all, not everyone can have a fibre optic Internet connection installed in their bedroom. The site, at **www.brendonwilson.com/projects/jxta**, provides chapter-sized PDFs for easy downloading by the bandwidth-challenged.

3. **Visit the book's web site:** Being a professional developer, you probably have Carpal Tunnel Syndrome and shudder at the idea of typing in example source code. Save yourself the trouble. Go to **www.brendonwilson.com/projects/jxta** and download the source code. And while you're there, why not download some of the chapters you're missing?

4. **Buy the book:** You knew there had to be a catch. Sure, the book's PDFs are free, but I'm hoping that enough of you like the book so much that you have to buy a copy. Either that, or none of you can stand to read the book from a screen (or, worse yet, print it all out <shudder>) and resort to paper to save what's left of your eyesight. The book is available at your local bookstore or from **Amazon.com** (at a **handsome discount**, I might add).

I now return to your regularly scheduled program: enjoy the book!

8

The Pipe Binding Protocol

Pipes are constructs within JXTA that send data to or receive data from a remote peer. Services typically use either a Resolver handler (refer to Chapter 5, "The Peer Resolver Protocol") or a pipe to communicate with another peer. Before a pipe can actually be used, it must be bound to a peer endpoint. Binding a pipe to an endpoint allows the peers to create either an input pipe for receiving data or an output pipe for sending data. The process of binding a pipe to an endpoint is defined by the Pipe Binding Protocol (PBP).

This chapter explains the Pipe Binding Protocol that JXTA peers use to bind a pipe to an endpoint. The PBP defines a set of messages that a peer can use to query remote peers to find an appropriate endpoint for a given Pipe Advertisement and respond to binding queries from other peers. After a pipe has been bound to an endpoint, a peer can use it to send or receive messages. Several examples in the section "The Pipe Service" demonstrate the use of both input and output pipes to send and receive data, respectively.

Introducing the Pipe Binding Protocol

The *endpoint* is the bottom-most element in the network transport abstraction defined by JXTA. Endpoints are encapsulations of the native network interfaces provided by a peer. These network interfaces typically provide access to low-level transport protocols such as TCP or UDP, although some can provide access to higher-level transport protocols such as HTTP. Endpoints are responsible for producing, sending, receiving, and consuming messages sent across the network. Other services in JXTA build on endpoints either directly or indirectly to provide network connectivity. The Resolver service, for example, builds directly on endpoints, whereas the Discovery service builds on endpoints indirectly via the Resolver service.

In addition to the Resolver service, JXTA offers another mechanism by which services can access a network transport without interacting directly with the endpoint abstraction: pipes. *Pipes* are an abstraction in JXTA that describe a connection between a sending endpoint and one or more receiving endpoints. A pipe is a convenience method layered on top of the endpoint abstraction. Although pipes might appear to provide access to a network transport, implementations of the endpoint abstraction are responsible for the actual task of sending and receiving data over the network.

To provide an abstraction that can encompass the simplest networking technology, JXTA specifies pipes as *unidirectional*, meaning that data travels in only one direction. Pipes are also *asynchronous*, meaning that data can be sent or received at any time, a feature that allows peers to act independently of other peers without any sort of state synchronization. The JXTA Protocols Specification does specify that other types of pipes (bidirectional, synchronous, or streaming) might exist in JXTA. However, only the unidirectional asynchronous variety of pipe is required by the specification.

Pipes are described by a Pipe Advertisement using the XML shown in Listing 8.1.

Listing 8.1 **The Pipe Advertisement XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PipeAdvertisement>
  <Id> . . . </Id>
  <Type> . . . </Type>
  <Name> . . . </Name>
</jxta:PipeAdvertisement>
```

The elements of the Pipe Advertisement provide required information that a peer can use to find and connect to a remote peer:

- **`Id`**—A required element containing an ID that uniquely identifies the pipe. This Pipe ID uses the standard JXTA URN format, as described in JXTA Protocols Specification.
- **`Type`**—A required element containing a description of the type of connection possible using the pipe. Currently, the reference implementation supports `JxtaUnicast`, `JxtaUnicastSecure`, and `JxtaPropagate`. The `JxtaUnicast` type of pipe provides a basic connection between one sending endpoint and one receiving endpoint. The `JxtaUnicastSecure` type of pipe provides the same functionality as the `JxtaUnicast` type of pipe, except that the connection is secured using the Transport Layer Security (TLS) protocol. The `JxtaPropagate` type of pipe provides a broadcast connection between many sending endpoints and multiple receiving endpoints.
- **`Name`**—An optional element containing a symbolic name for the pipe that can be used to discover the Pipe Advertisement using the Discovery service.

Notice that the Pipe Advertisement seems to be missing one important piece of information: a Peer ID. Pipe Advertisements are defined without specifying a specific peer to allow several peers to provide access to a service using the same Pipe Advertisement. The omission of a Peer ID is the reason that pipes must be resolved using the Pipe Binding Protocol.

When a peer wants to send data using a pipe, it needs to find a peer that has already bound a pipe with the same Pipe ID to an endpoint and that is listening for data. The PBP defines two messages to enable a peer to resolve a pipe:

- **The Pipe Binding Query Message**—A message format for querying a remote peer if it has bound a pipe with a matching Pipe ID.
- **The Pipe Binding Answer Message**—A message format for sending responses to the query.

The message formats are all that a peer needs to resolve the ID of a peer that has a bound pipe with a given Pipe ID. As shown in Figure 8.1, when a peer wants to bind to a specific pipe, it sends a Pipe Binding Query Message to all of its known peers and rendezvous peers. Peers respond with a Pipe Binding Answer Message that details whether they have a matching bound pipe.

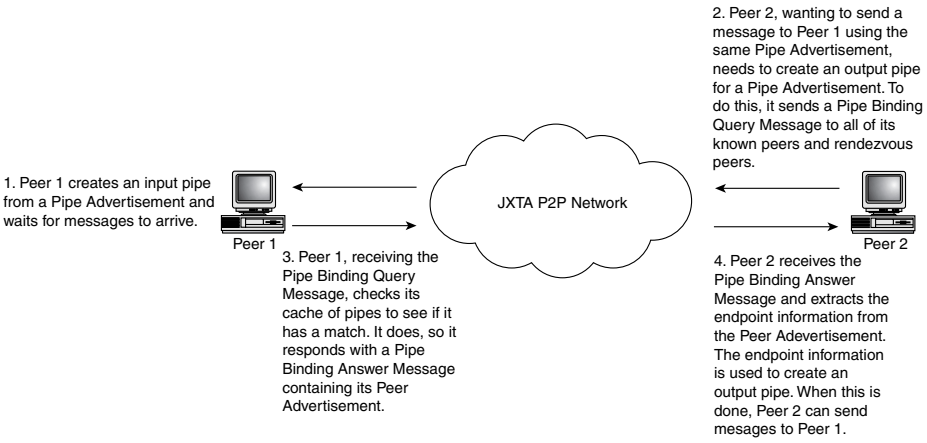

Figure 8.1 Exchange of Pipe Binding Messages.

Two important things should be noted from Figure 8.1. First, when Peer 1 creates an input pipe, nothing is sent to the network. Peer 1 simply begins listening on its local endpoints for incoming messages tagged with the Pipe ID specified in the Pipe Advertisement. Second, the Pipe Advertisement doesn't necessarily need to be communicated over the network. Although a Pipe Advertisement usually is discovered using the Discovery service, a Pipe Advertisement could also be hard-coded into an application or exchanged using the Resolver service.

After the receiving end of the pipe has been resolved to a particular endpoint on a remote peer, the peer can bind the other end of the pipe to its local endpoint. This pipe on the local peer is called an *output pipe* because the pipe has been bound to an endpoint for the purpose of sending output to the remote peer. The bound pipe on the remote peer is called an *input pipe* because the pipe has been bound to an endpoint for the purpose of accepting input. After the sending peer binds an output pipe, it can send messages to the remote peer.

Only the endpoint location of the pipe on a remote peer must be determined in the binding process to create an output pipe. When creating an input pipe, no binding process is necessary because the local peer already knows that it will be binding the Pipe Advertisement to its local endpoint for the purpose of accepting data.

It is important to reiterate that neither the input pipes nor the output pipes are actually responsible for sending or receiving data. The endpoints specified by the bound pipe are the elements responsible for handling the actual exchange of messages over the network.

In the case of propagation pipes (when the Pipe Advertisement's `Type` is set to `JxtaPropagate`), the implementation relies on the multicast or broadcast capabilities of the local endpoint. In this case, the PBP is not required because the sending endpoint doesn't need to find a listening endpoint before it can send data to the network.

The Pipe Binding Query Message

The Pipe Binding Query Message is sent by a peer to resolve the ID of a peer that has bound an input pipe with a specific Pipe ID. Listing 8.2 shows the format of the Pipe Binding Query Message.

Listing 8.2 **The Pipe Binding Query Message XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PipeResolver>
  <MsgType>Query</MsgType>
  <PipeId> . . . </PipeId>
  <Type> . . . </Type>
  <Cached> . . . </Cached>
  <Peer> . . . </Peer>
</jxta:PipeResolver>
```

The information sent in the Pipe Binding Query Message describes the pipe that the peer is seeking to resolve and tells whether to use cached information.

- **`MsgType`**—A required element containing a string that indicates the type of Pipe Binding Message. For the Pipe Binding Query Message, this element is hard-coded to `Query`.
- **`PipeId`**—A required element containing the ID of the pipe for which the requesting peer is attempting to resolve a Peer ID.
- **`Type`**—A required element containing the type of pipe being resolved. This corresponds to the `Type` field of the Pipe Advertisement, and it can have a value of `JxtaUnicast`, `JxtaUnicastSecure`, or `JxtaPropagate`.
- **`Cached`**—An optional element that specifies whether the remote peer being queried can use its local cache of resolved pipes to respond to the query. If this parameter is missing, the peer receiving the query assumes that it is allowed to use cached information.
- **`Peer`**—According to the specification, this optional element specifies the Peer ID of the only peer that should respond to the query. However, the current reference implementation does not send this parameter yet; which peers receive the query is specified by the service interface rather than the protocol.

The reference implementation doesn't define any classes to encapsulate the Pipe Binding Query Message.

The Pipe Binding Answer Message

A peer responds to a Pipe Binding Query Message using a Pipe Binding Answer Message. Note that response might or might not be sent to a given query. Responses received are useful only to update the local peer's cached set of resolved pipes. The Pipe Binding Answer Message comes in two forms: one to indicate that a matching pipe was not found and another to indicate a matching pipe was found. Listing 8.3 shows the format of the Pipe Binding Answer Message.

Listing 8.3 **The Pipe Binding Answer Message XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PipeResolver>
  <MsgType>Answer</MsgType>
  <PipeId> . . . </PipeId>
  <Type> . . . </Type>
  <Peer> . . . </Peer>
  <Found>false</Found>
  <PeerAdv> . . . </PeerAdv>
</jxta:PipeResolver>
```

The elements of the Pipe Binding Answer Message are nearly identical to those of the Pipe Binding Query Message, with the following exceptions:

- **`MsgType`**—A required element containing a string that indicates the type of Pipe Binding Message. For the Pipe Binding Response Message, this element is hard-coded to `Answer`.
- **`Found`**—An optional element that indicates whether a matching Pipe ID was found. If this element is missing, the reference implementation assumes that a peer with a matching Pipe ID was found.
- **`PeerAdv`**—An optional element containing the Peer Advertisement of the peer that has the matching Pipe ID. If no match was found, this element does not appear in the Pipe Binding Answer Message. The endpoint information required to contact a remote peer using a specific pipe is included as part of the Peer Advertisement.

As with the Pipe Binding Query Message, the reference implementation provides no classes to abstract the Pipe Binding Answer Message.

The Pipe Service

As with every other protocol in JXTA, the Pipe Binding Protocol is provided as a service. In the case of the PBP, the Pipe service is responsible for handling the details of creating input or output pipe objects and binding those pipe objects to endpoints. The Pipe service, as shown in Figure 8.2, is defined by the `PipeService` interface in the `net.jxta.pipe` package, with a reference implementation defined by the `PipeServiceImpl` class in the `net.jxta.impl.pipe` package.

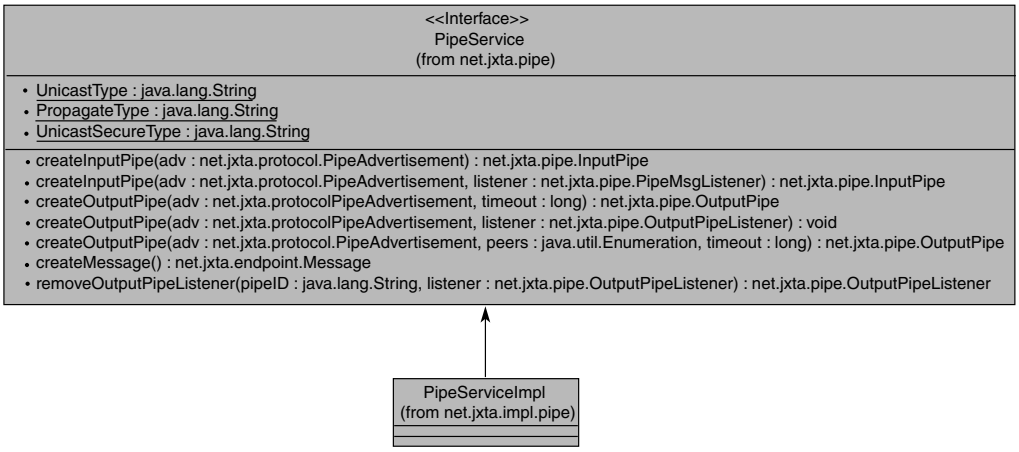

Figure 8.2 The Pipe Service interface and implementation.

One thing that should be noted about the reference implementation of the `PipeService` interface is its reliance on another service to implement the Pipe Binding Protocol. The `PipeResolver` service is a Resolver service handler that provides a convenience mechanism for `PipeServiceImpl`, freeing it to focus on matching resolved endpoints to pipe-implementation objects.

Pipe objects implement either the `InputPipe` or the `OutputPipe` interfaces defined in the `net.jxta.pipe` package. The reference implementation provides an implementation of these interfaces for each of the three types of pipe (unicast, secure unicast, and propagate), as shown in Figure 8.3.

A developer never creates these `InputPipe` or `OutputPipe` implementations directly. Instead, a developer obtains an `InputPipe` or `OutputPipe` instance using `PipeService`'s `createOutputPipe` or `createInputPipe` methods, respectively.

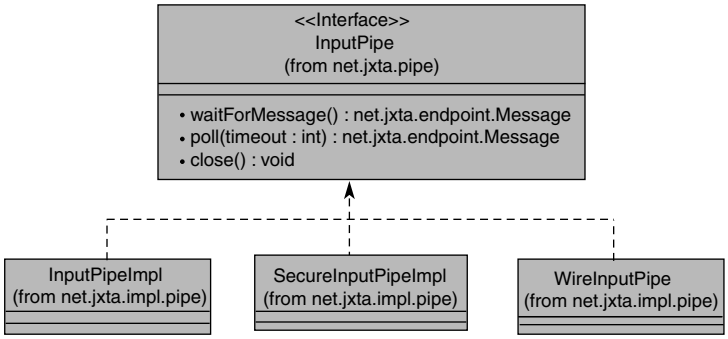

Figure 8.3 The pipe interfaces and classes.

Using the Pipe Service to Send and Receive Messages

The examples in the following sections demonstrate how to use the Pipe service and pipes to send and receive data. The example consists of three parts: an advertisement generator, a client, and a server.

Starting and Stopping the JXTA Platform

Unlike previous examples in this book, the examples in this chapter do not rely on the Shell to start the JXTA platform or provide the user interface. These applications start and stop the JXTA platform themselves by creating a Net Peer Group instance using this call:

```
PeerGroup peerGroup = PeerGroupFactory.newNetPeerGroup();
```

As you've seen in all the examples so far, all operations within JXTA are associated with a peer group. In the examples in all the previous chapters, a `PeerGroup` object obtained from the Shell environment was used to obtain an instance of a core service, such as the Discovery service, for a peer group.

The Net Peer Group is a special peer group, one that is described in greater detail in Chapter 10, "Peer Groups and Services." For the moment, just think of the Net Peer Group as a common peer group that peers belong to when the platform is started.

Unfortunately, after the platform starts, there currently isn't any nice way to shut down the JXTA platform in a controlled way. The only way, as unpleasant as it is, is to use this code:

```
System.exit(0);
```

The `exit` call takes an integer parameter, where `0` indicates no error occurred. To stop the JXTA platform after an error has occurred, the `exit` method should be called with a nonzero value, usually `1`.

Creating a Pipe Advertisement

Creating an input pipe or output pipe using the Pipe service requires a Pipe Advertisement. So, as shown in Listing 8.4, the first step in creating any solution that involves pipes is to create a Pipe Advertisement that describes the type of pipe, the Pipe ID, and an optional name for the pipe.

Listing 8.4 **Source Code for *PipeAdvPopulator.java***

```
package com.newriders.jxta.chapter8;

import java.io.FileWriter;
import java.io.IOException;

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredTextDocument;

import net.jxta.exception.PeerGroupException;

import net.jxta.id.ID;
import net.jxta.id.IDFactory;

import net.jxta.impl.peergroup.StdPeerGroup;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;

import net.jxta.pipe.PipeService;

import net.jxta.protocol.PipeAdvertisement;


/**
 * An example to create a set of common Pipe Advertisement to be used
 * by the PipeServer example application.
 */
public class PipeAdvPopulator
{
    /**
     * The peerGroup for the application.
     */
    private PeerGroup peerGroup = null;
```

continues

Listing 8.4 **Continued**

```
/**
 * Generates a Pipe Advertisement for the PipeClient/Server example.
 */
public void generatePipeAdv()
{
    // Create a new Pipe Advertisement object instance.
    PipeAdvertisement pipeAdv =
        (PipeAdvertisement) AdvertisementFactory.newAdvertisement(
            PipeAdvertisement.getAdvertisementType());

    // Create a unicast Pipe Advertisement.
    pipeAdv.setName("Chapter 8 Example Unicast Pipe Advertisement");
    pipeAdv.setPipeID((ID) IDFactory.newPipeID(
        peerGroup.getPeerGroupID()));
    pipeAdv.setType(PipeService.UnicastType);
    writePipeAdv(pipeAdv, "UnicastPipeAdv.xml");

    // Create a secure unicast Pipe Advertisement.
    pipeAdv.setName(
        "Chapter 8 Example Secure Unicast Pipe Advertisement");
    pipeAdv.setPipeID((ID) IDFactory.newPipeID(
        peerGroup.getPeerGroupID()));
    pipeAdv.setType(PipeService.UnicastSecureType);
    writePipeAdv(pipeAdv, "SecureUnicastPipeAdv.xml");

    // Create a propagate Pipe Advertisement.
    pipeAdv.setName("Chapter 8 Example Propagate Pipe Advertisement");
    pipeAdv.setPipeID((ID) IDFactory.newPipeID(
        peerGroup.getPeerGroupID()));
    pipeAdv.setType(PipeService.PropagateType);
    writePipeAdv(pipeAdv, "PropagatePipeAdv.xml");
}

/**
 * Starts the JXTA platform.
 *
 * @exception   PeerGroupException thrown if the platform can't
 *              be started.
 */
public void initializeJXTA() throws PeerGroupException
{
```

```
        peerGroup = PeerGroupFactory.newNetPeerGroup();
    }

    /**
     * Runs the application: starts the JXTA platform, generates the Pipe
     * Advertisements, and stops the JXTA platform.
     *
     * @param   args the command-line arguments passed to the application.
     */
    public static void main(String[] args)
    {
        PipeAdvPopulator p = new PipeAdvPopulator();

        try
        {
            // Initialize the JXTA platform.
            p.initializeJXTA();

            // Generate the Pipe Advertisements to be used by the examples.
            p.generatePipeAdv();

            // Stop the JXTA platform.
            p.uninitializeJXTA();
        }
        catch (PeerGroupException e)
        {
            System.out.println("Error starting JXTA platform: " + e);
            System.exit(1);
        }
    }

    /**
     * Stops the JXTA platform.
     */
    public void uninitializeJXTA()
    {
        // Currently, there isn't any nice way to do this.
        System.exit(0);
    }

    /**
```

continues

Listing 8.4 **Continued**

```
     * Writes the given Pipe Advertisement to a file
     * with the specified name.
     *
     * @param   pipeAdv the Pipe Advertisement to be written to file.
     * @param   fileName the name of the file to write.
     */
    private void writePipeAdv(PipeAdvertisement pipeAdv, String fileName)
    {
        // Create an XML formatted version of the Pipe Advertisement.
        try
        {
            FileWriter file = new FileWriter(fileName);
            MimeMediaType mimeType = new MimeMediaType("text/xml");
            StructuredTextDocument document =
                (StructuredTextDocument) pipeAdv.getDocument(mimeType);

            // Output the XML for the advertisement to the file.
            document.sendToWriter(file);
            file.close();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

The `PipeAdvPopulator` example creates a Pipe Advertisement for each possible pipe type, to allow you to experiment with all the pipe types in the following pipe examples. `PipeAdvPopulator` creates three files: `UnicastPipeAdv.xml`, `SecureUnicastPipeAdv.xml`, and `PropagatePipeAdv.xml`.

To compile and run `PipeAdvPopulator`, create a new directory and copy into it all the `JAR` files from the `lib` directory under the JXTA Demo install directory. Place `PipeAdvPopulator.java` in the same directory and compile it from the command line using this code:

```
javac -d . -classpath .;beepcore.jar;cms.jar;cryptix32.jar;
cryptix-asn1.jar;instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;
jxtashell.jar;log4j.jar;minimalBC.jar PipeAdvPopulator.java
```

Run the example using this code:

```
java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;
cryptix-asn1.jar;instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;
jxtashell.jar;log4j.jar;minimalBC.jar
com.newriders.jxta.chapter8.PipeAdvPopulator
```

Configure your peer as you did for the earlier Shell examples when prompted by the configuration screens. When you finish the configuration, the JXTA platform starts and `PipeAdvPopulator` creates the Pipe Advertisement files.

Creating an Input Pipe

An input pipe listens for messages being sent by other peers. The example in Listing 8.5 creates an `InputPipe` instance using the Pipe service.

Listing 8.5 **Source Code for *PipeServer.java***

```
package com.newriders.jxta.chapter8;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import java.awt.FlowLayout;
import java.awt.Container;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

import net.jxta.discovery.DiscoveryService;

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;

import net.jxta.endpoint.Message;

import net.jxta.exception.PeerGroupException;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
```

continues

Listing 8.5 **Continued**

```
import net.jxta.pipe.InputPipe;
import net.jxta.pipe.PipeMsgEvent;
import net.jxta.pipe.PipeMsgListener;
import net.jxta.pipe.PipeService;

import net.jxta.protocol.PipeAdvertisement;


/**
 * A server application to accept incoming messages on a pipe and display
 * them to the user.
 */
public class PipeServer implements PipeMsgListener
{
    /**
     * The frame for the user interface.
     */
    private JFrame serverFrame = new JFrame("PipeServer");

    /**
     * A label used to display the received message in the GUI.
     */
    private JLabel messageText =
        new JLabel("Waiting to receive a message...");

    /**
     * The peerGroup for the application.
     */
    private PeerGroup peerGroup = null;

    /**
     * Indicates whether the GUI has been initialized already.
     */
    private boolean initialized = false;

    /**
     * The input pipe used to receive messages.
     */
    private InputPipe inputPipe = null;
```

```
/**
 * Starts the JXTA platform.
 *
 * @exception   PeerGroupException thrown if the platform can't
 *              be started.
 */
public void initializeJXTA() throws PeerGroupException
{
    peerGroup = PeerGroupFactory.newNetPeerGroup();
}

/**
 * Load the Pipe Advertisement generated by PipeAdvPopulator.
 * This method tries to create an output pipe that can be used
 * to send messages.
 *
 * @param       fileName the name of the file from which to load
 *              the Pipe Advertisement.
 * @exception   FileNoteFoundException if the Pipe Advertisement
 *              file can't be found.
 * @exception   IOException if there is an error binding the pipe.
 */
public void loadPipeAdv(String fileName)
    throws FileNotFoundException, IOException
{
    FileInputStream file = new FileInputStream(fileName);
    MimeMediaType asMimeType = new MimeMediaType("text/xml");

    // Load the advertisement.
    PipeAdvertisement pipeAdv =
        (PipeAdvertisement) AdvertisementFactory.newAdvertisement(
            asMimeType, file);

    // Publish the discovery to allow peers to find and bind the pipe.
    DiscoveryService discovery = peerGroup.getDiscoveryService();
    discovery.publish(pipeAdv, DiscoveryService.ADV);
    discovery.remotePublish(pipeAdv, DiscoveryService.ADV);

    // Create an input pipe using the advertisement.
    PipeService pipeService = peerGroup.getPipeService();
    inputPipe = pipeService.createInputPipe(pipeAdv, this);
}
```

continues

Listing 8.5 **Continued**

```
/**
 * Runs the application: starts the JXTA platform, loads the
 * Pipe Advertisement from file, and creates an input pipe to
 * use to receive messages.
 *
 * @param   args the command-line arguments passed to the application.
 */
public static void main(String[] args)
{
    PipeServer server = new PipeServer();

    if (args.length == 1)
    {
        try
        {
            // Initialize the JXTA platform.
            server.initializeJXTA();

            // Load the Pipe Advertisement.
            server.loadPipeAdv(args[0]);

            // Show the user interface.
            server.showGUI();
        }
        catch (PeerGroupException e)
        {
            System.out.println("Error starting JXTA platform: " + e);
            System.exit(1);
        }
        catch (FileNotFoundException e2)
        {
            System.out.println("Unable to load Pipe Advertisement: "
                + e2);
            System.exit(1);
        }
        catch (IOException e3)
        {
            System.out.println("Error loading Pipe Advertisement: "
                + e3);
            System.exit(1);
```

```
        }
    }
    else
    {
        System.out.println(
            "Specify the name of the input Pipe Advertisement file.");
    }
}

/**
 * Handles an incoming message.
 *
 * @param   event the incoming event containing the arriving message.
 */
public void pipeMsgEvent(PipeMsgEvent event)
{
    // Extract the message.
    Message message = event.getMessage();

    // Set the user interface to display the message text.
    messageText.setText(message.getString("MessageText"));
}

/**
 * Configures and displays a simple user interface to display messages
 * received by the pipe. The GUI also allows the user to stop the
 * server application.
 */
public void showGUI()
{
    if (!initialized)
    {
        initialized = true;

        JButton quitButton = new JButton("Quit");

        // Populate the GUI frame.
        Container pane = serverFrame.getContentPane();
        pane.setLayout(new FlowLayout());
        pane.add(messageText);
        pane.add(quitButton);

        quitButton.addActionListener(
```

continues

Listing 8.5 **Continued**

```
                new ActionListener()
                {
                    public void actionPerformed(ActionEvent e)
                    {
                        serverFrame.hide();

                        // Stop the JXTA platform. Currently, there isn't
                        // any nice way to do this.
                        System.exit(0);
                    }
                }
            );

            // Pack and display the user interface.
            serverFrame.pack();
            serverFrame.show();
        }
    }
}
```

The `PipeServer` starts the JXTA platform, loads a Pipe Advertisement specified at the command line, creates an input pipe from the advertisement, and waits for messages to arrive that it can display in its user interface.

The `PipeServer` example creates an input pipe using this code:

```
inputPipe = pipeService.createInputPipe(pipeAdv, this);
```

As shown in Figure 8.4, this version of `createInputPipe` takes `PipeMsgListener` as its second parameter. The `PipeServer` class itself implements the `PipeMsgListener` interface to receive notification when new messages arrive through the newly created `InputPipe`.

This is the only mechanism for an application to register a listener because the `InputPipe` interface doesn't define any methods to register or unregister a listener object. The `PipeServer` example implements the `PipeMsgListener`'s `pipeMsgEvent` method to extract the received `Message` and update the `PipeServer` user interface.

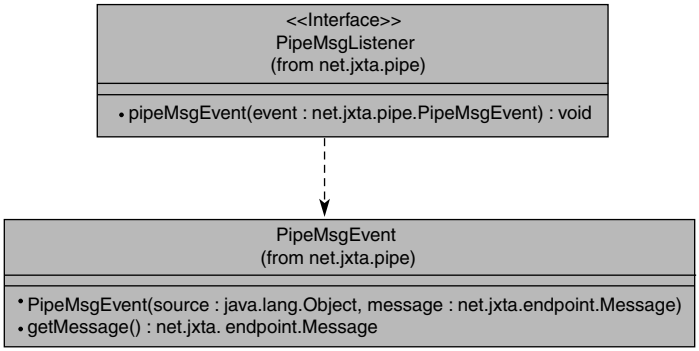

Figure 8.4 The `PipeMsgListener` interface and `PipeMsgEvent` class.

An application that doesn't use a `PipeMsgListener` can still retrieve messages received by `InputPipe` by using either the `poll` or `waitForMessage` methods defined by `InputPipe`:

```
public Message poll(int timeout) throws InterruptedException;

public Message waitForMessage() throws InterruptedException;
```

The `waitForMessage` method blocks indefinitely until a message arrives, at which point, it returns a `Message` object. Usually an application that wants to use this method spawns its own subclass of `Thread` to handle calling `waitForMessage` repeatedly and processing the `Message` objects as they arrive.

The `poll` method is similar to `waitForMessage`, except that a call to the `poll` method blocks only for the length of time specified. The `timeout` argument specifies the amount of time (in milliseconds) to wait for a `Message` to arrive before returning. If no message is received, the `poll` method returns `null`.

By itself, the `PipeServer` example isn't very useful. There's no point waiting for messages to arrive if no one's sending messages! Before you can use `PipeServer`, you need to create a client application that sends messages using the same Pipe Advertisement.

Creating an Output Pipe

An output pipe sends messages to a remote peer. The example in Listing 8.6 creates an output pipe to send simple text messages to a peer running the `PipeServer` example created in the previous section.

Listing 8.6 **Source Code for *PipeClient.java***

```
package com.newriders.jxta.chapter8;

import java.awt.FlowLayout;
import java.awt.Container;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import java.net.URL;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JTextField;

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;

import net.jxta.endpoint.Message;

import net.jxta.exception.PeerGroupException;

import net.jxta.id.IDFactory;

import net.jxta.peer.PeerID;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;

import net.jxta.pipe.OutputPipe;
import net.jxta.pipe.OutputPipeEvent;
import net.jxta.pipe.OutputPipeListener;
import net.jxta.pipe.PipeService;

import net.jxta.protocol.PipeAdvertisement;
```

```
/**
 * A client application, which sends messages over a pipe to a remote peer.
 */
public class PipeClient implements OutputPipeListener
{
    /**
     * The peerGroup for the application.
     */
    private PeerGroup peerGroup = null;

    /**
     * The pipe to use to send the message to the remote peer.
     */
    private OutputPipe outputPipe = null;

    /**
     * The frame for the user interface.
     */
    private JFrame clientFrame = new JFrame("PipeClient");

    /**
     * The text field in the user interface to accept the message
     * text to be sent over the pipe.
     */
    private JTextField messageText = new JTextField(20);

    /**
     * Indicates whether the pipe has been bound already.
     */
    private boolean initialized = false;


    /**
     * Starts the JXTA platform.
     *
     * @exception   PeerGroupException thrown if the platform can't
     *              be started.
     */
    public void initializeJXTA() throws PeerGroupException
    {
        peerGroup = PeerGroupFactory.newNetPeerGroup();
    }
```

continues

Listing 8.6 **Continued**

```
/**
 * Load the Pipe Advertisement generated by PipeAdvPopulator. This method
 * tries to create an output pipe that can be used to send messages.
 *
 * @param       fileName the name of the file from which to load
 *              the Pipe Advertisement.
 * @exception   FileNoteFoundException if the Pipe Advertisement file
 *              can't be found.
 * @exception   IOException if there is an error binding the pipe.
 */
public void loadPipeAdv(String fileName)
    throws FileNotFoundException, IOException
{
    FileInputStream file = new FileInputStream(fileName);
    MimeMediaType asMimeType = new MimeMediaType("text/xml");

    // Load the advertisement.
    PipeAdvertisement pipeAdv =
        (PipeAdvertisement) AdvertisementFactory.newAdvertisement(
             asMimeType, file);

    // Create an output pipe using the advertisement. This version of
    // createOutputPipe uses the PipeClient class as an
    // OutputPipeListener object.
    PipeService pipeService = peerGroup.getPipeService();
    pipeService.createOutputPipe(pipeAdv, this);
}

/**
 * Runs the application: starts the JXTA platform, loads the Pipe
 * Advertisement from file, and attempts to resolve the pipe.
 *
 * @param   args the command-line arguments passed to the application.
 */
public static void main(String[] args)
{
    PipeClient client = new PipeClient();

    if (args.length == 1)
    {
```

```
        try
        {
            // Initialize the JXTA platform.
            client.initializeJXTA();

            // Load the Pipe Advertisement.
            client.loadPipeAdv(args[0]);
        }
        catch (PeerGroupException e)
        {
            System.out.println("Error starting JXTA platform: " + e);
            System.exit(1);
        }
        catch (FileNotFoundException e2)
        {
            System.out.println("Unable to load Pipe Advertisement: "
                + e2);
            System.exit(1);
        }
        catch (IOException e3)
        {
            System.out.println("Error loading or binding Pipe"
                + " Advertisement: " + e3);
            System.exit(1);
        }
    }
    else
    {
        System.out.println("You must specify the name of the input"
            + " Pipe Advertisement file.");
    }
}

/**
 * The OutputPipeListener event that is triggered when an OutputPipe is
 * resolved by the call to PipeService.createOutputPipe.
 *
 * @param   event the event to use to extract the resolved output pipe.
 */
public void outputPipeEvent(OutputPipeEvent event)
{
    // We care about only the first pipe we manage to resolve.
```

continues

Listing 8.6 **Continued**

```
        if (!initialized)
        {
            initialized = true;

            // Get the bound pipe.
            outputPipe = event.getOutputPipe();

            // Show a small GUI to allow the user to send a message.
            showGUI();
        }
    }

    /**
     * Sends a message string to the remote peer using the output pipe.
     *
     * @param   messageString the message text to send to the remote peer.
     */
    private void sendMessage(String messageString)
    {
        PipeService pipeService = peerGroup.getPipeService();
        Message message = pipeService.createMessage();

        // Configure the message object.
        message.setString("MessageText", messageString);

        if (null != outputPipe)
        {
            try
            {
                // Send the message.
                outputPipe.send(message);
            }
            catch (IOException e)
            {
                // Show some warning dialog.
                JOptionPane.showMessageDialog(null, e.toString(), "Error",
                    JOptionPane.WARNING_MESSAGE);
            }
        }
        else
        {
```

```
            // Show some warning dialog.
            JOptionPane.showMessageDialog(null, "Output pipe is null!",
                "Error", JOptionPane.WARNING_MESSAGE);
        }
    }

    /**
     * Configures and displays a simple user interface to allow the user to
     * send text messages. The GUI also allows the user to stop the client
     * application.
     */
    private void showGUI()
    {
        JButton sendButton = new JButton("Send Message");
        JButton quitButton = new JButton("Quit");

        // Populate the GUI frame.
        Container pane = clientFrame.getContentPane();
        pane.setLayout(new FlowLayout());
        pane.add(messageText);
        pane.add(sendButton);
        pane.add(quitButton);

        // Set up listeners for the buttons.
        sendButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e)
                {
                    // Send the message.
                    sendMessage(messageText.getText());

                    // Clear the text.
                    messageText.setText("");
                }
            }
        );
        quitButton.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    clientFrame.hide();
```

continues

Listing 8.6 **Continued**

```
                    // Stop the JXTA platform. Currently, there isn't any
                    // nice way to do this.
                    System.exit(0);
                }
            }
        );

        // Pack and display the user interface.
        clientFrame.pack();
        clientFrame.show();
    }
}
```

The `PipeClient` example mirrors the `PipeServer` example. The `PipeClient` example starts the JXTA platform, loads a Pipe Advertisement specified at the command line, and creates an output pipe from the advertisement. After an output pipe is successfully created, the example displays a user interface that the user can use to input messages to be sent via the output pipe to a remote peer.

The `PipeClient` example creates an output pipe using this code:

```
pipeService.createOutputPipe(pipeAdv, this);
```

This version of `createOutputPipe` takes `OutputPipeListener`, shown in Figure 8.5, as its second parameter. The `PipeClient` itself implements the `OutputPipeListener` interface to receive notification when an output pipe has been successfully created.

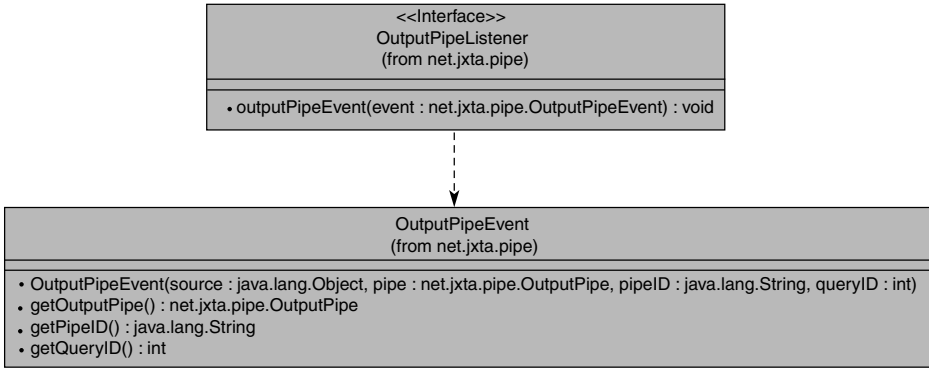

Figure 8.5 The `OutputPipeListener` interface and `OutputPipeEvent` class.

Unlike the PipeServer example, the PipeClient example doesn't display its user interface immediately. Instead, PipeClient's implementation of OutputPipeListener's outputPipeEvent method displays the user interface when a pipe has been bound to an endpoint successfully. Because an output pipe may be bound successfully to several endpoints, outputPipeEvent does this only the first time it is called. Text entered into the user interface is wrapped as a Message and sent over the resolved OutputPipe using OutputPipe's send method.

Using *PipeServer* and *PipeClient*

PipeServer and PipeClient each form one end of a complete communication connection. The PipeServer class listens for data on an input pipe, and the PipeClient class allows a user to send data using an output pipe. To prepare to run these examples, follow these steps:

1. Place the source code in the same directory that you created for the PipeAdvPopulator example.
2. Compile the source code by using the same command as before (replacing PipeAdvPopulator.java with the appropriate source filename, of course).
3. Create a copy of the entire directory. This is required so that you can run two independent instances of the PipeServer and PipeClient applications.

Next, start the PipeServer example in the original directory using this code:

```
java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;
instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;
log4j.jar;minimalBC.jar com.newriders.jxta.chapter8.PipeServer
UnicastPipeAdv.xml
```

Here, the UnicastPipeAdv.xml parameter specifies that PipeServer should use the UnicastPipeAdv.xml Pipe Advertisement file to create the input pipe. After the input pipe is created, the PipeServer example displays the user interface in Figure 8.6.

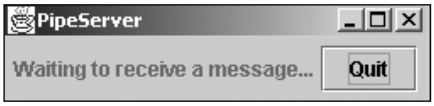

Figure 8.6 The PipeServer user interface.

Finally, start `PipeClient` in the copy of the original directory. For this to work, you need to force the JXTA platform to show the configuration interface by deleting the PlatformConfig file. Start `PipeClient` using this code:

```
java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;
instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;
log4j.jar;minimalBC.jar com.newriders.jxta.chapter8.PipeClient
UnicastPipeAdv.xml
```

When the configuration screen appears, choose a different TCP and HTTP port in the TCP and HTTP Settings sections of the Advanced tab. After you enter the configuration and started the platform, `PipeClient` attempts to bind an output pipe. When the output pipe has been successfully bound, `PipeClient` displays the user interface in Figure 8.7.

Figure 8.7 The `PipeClient` user interface.

You should now be able to enter message in `PipeClient`'s user interface and send it by clicking Send Message. The message is sent using the output pipe, and the `PipeServer` user interface displays the message.

Note that although this demonstration might make it appear as though the communication between the client and the server is reliable, JXTA does not guarantee message delivery. Even if the pipe is using an endpoint protocol built on top of a reliable network transport, such as TCP, a message is not guaranteed to be delivered. A message might be dropped en route by an overloaded intermediary peer or even by the destination peer itself. That said, reliable message delivery could be built on top of pipes fairly easily and will most likely be included in JXTA in the future.

Using Secure Pipes

A JXTA application can easily switch to using secure pipes just by changing the Pipe Advertisement used when creating the input and output pipes. To try using `PipeServer` and `PipeClient` with secure pipes, start the application the same way as in the previous section, but replace `UnicastPipeAdv.xml` in each command with `SecureUnicastPipeAdv.xml`.

Secure pipes use the Transport Security Layer protocol, a variant of SSL 3.0, to secure the communication channel. When you configure the platform for the first time, the platform generates a root certificate and private key that are used to secure communications. The root certificate is saved in the Personal Security Environment directory (`pse`) under the current directory when the platform executes, and the private key is protected using the password entered in the Security tab of the Configurator. The root certificate is also published within the Peer Advertisement.

Using secure pipes with `PipeServer` and `PipeClient` should not seem any different than using the nonsecure unicast pipes in the previous example.

Using Propagation Pipes

Propagation pipes are different than the other two types of pipes examined so far in this chapter. Propagation pipes provide a peer with a convenient mechanism to broadcast data to multiple peer endpoints. This might be useful in some applications, such as a chat application, in which one peer produces data for consumption by multiple remote peers.

In theory, you can use a propagation pipe by invoking `PipeClient` and `PipeServer` using the `PropagatePipeAdv.xml` Pipe Advertisement instead of the `UnicastPipeAdv.xml`. However, the current reference implementation of `PipeService` does not allow you to call `createOutputPipe` and provide an `OutputPipeListener`. This should be fixed shortly, but in case it isn't, Listing 8.7 shows a modified version of `PipeClient` that fixes the problem.

Listing 8.7 **Source Code for *PropagatePipeClient.java***

```
package com.newriders.jxta.chapter8;

import java.awt.FlowLayout;
import java.awt.Container;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import java.net.URL;

import javax.swing.JButton;
```

continues

Listing 8.7 **Continued**

```
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JTextField;

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;

import net.jxta.endpoint.Message;

import net.jxta.exception.PeerGroupException;

import net.jxta.id.IDFactory;

import net.jxta.peer.PeerID;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;

import net.jxta.pipe.OutputPipe;
import net.jxta.pipe.PipeService;

import net.jxta.protocol.PipeAdvertisement;


/**
 * A client application, which sends messages over a pipe to a remote peer.
 * This version is slightly different, to allow for use of a propagation
 * pipe.
 */
public class PropagatePipeClient
{
    /**
     * The peerGroup for the application.
     */
    private PeerGroup peerGroup = null;

    /**
     * The pipe to use to send the message to the remote peer.
     */
    private OutputPipe outputPipe = null;
```

```
/**
 * The frame for the user interface.
 */
private JFrame clientFrame = new JFrame("PropagatePipeClient");

/**
 * The text field in the user interface to accept the message
 * text to be sent over the pipe.
 */
private JTextField messageText = new JTextField(20);


/**
 * Starts the JXTA platform.
 *
 * @exception   PeerGroupException thrown if the platform can't
 * be started.
 */
public void initializeJXTA() throws PeerGroupException
{
    peerGroup = PeerGroupFactory.newNetPeerGroup();
}

/**
 * Load the Pipe Advertisement generated by PipeAdvPopulator. This
 * method tries to create an output pipe that can be used to send messages.
 *
 * @param       fileName the name of the file from which to load the
 *              Pipe Advertisement.
 * @exception   FileNoteFoundException if the Pipe Advertisement
 *              file can't be found.
 * @exception   IOException if there is an error binding the pipe.
 */
public void loadPipeAdv(String fileName)
    throws FileNotFoundException, IOException
{
    FileInputStream file = new FileInputStream(fileName);
    MimeMediaType asMimeType = new MimeMediaType("text/xml");

    // Load the advertisement.
    PipeAdvertisement pipeAdv =
```

continues

Listing 8.7 **Continued**

```
            (PipeAdvertisement) AdvertisementFactory.newAdvertisement(
                asMimeType, file);

        // Create an output pipe using the advertisement. This version of
        // createOutputPipe uses the PipeClient class as an
        // OutputPipeListener object.
        PipeService pipeService = peerGroup.getPipeService();
        outputPipe = pipeService.createOutputPipe(pipeAdv, 10000);

        // Because we can't use an OutputPipeListener when attempting to
        // create an output propagation pipe, the GUI must be displayed
        // immediately.
showGUI();
    }

    /**
     * Runs the application: starts the JXTA platform, loads the Pipe
     * Advertisement from file, and attempts to resolve the pipe.
     *
     * @param   args the command-line arguments passed to the application.
     */
    public static void main(String[] args)
    {
        PropagatePipeClient client = new PropagatePipeClient();

        if (args.length == 1)
        {
            try
            {
                // Initialize the JXTA platform.
                client.initializeJXTA();

                // Load the Pipe Advertisement.
                client.loadPipeAdv(args[0]);
            }
            catch (PeerGroupException e)
            {
                System.out.println("Error starting JXTA platform: " + e);
                System.exit(1);
            }
            catch (FileNotFoundException e2)
```

```
        {
            System.out.println("Unable to load Pipe Advertisement: "
                + e2);
            System.exit(1);
        }
        catch (IOException e3)
        {
            System.out.println("Error loading or binding Pipe"
                + " Advertisement: " + e3);
            System.exit(1);
        }
    }
    else
    {
        System.out.println("You must specify the name of the input"
            + " Pipe Advertisement file.");
    }
}

/**
 * Sends a message string to the remote peer using the output pipe.
 *
 * @param   messageString the message text to send to the remote peer.
 */
private void sendMessage(String messageString)
{
    PipeService pipeService = peerGroup.getPipeService();
    Message message = pipeService.createMessage();

    // Configure the message object.
    message.setString("MessageText", messageString);

    if (null != outputPipe)
    {
        try
        {
            // Send the message.
            outputPipe.send(message);
        }
        catch (IOException e)
        {
            // Show some warning dialog.
```

continues

Listing 8.7 **Continued**

```
                JOptionPane.showMessageDialog(null, e.toString(), "Error",
                    JOptionPane.WARNING_MESSAGE);
            }
        }
        else
        {
            // Show some warning dialog.
            JOptionPane.showMessageDialog(null, "Output pipe is null!",
                "Error", JOptionPane.WARNING_MESSAGE);
        }
    }

    /**
     * Configures and displays a simple user interface to allow the user to
     * send text messages. The GUI also allows the user to stop the client
     * application.
     */
    private void showGUI()
    {
        JButton sendButton = new JButton("Send Message");
        JButton quitButton = new JButton("Quit");

        // Populate the GUI frame.
        Container pane = clientFrame.getContentPane();
        pane.setLayout(new FlowLayout());
        pane.add(messageText);
        pane.add(sendButton);
        pane.add(quitButton);

        // Set up listeners for the buttons.
        sendButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e)
                {
                    // Send the message.
                    sendMessage(messageText.getText());

                    // Clear the text.
                    messageText.setText("");
                }
            }
```

```
            );
            quitButton.addActionListener(
                new ActionListener()
                {
                    public void actionPerformed(ActionEvent e)
                    {
                        clientFrame.hide();

                        // Stop the JXTA platform. Currently, there isn't any
                        // nice way to do this.
                        System.exit(0);
                    }
                }
            );

            // Pack and display the user interface.
            clientFrame.pack();
            clientFrame.show();
        }
    }
```

Instead of calling `createOutputPipe` in `loadPipeAdv` with an `OutputPipeListener` object, this version calls `createOutputPipe` with a timeout value. The user interface is shown in `loadPipeAdv` after the output pipe is bound rather than being shown by the `outputPipeEvent` method.

To see the propagate pipe in action, create another copy of the directory holding your source code and the JXTA JARs, and delete the `PlatformConfig` file. This time, run two `PipeServer` instances from different directories, and run a `PropagatePipeClient` instance from third directory. Remember to configure the platform running in the newest directory (the one that you copied at the beginning of this paragraph) to use another TCP and HTTP port as before. When running these applications, also be sure to use the `PropagatePipeAdv.xml` file as the source of the Pipe Advertisement.

When the `PipeServer` and `PropagatePipeClient` instances are running, you should be able to send a message using `PropagatePipeClient`. When the message is sent, it should be displayed by both `PipeServer` instances. By comparison, performing the same exercise using the `UnicastPipeAdv.xml` Pipe Advertisement would result in only one of the `PipeServer` instances receiving the message. In this case, only the first pipe instance resolved by the Pipe service would receive the message.

Bidirectional Pipes

The examples given so far in this chapter have demonstrated only unidirectional communication. To achieve bidirectional communication, you need two pipes: one to send data and one to receive data.

You can easily implement a bidirectional solution, but doing so requires you to write the code to bind both the input and output pipes. Instead of writing the code, you can use the `BidirectionalPipeService` class, shown in Figure 8.8, from the `net.jxta.impl.util` package to handle the common tasks of initializing pipes for two-way communications.

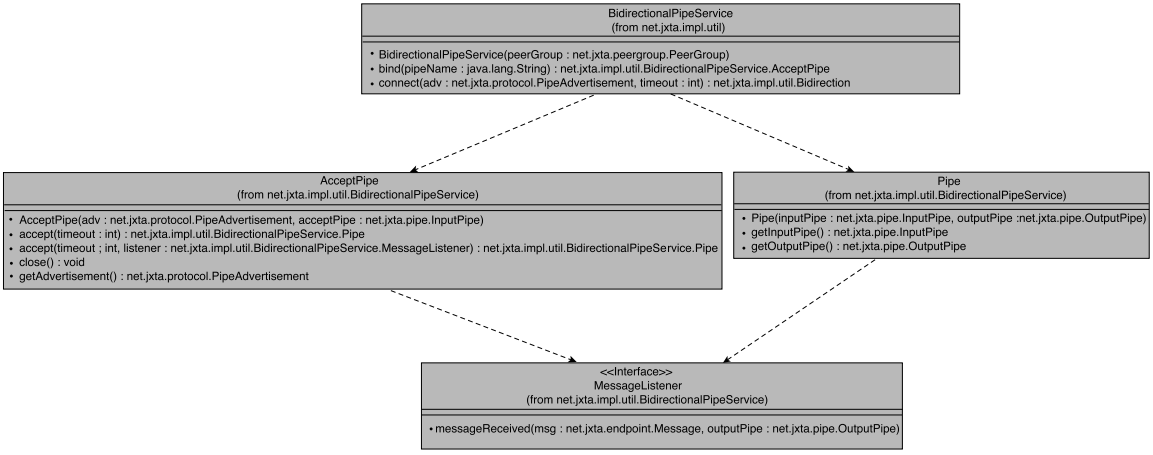

Figure 8.8 The `BidirectionalPipeService` class and supporting classes.

The `BidirectionalPipeService` class provided by the reference implementation isn't a real service. Unlike `PipeService` or any of the other core services, `BidirectionalPipeService` is not constantly running on a peer waiting to handle incoming messages. Instead, `BidirectionalPipeService` is simply a wrapper built on top of the Pipe and Discovery services. `BidirectionalPipeService`'s constructor takes a `PeerGroup` object as its sole argument, which it uses to extract the peer group's Discovery and Pipe service objects:

```
public BidirectionalPipeService (PeerGroup peerGroup);
```

As shown in Figure 8.9, `BidirectionalPipeService` provides only two other methods: `bind` and `connect`. The `bind` method is used to create an instance of `AcceptPipe`, an inner class defined by `BidirectionalPipeService`, which uses an input pipe to listen for connections from other peers. The `connect` method is used to connect to a remote peer that is already listening for connections.

`BidirectionalPipeService` and its support classes use a clever trick to require you to work directly with only one Pipe Advertisement.

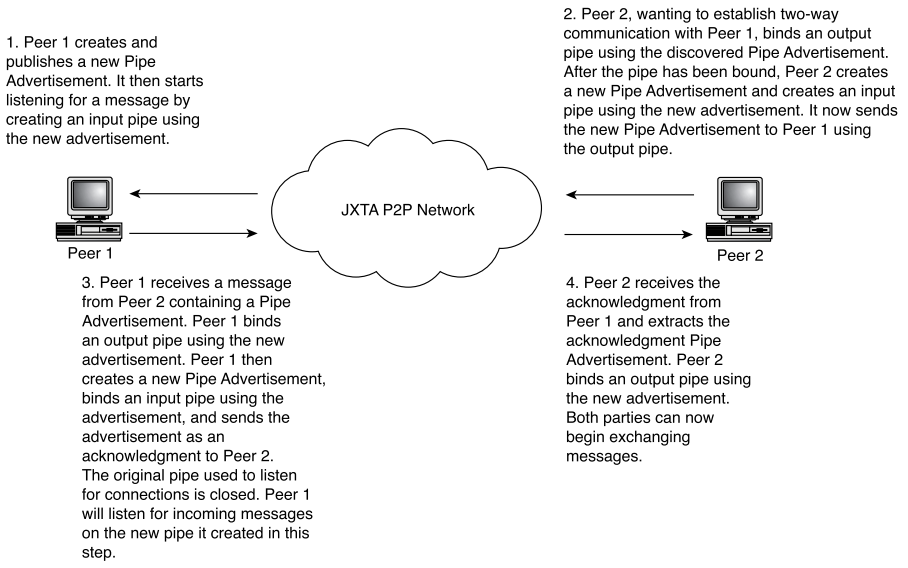

Figure 8.9 Flow of `BidirectionalPipeService` messages.

When the `connect` method is called, the Pipe Advertisement passed to the method binds an output pipe. If that output pipe is bound successfully, the `connect` method creates and binds a new input pipe. The `connect` method sends this new pipe's advertisement to the remote peer using the newly bound output pipe. On the remote peer, the `AcceptPipe` object listening for new connections receives the Pipe Advertisement and uses it to bind an output pipe. The remote peer can now use this output pipe to send messages back to the originating peer. The remote peer creates one more Pipe Advertisement and binds an input pipe using this advertisement. This advertisement is sent as an acknowledgement, which means that the original pipe used to negotiate the two-way communications channel is no longer used. The peer receiving the acknowledgement advertisement uses it rather than the original pipe to send messages to the remote peer. Voilá—two-way communication. Using `BidirectionalPipeService`, you can combine the earlier examples in this chapter to create the simple chat client in Listing 8.8.

Listing 8.8 **Source Code for *PipeClientServer.java***

```
package com.newriders.jxta.chapter8;

import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.Container;
```

continues

Listing 8.8 **Continued**

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.io.FileWriter;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import java.net.URL;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredTextDocument;

import net.jxta.endpoint.Message;

import net.jxta.exception.PeerGroupException;

import net.jxta.impl.util.BidirectionalPipeService;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;

import net.jxta.pipe.InputPipe;
import net.jxta.pipe.OutputPipe;
import net.jxta.pipe.PipeService;

import net.jxta.protocol.PipeAdvertisement;


/**
 * A simple text messaging application, which uses a bidirectional
 * pipe to send and receive messages.
 */
```

```
public class PipeClientServer
    implements BidirectionalPipeService.MessageListener
{
    /**
     * The peerGroup for the application.
     */
    private PeerGroup peerGroup = null;

    /**
     * The frame for the user interface.
     */
    private JFrame clientFrame = new JFrame("PipeClientServer");

    /**
     * The text field in the user interface to accept the message
     * text to be sent over the pipe.
     */
    private JTextField messageText = new JTextField(20);

    /**
     * A label used to display the received message in the GUI.
     */
    private JLabel receivedText = new JLabel(
        "Waiting to receive a message...");

    /**
     * Indicates whether the pipe has been bound already.
     */
    private boolean initialized = false;

    /**
     * The bidirectional pipe object to use to send and receive messages.
     */
    private BidirectionalPipeService.Pipe pipe = null;


    /**
     * Creates an input pipe and its advertisement using the
     * BidirectionalPipeService. This is used when starting this class up
     * in "server" mode. The advertisement is saved to file so that another
     * instance of this class can use the advertisement to start up in
     * "client" mode.
```

continues

Listing 8.8 **Continued**

```
 */
public void createPipeAdv() throws IOException
{
    BidirectionalPipeService pipeService = new
        BidirectionalPipeService(peerGroup);

    // Create an accept pipe to use to create an input pipe and
    // listen for connections. "PipeClientServer" is simply the
    // symbolic name that will appear in the Pipe Advertisement
    // created by the BidirectionalPipeService.
    BidirectionalPipeService.AcceptPipe acceptPipe =
        pipeService.bind("PipeClientServer");

    // Extract the Pipe Advertisement and write it to file.
    PipeAdvertisement pipeAdv = acceptPipe.getAdvertisement();
    try
    {
        FileWriter file = new FileWriter("PipeClientServer.xml");
        MimeMediaType mimeType = new MimeMediaType("text/xml");
        StructuredTextDocument document =
            (StructuredTextDocument) pipeAdv.getDocument(mimeType);

        // Output the XML for the advertisement to the file.
        document.sendToWriter(file);
        file.close();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }

    // "Accept" a connection, meaning set up the input pipe and listen
    // for messages. Set this object as the MessageListener so that
    // we can handle incoming messages without having to spawn a
    // thread to call waitForMessage on the input pipe.
    while (null == pipe)
    {
        try
        {
            pipe = acceptPipe.accept(30000, this);
        }
```

```
        catch (InterruptedException e)
        {
            System.out.println("Error trying to accept(): " + e);
        }
    }

    // Show the user interface.
    showGUI();
}

/**
 * Starts the JXTA platform.
 *
 * @exception   PeerGroupException thrown if the platform can't
 *               be started.
 */
public void initializeJXTA() throws PeerGroupException
{
    peerGroup = PeerGroupFactory.newNetPeerGroup();
}

/**
 * Starts the class in "client" mode, loading a Pipe Advertisement from
 * the given file. This advertisement is used to create an output pipe
 * to talk to the remote peer and set up the bidirectional
 * communications channel.
 *
 * @param       fileName the name of the file from which to load the
 *              Pipe Advertisement.
 * @exception   FileNoteFoundException if the Pipe Advertisement
 *              file can't be found.
 * @exception   IOException if there is an error binding the pipe.
 */
public void loadPipeAdv(String fileName)
    throws FileNotFoundException, IOException
{
    FileInputStream file = new FileInputStream(fileName);
    MimeMediaType asMimeType = new MimeMediaType("text/xml");

    // Load the advertisement.
    PipeAdvertisement pipeAdv =
        (PipeAdvertisement) AdvertisementFactory.newAdvertisement(
```

continues

Listing 8.8 **Continued**

```
            asMimeType, file);

        // Connect using the Pipe Advertisement and the
        // BidirectionalPipeService.
        BidirectionalPipeService pipeService =
            new BidirectionalPipeService(peerGroup);

        while (null == pipe)
        {
            try
            {
                System.out.println("Trying...");
                pipe = pipeService.connect(pipeAdv, 30000);
                System.out.println("Done Trying...");
            }
            catch (IOException e)
            {
                // Do nothing.
            }
        }

        // Show the user interface.
        showGUI();

        // There is no way to register a listener with the input pipe used
        // by the Pipe object to receive message. So, use the
        // waitForMessage method instead. Not the nicest way to do this,
        // but it gives you the idea.
        InputPipe input = pipe.getInputPipe();

        while (true)
        {
            try
            {
                Message message = input.waitForMessage();

                // Set the user interface to display the message text.
                receivedText.setText(message.getString("MessageText"));
            }
            catch (InterruptedException e)
```

```
        {
            // Do nothing, ignore the interruption.
        }
    }
}

/**
 * Runs the application. The application can run in either "server" or
 * "client" mode. In "server" mode, the application creates a new Pipe
 * Advertisement, writes it to a file, and binds an input pipe to start
 * listening for incoming messages. In "client" mode, a Pipe
 * Advertisement is read from a file and used to bind an output pipe to
 * a remote peer.
 *
 * @param   args the command-line arguments passed to the application.
 */
public static void main(String[] args)
{
    PipeClientServer client = new PipeClientServer();

    if (args.length == 0)
    {
        // No arguments, therefore we must be trying to
        // set up a new server. Create a input pipe and
        // write its advertisement to a file.
        try
        {
            // Initialize the JXTA platform.
            client.initializeJXTA();

            // Create the input connection and save the
            // Pipe Advertisement.
            client.createPipeAdv();
        }
        catch (PeerGroupException e)
        {
            System.out.println("Error starting JXTA platform: " + e);
            System.exit(1);
        }
        catch (FileNotFoundException e2)
        {
```

continues

Listing 8.8 **Continued**

```
                System.out.println("Unable to load Pipe Advertisement: "
                    + e2);
                System.exit(1);
            }
            catch (IOException e3)
            {
                System.out.println("Error loading or binding Pipe"
                    + " Advertisement: " + e3);
                System.exit(1);
            }
        }
        else if (args.length == 1)
        {
            // If there's one argument, then we need to try to
            // connect to an existing server using the Pipe Advertisement
            // in the file specified by the argument.
            try
            {
                // Initialize the JXTA platform.
                client.initializeJXTA();

                // Load the Pipe Advertisement.
                client.loadPipeAdv(args[0]);
            }
            catch (PeerGroupException e)
            {
                System.out.println("Error starting JXTA platform: " + e);
                System.exit(1);
            }
            catch (FileNotFoundException e2)
            {
                System.out.println("Unable to load Pipe Advertisement: "
                    + e2);
                System.exit(1);
            }
            catch (IOException e3)
            {
                System.out.println("Error loading or binding Pipe"
                    + " Advertisement: " + e3);
                System.exit(1);
            }
```

```
    }
    else
    {
        System.out.println("Usage:");
        System.out.println("'server' mode: PipeClientServer");
        System.out.println("'client' mode: PipeClientServer "
            + "<filename>");

    }
}

/**
 * Handles displaying an incoming message to the user interface.
 *
 * @param   message the message received by the input pipe.
 * @param   pipe an OutputPipe to use to send a response.
 */
public void messageReceived(Message message, OutputPipe pipe)
{
    // Set the user interface to display the message text.
    receivedText.setText(message.getString("MessageText"));
}

/**
 * Sends a message string to the remote peer using the output pipe.
 *
 * @param   messageString the message text to send to the remote peer.
 */
private void sendMessage(String messageString)
{
    PipeService pipeService = peerGroup.getPipeService();
    OutputPipe outputPipe = null;

    // Create and configure a message object.
    Message message = pipeService.createMessage();
    message.setString("MessageText", messageString);

    // Get the output pipe from the pipe.
    outputPipe = pipe.getOutputPipe();

    if (null != outputPipe)
    {
```

continues

Listing 8.8 **Continued**

```
            try
            {
                // Send the message.
                outputPipe.send(message);
            }
            catch (IOException e)
            {
                // Show some warning dialog.
                JOptionPane.showMessageDialog(null, e.toString(), "Error",
                   JOptionPane.WARNING_MESSAGE);
            }
        }
        else
        {
            // Show some warning dialog.
            JOptionPane.showMessageDialog(null, "Output pipe is null!",
                "Error", JOptionPane.WARNING_MESSAGE);
        }
    }

    /**
     * Configures and displays a simple user interface to allow the user to
     * send text messages. The GUI also allows the user to stop the client
     * application.
     */
    private void showGUI()
    {
        JButton sendButton = new JButton("Send Message");
        JButton quitButton = new JButton("Quit");

        JPanel receivePane = new JPanel();
        receivePane.setLayout(new FlowLayout());
        receivePane.add(receivedText);

        JPanel sendPane = new JPanel();
        sendPane.setLayout(new FlowLayout());
        sendPane.add(messageText);
        sendPane.add(sendButton);
        sendPane.add(quitButton);
```

```
        // Populate the GUI frame.
        Container pane = clientFrame.getContentPane();
        pane.setLayout(new BorderLayout());
        pane.add(receivePane, BorderLayout.NORTH);
        pane.add(sendPane, BorderLayout.SOUTH);

        // Set up listeners for the buttons.
        sendButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e)
                {
                    // Send the message.
                    sendMessage(messageText.getText());

                    // Clear the text.
                    messageText.setText("");
                }
            }
        );
        quitButton.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    clientFrame.hide();

                    // Stop the JXTA platform. Currently, there isn't any
                    // nice way to do this.
                    System.exit(0);
                }
            }
        );

        // Pack and display the user interface.
        clientFrame.pack();
        clientFrame.show();
    }
}
```

The PipeClientServer example has two modes of operation:

- **Server mode**—This mode is used when you want to start a new bidirectional pipe. It causes a new Pipe Advertisement to be written to the file PipeClientServer.xml. This advertisement is used by any peer that wants to send messages to the peer and initiate a bidirectional connection.
- **Client mode**—This mode is used when you want to connect to an existing bidirectional pipe. To connect, you need to provide a Pipe Advertisement as part of the command-line arguments. In this example, the advertisement that must be provided is the PipeClientServer.xml file written by another instance of PipeClientServer, running in server mode.

To see this example in operation, you need to use two separate instances of PipeClientServer. This requires two separate directories containing the compiled source code and JXTA JARs. As in previous examples, you need to configure the JXTA platform for each directory to use different TCP and HTTP ports.

After you create and configure the two directories, run one instance of PipeClientServer in server mode from one directory using this code:

```
java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;
instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;
log4j.jar;minimalBC.jar com.newriders.jxta.chapter8.PipeClientServer
```

After the bidirectional pipe has successfully created and published a Pipe Advertisement, you should see a message on the console similar to this:

```
Published bidir pipe urn:jxta:uuid-59616261646162614E50472050325033D7F4C6E7
B2BD4572B6628F1DFEE6B34404
```

This indicates that the bidirectional pipe has created an input pipe and published the pipe's advertisement. The PipeClientServer extracts this advertisement and writes it to the file PipeClientServer.xml.

Now that an input pipe has been started, you need to start a second instance of PipeClientServer, this time in client mode. To do this, you need to provide a Pipe Advertisement. Copy the PipeClientServer.xml file from the first PipeClientServer instance's directory to the directory where the second instance of PipeClientServer will be started. Start the second instance of PipeClientServer from this directory using this code:

```
java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;
instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;
log4j.jar;minimalBC.jar com.newriders.jxta.chapter8.PipeClientServer
PipeClientServer.xml
```

The second instance loads the Pipe Advertisement from `PipeClientServer.xml` and attempts to bind an output pipe. After this has been done, both instances of `PipeClientServer` should display their user interfaces, as shown in Figure 8.10.

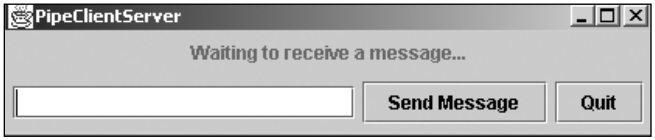

Figure 8.10 The `PipeClientServer` user interface.

You should now be able to send messages between the two `PipeClientServer` instances. Note that if you stop both instances and start them again, you need to recopy the `PipeClientServer.xml` file created by the server mode instance. The bidirectional pipe creates a new Pipe Advertisement each time.

Summary

In this chapter, you learned how a pipe can be bound to an endpoint to send data to or receive data from a remote peer. To demonstrate the use of the Pipe service, this chapter used a set of Pipe Advertisement files generated by the `PipeAdvPopulator` class. Although these files simplified the examples, it should be realized that in real applications, Pipe Advertisements usually are obtained using the Discovery service.

This chapter also examined the `BidirectionalPipeService`, a pseudo-service built on top of the Pipe service. The `BidirectionalPipeService` provides a simple mechanism for peers to establish two-way communications using two pipes. The advantage of this mechanism is that only one Pipe Advertisement must be published or discovered because the `BidirectionalPipeService` handles negotiation of a second Pipe Advertisement. This mechanism also has the advantage that it eliminates some of the code required to manage two pipes.

Pipe Advertisements aren't usually published by themselves, but they are usually contained within another advertisement. As you'll see in Chapter 10, "Peer Groups and Services," a Pipe Advertisement is usually associated with a service, allowing a remote peer to interact with a service through the pipe.