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!

9

The Endpoint Routing Protocol

Due to the ad hoc nature of a P2P network, a message between two endpoints might need to travel through intermediaries. An intermediary might be used to allowing peers with incompatible network transports to communicate by using the intermediary as a gateway. To determine how a message should be sent between two endpoints, a mechanism is required to allow a peer to discover route information. The Endpoint Routing Protocol (ERP) provides peers with a mechanism for determining a route to an endpoint, allowing the peer to send data to the remote endpoint.

Before learning about the Endpoint Routing Protocol, it is necessary to understand how endpoints work. In Chapter 8, "The Pipe Binding Protocol," you learned that although pipes provide a transmission mechanism, the pipes themselves are not responsible for the actual transmission and reception of data. Pipes are an abstraction built on top of endpoints to provide a convenient programming model. Endpoints are the entity responsible for conducting the actual exchange of information over a network. Endpoints encapsulate a set of network interfaces, allowing a peer to send and receive data independently of the type of network transport being employed.

Although JXTA provides the Resolver and Pipe services to enable high-level use of endpoints, some services might want to use endpoints directly. This chapter first explores the use of endpoints for conducting network communication and then details the Endpoint Routing Protocol and its relationship to endpoints.

Introduction to Endpoints

JXTA offers two simple ways to send and receive messages: the Resolver service and the Pipe service. However, as revealed in Chapter 5, "The Peer Resolver Protocol," and Chapter 8, these services are simply convenient wrappers for sending and receiving messages using a peer's local endpoints. An *endpoint* is an interface to a set of network transports that allows data to be sent across the network. In JXTA, network transports are assumed to be unreliable, even though actual endpoint protocol implementations might use reliable transports such as TCP/IP.

Unlike other areas of the JXTA platform, endpoint functionality doesn't have a protocol definition. Details on how data is to be formatted for transport across the network is the responsibility of a particular endpoint protocol implementation. The only functionality exposed to the developer is provided by the Endpoint service, which aggregates the registered endpoint protocol implementations for use by a developer. Although a developer could use the Endpoint service implementation to obtain a particular endpoint protocol implementation and use it directly, this is not desirable in most cases. Using a particular endpoint protocol implementation directly makes a solution less flexible by making the solution dependent on a particular network transport.

The Endpoint Service

The Endpoint service provides an access point to all the endpoint protocol implementations installed on a peer, allowing a programmer to send a message using these endpoint protocol implementations. Unlike the other core services in JXTA, the Endpoint service is independent of a peer group. All peer groups share the same Endpoint service, which makes sense, considering that the Endpoint service provides the communication layer closest to the network transport layer. By default, a peer group in the reference implementation inherits the Endpoint service provided by its parent group. However, developers can provide a custom Endpoint service implementation for a peer group that they create by loading a custom Endpoint service. This Endpoint service implementation is loaded just like any other custom service, using the techniques demonstrated in Chapter 10, "Peer Groups and Services."

In the reference implementation, the Endpoint service, shown in Figure 9.1, is defined by the `EndpointService` interface in the `net.jxta.endpoint` package and is implemented by the `EndpointServiceImpl` class in the `net.jxta.impl.endpoint` package.

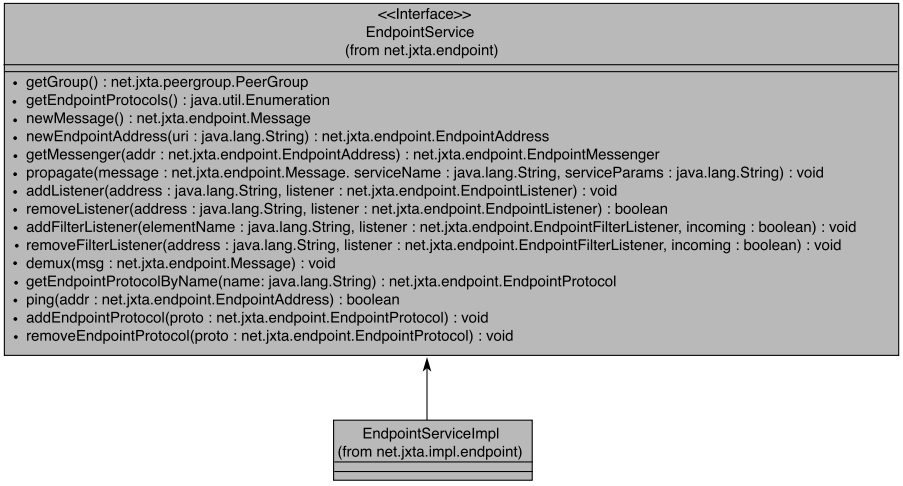

Figure 9.1 The Endpoint service interface and implementation.

Although the endpoint protocol implementations define the format for data crossing the network, the Endpoint service does add one piece of information when propagating a message. The Endpoint service adds a message element named `jxta:EndpointHeaderSrcPeer` to the outgoing messages. If visualized as XML, remembering that messages aren't necessarily rendered to XML, the message element's format would be as follows:

```
<jxta:EndpointHeaderSrcPeer>
  . . .
</jxta:EndpointHeaderSrcPeer>
```

The `jxta:EndpointHeaderSrcPeer` element contains the ID of the peer propagating the message. This Peer ID is used by the Endpoint service that receives the message to eliminate loopback by discarding messages whose source Peer ID matches the local Peer ID. The remainder of the formatting of an outgoing message is the responsibility of a particular endpoint protocol implementation registered with an `EndpointService` instance using the `EndpointService.addEndpointProtocol` method.

As shown in Figure 9.2, an endpoint protocol implementation realizes the `EndpointProtocol` interface from the `net.jxta.endpoint` package.

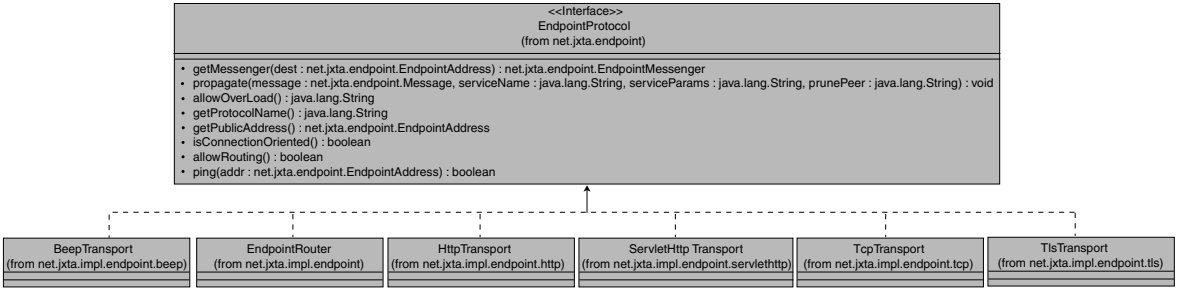

Figure 9.2 The `EndpointProtocol` interface and implementations.

An `EndpointProtocol` implementation allows a peer to propagate a message to as many peers as possible. In a TCP endpoint protocol implementation, for example, TCP multicast capabilities are used to send a message to as many peers on the local LAN segment as possible. An `EndpointProtocol` implementation is also responsible for allowing a peer to send a message directly to a peer located at a specific Endpoint Address. This functionality is provided by an implementation of the `EndpointMessenger` interface obtained using the `EndpointProtocol` implementation's `getMessenger` method. The `EndpointMessenger` interface, shown in Figure 9.3, is defined in `net.jxta.endpoint`.

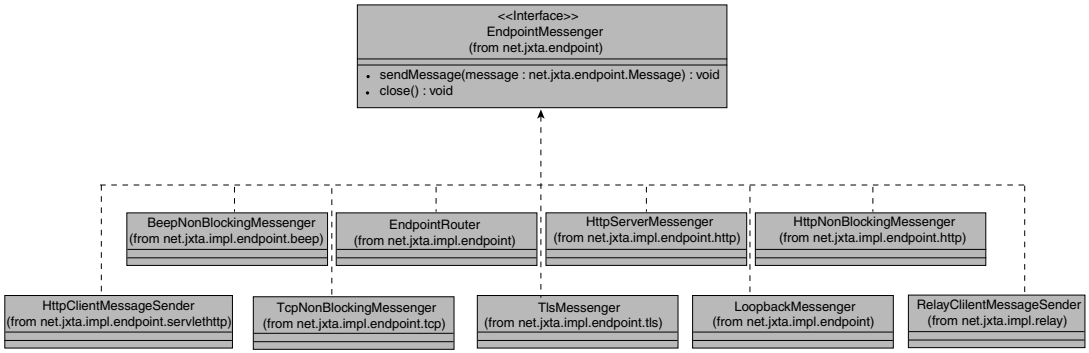

Figure 9.3 The `EndpointMessenger` interface and implementations.

An implementation of `EndpointMessenger` is usually obtained using the `EndpointService.getMessenger` method. The `EndpointService.getMessenger` method takes a `net.jxta.endpoint.EndpointAddress` argument that identifies the remote peer's transport-specific location. This address is used to determine which

`EndpointProtocol` provides connectivity to the remote peer, and it calls `getMessenger` on the `EndpointProtocol` implementation. This encapsulation and abstraction eliminate the need for a developer to ever instantiate an `EndpointMessenger` implementation directly.

Types of Endpoint Transport Implementations

In the Java reference implementation, currently five endpoint protocol implementations are available:

- **TCP (`net.jxta.impl.endpoint.tcp`)**—This provides a TCP `EndpointProtocol` implementation that uses a `MulticastSocket` to send data to peers on the local LAN segment. A TCP-based `EndpointMessenger` implementation uses a `Socket` to connect directly to a remote peer.
- **HTTP (`net.jxta.impl.endpoint.http`)**—This provides an HTTP `EndpointProtocol` and `EndpointMessenger`. This endpoint protocol is slightly different from a typical endpoint protocol implementation because the HTTP endpoint protocol implementation provides the router peer functionality that allows peers to perform firewall traversal. The HTTP implementation of `EndpointProtocol` does not provide broadcast capabilities.
- **Servlet HTTP (`net.jxta.impl.endpoint.servlethttp`)**—Similar to the HTTP implementation, the Servlet HTTP implementation provides HTTP transport functionality that can be plugged into application servers that support the Java Servlet APIs.
- **TLS (`net.jxta.impl.endpoint.tls`)**—This is the Transport Layer Security protocol endpoint protocol implementation. This endpoint protocol implementation does not provide broadcast capabilities because the TLS implementation is designed only for securing one-to-one communications. This implementation is built on top of libraries provided by the Cryptix project (`www.cryptix.org`).
- **BEEP (`net.jxta.impl.endpoint.beep`)**—This is the Block Extensible Exchange Protocol (IETF RFC 3080) implementation. BEEP is basically a framework for building application protocols. This endpoint protocol implementation does not provide broadcast capabilities. This implementation is built on top of libraries provided by `beepcore.org`.

One other implementation, the Endpoint Router implementation, provides a transport that handles finding routes to remote peers via gateways. This transport provides the implementation of the Endpoint Routing Protocol that will be discussed later in this chapter.

Each endpoint protocol implementation made available by a peer is identified by a Transport Advertisement in the peer's Peer Advertisement. However, the format of this Transport Advertisement varies by endpoint protocol implementation. Only the root element is common to all implementations:

```
<jxta:TransportAdvertisement>
  . . .
</jxta:TransportAdvertisement>
```

By default, the endpoint protocol implementations are configured when the JXTA platform boots. A default set of endpoint protocol implementations is added to the Endpoint service based on the settings provided by the user to the JXTA configuration tool. Currently, the default transports loaded include TCP, HTTP, and TLS.

Endpoint Addresses

Endpoint Addresses provide the network transport-specific information required to route a message over a particular endpoint protocol implementation to a specific peer and service. In general, the format of an Endpoint Address in the reference implementation takes this form:

```
<protocol>://<network address>/<service name>/<service parameters>
```

The following definitions are used for each section of the Endpoint Address:

- **<protocol>**—The name of the network transport to use when sending the message. Example values include `tcp`, `http`, and `jxtatls`.
- **<protocolAddress>**—The network transport-specific address used to locate the destination peer on the network. For example, a TCP Endpoint Address would use an IP address and port number for this value.
- **<serviceName>**—An identifier that uniquely specifies the destination service on the remote peer. This effectively allows messages arriving over a single network transport to be demultiplexed by the Endpoint service and passed to the appropriate service. To associate a service with a particular peer group, the service name is usually a combination of a common name for the service and the Peer Group ID.
- **<serviceParameters>**—Some unique identifying parameters being passed to the service. These parameters might be used by a particular destination service to provide information required to route the message to a particular handler instance before parsing the message itself.

For example, a message destined for the Pipe service on a remote peer using the TCP endpoint protocol implementation would use an Endpoint Address that looks like this:

```
tcp://10.6.18.38:80/PipeService/<Pipe ID>
```

In this example, Endpoint Address, `10.6.18.38:80` is the destination's IP address (10.6.18.38) and port number (80), `PipeService` is the name of the service, and `<Pipe ID>` is the parameter to the Pipe service.

Only the protocol and the network address are required elements in the Endpoint Address. If the Endpoint Address has no service specified, the form of the address changes slightly to this:

```
<protocol>://<network address>/
```

Note that, in this case, no service parameters are specified because no service has been specified. An address may also specify a service name but no service parameters, in which case the form of the address is as follows:

```
<protocol>://<network address>/<service name>
```

The reference implementation defines the `EndpointAddress` interface in `net.jxta.endpoint` and the `Address` implementation in `net.jxta.impl.endpoint`, shown in Figure 9.4, to handle the details of manipulating Endpoint Addresses.

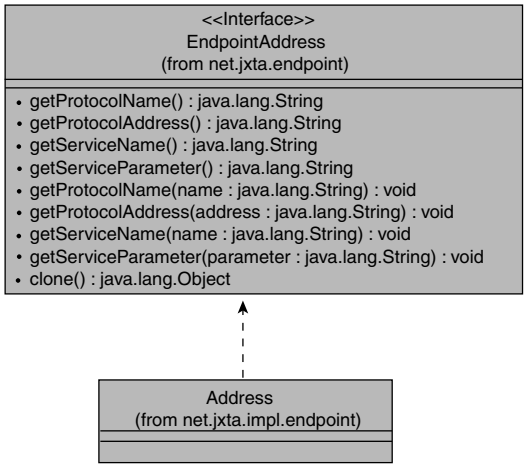

Figure 9.4 The `EndpointAddress` interface and implementation.

Instead of specifying an Endpoint Address in a transport-specific form, such as `tcp://10.6.18.38`, higher-level services in JXTA use a transport-neutral Endpoint Address of this form:

```
jxta://<unique Peer ID>
```

The `jxta` protocol specifier is used to indicate the JXTA-specific Endpoint Routing Protocol. This form of Endpoint Address is used to allow JXTA peers to act independently of the network transport. By using the `jxta` form of the address, a peer can send messages via the Endpoint Routing Protocol as if connecting directly to the remote peer. In fact, the message might travel

through several peers, a fact that is unknown to the peer. In this way, the Endpoint Routing Protocol abstracts the details of the underlying network topology, allowing a peer to act as if it is capable of connecting directly to a remote peer.

Message Formatting

Unlike the other services in JXTA, no corresponding protocol defines the format of messages sent by the Endpoint service. Although the endpoint protocol implementations are ultimately responsible for handling the details of formatting a message, the reference protocol implementations share code to render a message from the internal XML object structure into a format suitable for transport over the network. Currently, a transport can use two possible output formats to render a `Message` object:

- **Binary message format**—The message elements are rendered into simple binary byte stream. This functionality is encapsulated in the `MessageWireFormatBinary` class in the `net.jxta.impl.endpoint` package. This format of the output produced by this class is specified by the `application/x-jxta-msg` MIME type.
- **XML message format**—The message is rendered from the `Message` object's representation of an XML tree into real XML output. This functionality is encapsulated in the `MessageWireFormatXML` class in the `net.jxta.impl.endpoint` package. This format of the output produced by this class is specified by the `text/xml` MIME type.

Both `MessageWireFormatXML` and `MessageWireFormatBinary` extend the `MessageWireFormat` abstract class. Endpoint protocol implementations create an instance of a specific type of wire-formatting object using the `MessageWireFormatFactory` class and specifying the appropriate MIME type to the `newMessageWireFormat` method. It is up to the endpoint protocol implementation to choose the output format most appropriate to its particular network transport.

Using the Endpoint Service

To demonstrate the use of the Endpoint service, this section develops an application similar to the one in Chapter 7, "The Peer Information Protocol." The difference is that this example uses the Endpoint service on which pipes are built to provide the messaging functionality.

Receiving Incoming Messages

It should come as no surprise that the `EndpointService` is structured in a similar way to those services built on top of it. The Endpoint service provides the `EndpointListener` interface in `net.jxta.endpoint`, shown in Figure 9.5, to allow other services to receive notification of arriving messages.

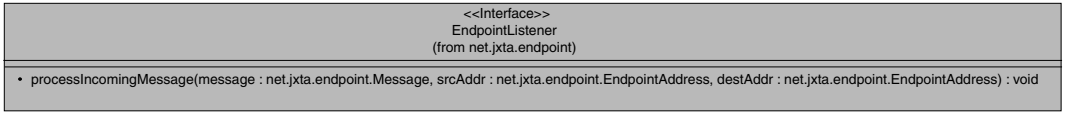

Figure 9.5 The `EndpointListener` interface.

The sole method that developers need to implement, `processIncomingMessage`, accepts the arriving `Message` object as well as the source and destination Endpoint Addresses:

```
public void processIncomingMessage(Message message,
    EndpointAddress source, EndpointAddress destination);
```

To listen for messages arriving for a specific service, a developer needs only to register an `EndpointListener` instance with the `EndpointService` instance using the `EndpointService.addListener` method:

```
public void addListener(String address, EndpointListener listener)
        throws IllegalArgumentException;
```

The `EndpointListener`'s `processIncomingMessage` method is called whenever a `Message` arriving at the peer contains a destination Endpoint Address that specifies a service matching the `address` value.

Note that when registering an `EndpointListener`, the `address` value that you pass shouldn't be just the name of the destination service. Instead, the `address` should be the concatenation of the destination service name and service parameters that are part of the destination Endpoint Address. This is an area that is ambiguous in the current reference implementation and will be refined in future releases.

To demonstrate the use of the `EndpointListener` and `EndpointService` interfaces, the `EndpointServer` example in Listing 9.1 starts the JXTA platform and adds itself to the `EndpointService` instance as a listener for messages addressed to a service with the name `EndpointServer` plus the same Peer Group ID with the parameters `012345`.

Listing 9.1 **Source Code for *EndpointServer.java***

```
package com.newriders.jxta.chapter9;

import java.util.Enumeration;

import net.jxta.endpoint.EndpointAddress;
import net.jxta.endpoint.EndpointProtocol;
import net.jxta.endpoint.EndpointService;
import net.jxta.endpoint.EndpointListener;
import net.jxta.endpoint.Message;

import net.jxta.exception.PeerGroupException;

import net.jxta.impl.endpoint.Address;

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


/**
 * A simple server that listens on an endpoint, looking for
 * Messages destined for a service named EndpointServer
 * concatenated with the Peer Group ID, with service
 * parameters 012345.
 */
public class EndpointServer implements EndpointListener
{
    /**
     * The peer group for the application.
     */
    private PeerGroup peerGroup = null;

    /**
     * The service name to use when listening for messages.
     * This service name will be appended with the Peer Group ID
     * of the peer group when the JXTA platform is started.
     */
    private String serviceName = "EndpointServer";

    /**
     * The service parameters to use when listening for
     * messages.
```

```
 */
private String serviceParameters = "012345";


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

    // Add the Peer Group ID of the group to the service
    // name so the endpoint listener is specific to
    // the peer group.
    serviceName += peerGroup.getPeerGroupID().toString();
}

/**
 * Runs the application: starts the JXTA platform, starts
 * listening on the Endpoint service for messages.
 *
 * @param   args the command-line arguments passed to
 *      the application.
 */
public static void main(String[] args)
{
    EndpointServer server = new EndpointServer();

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

        // Start the server.
        server.start();
    }
    catch (PeerGroupException e)
    {
        System.out.println("Error starting JXTA platform: "
```

continues

Listing 9.1 **Continued**

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

    /**
     * The EndpointListener implementation. Accepts an incoming
     * message for processing.
     *
     * @param   message the Message that has arrived for
     *           processing.
     * @param   source the EndpointAddress of the peer sending
     *           the message.
     * @param   destination the EndpointAddress of the
     *           destination peer for the message.
     */
    public void processIncomingMessage(Message message,
        EndpointAddress source, EndpointAddress destination)
    {
        System.out.println("Message received from " + source
            + " for " + destination + ":");
        System.out.println(message.getString("MessageText"));
    }

    /**
     * Start the server listening on the Endpoint service.
     */
    public void start()
    {
        EndpointService endpoint =
            peerGroup.getEndpointService();

        // Print out all of the endpoint protocol addresses.
        // These can be used by the EndpointClient to send a
        // message to the EndpointServer.
        EndpointProtocol aProtocol = null;
        Enumeration protocols = endpoint.getEndpointProtocols();
        while (protocols.hasMoreElements())
```

```
            {
                aProtocol =
                    (EndpointProtocol) protocols.nextElement();

                // Print out the address.
                System.out.println("Endpoint address: "
                    + aProtocol.getPublicAddress().toString());
            }

            // Add ourselves as a listener to the Endpoint service.
            endpoint.addListener(serviceName + serviceParameters,
                this);
        }
    }
```

By itself, the `EndpointServer` example isn't very useful without another peer capable of sending messages to the EndpointServer service for the peer group. Peers can propagate a message to many peers using the Endpoint service or send a message directly to a specific peer using an `EndpointMessenger`.

Propagating Messages Using the Endpoint Service

Propagating a message to a number of remote peers works in a similar fashion to using propagation pipes, but without the requirement for you to find and bind an output pipe. However, unlike propagation pipes, the Endpoint service cannot propagate messages across firewall and NAT boundaries. Propagation across firewalls and network boundaries is a feature offered by the Rendezvous service, explained in Chapter 6, "The Rendezvous Protocol," which builds on the Endpoint service to provide this capability. Propagation using the Endpoint service is built on the capabilities of registered endpoint protocol implementations to broadcast to a number of Endpoint Addresses simultaneously. This functionality is not available in all network transports, such as HTTP, but it is available in low-level network transports, such as TCP. The reference implementation of the Endpoint service provides propagation using only the TCP endpoint protocol implementation and is thus limited to propagating messages within the boundaries of a LAN segment.

`EndpointPropagateClient` in Listing 9.2 provides a simple example of the elements necessary to propagate a message using the `EndpointService` instance.

Listing 9.2 **Source Code for *EndpointPropagateClient.java***

```
package com.newriders.jxta.chapter9;

import java.io.IOException;

import net.jxta.endpoint.EndpointAddress;
import net.jxta.endpoint.EndpointService;
import net.jxta.endpoint.Message;

import net.jxta.exception.PeerGroupException;

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


/**
 * A simple client that uses the Endpoint service to propagate
 * messages to a service named EndpointServer concatenated
 * with the Peer Group ID, with the service parameters 012345
 * on all peers in the local LAN segment.
 */
public class EndpointPropagateClient
{
    /**
     * The peer group for the application.
     */
    private PeerGroup peerGroup = null;

    /**
     * The service name to use when listening for messages.
     * This service name will be appended with the Peer Group ID
     * of the peer group when the JXTA platform is started.
     */
    private String serviceName = "EndpointServer";

    /**
     * The service parameters to use when listening for
     * messages.
     */
    private String serviceParameters = "012345";
```

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

    // Add the Peer Group ID of the group to the service
    // name so the message is sent to the version of the
    // endpoint listener specific to this peer group.
    serviceName += peerGroup.getPeerGroupID().toString();
}

/**
 * Runs the application: starts the JXTA platform, accepts
 * user input messages, and propagates them to other peers.
 *
 * @param   args the command-line arguments passed to the
 *      application.
 */
public static void main(String[] args)
{
    EndpointPropagateClient client =
        new EndpointPropagateClient();

    try
    {
        boolean done = false;
        String messageString = null;

        // Initialize the JXTA platform.
        client.initializeJXTA();

        while (!done)
        {
            // Reset the message string.
            messageString = null;

            // Get the message; if the message is '.',
```

continues

Listing 9.2 **Continued**

```
                // then quit the application.
                System.out.print("Enter a message (or '.' "
                    + "to quit): ");
                messageString = client.readInput();

                if ((messageString.length() > 0)
                    && (!messageString.equals(".")))
                {
                    // Send a message to the server.
                    client.sendMessage(messageString);
                }
                else
                {
                    // We're done.
                    done = true;
                }
            }

            // Stop the JXTA platform. Currently, there isn't
            // any nice way to do this.
            System.exit(0);
        }
        catch (PeerGroupException e)
        {
            System.out.println("Error starting JXTA platform: "
                + e);
            System.exit(1);
        }
    }

    /**
     * Read a line of input from the system console.
     *
     * @return  the String read from the System.in InputStream.
     */
    public String readInput()
    {
        StringBuffer result = new StringBuffer();
        boolean done = false;
        int character;
```

```
    while (!done)
    {
        try
        {
            // Read a character.
            character = System.in.read();

            // Check to see if the character is a newline.
            if ((character == -1)
                || ((char) character == '\n'))
            {
                done = true;
            }
            else
            {
                // Add the character to the result string.
                result.append((char) character);
            }
        }
        catch (IOException e )
        {
            done = true;
        }
    }

    return result.toString().trim();
}

/**
 * Sends a message. In this case, the message string is
 * propagated to all peers in the peer group on the local
 * LAN segment.
 *
 * @param   messageString the message to send to other
 *          peers.
 */
public void sendMessage(String messageString)
{
    EndpointService endpoint =
        peerGroup.getEndpointService();

    // Create a new message.
```

continues

Listing 9.2 **Continued**

```
            Message message = endpoint.newMessage();

            // Populate the message contents with the messageString.
            message.setString("MessageText", messageString);

            try
            {
                // Propagate the message within the peer group.
                endpoint.propagate(message, serviceName,
                    serviceParameters);
            }
            catch (IOException e)
            {
                System.out.println("Error sending message: " + e);
            }
        }
    }
```

To propagate a message, create a `Message` object using the `EndpointService`'s `createMessage` method, and populate it in the same fashion as when sending a message using a pipe. As with any `Message`, multiple elements can be added. In the example, a single element called `MessageText` containing the outgoing text being sent to the remote peer is added using the following code:

```
// Populate the message contents with the messageString.
message.setString("MessageText", messageString);
```

It is propagated to other peers using this code:

```
// Propagate the message within the peer group.
endpoint.propagate(message, serviceName, serviceParameters);
```

The `EndpointService.propagate` method takes not only the message being propagated, but also the name of the destination service and parameters to pass to the destination service.

Using *EndpointServer* and *EndpointPropagateClient*

As with the `PipeServer` and `PipeClient` examples created in Chapter 8, using `EndpointServer` and `EndpointPropagateClient` requires two separate instances of the JXTA platform. To prepare to run the `EndpointServer` and `EndpointPropagateClient` examples, follow these steps:

1. Create two directories, placing the `EndpointServer` source code in one directory and the `EndpointPropagateClient` source code in the other.
2. Copy all of the JAR files from the `lib` directory under the JXTA Demo `install` directory into each directory.
3. Start a command console and change to the directory containing the `EndpointServer` code.
4. Compile `EndpointServer` using `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 EndpointServer.java`.
5. Start the EndpointServer using `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.chapter9.EndpointServer`.

   `EndpointServer` starts and prints the Endpoint Address for each of the protocols registered with the `EndpointService` instance. This isn't used in this example, but it will be used when demonstrating the use of `EndpointMessenger`.
6. Start a second command console and change to the directory containing the `EndpointPropagateClient` code.
7. Compile `EndpointPropagateClient` using `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 ➥EndpointPropagateClient.java`.
8. Start `EndpointPropagateClient` using `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.chapter9.EndpointPropagateClient`.

`EndpointPropagateClient` starts and prompts for a message to send. Each message can be only one line long, and the client continues to prompt for a message until a `.` is entered as a message. The client then quits.

Each message entered into `EndpointPropagateClient` should appear in the output of `EndpointServer`. However, to truly see the effect of propagation, you might want to create a copy of the directory containing the `EndpointServer` code to run a second instance of `EndpointServer`. This enables you to see multiple peers receiving the message propagated by the client and illustrates the difference between propagation and the technique used by the example in the next section. When starting a second instance of `EndpointServer`, be sure to configure a different TCP and HTTP port for the JXTA platform.

Sending Messages Directly Using *EndpointMessenger*

The disadvantage of the propagation demonstrated in the previous example is that it's wasteful. Peers that might not be interested in the message receive the message, only to discard it. In the reference implementation, the TCP endpoint protocol implementation's use of TCP multicast limits this inefficiency to peers on the local LAN segment. To improve efficiency, it would be useful if a message could be sent to one specific peer using the `EndpointService` instance.

In fact, `EndpointService` does support this functionality through the `EndpointMessenger` interface. The `EndpointProtocol` interface allows a developer to obtain an `EndpointMessenger` instance for the endpoint protocol implementation. This object can be used to send messages to a specific peer located at a specific `EndpointAddress`. This functionality is used by the reference implementation to provide an implementation of the `OutputPipe` interface.

When starting `EndpointServer` in the "Using *EndpointServer* and *EndpointPropagateClient*" section, the `EndpointServer` prints the `EndpointAddress` for each protocol currently registered with the `EndpointService` instance. Each address follows the same basic format outlined in the "Endpoint Addresses" section earlier in this chapter. The example in Listing 9.3 prompts the user for a message and a destination address, and attempts to send the message using `EndpointMessenger`.

Listing 9.3 **Source Code for *EndpointMessengerClient.java***

```
package com.newriders.jxta.chapter9;

import java.io.IOException;

import net.jxta.endpoint.EndpointAddress;
import net.jxta.endpoint.EndpointMessenger;
import net.jxta.endpoint.EndpointService;
import net.jxta.endpoint.Message;

import net.jxta.exception.PeerGroupException;

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


/**
 * A simple Endpoint client that sends a message directly to a
 * service named EndpointServer concatenated with the Peer
 * Group ID, with service parameters 012345 on a specific peer
 * located at an Endpoint Address using an EndpointMessenger.
```

```
*/
public class EndpointMessengerClient
{
    /**
     * The peer group for the application.
     */
    private PeerGroup peerGroup = null;

    /**
     * The service name to use when listening for messages.
     * This service name will be appended with the Peer Group ID
     * of the peer group when the JXTA platform is started.
     */
    private String serviceName = "EndpointServer";

    /**
     * The service parameters to use when listening for
     * messages.
     */
    private String serviceParameters = "012345";


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

        // Add the Peer Group ID of the group to the service
        // name so the message is sent to the version of the
        // endpoint listener specific to this peer group.
        serviceName += peerGroup.getPeerGroupID().toString();
    }

    /**
     * Runs the application: starts the JXTA platform, accepts
     * user input message and endpoint info, and sends the
     * message to the Endpoint Address specified.
     *
```

continues

Listing 9.3 **Continued**

```
* @param   args the command-line arguments passed to the
*      application.
*/
public static void main(String[] args)
{
    EndpointMessengerClient client =
        new EndpointMessengerClient();

    try
    {
        boolean done = false;
        String messageString = null;
        String addressString = null;

        // Initialize the JXTA platform.
        client.initializeJXTA();

        while (!done)
        {
            // Reset the strings.
            addressString = null;
            messageString = null;

            // Get the message; if the message is ., then
            // quit the application.
            System.out.print("Enter a message (or '.' "
                + " to quit): ");
            messageString = client.readInput();

            if ((messageString.length() > 0)
                && (!messageString.equals(".")))
            {
                // Get the destination Endpoint Address
                // from the user.
                System.out.print(
                    "Enter an endpoint address: ");
                while ((addressString == null)
                    || (addressString.length() == 0))
                {
                    addressString = client.readInput();
                }
```

```
                // Send a message to the server.
                client.sendMessage(
                    messageString, addressString);
            }
            else
            {
                // We're done.
                done = true;
            }
        }

        // Stop the JXTA platform. Currently, there isn't
        // any nice way to do this.
        System.exit(0);
    }
    catch (PeerGroupException e)
    {
        System.out.println("Error starting JXTA platform: "
            + e);
        System.exit(1);
    }
}

/**
 * Read a line of input from the system console.
 *
 * @return  the String read from the System.in InputStream.
 */
public String readInput()
{
    StringBuffer result = new StringBuffer();
    boolean done = false;
    int character;

    while (!done)
    {
        try
        {
            // Read a character.
            character = System.in.read();

            // Check to see if the character is a newline.
            if ((character == -1)
                || ((char) character == '\n'))
```

continues

Listing 9.3 **Continued**

```
                {
                    done = true;
                }
                else
                {
                    // Add the character to the result string.
                    result.append((char) character);
                }
            }
            catch (IOException e )
            {
                done = true;
            }
        }

        return result.toString().trim();
    }

    /**
     * Sends a message. In this case, the message string is sent
     * to the Endpoint Address specified, provided that the
     * Endpoint Address responds to a ping.
     *
     * @param   messageString the message to send to the peer.
     * @param   addressString the Endpoint Address of the
     *      destination peers.
     */
    public void sendMessage(String messageString,
        String addressString)
    {
        EndpointService endpoint =
            peerGroup.getEndpointService();
        EndpointAddress endpointAddress =
            endpoint.newEndpointAddress(addressString);

        // Manipulate the Endpoint Address to include the
        // appropriate destination service name and parameters.
        endpointAddress.setServiceName(serviceName);
        endpointAddress.setServiceParameter(serviceParameters);
```

```
        // Check that we can reach the Endpoint Address.
        if (endpoint.ping(endpointAddress))
        {
            // Create a new message.
            Message message = endpoint.newMessage();

            // Populate the message contents with the
            // messageString.
            message.setString("MessageText", messageString);

            try
            {
                EndpointMessenger messenger =
                    endpoint.getMessenger(endpointAddress);

                if (messenger != null)
                {
                    // Send the message directly to the Endpoint
                    // Address specified.
                    messenger.sendMessage(message);
                }
                else
                {
                    System.out.println("Unable to create "
                        + "messenger for given address.");
                }
            }
            catch (IOException e)
            {
                System.out.println("Error creating messenger "
                    + "or sending message: " + e);
            }
        }
        else
        {
            System.out.println("Unable to reach specified "
                + "address!");
        }
    }
}
```

Running the `EndpointMessengerClient` example requires similar steps to those outlined in the section "*Using EndpointServer* and *EndpointPropagateClient*":

1. Start an instance of `EndpointServer`.
2. Copy the `EndpointMessengerClient` source code into the same directory where you previously copied `EndpointPropagateClient`.
3. Compile `EndpointMessengerClient` using `javac -d . -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptixasn1.jar;instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;log4j.jar;minimalBC.jar EndpointMessengerClient.java`.
4. Start `EndpointMessengerClient` using `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.chapter9.EndpointMessengerClient`.

`EndpointMessengerClient` starts and prompts you to enter a message. After you have entered a message, the client prompts for a destination Endpoint Address. Enter one of the Endpoint Addresses printed by `EndpointServer` when it started.

The difference between `EndpointMessengerClient` and `EndpointPropagateClient` will become obvious if you copy the client directory and start a second instance of `EndpointServer`. Unlike in the `EndpointPropagateClient` example, only the server specified by the Endpoint Address entered into `EndpointMessengerClient` will receive the message.

The Endpoint Filter Listener

One other feature offered by the `EndpointService` interface is the capability to add filter listeners implementing the `EndpointFilterListener` interface (shown in Figure 9.6), defined in `net.jxta.endpoint`. `EndpointFilterListener` implementations can be registered with the `EndpointService` instance to allow a developer to arbitrarily preprocess incoming messages before they are handed off to the registered `EndpointListener` implementations.

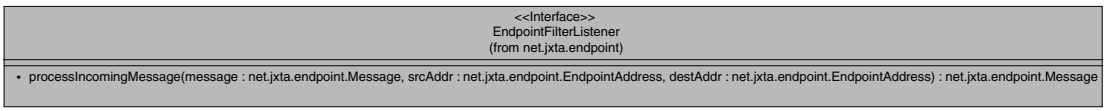

Figure 9.6 The `EndpointFilterListener` interface.

Currently, the `EndpointFilterListener` interface is implemented by the `EndpointServiceStatsFilter` class in `net.jxta.impl.util`. This class is used to collect the message throughput statistics delivered by the Peer Information Protocol. The Rendezvous service reference implementation, `RendezVousServiceImpl`, also uses an inner class, `FilterListener`, to implement `EndpointFilterListener`. This implementation is used to prevent uncontrolled propagation and loopbacks.

Filter listeners are added to the Endpoint service using the `EndpointService.addFilterListener` method:

```
public void addFilterListener(String elementName,
    EndpointFilterListener listener, boolean incoming)
        throws IllegalArgumentException;
```

When registering a filter listener, the caller specifies whether the listener should be called to process incoming or outgoing messages. In addition, the caller specifies the name of a message element that a message must contain before the filter will be applied. Only those messages containing an element with a matching element name will have the filter applied to the message.

To understand how filters are applied to incoming messages, it is necessary to understand how incoming messages flow from an endpoint protocol implementation to registered `EndpointListener` instances. When an endpoint protocol implementation receives a complete message from a remote peer, it calls the `EndpointService.demux` method. The `demux` method implementation is responsible for first preprocessing the message using the registered `EndpointFilterListener` instances and then notifying registered `EndpointListener` instances. The `demux` method acts as a callback, freeing an endpoint protocol implementation from the duty of applying filters and notifying listeners itself.

In the current reference implementation, filters are not applied on outgoing messages. However, there is already some code in place, indicating that this feature will be implemented soon.

Although both `EndpointListener` and `EndpointFilterListener` define only a single `processIncomingMessage` method, there is one important difference between the two interfaces. Unlike `EndpointListener`, `EndpointFilterListener`'s version of `processIncomingMessage` returns a `Message` object. This object is used as input into subsequent filters and finally is used to either send the outgoing message to other peers or notify registered `EndpointListener` instances. If an `EndpointFilterListener` returns a `null` object, the message is discarded.

Introducing the Endpoint Routing Protocol

After examining the example code and the explanation of the `EndpointProtocol` and `EndpointMessenger` interfaces, you've probably realized that JXTA needs a mechanism to send messages between peers that aren't directly connected. Although the HTTP endpoint protocol implementation in the reference implementation provides router peer functionality that allows a message to traverse a firewall, a peer still needs some way to learn of the existence of the router peer in the first place. Because router peers may enter or leave the network spontaneously, a peer needs a routing mechanism that works even in situations in which the route between two peers is constantly changing. Enter the Endpoint Routing Protocol.

If two peers cannot communicate directly using a common endpoint protocol implementation, the Endpoint Routing Protocol provides each peer with a way to discover how it can send messages to the other peer via an intermediary, using only available endpoint protocol implementations (see Figure 9.7).

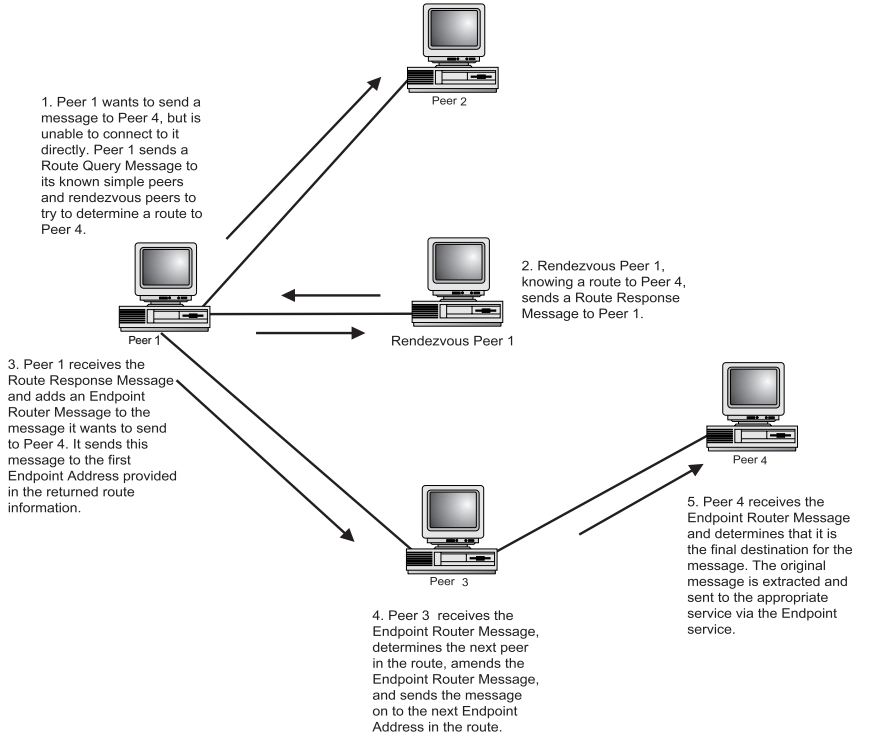

Figure 9.7 Flow of the Endpoint Routing Protocol.

The Endpoint Routing Protocol, also called the Peer Endpoint Protocol, provides a mechanism for a message to be sent to a remote peer using discovered route information. Each intermediary along the message route is responsible for passing the message on to the next peer described by the route information until the message reaches its ultimate destination.

For now, only two messages are required to determine route information: the Route Query Message and the Route Response Message. The current JXTA Protocols Specification defines three other messages for the Endpoint Routing Protocol: the Ping Query Message, the Ping Response Message, and the NACK Message. These messages allow a peer to test that a message can be routed to a destination peer and also allow an intermediary peer to signal the sender that an attempt to route a message has failed. These messages are not currently available in the reference implementation and will not be discussed.

The Endpoint Routing Protocol defines one other message, the Endpoint Router Message, which is used to pass route information along with a message. Peers along the message's path as it travels to its destination use the extra information provided by the Endpoint Router Message to determine the next peer en route to the destination.

The Route Query Message

A Route Query Message is sent by a peer when it wants to determine the set of ordered peers to use to send a message to a given Endpoint Address. Listing 9.4 shows the elements of the Router Query Message.

Listing 9.4 **The Route Query Message XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:EndpointRouter>
  <Type>RouteQuery</Type>
  <DestPeer> . . . </DestPeer>
  <RoutingPeerAdv> . . . </RoutingPeerAdv>
</jxta:EndpointRouter>
```

Each element in the Route Query Message describes one aspect required to perform the search for route information:

- **`Type`**—A required element describing the type of Endpoint Router message being sent. For the Route Query Message, this element is set to `RouteQuery`.

- **DestPeer**—An optional element containing the Endpoint Address of the final destination peer in the route being discovered. Any route returned in response to this Route Query Message provides a route that allows a message to be sent from the local peer to the peer specified by `DestPeer`.
- **RoutingPeerAdv**—An optional element containing the Peer Advertisement of the peer requesting route information.

To discover route information, a peer sends a Route Query Message to other peers that it has previously discovered. In addition to finding peers by peer discovery, a peer may learn of another peer's existence by processing a Route Query Message and extracting the `RoutingPeerAdv`, if one has been passed. By reusing this Peer Advertisement, the peer can save network bandwidth and potentially reduce the time required to obtain route information, resulting in improved performance.

As with any of the core protocols, a query might not result in a response or might result in multiple responses.

The Route Response Message

To provide a reply to a Route Query Message, a peer sends a Route Response Message describing a set of ordered Endpoint Addresses to use to send a message to a given destination peer. The Route Response Message has the format shown in Listing 9.5.

Listing 9.5 **The Route Response Message XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:EndpointRouter>
  <Version>2</Version>
  <Type>RouteResponse</Type>
  <DestPeerIdTag> . . . </DestPeerIdTag>
  <RoutingPeerIdTag> . . . </RoutingPeerIdTag>
  <NbOfHops> . . . </NbOfHops>
  <RoutingPeerAdvTag> . . . </RoutingPeerAdvTag>
  <GatewayForward> . . . </GatewayForward>
</jxta:EndpointRouter>
```

The Route Response Message contains similar information to the Route Query Message, with the exception that it contains the route information requested by the peer:

- **`Version`**—A required element containing an integer describing the version of the Endpoint Routing Protocol being employed in the protocol conversation. Although the Protocols Specification defines this as a required element for both the Route Query and Response Message formats, the reference implementation currently adds it only in the Route Response Message. At this time, the `Version` is set to 2.
- **`Type`**—A required element containing a string describing the type of Endpoint Router Message being sent. For the Route Response Message, this element is set to `RouteResponse`.
- **`DestPeerIdTag`**—An optional element containing the Endpoint Address of the final destination peer in the route being discovered. This should match the Endpoint Address passed in the original Route Query Message's `DestPeerIdTag` element.
- **`RoutingPeerIdTag`**—An optional element containing the Endpoint Address of the peer that is acting as a source of route information. This will most likely be called `RoutingPeer` in the future because it contains an Endpoint Address rather than a Peer ID.
- **`RoutingPeerAdvTag`**—An optional element containing the Peer Advertisement of the peer requesting route information.
- **`NbOfHops`**—An optional element containing the number of network hops in the route to the destination peer.
- **`GatewayForward`**—An optional element containing the Endpoint Address of a peer along the route to the destination peer. There may be several `GatewayForward` elements in a Route Response Message, and the route depends on the order of these elements. The `GatewayForward` elements describe the path in order of Endpoint Addresses that a message must visit to reach a destination peer.

When a peer receives a Route Query Message, it checks to see if it knows how to route a message to the specified destination peer. If so, it returns the route information in a Route Response Message; otherwise, the current reference implementation discards the query and returns no response.

The Endpoint Router Message

The Endpoint Router Message provides the information required to route a message to its destination after the message has left its source peer. Rather than encapsulating the message being routed, the Endpoint Router Message simply

adds routing information alongside the other content of a message. The Endpoint Router Message provides the route information in the format in Listing 9.6.

Listing 9.6 **The Endpoint Router Message XML**

```
<jxta:JxtaEndpointRouter>
  <jxta:Src> . . . </jxta:Src>
  <jxta:Dest> . . . </jxta:Dest>
  <jxta:Last> . . . </jxta:Last>
  <jxta:NBOH> . . . </jxta:NBOH>
  <jxta:GatewayForward> . . . </jxta:GatewayForward>
  <jxta:GatewayReverse> . . . </jxta:GatewayReverse>
</jxta:JxtaEndpointRouter>
```

Each element provides information required to route a message to the destination peer and also how to route a response message to the source peer:

- **`Src`**—A required element containing the Endpoint Address of the original peer responsible for sending the message.
- **`Dest`**—A required element containing the Endpoint Address of the destination peer for the message.
- **`Last`**—An optional element containing the Endpoint Address of the previous peer in the routing order. This address corresponds to the peer responsible for sending a message to the current peer.
- **`NBOH`**—An optional element containing the number of network hops contained in the reverse route. If this parameter is set to `0`, it indicates that the message doesn't contain reverse routing information.
- **`GatewayForward`**—An optional element containing the Endpoint Address of a peer along the route to the destination peer. There may be several `GatewayForward` elements in a Route Response Message, and the route depends on the order of these elements. The `GatewayForward` elements describe the path in order of Endpoint Addresses that a message must visit to reach a destination peer.
- **`GatewayReverse`**—An optional element containing the Endpoint Address of a peer along the route from the destination peer to the source peer. There may be several `GatewayReverse` elements in a Route Response Message, and the reverse route depends on the order of these elements. The `GatewayReverse` elements describe the path in order of Endpoint Addresses that a message must visit to reach the original source peer.

As a peer receives an Endpoint Router Message, it determines the next peer in the route, modifies the Endpoint Router Message, and sends the message on to the next peer. The next peer in the route can be determined by either sending a Route Query Message or consulting the `GatewayForward` elements in the Endpoint Router Message accompanying the message.

Although a peer isn't required to populate the `GatewayForward` and `GatewayReverse` elements of the Endpoint Router Message before sending the message to the next peer, the JXTA Protocols Specification encourages peers to add this information. Adding this information not only reduces the processing and route query overhead required at each point along the route, but it also improves the performance of the routing process.

The Endpoint Router Transport Protocol

Up to this point, you might have assumed that the Endpoint Routing Protocol is implemented as a service, just like all the other core protocols in JXTA. However, to simplify the implementation of the Endpoint Routing Protocol, it is implemented as an endpoint protocol implementation, bound within the Endpoint service to the `jxta` protocol specifier. This endpoint protocol implementation, called the Endpoint Router Transport Protocol, is invoked when a message is sent to an Endpoint Address of the form `jxta://<Peer ID unique format>`. The mechanism is invoked in exactly the same fashion that the TCP endpoint protocol implementation gets invoked when sending a message to an Endpoint Address of the form `tcp://10.6.18.38`.

Because the Endpoint Router Transport Protocol is invoked automatically to handle messages being sent to Endpoint Addresses for the Endpoint Router, the developer never has to interact with the endpoint protocol implementation directly. To send a message to a remote peer via the Endpoint Router Transport, a developer needs only to create an Endpoint Router Endpoint Address from the Peer ID of the destination peer:

```
PeerID peerId;
EndpointServer endpoint;
. . .
String asString = "jxta://" + peerId.getUniqueValue().toString();
EndpointAddress address = endpoint.newEndpointAddress(asString);
```

After the `EndpointAddress` has been created, the service name and service parameters can be set, just as with any other Endpoint Address. The message can then be sent to the remote peer via the Endpoint Routing Transport Protocol by using the `EndpointService` to obtain an `EndpointMessenger` object for the `EndpointAddress`.

The Endpoint Router Transport Protocol in the reference implementation is provided by the `EndpointRouter` class in the `net.jxta.impl.endpoint` package. When the `getMessenger` method is called via the `EndpointService.getMessenger` method, `EndpointRouter` transparently handles determining route information, either from cached information or from sending Route Query Messages. If a direct connection is possible using one of the registered endpoint protocol implementations, the method returns the appropriate messenger; otherwise, the method returns an `EndpointRouter.EndpointRouterMessenger` object. This class implements `EndpointMessenger` and adds an Endpoint Router Message to outgoing message. The important point to realize here is that the Endpoint service is responsible for masking all of this from the developer. As long as the `jxta://` form of the Endpoint Address is used, the `EndpointService` instance handles the details of finding the right endpoint protocol implementation or routing information in a transparent fashion.

The Endpoint Routing Transport Protocol is incapable of propagating messages to multiple peers, and the reference implementation provides an empty implementation for `EndpointProtocol.propagate`. However, propagation isn't really the responsibility of the Endpoint Routing Transport Protocol. The Rendezvous service is responsible for propagating messages via known rendezvous peers when the peer is behind a firewall. Messages to individual rendezvous peers sent by the `RendezvousServiceImpl` use `EndpointMessenger`, allowing Endpoint Router–formatted Endpoint Addresses to correctly invoke the Endpoint Routing protocol implementation.

Summary

In this chapter, you explored the Endpoint service and the Endpoint Routing Protocol. These two elements are responsible for encapsulating and abstracting network transport-specific details and hiding those details from higher services. All that remains is to learn how to create new services and applications of your own. To do this, Chapter 10 discusses services and peer groups, how they relate, and how to create your own peer group and associate services with it.