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!

7

The Peer Information Protocol

The Peer Information Protocol (PIP) allows peers to obtain status information from previously discovered peers. This status information is currently limited to include only data on the uptime of the peers and the amount of traffic processed by the peer. Future work on the PIP will most likely extend this basic protocol to provide ways for developers to extend the protocol's default status-monitoring capabilities.

Introducing the Peer Information Protocol

After a remote peer has been discovered using the Discovery service and the Peer Discovery Protocol, a peer might want to monitor the remote peer's status to make additional decisions about how to use the remote peer most effectively or to make the use of its services by other peers more efficient. Monitoring is an essential part of a P2P network, providing information that peers can use to leverage the resources of the P2P network in the most efficient manner. For example, in a file-sharing P2P application, a peer could use status information describing the current network traffic on a remote peer to decide whether to use the remote peer as a source of services. If the remote peer is under an extreme load, it's in the interests of both the client peer and the P2P network in general to shift usage away from that remote peer.

The Peer Information Protocol (PIP) is an optional JXTA protocol that allows a peer to monitor a remote peer and obtain information on the remote peer's current status. As with all the protocols described up to this point in the book, the PIP requires only two types of messages:

- **Peer Info Query Message**—A message format for querying a remote peer's status
- **Peer Info Response Message**—A message format for providing a peer's status to other peers

These two messages are responsible for providing access to a peer's status information using the protocol shown in Figure 7.1.

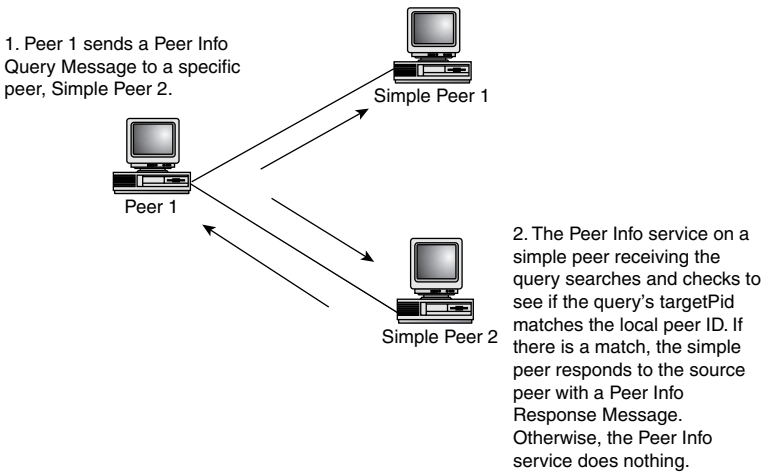

Figure 7.1 Exchange of Peer Info messages.

Although Figure 7.1 shows that a peer that receives a Peer Info Query Message not addressed to it does nothing with the message, the reference implementation is slightly different. The reference implementation provides for the possibility that a query message might be propagated to a peer instead of sent to a specific peer. If a peer receives a query to which it isn't the subject of the query, the peer propagates the query to the peer group.

The Peer Info service implements the PIP by leveraging the Resolver and Rendezvous services. This implementation follows a similar pattern to the Discovery service. The Peer Info service uses Resolver Query and Response Messages and the Resolver service to handle the details of sending a query to

a named handler and generating a response. The Resolver service is responsible for handling the details of propagating messages to other simple peers and rendezvous peers for the Resolver service. As with all the protocols in JXTA that you've seen so far, a PIP query to a remote peer might not result in a response.

The Peer Info Query Message

The Peer Info Query Message is very simple, if not a little limited, as shown in Listing 7.1.

Listing 7.1 **The Peer Info Query Message**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PeerInfoQueryMessage xmlns:jxta="http://jxta.org">
  <sourcePid> . . . </sourcePid>
  <targetPid> . . . </targetPid>
  <request> . . . </request>
</jxta:PeerInfoQueryMessage>
```

The Peer Info Query Message specifies three parameters:

- **`sourcePid`**—A required element containing the ID of the peer requesting status information from a remote peer. This Peer ID is a string encoding of the JXTA URN for the peer that generated the query.
- **`targetPid`**—A required element containing the ID of the remote peer from which status information is being solicited. This Peer ID is a string encoding of the JXTA URN for the peer that is the target of the query.
- **`request`**—An optional element containing a string specifying the status information being requested from the remote peer. The format of this request string is unspecified; it is the responsibility of the recipient to know how to decode it.

Unfortunately, the current reference implementation provides no mechanism to allow a developer to handle requests specified by the contents of the `request` element in the query. It appears that this is work that will be undertaken in the future to allow developers to add their own code to unmarshall the contents of the `request` element and provide response information to the peer requesting status information.

Unlike other protocols that you've seen so far, each rendezvous peer that propagates this message does not generate a response to the query. When the Peer Info service receives a Peer Info Query Message, it checks to see if the

local peer's ID matches the `targetPid`. If the IDs match, the service generates a response that is sent by the Resolver service to the source peer. Otherwise, no response is generated and the message is propagated to other peers that might be capable of providing the response. Theoretically, it should be possible to propagate a Peer Info Query Message in the reference implementation, but functionality to handle this case has been added as a precaution.

The Peer Info Query Message is different from the other protocol implementations in another significant way: The abstract `PeerInfoQueryMessage` class in net.jxta.protocol and the reference implementation `PeerInfoQueryMsg` in net.jxta.impl.protocol aren't used throughout by the reference implementation! Although some areas of the reference implementation do use these objects, their use is inconsistent.

This inconsistency suggests that the implementation of the PIP is in a much earlier stage of development compared to some of the other protocols. The lack of a mechanism to allow a developer to handle the contents of the `request` element of the Peer Info Query Message further underlines the fact that the PIP is still a work in progress.

The Peer Info Response Message

The counterpart to the Peer Info Query Message, the Peer Info Response Message, is significantly more detailed, as shown in Listing 7.2.

Listing 7.2 **The Peer Info Response Message**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PeerInfoResponse xmlns:jxta="http://jxta.org">
  <sourcePid> . . . </sourcePid>
  <targetPid> . . . </targetPid>
  <uptime> . . . </uptime>
  <timestamp> . . . </timestamp>
  <response> . . . </response>
  <traffic>
    . . .
  </traffic>
</jxta:PeerInfoResponse>
```

The Peer Info Response Message provides a variety of status information, most of which is oriented to providing information on the network traffic load on the remote peer:

- **sourcePid**—A required element containing the ID of the peer requesting status information from the peer. This Peer ID is a string encoding of the JXTA URN for the peer that generated the original Peer Info Query Message.
- **targetPid**—A required element containing the ID of the remote peer from which status information is being solicited. This Peer ID is a string encoding of the JXTA URN for the peer providing the response to the Peer Info Query Message.
- **uptime**—An optional element containing the amount of time, in milliseconds, since the peer joined the P2P network. In the reference implementation, the uptime corresponds to the amount of time that has elapsed since the Peer Info service started.
- **timestamp**—An optional element containing a timestamp describing the time when the peer generated the status information contained in the response. This timestamp is given in milliseconds since the epoch (January 1, 1970, 00:00:00 GMT).
- **response**—An optional element containing a string specifying the status information being returned in response to the query's `request` element's content. The format of this response string is unspecified; it is the responsibility of the recipient to know how to decode it. The reference PIP implementation does not currently provide a mechanism to allow a developer to populate the `response` element to provide the requested information.
- **traffic**—An optional element that contains details on the network traffic handled by the peer. The format of the contents of the `traffic` element is shown in Listing 7.3.

Listing 7.3 **The Format of the *traffic* Element Contents**

```
<traffic>
  <lastIncomingMessageAt> . . . </lastIncomingMessageAt>
  <lastOutgoingMessageAt> . . . </lastOutgoingMessageAt>
  <in>
    <transport endptaddr=" . . . "> . . . </transport>
  </in>
  <out>
    <transport endptaddr=" . . . "> . . . </transport>
  </out>
</traffic>
```

The contents of the `traffic` element in the Peer Info Response Message describe in detail the network traffic handled by the peer:

- **`lastIncomingMessageAt`**—An optional element containing a timestamp specifying the last time that the peer's endpoints handled an incoming message. The timestamp is given in milliseconds since the epoch.
- **`lastOutgoingMessageAt`**—An optional element containing a timestamp specifying the last time that the peer's endpoints handled an outgoing message. The timestamp is given in milliseconds since the epoch.
- **`in`**—An optional element containing details on the inbound traffic seen by the peer's endpoints. The `in` element may contain zero or more `transport` elements.
- **`transport`**—An optional element containing the number of bytes processed by the endpoint address specified by the `endptaddr` attribute. When used inside the `in` element, this element specifies the number of bytes received by the endpoint address specified. When used inside the `out` element, this element specifies the number of bytes sent by the endpoint address specified. The format of the endpoint address is covered in Chapter 9, "The Endpoint Routing Protocol."
- **`out`**—A container element to hold details on the outbound traffic seen by the peer's endpoints. The `out` element may contain zero or more `transport` elements.

The reference implementation of the PIP has one oversight in its current form: Peer Info Query Messages are propagated indiscriminately. When a peer receives a Peer Info Query in which the `targetPid` matches its local Peer ID, it generates a Peer Info Response Message that the Resolver service sends to the peer that generated the query. Unfortunately, the Resolver service still propagates the query, even though the target peer has responded.

Similar to the Peer Info Query Message, the reference implementation provides the `PeerInfoResponseMessage` abstract class in net.jxta.protocol and the `PeerInfoResponseMsg` implementation class in net.jxta.impl.protocol. These classes are shown in Figure 7.2.

Unlike the `PeerInfoQueryMessage` and `PeerInfoQueryMessage` classes, the `PeerInfoResponseMessage` and `PeerInfoResponseMsg` classes are used throughout the reference implementation to handle parsing and formatting Peer Info Response Messages.

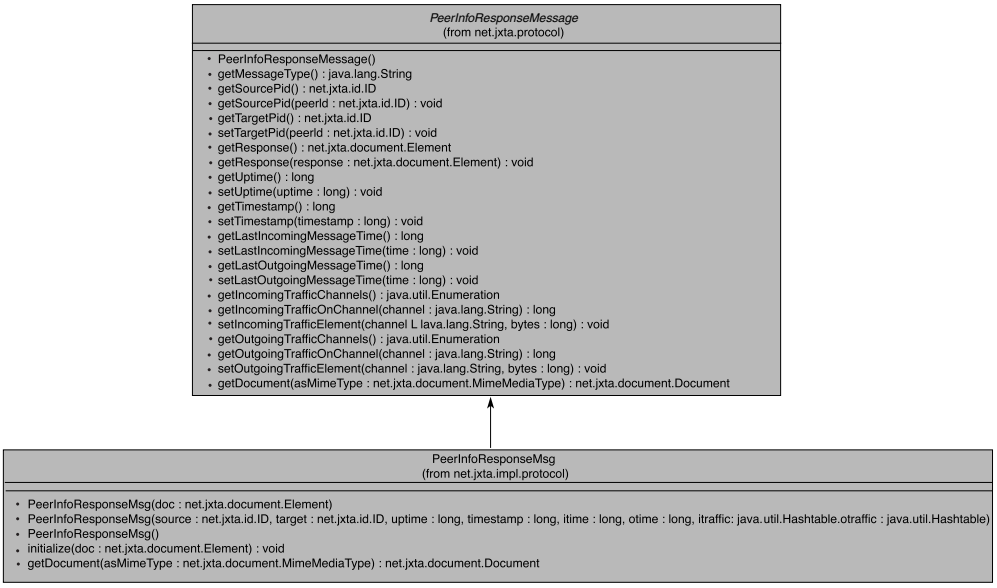

Figure 7.2 The Peer Info Response Message classes.

The Peer Info Service

As with the other protocols, the PIP is encapsulated as a service, as shown in Figure 7.3, freeing the developer from dealing with the details of the Peer Info Query and Response Messages.

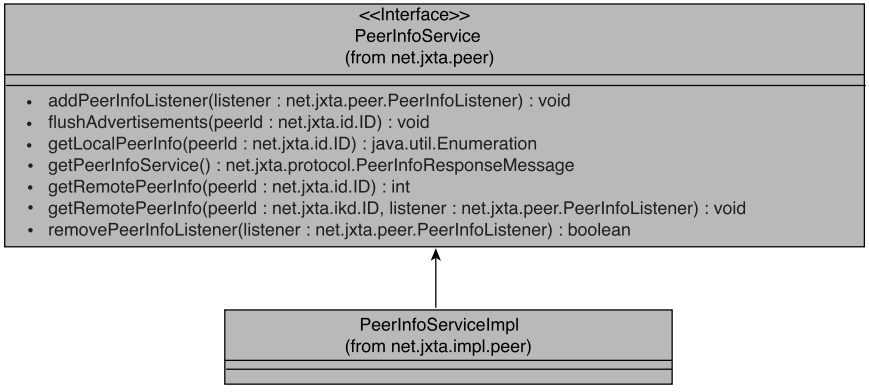

Figure 7.3 The Peer Info service interface and implementation.

The definition of the `PeerInfoService` interface is very similar to the `DiscoveryService` interface, providing methods to retrieve remote and local peer status information. Like the Discovery service, the Peer Info service provides a mechanism to register a listener that will be notified when the Peer Info service receives a Peer Info Response Message.

The reference implementation of the Peer Info service is a `QueryHandler` implementation, whose `processQuery` is responsible for generating a response, if any. Unfortunately, there is no way for the `processQuery` implementation to signal to the Resolver service that a response has been generated and that the original query should not be propagated. If the `processQuery` implementation threw a `DiscardResponseException`, the Resolver service wouldn't propagate the query. However, throwing this exception would prevent `processQuery` from returning a response to be sent to the source of the original query. This incapability to prevent propagation is responsible for the current reference implementation's undesirable propagation of Peer Info Query Messages.

The *PeerInfoListener* Interface

As shown in Figure 7.4, to receive notifications of incoming Peer Info Response Messages, a developer can create and register an implementation of the `PeerInfoListener` interface.

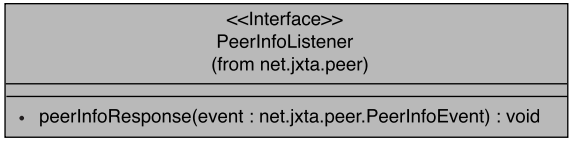

Figure 7.4 The `PeerInfoListener` interface.

Like `DiscoveryListener`, `PeerInfoListener` provides only one method that gets invoked when the Peer Info service receives a Peer Info Response Message. The `peerInfoResponse` method accepts a `PeerInfoEvent` object, shown in Figure 7.5, that can be used by a `PeerInfoListener` implementation to extract the Peer Info Response Message.

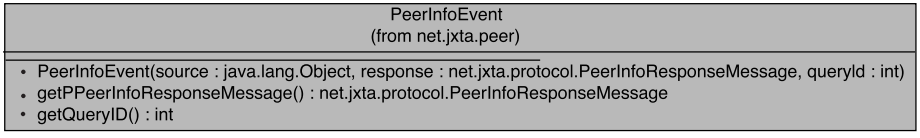

Figure 7.5 The `PeerInfoEvent` class.

The `PeerInfoEvent`'s `getPPeerInfoResponseMessage` returns a `PeerInfoResponseMessage` instance that contains the response received from a remote peer to a Peer Info Query Message.

Using the Peer Info Service

To demonstrate the use of the Peer Info service, you'll implement a simple `PeerInfoListener` and use the peer group's `PeerInfoService` instance to obtain a remote peer's status information.

By this point in the book, you've probably noticed a number of common architectural patterns employed by the JXTA reference implementation. These patterns include the use of listener objects to provide callback functionality, the division of all implementations into an abstract class defining the Java implementation's API, and a concrete class providing the reference implementation. Because of the similarities between the Peer Info and Discovery services, this example skims over some of the basic details that were explained in Chapter 4, "The Peer Discovery Protocol."

Implementing *PeerInfoListener*

Implementing the `PeerInfoListener` interface is very similar to implementing the `DiscoveryListener` interface. A developer needs only to create a class that implements the `peerInfoResponse` method, as shown in Listing 7.4, and register an instance of the implementation with the Peer Info service.

Listing 7.4 **Source Code for *ExampleListener.java***

```
package net.jxta.impl.shell.bin.example7_1;

import java.io.StringWriter;

import java.util.Enumeration;

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

import net.jxta.peer.PeerInfoEvent;
import net.jxta.peer.PeerInfoListener;

import net.jxta.protocol.PeerInfoResponseMessage;


/**
```

continues

Listing 7.4 **Continued**

```
 * A simple implementation of PeerInfoListener to print out details on
 * the received Peer Info Response Messages.
 */
public class ExampleListener implements PeerInfoListener
{
    /**
     * A simple handler that prints out the details on the received
     * peer status information.
     *
     * @param   event the event detailing the received response.
     */
    public void peerInfoResponse(PeerInfoEvent event)
    {
        // Extract the peer info response from the event.
        PeerInfoResponseMessage response =
            event.getPPeerInfoResponseMessage();

        // Print out the peer info.
        System.out.println("Uptime: " + response.getUptime());
        System.out.println("Timestamp: " + response.getTimestamp());
        System.out.println("Target: " + response.getTargetPid());
        System.out.println("Source: " + response.getSourcePid());
        System.out.println("Last Incoming Message: "
            + response.getLastIncomingMessageTime());
        System.out.println("Last Outcoming Message: "
            + response.getLastOutgoingMessageTime());

        // Print out the incoming channel statistics.
        Enumeration incoming = response.getIncomingTrafficChannels();
        if (incoming != null)
        {
            while (incoming.hasMoreElements())
            {
                String incomingchannel = (String) incoming.nextElement();
                long incomingbytes =
                    response.getIncomingTrafficOnChannel(incomingchannel);

                System.out.println(
                    incomingbytes + " incoming bytes on channel "
                        + incomingchannel);
            }
```

```
            }

            // Print out the outgoing channel statistics.
            Enumeration outgoing = response.getOutgoingTrafficChannels();
            if (outgoing != null)
            {
                while (outgoing.hasMoreElements())
                {
                    String outgoingchannel = (String) outgoing.nextElement();
                    long outgoingbytes =
                        response.getOutgoingTrafficOnChannel(outgoingchannel);

                    System.out.println(
                        outgoingbytes + " incoming bytes on channel "
                            + outgoingchannel);
                }
            }

            System.out.println("Done with status info...");
        }
    }
```

Extracting the Peer Info Response Message is as simple as a call to `getPPeerInfoResponseMessage`:

```
PeerInfoResponseMessage peerinfo =
    event.getPPeerInfoResponseMessage();
```

The returned `PeerInfoResponseMessage` can then be used to obtain the `timestamp`, `uptime`, and other Peer Info Response Message elements' contents.

Registering a *PeerInfoListener*

Before a `PeerInfoListener` implementation will begin receiving notification of incoming Peer Info Response Messages, the implementation must be registered with the Peer Info service for a specific peer group. A `PeerInfoListener` is registered using the `PeerInfoService` interface `addPeerInfoListener` method:

```
public void addPeerInfoListener (PeerInfoListener listener)
```

Like the Discovery service, the reference implementation of the Peer Info service is implemented as a Resolver handler. The `PeerInfoServiceImpl` implements the `QueryHandler` interface and handles invoking the registered listeners' `peerInfoResponse` method.

An alternative to registering a handler with the `PeerInfoService` is to pass a `PeerInfoListener` instance when querying remote peers for status information. The `PeerInfoService.getRemotePeerInfo` method accepts a `PeerInfoListener` instance:

```
peerinfo.getRemotePeerInfo(peerIdObject, new ExampleListener());
```

If `getRemotePeerInfo` is called with a `PeerInfoListener` implementation, the given listener object is invoked when replies for the query arrive. In the reference implementation of `PeerInfoService`, `PeerInfoServiceImpl`, the listener is stored to a `Hashtable` using a query ID as the key. The query ID is used in the creation of the Resolver Query Message, and the response sent by a remote peer should use the same query ID. When a Peer Info Response message arrives, wrapped in a Resolver Response Message, the `PeerInfoServiceImpl` extracts the query ID from the `ResolverResponseMsg` and uses it to find a listener in the `Hashtable` with the matching query ID. If a listener is found, its `peerInfoResponse` method is invoked. This is done in addition to invoking the `peerInfoResponse` method of all the listeners registered using `addPeerInfoListener`.

As with the Discovery service, listeners can be removed from the `PeerInfoService` instance. Removing a listener stops it from receiving notification of new incoming Peer Info Response Messages. To remove a listener, a reference to the listener object is required:

```
public boolean removePeerInfoListener ( PeerInfoListener listener);
```

The `removePeerInfoListener` method returns `true` if the `PeerInfoService` has successfully removed the listener. If the method returns `false`, it indicates that the listener object could not be found in the service's set of registered listeners. Unfortunately, listeners that are added to the service by invoking `getRemotePeerInfo` with a listener object cannot be removed using `removePeerInfoListener`.

Finding Remote Peer Information

Using the `ExampleListener` shown in Listing 7.4, it's simple to create a Shell command to send a query to remote peers for peer status information, shown in Listing 7.5.

Listing 7.5 **Source Code for *example7_1.java***

```
package net.jxta.impl.shell.bin.example7_1;

import java.net.URL;

import net.jxta.id.IDFactory;
```

```
import net.jxta.peer.PeerID;
import net.jxta.peer.PeerInfoService;

import net.jxta.peergroup.PeerGroup;

import net.jxta.impl.shell.GetOpt;
import net.jxta.impl.shell.ShellApp;
import net.jxta.impl.shell.ShellEnv;
import net.jxta.impl.shell.ShellObject;


/**

 * A simple Shell command to demonstrate the use of the Peer Info
 * service and the PeerInfoListener interface to query remote peers for
 * status info.
 */
public class example7_1 extends ShellApp
{
    /**
     * The ID of the peer from whom peer info is being solicited.
     */
     private String peerid = null;


    /**
     * Parses the command-line arguments and initializes the command
     *
     * @param        args the arguments to be parsed.
     * @exception    IllegalArgumentException if an invalid parameter
     *                is passed.
     */
    private void parseArguments(String[] args)
        throws IllegalArgumentException
    {
        int option;

        // Parse the arguments to the command.
        GetOpt parser = new GetOpt(args, "p:");

        while ((option = parser.getNextOption()) != -1)
        {
```

continues

Listing 7.5 **Continued**

```
            switch (option)
            {
                case 'p' :
                {
                    // Set the ID of the peer used to retrieve peer info.
                    peerid = parser.getOptionArg();

                    break;
                }
            }
        }
    }

    /**
     * Sends a query to a remote peer.
     *
     * @param   aPeerId the ID of the peer from whom to solicit status info.
     * @param   peerinfo the PeerInfoService to use to perform the query,
     */
    private void sendRemoteRequest(String aPeerId, PeerInfoService peerinfo)
    {
        try
        {
            // Transform the Peer ID string into a Peer ID object.
            PeerID peerIdObject = (PeerID) IDFactory.fromURL(
                new URL((aPeerId)));

            // Use the Peer Info service to query for the peer info.
            peerinfo.getRemotePeerInfo(peerIdObject, new ExampleListener());
        }
        catch (Exception e)
        {
            System.out.println("Error parsing Peer ID string: " + e);
        }
    }

    /**
     * The implementation of the Shell command, invoked when the command
     * is started by the user from the Shell.
```

```
     *
     * @param   args the command-line arguments passed to the command.
     * @return  a status code indicating the success or failure of
     *           the command.
     */
    public int startApp(String[] args)
    {
        String peerid = null;
        int result = appNoError;

        // Get the shell's environment.
        ShellEnv theEnvironment = getEnv();

        // Use the environment to obtain the current peer group.
        ShellObject theShellObject = theEnvironment.get("stdgroup");
        PeerGroup currentGroup = (PeerGroup) theShellObject.getObject();

        // Get the Peer Info service for the current peer group.
        PeerInfoService peerinfo = currentGroup.getPeerInfoService();

        // Default to getting the local peer's status info.
        peerid = currentGroup.getPeerID().toString();

        try
        {
            // Parse the command-line arguments.
            parseArguments(args);
        }
        catch (IllegalArgumentException e)
        {
            println("Incorrect parameters passed to the command.");
            result = ShellApp.appParamError;
        }

        // Send a remote peer info request.
        System.out.println("Running example7_1...");
        sendRemoteRequest(peerid, peerinfo);

        return result;
    }
}
```

The `example7_1` command sends a query to a remote peer using the `PeerInfoService`'s `getRemotePeerInfo` method, passing an instance of the `ExampleListener` class to handle the responses.

It's important to note that `getRemotePeerInfo` requires a real Peer ID to be passed to the method. Although the reference implementation allows `null` to be passed to `getRemotePeer`, the `targetPid` element for the Peer Info Query Message is empty. The reference implementation of Peer Info service that receives the query checks the `targetPid` against the local Peer ID and, because they don't match, does nothing. As a result, a response is never generated in response to the query.

By default, `example7_1` uses the local Peer ID as the `targetPid`, resulting in the status information for the local peer. To retrieve remote peer information, the command must be invoked with a Peer ID, such as in this code:

```
JXTA>example7_1 -purn:jxta:uuid-59616261646162614A787461503250332A2C6697AF84
4127A89E8F30B01CA1C403
```

To use `example7_1` to retrieve the peer information for a specific peer, you first need the ID of the remote peer. Run the `peers` command and choose a peer from which you want to solicit status information. View the peer's advertisement using `cat`:

```
JXTA>cat peer0
```

Find the `PID` element in the Peer Advertisement, and use that value to invoke the `example7_1` command with the `-p` option. Unfortunately, the Shell doesn't currently support paste operations on all platforms, so you can't cut and paste the `PID` value.

An easier way to invoke the command is by using a *Shell script*. A Shell script is any plain text file that contains Shell commands. To run a Shell script, do the following:

```
JXTA>Shell -ftest.txt
```

This example runs the script `test.txt` from the Shell's current directory. In the Shell, the current directory is the Shell subdirectory of the JXTA Demo install directory. To run the `example7_1` command from a script, follow these steps:

1. Create a text file in the Shell subdirectory of the JXTA Demo installation directory. For this example, call the file `test.txt`.
2. Use the right-click pop-up menu in the Shell to copy the `PID` from the remote peer that you want to query.
3. Edit `test.txt`.

4. Add the text **example7_1 –p** to the text file, and then paste the `PID` directly after the `-p` option.
5. Save the `test.txt` file.
6. From the Shell, run the script using this command:

```
JXTA>Shell -ftest.txt
```

This runs the `example7_1` command and queries the peer specified by the `PID` that you entered in the script. When the `ExampleListener` receives responses, the peer information details are printed to the console (not the Shell console) and resemble Listing 7.6.

Listing 7.6 **Example Output from *ExampleListener***

```
Uptime: 7330
Timestamp: 1007439754658
Target: urn:jxta:uuid-59616261646162614A787461503250332A2C6697AF84
4127A89E8F30B01CA1C403
Source: urn:jxta:uuid-59616261646162614A78746150325033AEB5D26090CD4EC683
E18ABE877ABE2703
Last Incoming Message: 0
Last Outcoming Message: 0
Done with status info...
```

Note

As of build 49b of the Java reference implementation of the JXTA platform, the `PeerInfoService` implementation is disabled. This is a result of ongoing work to resolve issues within the implementation. If the examples in this chapter do not produce any output, it is most likely due to this ongoing development work. Consult the `platform.jxta.org` web site for more information on the current status of the PIP implementation.

Finding Cached Peer Information

Just as the Discovery service enables you to query the local cache of advertisements, the Peer Info service provides a mechanism for retrieving cached status information. The example shown in Listing 7.7 provides a simple command for retrieving the locally cached peer information using the Peer Info service.

Listing 7.7 **Source Code for *example7_2.java***

```
package net.jxta.impl.shell.bin.example7_2;

import java.io.IOException;
import java.io.StringWriter;

import java.net.URL;

import java.util.Enumeration;

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

import net.jxta.id.IDFactory;

import net.jxta.peer.PeerID;
import net.jxta.peer.PeerInfoListener;
import net.jxta.peer.PeerInfoService;

import net.jxta.peergroup.PeerGroup;

import net.jxta.protocol.PeerInfoResponseMessage;

import net.jxta.impl.shell.GetOpt;
import net.jxta.impl.shell.ShellApp;
import net.jxta.impl.shell.ShellEnv;
import net.jxta.impl.shell.ShellObject;


/**
 * A simple Shell command to demonstrate the use of the Peer Info
 * service and the PeerInfoListener interface to retrieve locally cached
 * status info.
 */
public class example7_2 extends ShellApp
{
    /**
     * The ID of the peer from which peer info is being solicited.
     */
    private String peerid = null;
```

```
/**
 * Parses the command-line arguments and initializes the command
 *
 * @param        args the arguments to be parsed.
 * @exception    IllegalArgumentException if an invalid parameter
 *               is passed.
 */
private void parseArguments(String[] args)
    throws IllegalArgumentException
{
    int option;

    // Parse the arguments to the command.
    GetOpt parser = new GetOpt(args, "p:l");

    while ((option = parser.getNextOption()) != -1)
    {
        switch (option)
        {
            case 'p' :
            {
                 // Set the ID of the peer used to retrieve peer info.
                 peerid = parser.getOptionArg();

                break;
            }
        }
    }
}

/**
* Retrieves peer information from the local cache.
*
* @param   aPeerId the ID of the peer from which to solicit status info.
* @param   peerinfo the PeerInfoService to use to perform the query,
*/
private void sendLocalRequest(String aPeerId, PeerInfoService peerinfo)
{
    try
    {
        PeerID peerIdObject = (PeerID) IDFactory.fromURL(
            new URL((aPeerId)));
```

continues

Listing 7.7 **Continued**

```
Enumeration enum = peerinfo.getLocalPeerInfo(peerIdObject);

// Iterate through the response messages.
while (enum.hasMoreElements())
{
     // Extract the peer info response from the event.
     PeerInfoResponseMessage response =
          (PeerInfoResponseMessage) enum.nextElement();

     // Print out the peer info.
     System.out.println("Uptime: " + response.getUptime());
     System.out.println("Timestamp: " + response.getTimestamp());
     System.out.println("Target: " + response.getTargetPid());
     System.out.println("Source: " + response.getSourcePid());
     System.out.println("Last Incoming Message: "
          + response.getLastIncomingMessageTime());
     System.out.println("Last Outcoming Message: "
          + response.getLastOutgoingMessageTime());

     // Print out the incoming channel statistics.
     Enumeration incoming =
          response.getIncomingTrafficChannels();
     if (incoming != null)
     {
          while (incoming.hasMoreElements())
          {
               String incomingchannel =
                  (String) incoming.nextElement();
               long incomingbytes =
                    response.getIncomingTrafficOnChannel(
                         incomingchannel);

               System.out.println(incomingbytes
                    + " incoming bytes on channel "
                    + incomingchannel);
          }
     }

     // Print out the outgoing channel statistics.
     Enumeration outgoing =
          response.getOutgoingTrafficChannels();
```

```
            if (outgoing != null)
            {
                while (outgoing.hasMoreElements())
                {
                    String outgoingchannel =
                        (String) outgoing.nextElement();
                    long outgoingbytes =
                        response.getOutgoingTrafficOnChannel(
                            outgoingchannel);

                    System.out.println(outgoingbytes
                        + " incoming bytes on channel "
                        + outgoingchannel);
                }
            }

            System.out.println("Done with status info...");
        }
    }
    catch (IOException e)
    {
        println("Error retrieving local peer info responses!" + e);
    }
}

/**
 * The implementation of the Shell command, invoked when the command
 * is started by the user from the Shell.
 *
 * @param   args the command-line arguments passed to the command.
 * @return  a status code indicating the success or failure of
 *          the command.
 */
public int startApp(String[] args)
{
    String peerid = null;
    int result = appNoError;

    // Get the shell's environment.
    ShellEnv theEnvironment = getEnv();

    // Use the environment to obtain the current peer group.
```

continues

Listing 7.7 **Continued**

```
            ShellObject theShellObject = theEnvironment.get("stdgroup");
            PeerGroup currentGroup = (PeerGroup) theShellObject.getObject();

            // Get the PeerInfo service for the current peer group.
            PeerInfoService peerinfo = currentGroup.getPeerInfoService();

            // Default to getting the local peer's status info.
            peerid = currentGroup.getPeerID().toString();

            try
            {
                // Parse the command-line arguments.
                parseArguments(args);
            }
            catch (IllegalArgumentException e)
            {
                println("Incorrect parameters passed to the command.");
                result = ShellApp.appParamError;
            }

            peerid = null;

            // Send a local peer info request.
            System.out.println("Running example7_2...");
            sendLocalRequest(peerid, peerinfo);

            return result;
        }
    }
```

The `getLocalPeerInfo` method provided by the `PeerInfoService` interface allows a developer to retrieve cached status information. Unlike `getRemotePeerInfo`, the method returns an `Enumeration` of matching `PeerInfoResponseMessages` retrieved from the cache. Because this is a local request, registered implementations of `PeerInfoListener` are never invoked as a result of calling the `getLocalPeerInfo` method.

Unlike `getRemotePeerInfo`, `getLocalPeerInfo` can be passed a `null` Peer ID `String`. Passing `null` to `getLocalPeerInfo` returns all the peer information in the local cache.

To run the example, modify the `test.txt` script to use `example7_2` instead of `example7_1`. When run, the output produced by `example7_2` should be roughly the same as `example7_1`, with the distinction that the information is from the cache.

Summary

In its current state, the PIP isn't especially useful. However, work is under way by the Project JXTA team to augment the PIP to provide a more generic status-monitoring framework. It's not clear what features will emerge from this work. Currently, it appears that the PIP implementation will be augmented to provide a way to handle the content of the `request` element sent in the Peer Info Query Message to a remote peer and generate content for the `response` element in the Peer Info Response Message returned by the remote peer. This work will undoubtedly build on the existing classes, so it has been important in this chapter to understand the current PIP, if only to prepare for the arrival of these additional features.

Now that you've seen how the Peer Information Protocol allows a peer to monitor a remote peer, the next chapter explores another mechanism used to transport data between peers: pipes. The Pipe Binding Protocol described in the next chapter is used to establish pipe connections between peers. After a connection is established, pipes allow peers to send data across a virtual connection, abstracting the network transport layer in a generic fashion.