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!

4

The Peer Discovery Protocol

As described in Chapter 2, "P2P Concepts," advertisements are the basic unit of data exchanged between peers to provide information on available services, peers, peer groups, pipes, and endpoints. With advertisements, the problem of finding peers and all their different types of resources can be reduced to a problem of finding advertisements describing those resources.

The Peer Discovery Protocol (PDP) defines a protocol for requesting advertisements from other peers and responding to other peers' requests for advertisements. This chapter describes the format of the messages of the PDP and tells how to discover advertisements using the Java reference implementation of JXTA.

Introducing the Peer Discovery Protocol

In Chapter 2, you saw that peers discover resources by sending a request to another peer, usually a rendezvous peer, and receiving responses containing advertisements describing the available resources on the P2P network.

The Peer Discovery Protocol consists of only two messages that define the following:

- A request format to use to discover advertisements
- A response format for responding to a discovery request

These two message formats, the Discovery Query Message and the Discovery Response Message, define all the elements required to perform a discovery transaction between two peers, as shown in Figure 4.1.

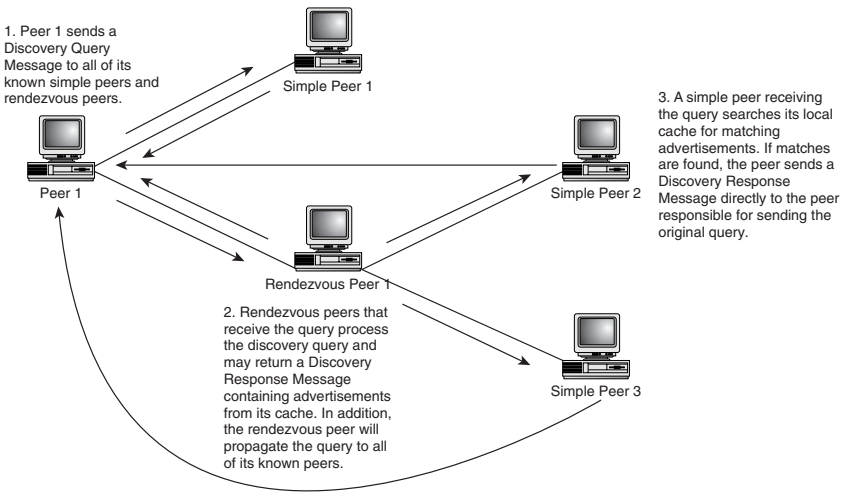

Figure 4.1 Exchange of discovery messages.

Although the messages define a request and a response to that request, it is important to note that a peer might not expect a Discovery Response Message in response to a given Discovery Query Message. A response to a request might not be received for a variety of reasons—for example, the request didn't generate any results, or the request was ignored by an overloaded peer.

The Discovery Query Message

The Discovery Query Message is sent to other peers to find advertisements. It has a simple format, as shown in Listing 4.1.

Listing 4.1 **The Discovery Query Message XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:DiscoveryQuery>
  <Type> . . . </Type>
  <Threshold> . . . </Threshold>
  <PeerAdv> . . .</PeerAdv>
  <Attr> . . . </Attr>
  <Value> . . .</Value>
</jxta:DiscoveryQuery>
```

The root element for the Discovery Query Message is the `jxta:DiscoveryQuery` element. Developers familiar with XML might recognize the `jxta:` prefix in the root element as an XML namespace specifier and wonder if the `jxta` namespace is used or enforced within the Java implementation. Although the `jxta` prefix does specify a namespace, the current Java implementation of JXTA does not understand XML namespaces and treats `jxta:DiscoveryQuery` as the element name rather than recognizing `DiscoveryQuery` as an XML tag from the `jxta` namespace.

The elements of the Discovery Query Message describe the discovery parameters for the query. Only advertisements that match all the requirements described by the query's discovery parameters are returned by a peer. The discovery parameters described by the Discovery Query Message are listed here:

- **`Type`**—A required element containing an integer value specifying the type of advertisement being discovered. A value of `0` represents a query for Peer Advertisements, `1` represents a query for Peer Group Advertisements, and `2` represents a query for any other type of advertisement.
- **`Threshold`**—An optional element containing a number specifying the maximum number of advertisements that should be sent by a peer responding to the query.
- **`PeerAdv`**—An optional element containing the Peer Advertisement for the peer making the discovery query. The Peer Advertisement contains details that uniquely identify the peer on the network to enable another peers to respond to the query.
- **`Attr` and `Value`**—An optional pair of elements that together specify the criteria that an advertisement must fulfill to be returned as a response to this query. `Attr` specifies the name of an element, and `Value` specifies the value that the element must have to be returned as a response to the query.

A couple special exceptions to these rules apply:

- When the `Type` is set to `0` (representing a query for Peer Advertisements) and the threshold is set to `0`, the peer sending the Discovery Query Message is seeking to obtain as many Peer Advertisements as possible. All peers that receive the query should respond to the query with their Peer Advertisement.
- When values for the `Attr` and `Value` elements are absent, each peer responds with a random set of advertisements of the requested `Type`, up to the maximum specified by the `Threshold` element.

In the Java reference implementation, the Discovery Query Message's definition is split into an abstract class definition and a reference implementation provided by Project JXTA, as shown in Figure 4.2. The purpose of this division is to allow third-party developers to maintain API compatibility with the Java reference implementation when providing their own implementation for message parsing and formatting.

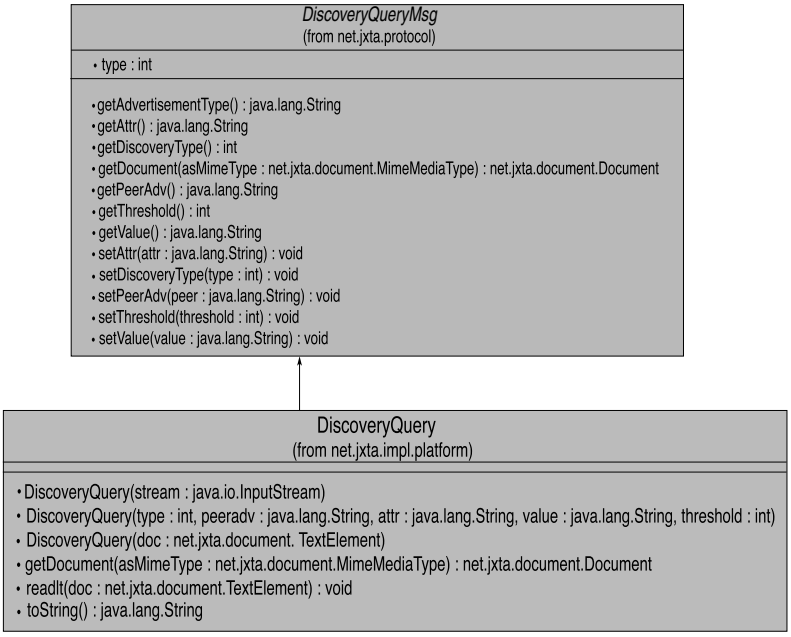

Figure 4.2 The Discovery Query Message classes.

The abstract definition of the Discovery Query Message can be found in the `net.jxta.protocol.DiscoveryQueryMsg` class, and the reference implementation of the abstract class can be found in the `net.jxta.impl.protocol.DiscoveryQuery` class.

Listing 4.2 provides the `shell` command to create a Discovery Query Message using the `DiscoveryQuery` implementation and prints it to the Shell's standard output for examination.

Listing 4.2 **Source Code for *example4_1.java***

```
package net.jxta.impl.shell.bin.example4_1;

import java.io.StringWriter;

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

import net.jxta.discovery.DiscoveryService;

import net.jxta.impl.protocol.DiscoveryQuery;

import net.jxta.impl.shell.ShellApp;


/**
 * A Shell command to create and output a Discovery Query Message.
 */
public class example4_1 extends ShellApp
{
    /**
     * 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)
    {
        int result = appNoError;

        int type = DiscoveryService.PEER;
        String attribute = null;
        String value = null;
        int threshold = 0;
        String advertisementString = "This is my Peer Advertisement";

        // Construct a discovery query message.
```

continues

Listing 4.2 **Continued**

```
            DiscoveryQuery query =
                new DiscoveryQuery(type, advertisementString, attribute,
                    value, threshold);

            // Create an XML formatted string version of the discovery query.
            StringWriter buffer = new StringWriter();
            MimeMediaType mimeType = new MimeMediaType("text/xml");
            // MimeMediaType mimeType = new MimeMediaType("text/plain");
            try
            {
                StructuredTextDocument document =
                    (StructuredTextDocument) query.getDocument(mimeType);
                document.sendToWriter(buffer);
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }

            // Print out the formatted message.
            println(buffer.toString());

            return result;
        }
    }
```

Place the example's code in a file called `example4_1.java` in the `Shell` subdirectory of the JXTA demo installation. Compile the example using this code:

```
javac -d . -classpath ..\lib\beepcore.jar;..\lib\cms.jar;
..\lib\cryptix-asn1.jar;..\lib\cryptix32.jar;..\lib\instantp2p.jar;
..\lib\jxta.jar;..\lib\jxtaptls.jar;..\lib\jxtasecurity.jar;
..\lib\jxtashell.jar;..\lib\log4j.jar;..\lib\minimalBC.jar example4_1.java
```

After the example has compiled, run the Shell application using this code:

```
java -classpath ..\lib\beepcore.jar;..\lib\cms.jar;
..\lib\cryptix-asn1.jar;..\lib\cryptix32.jar;..\lib\instantp2p.jar;
..\lib\jxta.jar;..\lib\jxtaptls.jar;..\lib\jxtasecurity.jar;
..\lib\jxtashell.jar;..\lib\log4j.jar;..\lib\minimalBC.jar;.
 net.jxta.impl.peergroup.Boot
```

When the Shell has loaded, run the example using this command:

```
JXTA>example4_1
```

The `example4_1` command produces the XML-formatted Discovery Query Message containing the parameters for the discovery, shown in Listing 4.3.

Listing 4.3 **Output of the *example4_1* Shell Command**

```
<?xml version="1.0"?>

<!DOCTYPE jxta:DiscoveryQuery>

<jxta:DiscoveryQuery xmlns:jxta="http://jxta.org">
      <Type>
            0
      </Type>
      <Threshold>
            0
      </Threshold>
      <PeerAdv>
            This is my Peer Advertisement
      </PeerAdv>
</jxta:DiscoveryQuery>
```

The `DiscoveryQuery` constructor uses a `String` representation for the Peer Advertisement instead of an object, and the `String` passed to the constructor is used directly in the output. The Discovery Query Message output produced by the example isn't valid because the `PeerAdv` element doesn't actually contain a valid Peer Advertisement. Producing a valid Discovery Query Message using the `DiscoveryQuery` class requires the developer to create a Peer Advertisement object and format it as a `String` in the same manner that the example uses to create a `String` from the `query` object. This `String` then set as the contents of `PeerAdv` element using `DiscoveryQuery`'s `setPeerAdv` method. For now, you'll avoid creating the Peer Advertisement object; we'll focus on it later in this chapter when advertisement instantiation is explored.

The mechanism for formatting the `query` object as a `String` is entirely abstracted through the `net.jxta.document.Document` interface. The `Document` interface defines a generic container for MIME media that can be read from an `InputStream` or written to an `OutputStream`. All advertisement and message objects used by the Java implementation of JXTA use an implementation of the `StructuredTextDocument` interface, derived from the `Document` interface, to provide a representation of the class as a structured MIME text document.

Using the `StructuredTextDocument` interface, the `query` object in the example is written out to XML by providing a `MimeMediaType` object for the `text/xml` MIME type to the `query` object's `getDocument` method. Because the formatting framework is so flexible, the output format could be easily changed to print plain text instead of XML by changing the following line in the example:

```
MimeMediaType mimeType = new MimeMediaType("text/xml");
```

To print plain text, create a `MimeMediaType` object for the `text/plain` MIME type instead of `text/xml` using the following line:

```
MimeMediaType mimeType = new MimeMediaType("text/plain");
```

When this change is in place, recompile the example and restart the Shell application. Running the `example4_1` command this time produces the result shown in Listing 4.4.

Listing 4.4 **Output of the Modified *example4_1* Shell Command**

```
jxta:DiscoveryQuery :
      Type : 0
      Threshold : 0
      PeerAdv : This is my Peer Advertisement
```

The format of the output is determined by the `MimeMediaType` object passed to `getDocument`. The `query` object's `getDocument` method uses this MIME type and the `StructuredDocumentFactory` to produce an implementation of the `StructuredDocument` interface. The available implementations of `StructuredDocument` are defined in the `StructuredDocumentInstanceTypes` property of the `config.properties` property file located in the `net.jxta.impl` package. Currently, only two implementations, `LiteXMLDocument` and `PlainTextDocument`, are available, corresponding to the `text/xml` and `text/plain` MIME types, respectively. The abstraction of message and advertisement formatting means that the Java reference implementation could switch easily from XML to another, possibly binary, format without requiring major changes to the implementation architecture.

The example demonstrates only how to create a Discovery Query Message, not how to send it to other peers to perform the actual discovery. An application developer never actually needs to formulate a Discovery Query Message and send it to other peers themselves; in fact, there is no abstract way of instantiating a `DiscoveryQueryMsg` implementation in the Java reference implementation. The `DiscoveryQueryMsg` is an abstract class defining an interface that `DiscoveryQuery` implements. Although a developer can use the `DiscoveryQuery` implementation directly, this prevents a developer from using another implementation without changing all the code. As you'll see, developers discover advertisements using the Discovery service instead of using the `DiscoveryQueryMsg` class or its implementations directly, thereby abstracting the developer from a particular implementation of `DiscoveryQueryMsg`.

The Discovery Response Message

To reply to a Discovery Query Message, a peer creates a Discovery Response Message that contains advertisements that match the query's search criteria, such as the `Attr/Value` combination or `Type` of advertisement. The Discovery Response Message is formatted as shown in Listing 4.5.

Listing 4.5 **The Discovery Response Message XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:DiscoveryResponse>
  <Type> . . . </Type>
  <Count> . . . </Count>
  <PeerAdv> . . . </PeerAdv>
  <Attr> . . . </Attr>
  <Value> . . . </Value>
  <Response Expiration="expiration time">
    . . .
  </Response>
</jxta:DiscoveryResponse>
```

The elements of the Discovery Response Message closely correspond to those of the Discovery Query Message:

- **`Type`**—Similar to the `Type` element passed in the Discovery Query Message, the `Type` element here is a required element containing an integer value that represents the type of all the advertisements contained within the `Response` elements of the message. As before, a value of `0` represents Peer Advertisements, `1` represents Peer Group Advertisements, and `2` represents all other types of advertisements.
- **`Count`**—An optional element containing an integer representing the total number of `Response` elements in the message.
- **`PeerAdv`**—An optional element containing the Peer Advertisement of the peer responding to the original Discovery Query Message.
- **`Attr` and `Value`**—An optional pair of elements that together specify the original search criteria that generated this response. These have the same value as the `Attr` and `Value` in the Discovery Query Message; if these elements were not present in the original query, they are omitted from the response.

- **`Response`**—An optional element containing an advertisement that matched the search criteria in the Discovery Query Message. Each Discovery Response Message can contain multiple `Response` elements, each containing one advertisement in response to the original query. The total number of `Response` elements equals the value held by the `Count` element. The `Expiration` attribute on the `Response` elements specifies the length of time that this advertisement should be considered valid. In the Java reference implementation, this time is implicitly expressed in milliseconds.

The abstract definition of the Discovery Response Message is defined in the `net.jxta.protocol.DiscoveryResponseMsg` class, shown in Figure 4.3, and the reference implementation is defined in the `net.jxta.impl.protocol.DiscoveryResponse` class.

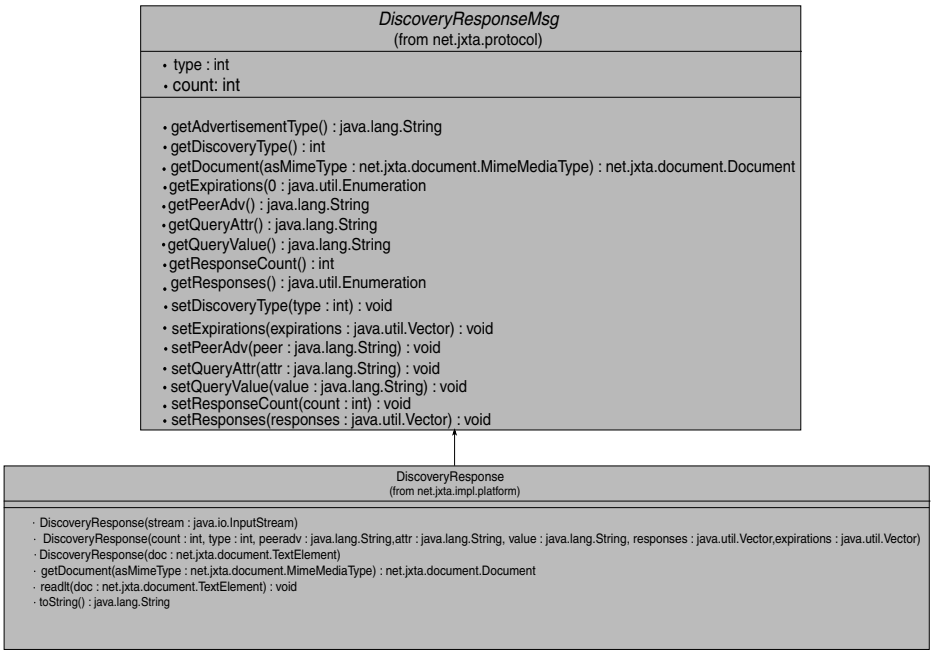

Figure 4.3 The Discovery Response Message classes.

Unlike with `DiscoveryQueryMsg` class, a developer uses the `DiscoveryResponseMsg` class in conjunction with the Discovery service to process responses to queries. The `DiscoveryResponseMsg` class provides developers with an easy mechanism to extract response advertisements; this is demonstrated in the example in the next section.

The Discovery Service

All the protocols defined by the JXTA Protocols Specification are implemented as services called *core services*. The core services include the following:

- Discovery
- Pipe
- Endpoint
- Rendezvous
- Peer Info
- Resolver

An instance of a service is associated with a specific peer group. Only peers that are members of the same peer group are capable of communicating with each other via their services. By default, all peers belong to a common peer group, called *Net Peer Group*, thereby allowing all peers and their advertisements to be discovered.

Services provide developers with a level of abstraction, insulating them somewhat from the raw message objects used to send information between peers. The Discovery service provides a mechanism for the following:

- Retrieving remote advertisements
- Retrieving local advertisements
- Publishing advertisements locally
- Publishing advertisements remotely
- Flushing local advertisements

In the Java reference implementation, the Discovery service is defined by the `DiscoveryService` interface in `net.jxta.discovery` and is implemented by the `DiscoveryServiceImpl` class in net.jxta.impl.discovery, as shown in Figure 4.4.

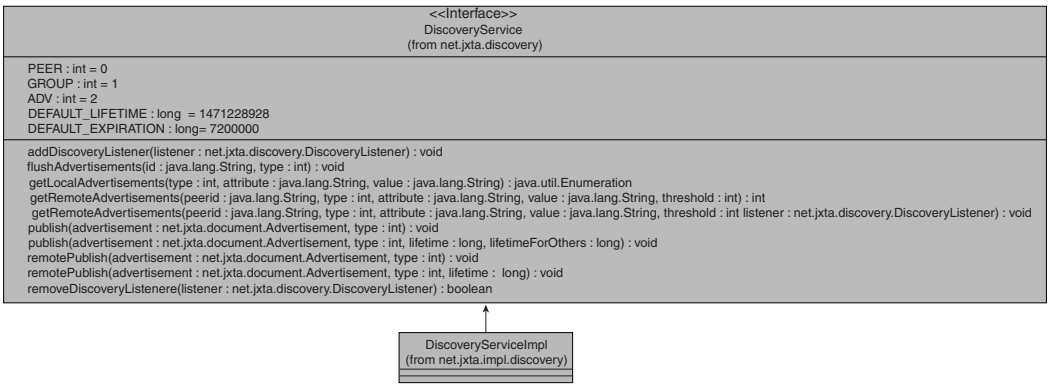

Figure 4.4 The `DiscoveryService` interface and implementation.

The `DiscoveryService` interface provides a simple mechanism for developers to send discovery queries and process discovery responses. A small set of convenience methods allows developers to send Discovery Query Messages without requiring the developer to create and populate a `DiscoveryQuery` object beforehand.

The *DiscoveryListener* Interface

An application requires some way of being notified of responses to a discovery query to allow the application to extract advertisements from the response. In the Java reference implementation, developers can register a *listener* object that will be notified by the `DiscoveryService` when Discovery Response Messages are received.

Java developers are probably most familiar with the concept of a listener from the Java Foundation Classes (JFC). In the JFC, a listener interface is defined for each type of event that can be generated from a user interface widget, such as a button. An object that wants to be informed when a button is clicked implements the appropriate listener interface and registers itself with the button. When the button is clicked, the button widget calls the appropriate method of each listener implementation instance that has registered with the widget.

The Java reference implementation uses a similar mechanism to allow developers to be informed when a new Discovery Response Message is received by the `DiscoveryService`. A developer wanting to be notified of the arrival of a new Discovery Response Message needs to create an implementation of the `DiscoveryListener` interface, as shown in Figure 4.5.

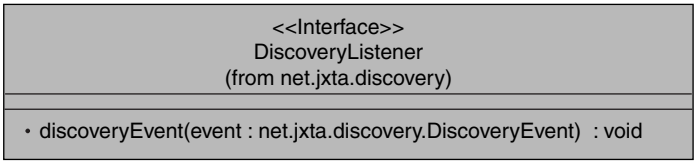

Figure 4.5 The `DiscoveryListener` interface.

To receive notification, the developer registers the implementation of the `DiscoveryListener` interface with an instance of the `DiscoveryService` using the `addDiscoveryListener` method defined in the `net.jxta.discovery.Discovery` interface:

```
public void addDiscoveryListener(
    DiscoveryListener listener);
```

Each time the DiscoveryService instance receives a Discovery Response Message, the listener's discoveryEvent method is called with an event detailing the response received by the service.

To stop receiving notifications, the listener object must be removed from the DiscoveryService using the removeDiscoveryListener method defined in the net.jxta.discovery.Discovery interface:

```
public boolean removeDiscoveryListener(
    DiscoveryListener listener);
```

A reference to the original listener object is required to be capable of removing the listener object from the DiscoveryService instance. The call to the removeDiscoveryListener returns true if the given listener object is removed from the DiscoveryService instance, or false if the listener object isn't currently registered with the DiscoveryService instance.

The *DiscoveryEvent* Class

As shown in Figure 4.6, the DiscoveryEvent defined in net.jxta.discovery is provided to the discoveryEvent method of the DiscoveryListener implementation to provide details on the Discovery Response Message received by a DiscoveryService instance.

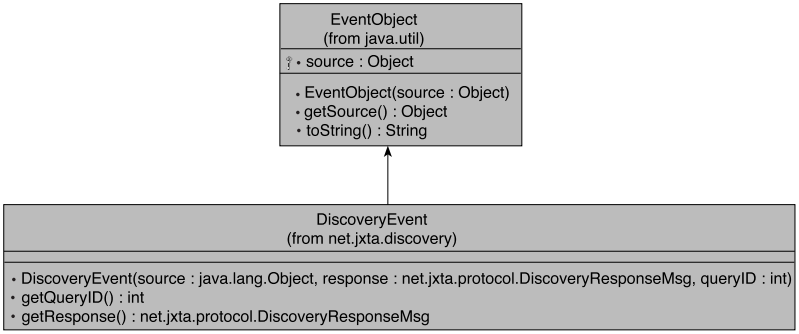

Figure 4.6 The DiscoveryEvent class.

The listener can extract the DiscoveryResponseMsg from the event using the getResponse method of DiscoveryEvent:

```
public DiscoveryResponseMsg getResponse()
```

Use the getResponses method of DiscoveryResponseMsg, as shown in Listing 4.6, to obtain an Enumeration object that can be used to iterate over the advertisements returned in the DiscoveryResponseMsg.

Listing 4.6 **Extracting Responses from a *DiscoveryEvent* Object**

```
public void discoveryEvent(DiscoveryEvent event)
{
    DiscoveryResponseMsg response = event.getResponse();
    Enumeration enum = response.getResponses();

    while (enum.hasMoreElements())
    {
        String advString =
            (String) enum.nextElement();

        // Extract the advertisement from the string here.
    }
}
```

The `DiscoveryResponseMsg` interface also provides the `getExpirations` method, allowing a developer to obtain an `Enumeration` of the expiration times for each of the advertisements returned in the response.

Using *DiscoveryListener* and *DiscoveryEvent*

To try out handling discovery responses, you'll create a `shell` command to handle registering and unregistering your own `DiscoveryListener` implementation. First, you need an implementation of the `DiscoveryListener` interface, as shown in Listing 4.7.

Listing 4.7 **Source Code for *ExampleListener.java***

```
package net.jxta.impl.shell.bin.example4_2;

import java.util.Enumeration;

import net.jxta.document.Advertisement;

import net.jxta.discovery.DiscoveryEvent;
import net.jxta.discovery.DiscoveryListener;

import net.jxta.protocol.DiscoveryResponseMsg;


/**
 * A simple listener to notify the user when a discovery event has
```

```
 * been received.
 */
public class ExampleListener implements DiscoveryListener
{
    /**
     * The DiscoveryListener's event method, used for handling
     * notification of a received Discovery Response Message from
     * the Discovery service.
     *
     * @param   event the event containing the received response.
     */
    public void discoveryEvent(DiscoveryEvent event)
    {
        DiscoveryResponseMsg response = event.getResponse();

        System.out.println("Received a response containing "
            + response.getResponseCount() + " advertisements");
    }
```

For this simple example, you don't need anything fancy—just a notification that a response has been received and details on the number of advertisements contained in the response. Next, you need to create a Shell command called `example4_2` to handle registering and unregistering your listener object. This is shown in Listing 4.8.

Listing 4.8 **Source Code for *example4_2.java***

```
package net.jxta.impl.shell.bin.example4_2;

import net.jxta.discovery.DiscoveryService;
import net.jxta.discovery.DiscoveryListener;

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;


package net.jxta.impl.shell.bin.example4_2;
```

continues

Listing 4.8 **Continued**

```
import net.jxta.discovery.DiscoveryService;
import net.jxta.discovery.DiscoveryListener;

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 example shell application to register or unregister a
 * DiscoveryListener.
 */
public class example4_2 extends ShellApp
{
    /**
     * The shell environment holding the store of environment variables.
     */
    ShellEnv theEnvironment;

    /**
     * A flag indicating whether to add or remove the listener.
     */
    boolean addListener = true;

    /**
     * The name used to store the listener in the environment.
     */
    String name = "Default";


    /**
     * Manages adding or removing the listener.
     *
     * @param   discovery the Discovery service to use to manage
     *           the listener.
     */
    private void manageListener(DiscoveryService discovery)
    {
```

```
    if (name != null)
    {
        // Check if a listener already exists.
        ShellObject theShellObject = theEnvironment.get(name);

        if (addListener)
        {
            if (theShellObject == null)
            {
                // Create a new listener.
                DiscoveryListener listener = new ExampleListener();

                // Add the listener to the discovery service.
                discovery.addDiscoveryListener(listener);

                // Add the listener object to the environment.
                theEnvironment.add(name,
                    new ShellObject(name, listener));
            }
        }
        else
        {
            if (theShellObject != null)
            {
                DiscoveryListener listener =
                    (DiscoveryListener) theShellObject.getObject();

                // Remove the listener from the discovery service.
                discovery.removeDiscoveryListener(listener);

                // Remove the listener object from the environment.
                theEnvironment.remove(name);
            }
        }
    }
}

/**
 * Parses the command-line arguments and initializes the command
 *
 * @param        args the arguments to be parsed.
 * @exception   IllegalArgumentException if an invalid parameter
```

continues

Listing 4.8 **Continued**

```
     *               is passed.
     */
    private void parseArguments(String[] args)
        throws IllegalArgumentException
    {
        int option;

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

        while ((option = parser.getNextOption()) != -1)
        {
            switch (option)
            {
                case 'r' :
                {
                    // Remove the listener.
                    addListener = false;
                    break;
                }

                case 'n' :
                {
                    // Get the name used to store the listener object.
                    String argument= null;

                    if ((argument = parser.getOptionArg()) != null)
                    {
                        name = argument;
                    }

                    break;
                }
            }
        }
    }

    /**
     * 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)
    {
        int result = appNoError;

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

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

        // Get the Discovery service for the current peer group.
        DiscoveryService discovery = currentGroup.getDiscoveryService();

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

        // Manage the listener to the Discovery service. This
        // adds or removes the listener as specified by the
        // command-line parameters.
        manageListener(discovery);

        return result;
    }
}
```

By default, the `example4_2` command creates a listener, adds it to the current peer group's `DiscoveryService`, and stores the listener in a Shell environment variable named `Default`. Storing the listener object is essential; otherwise, the listener can't be removed from the `DiscoveryService` at a later time.

Note

Even if you already configured the Shell in the past, you will be prompted each time you start the Shell to provide your username and password. When trying out the examples, this can become annoying. To avoid having to enter your username and password each time, you can pass in your username and password as system properties to the Java runtime. Use this command to pass in your `username` and `password` as system properties:

```
java -Dnet.jxta.tls.password=password
-Dnet.jxta.tls.principal=username . . .
```

This sets a system property called net.jxta.tls.password to the password value provided after the equals (=) sign and a system property called `net.jxta.tls.principal` to the username provided. When you start the Shell from the command line and include these parameters, the Shell starts immediately without prompting for your username and password.

Place the source code in the `Shell` subdirectory of the JXTA installation and compile it in the same way that you compiled the previous example. Start the Shell from the command line. After the Shell has loaded, clear the local cache of Peer Advertisements using this line:

```
JXTA>peers -f
```

Register an `ExampleListener` instance by running the `example4_2` command:

```
JXTA>example4_2
```

You can check that a `Shell` variable has been created using the variable name `Default` by checking the output of the `env` command. At this point, a `DiscoveryListener` has been registered to be notified when Discovery Response Messages are received by the current peer group's `DiscoveryService`. The code responsible for retrieving the current peer group and the peer group's `DiscoveryService` is shown in Listing 4.9.

Listing 4.9 **Obtaining *DiscoveryService***

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

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

// Get the Discovery service for the current peer group.
DiscoveryService discovery = currentGroup.getDiscoveryService();
```

This code retrieves the current peer group's `PeerGroup` object from the Shell's environment, where it is always stored using the name `stdgroup`. This object obtains a reference to the `DiscoveryService` object that is used by the `manageListener` method to either add or remove the listener.

To see the listener in action, send a discovery query using the `peers -r` command:

```
JXTA>peers -r
```

Every time your peer receives responses to the query, the `ExampleListener` object's `discoveryEvent` method prints the number of advertisements in the response message:

```
Received a response containing 4 advertisements
```

This output appears not in the Shell itself, but in the standard output of the command shell used to start the Shell application. Although you could print the output to the Shell console, it would require delving into the use of pipes, which isn't appropriate at this point.

Instead of sending an active discovery query, try using the `peers` command to retrieve local Peer Advertisements, and observe the behavior of the `ExampleListener` output. You should observe that the `ExampleListener` never receives notification of responses to a local discovery query. The `DiscoveryService` uses the local cache to provide immediate responses to a call to send a local discovery query; therefore, registered listeners never receive a notification of a response to a local discovery query.

The `example4_2` command takes two optional parameters, `-r` and `-n`. The `-r` option indicates to the command that the listener object should be removed from the `DiscoveryService`, and the `-n` option indicates the name of the variable storing the listener instance. For example, issuing the following line attempts to retrieve a `DiscoveryListener` object from an environment variable named `MyListener` and remove the retrieved listener object from the `DiscoveryService` instance:

```
JXTA>example4_2 -r -nMyListener
```

The arguments to the `example4_2` command are parsed easily using the `GetOpt` object in the example:

```
GetOpt parser = new GetOpt(args, "rn:");
```

The second argument to the `GetOpt` constructor, called the *format string*, specifies the command's options and whether the option has any arguments. If a character is followed by the `:` (colon) character, that option requires an

argument; if it is followed by the ; (semicolon), the option has an optional argument. This functionality will be used again in later examples.

At this point, you know how to receive notification of a response to discovery query but not how to send the actual discovery query itself. The next section provides an example of how to send a discovery query to a remote peer using the `DiscoveryService`.

Finding Remote Advertisements

Rather than force developers to create a `DiscoveryQueryMsg` instance themselves, the `DiscoveryService` interface provides an easy way for developers to send a Discovery Query Message to other peers using the `getRemoteAdvertisements` method:

```
public int getRemoteAdvertisements (String peerid,
    int type, String attribute, String value,
    int threshold, DiscoveryListener listener);
```

Each parameter passed to `getRemoteAdvertisements` corresponds to a field in the Discovery Query Message, with the exception of the `peerid` and `listener` parameters. The `peerid` parameter is a parameter that uniquely identifies the peer to query for advertisements; if this parameter is `null`, the message is sent to all peers on the local network and is propagated via available rendezvous peers. More information on identifiers is provided in the section "Working with Advertisements" later in this chapter.

The `listener` parameter provides a `DiscoveryListener` object that is called only when responses arrive in response to this particular call to `getRemoteAdvertisements`. Providing a `listener` object provides a way to receive notification without registering a listener with the `DiscoveryService`. Registered listeners are notified of incoming responses regardless of whether a `null` or non-`null` listener is passed to `getRemoteAdvertisements`.

To try out the `getRemoteAdvertisements` method, the following example `shell` command shown in Listing 4.10 allows a user to send remote queries and specify the desired advertisement type and maximum responses.

Listing 4.10 **Source Code for *example4_3.java***

```
package net.jxta.impl.shell.bin.example4_3;

import java.io.IOException;

import net.jxta.discovery.DiscoveryService;
```

```
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 example shell application to enable a user to send remote
 * discovery queries using the current peer group's Discovery service.
 */
public class example4_3 extends ShellApp
{
    /**
     * The type of advertisement to discover. Defaults to peer
     * advertisements.
     */
    private int type = DiscoveryService.PEER;

    /**
     * The maximum number of responses requested.
     */
    private int threshold = 10;

    /**
     * The Discovery service being used to discover advertisements.
     */
    private DiscoveryService discovery = 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;
```

continues

Listing 4.10 **Continued**

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

        while ((option = parser.getNextOption()) != -1)
        {
            switch (option)
            {
                case 'a' :
                {
                    // Set the type of advertisement to discover.
                    type = Integer.parseInt(parser.getOptionArg());

                    // Validate the type.
                    if ((type < 0) || (type > 2))
                    {
                        // Default to the peer type.
                        type = DiscoveryService.PEER;
                    }

                    break;
                }

                case 't' :
                {
                    String argument = null;

                    if ((argument = parser.getOptionArg()) != null)
                    {
                        // Set the threshold.
                        threshold = Integer.parseInt(argument);
                    }

                    break;
                }
            }
        }
    }

    /**
     * Send a discovery request to remote peers via the Discovery service.
     *
```

```
 * @param   type the type of advertisement to discover.
 * @param   threshold the maximum number of advertisements to be
 *           returned by any single peer.
 */
private void sendRemoteDiscovery(int type, int threshold)
{
    discovery.getRemoteAdvertisements(null, type, null, null,
        threshold, null);
}

/**
 * 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)
{
    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 Discovery service for the current peer group.
    discovery = currentGroup.getDiscoveryService();

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

continues

Listing 4.10 **Continued**

```
            // Send a remote discovery request.
            sendRemoteDiscovery(type, threshold);

            return result;
        }
    }
```

The example is essentially a replacement for the `peers -r` command. When run in conjunction with `example4_2`, it allows a user to send queries and be notified when responses arrive.

To see the `example4_3` command in action, first register a listener using the `example4_2` command:

```
JXTA>example4_2
```

Then send a Discovery Query Message that searches for Peer Advertisements, with a maximum of 10 responses from any given peer:

```
JXTA>example4_3
```

The peer sends a Discovery Query Message to all known peers requesting a response containing matching advertisements. The `ExampleListener` registered using the `example4_2` command prints information each time that a response to this query is received by the `DiscoveryService` instance.

Finding Cached Advertisements

In the Java reference implementation, advertisements in responses to a Discovery Query Message are automatically added to a local cache of advertisements. `DiscoveryListener` implementations don't have to provide caching functionality themselves.

To find advertisements using the local cache, a developer can use the `getLocalAdvertisements` method of the `DiscoveryService` interface. Unlike performing an active discovery to find advertisements on remote peers, performing discovery using the local cache returns results immediately and does not require an implementation of the `DiscoveryListener` interface.

To see how local discovery works, Listing 4.11 shows another example Shell command that replaces some of the functionality of the `peers` command.

Listing 4.11 **Source Code for *example4_4.java***

```
package net.jxta.impl.shell.bin.example4_4;

import java.io.IOException;

import java.util.Enumeration;

import net.jxta.discovery.DiscoveryService;

import net.jxta.document.Advertisement;

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 example shell application to enable a user to send local
 * discovery queries using the current peer group's Discovery service.
 */
public class example4_4 extends ShellApp
{
    /**
     * The type of advertisement to discover. Defaults to
     * peer advertisements.
     */
    private int type = DiscoveryService.PEER;

    /**
     * The Discovery service being used to discover advertisements.
     */
    private DiscoveryService discovery = null;

    /**
     * The name of the element to match.
     */
    private String attribute = null;
```

continues

Listing 4.11 **Continued**

```
/**
 * The value to match for the element specified by the attribute
 * variable.
 */
private String value = 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, "a:k:v:");

    while ((option = parser.getNextOption()) != -1)
    {
        switch (option)
        {
            case 'a' :
            {
                // Set the type of advertisement to discover.
                type = Integer.parseInt(parser.getOptionArg());

                // Validate the type.
                if ((type < 0) || (type > 2))
                {
                    // Default to the peer type.
                    type = DiscoveryService.PEER;
                }

                break;
            }
```

```
            case 'k' :
            {
                // Set the attribute to match.
                attribute = parser.getOptionArg();

                break;
            }

            case 'v' :
            {
                // Set the value for the attribute being matched.
                value = parser.getOptionArg();

                break;
            }
        }
    }

    // Both attribute and value must be specified.
    if (!((null != attribute) && (null != value)))
    {
        // Set both to null.
        attribute = null;
        value = null;
    }
}

/**
 * Sends a local discovery request using the Discovery service.
 */
private void sendLocalDiscovery()
{
    try
    {
        int count = 0;
        Enumeration enum =
            discovery.getLocalAdvertisements(type, attribute, value);
        Advertisement advertisement;

        // Iterate through the response advertisements.
        while (enum.hasMoreElements())
        {
```

continues

Listing 4.11 **Continued**

```
                // Get the next element from the enumeration.
                advertisement = (Advertisement) enum.nextElement();

                println("Found a matching advertisement!");

                // Increment the counter.
                count++;
            }

            println("Found " + count + " advertisements!");
        }
        catch (IOException e)
        {
            println("Error discovering local advertisements!" + 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)
    {
        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 Discovery service for the current peer group.
        discovery = currentGroup.getDiscoveryService();

        try
        {
```

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

        // Send a local discovery request.
        sendLocalDiscovery();

        return result;
    }
```

Instead of using a `DiscoveryListener` implementation to handle advertisements returned in a `DiscoveryResponseMsg` response, `getLocalAdvertisements` returns an `Enumeration` of advertisements immediately that match the query parameters provided.

This example allows the user to provide an attribute and value to match, allowing the user to search for an advertisement that matches specific criteria. For example, assume that the `peers` command returns the following:

```
peer0: name = Cadillac
peer1: name = spec
peer2: name = spiro
peer3: name = zynevich
```

The `example4_4` command could be used to search the local cache for any Peer Advertisement in which the peer has a given name. The name of a peer is described by the `Name` element in its advertisement and is displayed as the `name` in the list returned by the `peers` command. To discover Peer Advertisements in which the `Name` is spec using the `example4_4` command, do the following:

```
JXTA>example4_4 -a0 -kName -vspec
Found a matching advertisement!
Found 1 advertisements!
```

The `-a` option specifies the advertisement type to discover, and the `-k` and `-v` options together specify a tag and value that an advertisement must contain to be part of a peer's response to the query. Discovering an advertisement that matches a given tag and value combination can even use a wildcard in the

value string. Discovering Peer Advertisements with a tag called `Name` whose value starts with the letter *s* could be accomplished as follows:

```
JXTA>example4_4 -a0 -kName -vs*
Found a matching advertisement!
Found 2 advertisements!
```

The wildcard symbol, `*`, can be used anywhere within the value term; however, the wildcard symbol can't be used by itself, and the value to be matched must consist of at least one nonwildcard character. Wildcards can even be used in multiple places in the search string:

```
JXTA>example4_4 -a0 -kName -v*ill*
Found a matching advertisement!
Found 1 advertisements!
```

The Cache Manager

The local cache, implemented by the Cache Manager class `Cm` in the `net.jxta.impl.cm` package, handles storing discovered advertisements in a local file and directory structure. The Cache Manager is responsible not only for providing search capabilities for local discovery requests, but also for finding advertisements that match Discovery Query Messages sent by other peers. The Cache Manager stores cached advertisements in a directory called `cm` under the current directory when the JXTA application is executed.

Flushing Advertisements

At some point, an application might need to clear the entire cache; this might be required when an application has not been used in a long time and all advertisements are suspected to be stale. As shown in Listing 4.12, the `DiscoveryService` provides a simple mechanism to allow an application to clear the cache of specific advertisement types that match a given type of advertisement and identifier string.

Listing 4.12 **Source Code for *example4_5.java***

```
package net.jxta.impl.shell.bin.example4_5;

import java.io.IOException;

import net.jxta.discovery.DiscoveryService;

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 example shell application to enable a user to flush
 * discovered advertisements from the local cache using the current peer
 * group's Discovery service.
 */
public class example4_5 extends ShellApp
{
    /**
     * The type of advertisement to flush. Defaults to peer advertisements.
     */
    private int type = DiscoveryService.PEER;

    /**
     * The ID of the advertisement to flush. Defaults to null.
     */
    private String id = 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, "a:i:");

        while ((option = parser.getNextOption()) != -1)
        {
            switch (option)
            {
                case 'a' :
```

continues

Listing 4.12 **Continued**

```
                {
                    // Set the type of advertisement to flush.
                    type = Integer.parseInt(parser.getOptionArg());

                    // Validate the type.
                    if ((type < 0) || (type > 2))
                    {
                        // Default to the peer type.
                        type = DiscoveryService.PEER;
                    }

                    break;
                }

                case 'i' :
                {
                    // Set the ID string of the advertisement to flush.
                    id = parser.getOptionArg();

                    break;
                }
            }
        }
    }

    /**
     * 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)
    {
        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 DiscoveryService service for the current peer group.
            DiscoveryService discovery = currentGroup.getDiscoveryService();

            try
            {
                // Parse the command-line arguments.
                parseArguments(args);

                // Flush all of the advertisements of the given type and ID.
                discovery.flushAdvertisements(id, type);
            }
            catch (IllegalArgumentException e)
            {
                println("Incorrect parameters passed to the command.");
                result = ShellApp.appParamError;
            }
            catch (IOException e)
            {
                println("Error flushing advertisements: " + e);
                result = appMiscError;
            }

            return result;
        }
    }
```

To remove all the Peer Advertisements from the local cache using the `example4_5` command, type the following:

```
JXTA>example4_5 -a0
```

Invoking the `peers` command after this command should return an empty list.

To remove only a specific advertisement, the unique identifier for the advertisement is required; in the case of a Peer Advertisement, this identifier is given by the `PID` element. The `PID` can be obtained using the `cat` command to view a Peer Advertisement stored in the Shell's environment variables, as demonstrated in Chapter 3, "Introducing JXTA P2P Solutions." For example, imagine that the Peer Advertisement to be flushed from the cache has this ID:

```
urn:jxta:uuid-596162616461626l4A78746150
32503323EC5B06B634476AB7418CB18BA45DCA03
```

It can be removed from the cache using this command:

```
JXTA> example4_5 -a0 -iurn:jxta:uuid-596162616461626l4A7874615032503
323EC5B06B634476AB7418CB18BA45DCA03
```

Executing this command removes the advertisement from the cache by deleting its corresponding advertisement from within the Cache Manager's `cm` directory.

Working with Advertisements

At this point, this chapter has only really discussed advertisements in generic terms. You have not delved into the specifics of what information is contained by a particular advertisement or how advertisements are used within the Java reference implementation. Although you know how to discover advertisements, how do you use them?

All advertisements in the Java reference implementation extend the `net.jxta.document.Advertisement` abstract class. The `Advertisement` class defines several methods, the most important being the `getDocument` method for transforming an `Advertisement` into a `Document` instance corresponding to a particular MIME type. As shown in Figure 4.7, each type of advertisement is split into an abstract class in the `net.jxta.protocol` package and an implementation class in `net.jxta.impl.protocol`.

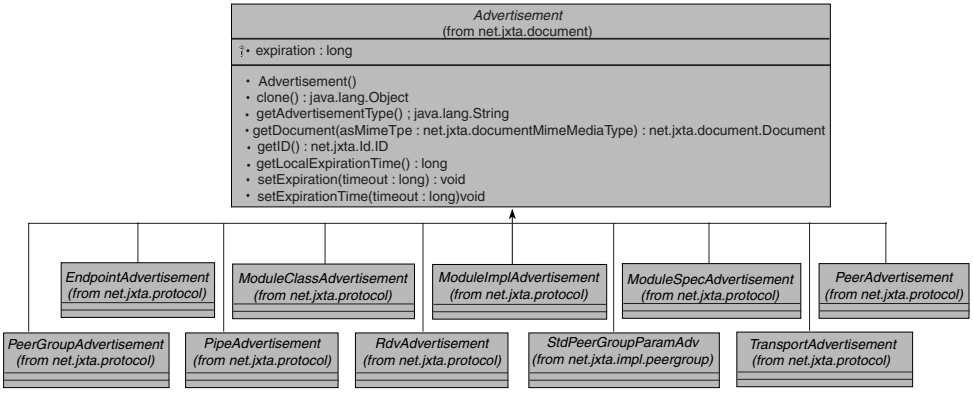

Figure 4.7 The `Advertisement` abstract implementation classes.

The `net.jxta.protocol` abstract classes augment the `Advertisement` class with attributes specific to the type of advertisement and accessor methods to set and retrieve the value of those fields. The `net.jxta.impl.protocol` implementation classes provide the implementation of the `getDocument` method.

Instantiating an Advertisement

To insulate a developer from knowing about a specific advertisement implementation class, advertisements are instantiated using the `AdvertisementFactory` class in the `net.jxta.document` package. The simplest way to create an advertisement instance is to use the factory's static `newAdvertisement` method, providing a `String` containing the type of advertisement to create.

An advertisement type in the Java reference implementation is a `String` containing the root element of the advertisement that it is associated with. Although developers could construct the advertisement type `String` themselves, it is easier to use the static `getAdvertisementType` defined by the `Advertisement` class. For example, a `PeerAdvertisement` could be instantiated using either

```
PeerAdvertisement advertisement =
    (PeerAdvertisement)
        AdvertisementFactory.newAdvertisement(
            "jxta:PA");
```

or

```
PeerAdvertisement advertisement =
    (PeerAdvertisement)
        AdvertisementFactory.newAdvertisement(
            PeerAdvertisement.getAdvertisementType());
```

Each of the `net.jxta.protocol` subclasses of `Advertisement` provides an implementation of `getAdvertisementType` that allows a developer to get a specific type of advertisement without knowing which concrete class is providing the implementation.

Publishing Advertisements

Constructing an advertisement by itself doesn't make the advertisement known to either the local peer or any other peer on the network. For an advertisement to be available on the P2P network, it needs to be published locally, remotely, or both.

Publishing an advertisement locally places the advertisement in the local peer's cache of advertisements; other peers can find this advertisement using a standard Discovery Query Message. The `DiscoveryService` interface provides a simple mechanism for publishing the advertisement to the local cache using either

```
public void publish(Advertisement advertisement,
    int type) throws IOException;
```

or

```
public void publish (Advertisement adv, int type,
    long lifetime, long lifetimeForOthers)
        throws IOException;
```

The second version of the `publish` method is more explicit, allowing the caller to specify not only the advertisement and its type, but also the length of time that the advertisement will remain in the local cache and the length of time that the advertisement will be available to be discovered by other peers. The length of time in both cases is expressed in milliseconds, and the type of advertisement corresponds to the values used by the Discovery Query and Response Messages (`0` = peer, `1` = peer group, `2` = other advertisements).

The first version of the `publish` method publishes an advertisement to the local cache using default values for the local and remote lifetimes of the advertisement. The default local lifetime is one year, and the default lifetime for other peers is two hours.

To help accelerate the process of distributing an advertisement within the membership of a peer group, an advertisement can be published remotely. Publishing an advertisement remotely broadcasts the advertisement directly to other known peers or indirectly via known rendezvous peers to other members of the peer group associated with the `DiscoveryService` service instance. This broadcast uses a Discovery Response Message to push the advertisement to peers.

The `DiscoveryService` interface provides two methods, similar to the `publish` methods, to publish an advertisement to a remote peer. An advertisement can be remotely published using either

```
public void remotePublish (
    Advertisement adv, int type);
```

or

```
public void remotePublish (Advertisement adv, int
    type, long lifetime);
```

Although the documentation in the `DiscoveryService` interface specifies that the type can be set to indicate a peer, peer group, or other type of advertisement, the current implementation does not remotely publish Peer Advertisements. However, the reference implementation of `DiscoveryService`, `DiscoveryServiceImpl`, automatically adds the Peer Advertisement contained in any Discovery Query Messages that it receives, providing the same functionality.

To demonstrate the use of the `publish` and `remotePublish` methods, the `shell` command in Listing 4.13 creates a Peer Group Advertisement using the current peer group as a template, and publishes the advertisement locally and remotely.

Listing 4.13 **Source Code for *example4_6.java***

```
package net.jxta.impl.shell.bin.example4_6;

import java.io.IOException;

import net.jxta.discovery.DiscoveryService;

import net.jxta.document.AdvertisementFactory;

import net.jxta.id.IDFactory;

import net.jxta.peergroup.PeerGroup;

import net.jxta.protocol.PeerGroupAdvertisement;

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


/**
 * A simple example shell application to publish a peer group
 * advertisement based on the Shell's current peer group.
 */
public class example4_6 extends ShellApp
{
    /**
     * 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)
    {
        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();
```

continues

Listing 4.13 **Continued**

```
        // Get the Discovery service for the current peer group.
        DiscoveryService discovery = currentGroup.getDiscoveryService();

        try
        {
            // Create an advertisement.
            PeerGroupAdvertisement advertisement =
                (PeerGroupAdvertisement)
                    AdvertisementFactory.newAdvertisement(
                        PeerGroupAdvertisement.getAdvertisementType());

            // Populate the various fields. For most of this, we'll create
            // our own values, but we'll need the Module Spec ID of our
            // current peer group.
            PeerGroupAdvertisement currentAdvertisement =
                currentGroup.getPeerGroupAdvertisement();

            // Set the values that must be unique for the new advertisement.
            advertisement.setName("NewGroup");
            advertisement.setDescription("PG for example4_6");
            advertisement.setPeerGroupID(IDFactory.newPeerGroupID());
            advertisement.setModuleSpecID(
                currentAdvertisement.getModuleSpecID());

            // Publish the advertisement locally.
            discovery.publish(advertisement, DiscoveryService.GROUP,
                10000, 1000);

            // Publish the advertisement remotely.
            discovery.remotePublish(advertisement,
                DiscoveryService.GROUP, 1000);
        }
        catch (IOException e)
        {
            println("Error publishing the advertisement to cache." + e);
            result = ShellApp.appMiscError;
        }

        return result;
    }
}
```

Not all the values for the newly created Peer Group Advertisement are exact copies; most important, the identifier for the peer group must be a new, unique ID. Creating a new ID is achieved using the `net.jxta.id.IDFactory` to create a new Peer Group ID:

```
advertisement.setPeerGroupID(IDFactory.newPeerGroupID());
```

The `IDFactory` class generates a unique identifier for a variety of advertisements that require a unique identifier, including peers, peer groups, pipes, and services.

One other ID that is added to the new Peer Group Advertisement is a Module Specification ID:

```
advertisement.setModuleSpecID(currentAdvertisement.getModuleSpecID());
```

This ID uniquely identifies a Module Specification Advertisement, which defines the set of services provided by the peer group. For this example, you simply copy the value, thereby associating your Peer Group Advertisement with the same Module Specification Advertisement as the current group.

When you explore services and peer groups in Chapter 10, "Peer Groups and Services," you learn how to create a new Module Specification Advertisement and use it to create a new peer group and start the group's services. It's important to note that this example only publishes the new peer group's advertisement but does not actually start the new peer group's services.

To try out the `example4_6` command, start the Shell and flush the cache of Peer Group Advertisements:

```
JXTA>groups -f
```

After flushing the cached Peer Group Advertisements, executing the `groups` command again should result in an empty list. Implicitly, the peer is still aware of the default `NetPeerGroup`, and executing the `example4_6` command clones that group's advertisement and publishes the resulting advertisement both locally and remotely:

```
JXTA>example4_6
```

Another call to groups should display the newly published group:

```
JXTA>groups
group0: name = NewGroup
```

The `example4_6` command sets the local lifetime to 10 seconds (10,000 milliseconds) when it publishes the advertisement locally:

```
discovery.publish(advertisement, Discovery.GROUP,
    10000, 1000);
```

After 10 seconds, the Cache Manager clears the advertisement from the cache. Executing the `groups` command again returns an empty list, as expected.

Summary

This chapter demonstrated how the JXTA platform manages peer discovery and how the Java reference implementation provides a developer with the capability to send Discovery Query Messages to other peers and process the Discovery Responses Messages sent in response to queries.

In addition to performing discovery, the `DiscoveryService` interface and the implementation provided by the Java reference implementation can be used to publish advertisements to both the local cache and remote peers.

In the next chapter, you explore the Peer Resolver Protocol and the Resolver service. The Peer Resolver Protocol allows a peer to process and respond to generic queries. As you'll see, the Peer Resolver Protocol and the Resolver service provide the Discovery service with the capability to send queries to remote peers, process queries from other peers, and send responses to queries.