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!

10

Peer Groups and Services

The concept of a peer group is central to all aspects of the JXTA platform. Peer groups provide a way of segmenting the P2P network space into distinct communities of peers organized for a specific purpose. All the core protocols explored in this book so far have depended on a peer group to provide the context for operations performed by services. Before creating a sample application, it is necessary to understand how JXTA peer groups are created and used to gain access to services.

This chapter provides the information you need to use peer groups and to understand how to create, join, or leave peer groups. Part of this coverage includes how JXTA allows peer groups to create private peer groups that are accessible to only authorized peers. In addition to the coverage of peer group semantics, the chapter explores how to create new services and create a peer group that can use these services.

Modules, Services, and Applications

Before diving into the specifics of working with peer groups, it is essential to understand the module framework employed by JXTA. The module framework is designed to allow a developer to provide functionality within JXTA in an extensible manner. Modules managed by the framework are responsible for providing all aspects of JXTA's functionality, including the implementation of the peer group mechanism as well as services and applications that are provided by a peer group.

Understanding modules is essential to being able to write new services for JXTA. To understand how these peer groups, services, and applications are specified, you first need to understand the JXTA concept of a module. Simply put, a *module* is some distributable chunk of functionality that a peer can initialize, start, and stop. In addition to a plain module, JXTA provides the concept of a *service module*, a component used by a peer to run a service, and an *application module*, a component used by a peer to run an application. An application module is different from an application that invokes JXTA. An application that invokes JXTA can be comprised of several service and application modules.

To enable peers to discover modules, the definition of a module is divided into three types of advertisements: a Module Class Advertisement, a Module Specification Advertisement, and a Module Implementation Advertisement. Before diving into the specifics of each type of advertisement, it's important to understand how they are related.

Consider some of the problems inherent in creating a framework for modules in JXTA:

- Because JXTA is supposed to be language/platform agnostic, the module framework needs to support multiple implementations of a given module. For example, the Discovery service module might be implemented as a Java class or as a C++ COM class. Therefore, the framework needs to be capable of distinguishing among these module implementations, possibly using some external metadata representation, such as an advertisement.
- The capabilities of the module will invariably change over time. The person or organization responsible for defining the module might want to add or remove functionality, thus changing the specification of the module. For example, the Peer Discovery Protocol specification might change over time to add new search capabilities. Therefore, the module framework must be capable of distinguishing among various versions of a module, again using some metadata representation. In addition,

each version of the module's specification might have multiple implementations, necessitating some link between the metadata describing a module implementation and the metadata describing the module's specification.

- There must be some way of referring to a module that provides a class of functionality independent of a particular specification or implementation of the module. For example, the JXTA Discovery service module is a class of module that provides discovery services. Again, there must be some relationship between the metadata describing a module specification and the metadata describing the module's class.

Each of these aspects is encapsulated by the Module Implementation, Module Specification, and Module Class Advertisements, respectively. The discussion of these advertisements in the following sections starts from the most general advertisement describing a module, the Module Class Advertisement.

The Module Class Advertisement

The first advertisement, the Module Class Advertisement, doesn't provide information on a module implementation; it exists solely to announce the existence of a class of module. The Module Class Advertisement provides only the few pieces of information shown in Listing 10.1.

Listing 10.1 **The Module Class Advertisement XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:MCA>
  <MCID> . . . </MCID>
  <Name> . . . </Name>
  <Desc> . . . </Desc>
</jxta:MCA>
```

These pieces of information can be used by a peer to search for a module based on one of the advertisement's elements:

- **`MCID`**—A required element containing a Module Class ID. This ID uniquely identifies a class of modules. The Module Class ID is used as the basis for the IDs contained in the Module Specification and Implementation Advertisements.
- **`Name`**—An optional element containing a simple name for the module class. This string is not necessarily unique.
- **`Desc`**—An optional element containing a description of the module class.

As with all other advertisement types in the reference implementation, the implementation of the Module Class Advertisement is split into the abstract definition `ModuleClassAdvertisement`, defined in `net.jxta.protocol`, and the reference implementation `ModuleClassAdv`, defined in `net.jxta.impl.protocol`. These classes are shown in Figure 10.1.

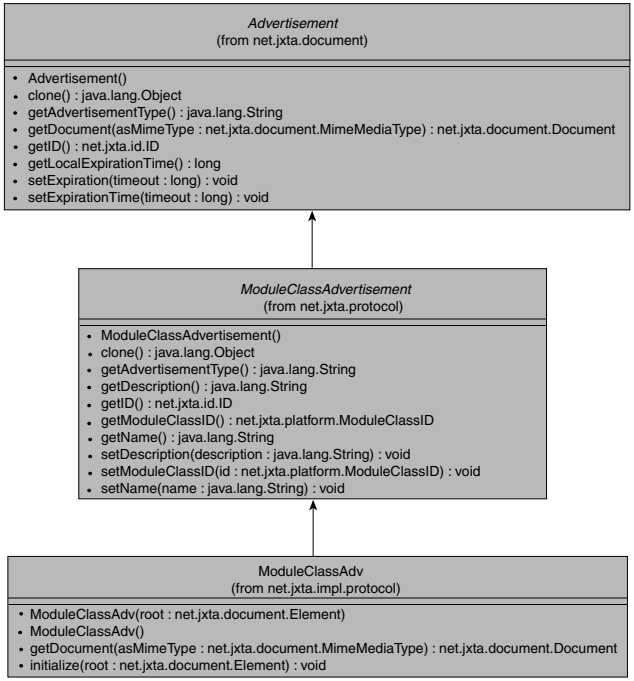

Figure 10.1 The Module Class Advertisement classes.

Each class of modules has a unique Module Class Advertisement. For example, the reference implementation of the Discovery service is associated with a Module Class Advertisement that defines a class of modules responsible for providing discovery capabilities. Modules that provide this capability also use this same Module Class Advertisement. A different class of module, such as a module that provides routing capabilities, is associated with a different Module Class Advertisement.

The Module Specification Advertisement

The second advertisement responsible for defining a module is the Module Specification Advertisement. The purpose of a Module Specification Advertisement, shown in Listing 10.2, is to uniquely identify a set of protocol-compatible modules.

Listing 10.2 **The Module Specification Advertisement XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:MSA>
  <MSID> . . . </MSID>
  <Name> . . . </Name>
  <Crtr> . . . </Crtr>
  <SURI> . . . </SURI>
  <Vers> . . . </Vers>
  <Desc> . . . </Desc>
  <Parm> . . . </Parm>
  <jxta:PipeAdvertisement> . . . </jxta:PipeAdvertisement>
  <Proxy> . . . </Proxy>
  <Auth> . . . </Auth>
</jxta:MSA>
```

The advertisement provides metadata on a version of the module's specification using the following elements:

- **`MSID`**—A required element containing a Module Specification ID that uniquely identifies the module specification. The Module Specification ID includes within it the Module Class ID identifying the class of module to which this specification belongs.
- **`Name`**—An optional element containing a simple name for the module specification. This string is not necessarily unique.
- **`Crtr`**—An optional element containing the name of the creator of the module specification.
- **`SURI`**—An optional element containing a URI that points to a specification document that describes the purpose and protocol, if any, defined by the module.
- **`Vers`**—A required element containing information on the specification version embodied by this Module Specification Advertisement.
- **`Desc`**—An optional element containing a description of the module specification.
- **`Parm`**—An optional element containing parameters for the specification. The format and meaning of these parameters is defined by the module's specification.
- **`jxta:PipeAdvertisement`**—An optional element containing a Pipe Advertisement describing a pipe that can be used to send data to the module. This element is actually the root element of the Pipe Advertisement, not an element that contains a Pipe Advertisement. The

module that implements this specification binds an input pipe to the pipe identified by the Pipe Advertisement, allowing third parties to communicate with the module.

- **`Proxy`**—An optional element containing the Module Specification ID of a module that can be used to proxy communication with a module defined by this module specification. This is not really used in the reference implementation, and its use in modules is discouraged.
- **`Auth`**—An optional element containing the Module Specification ID of a module that provides authentication services for a module defined by this module specification.

The implementation of the Module Specification Advertisement is split into the abstract definition `ModuleSpecAdvertisement`, defined in `net.jxta.protocol`, and the reference implementation `ModuleSpecAdv`, defined in `net.jxta.impl.protocol`. These classes are shown in Figure 10.2.

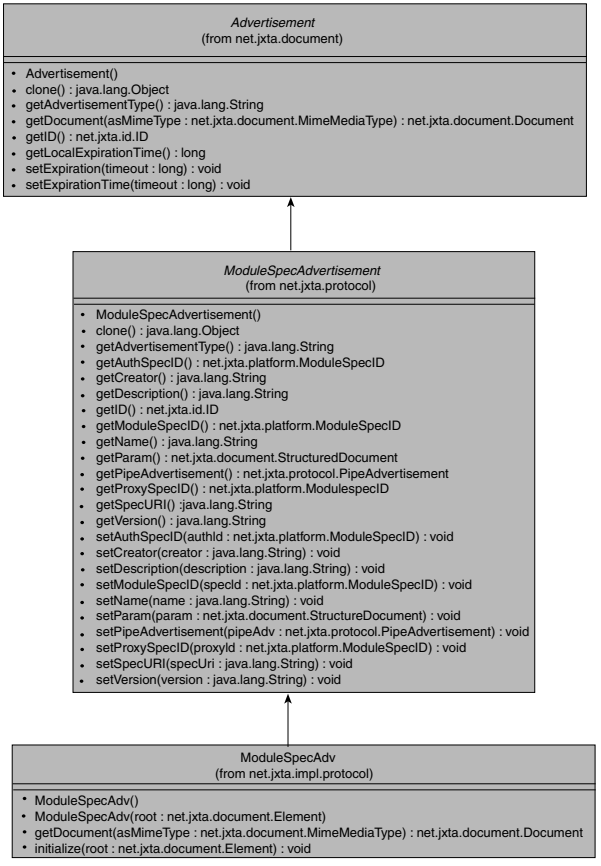

Figure 10.2 The Module Specification Advertisement classes.

For every Module Class Advertisement, there can be one or more different Module Specification Advertisements, each specifying a different version of the module. For example, if a new version of the Peer Discovery Protocol is released, the module responsible for implementing the PDP in the reference implementation will be associated with a new Module Specification Advertisement that identifies the new version of the protocol that it implements.

The Module Implementation Advertisement

The final advertisement responsible for defining a module, the Module Implementation Advertisement, provides information on a particular implementation of a module specification, as shown in Listing 10.3.

Listing 10.3 **The Module Implementation Advertisement XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:MIA>
  <MSID> . . . </MSID>
  <Comp> . . . </Comp>
  <Code> . . . </Code>
  <PURI> . . . </PURI>
  <Prov> . . . </Prov>
  <Desc> . . . </Desc>
  <Parm> . . . </Parm>
</jxta:MIA>
```

The Module Implementation Advertisement provides information on a concrete implementation of a module specification:

- **`MSID`**—A required element containing the Module Specification ID identifying the module specification that this module is implementing.
- **`Comp`**—A required element containing compatibility information. The format of the information contained by this element depends on the possible deployment platforms for modules. Currently, the reference implementation defines an XML format for the compatibility information that details the JVM and binding. Future work to provide other bindings will result in a variety of formats for this information.
- **`Code`**—A required element containing any information required to run the code of the module implementation. The format of this information is again defined by the deployment platform in which the module will be running. Although this information is usually provided in addition to

the package information provided by the `PURI`, the information in this element could provide the code for the implementation, eliminating the need for a `PURI` element.

- **`PURI`**—An optional element containing a URI that points to a package that contains the code responsible for providing the module implementation. In the reference implementation, the `Code` element provides the fully qualified class name for the module implementation, and the `PURI` element points to the location of a JAR file containing the class described by the `Code` element. Together, these elements can be used to download the module implementation if it doesn't exist locally and to start the module.
- **`Prov`**—An optional element containing the name of the entity that is providing the module implementation specified by this advertisement's `Code` or `PURI` elements.
- **`Desc`**—An optional element containing a description of the module implementation.
- **`Parm`**—An optional element containing parameters for the implementation. The format and meaning of these parameters is defined by the module's implementation.

The implementation of the Module Implementation Advertisement is similarly split into the abstract definition `ModuleImplAdvertisement`, defined in `net.jxta.protocol`, and the reference implementation `ModuleImplAdv`, defined in `net.jxta.impl.protocol`. These classes are shown in Figure 10.3.

For every Module Specification Advertisement, there can be one or more different Module Implementation Advertisements, each specifying a different version of the module. Modules that are described by different module implementations that point to the same Module Specification Advertisement are compatible.

For example, if you have a C++ module and Java module each implementing the same version of the PDP, both will be associated with the same Module Specification Advertisement. Each implementation will be associated with different Module Implementation Advertisements that point to the same Module Specification Advertisement. The implementation-specific details, such as where to locate and download the module code, are specified in each module's Module Implementation Advertisement.

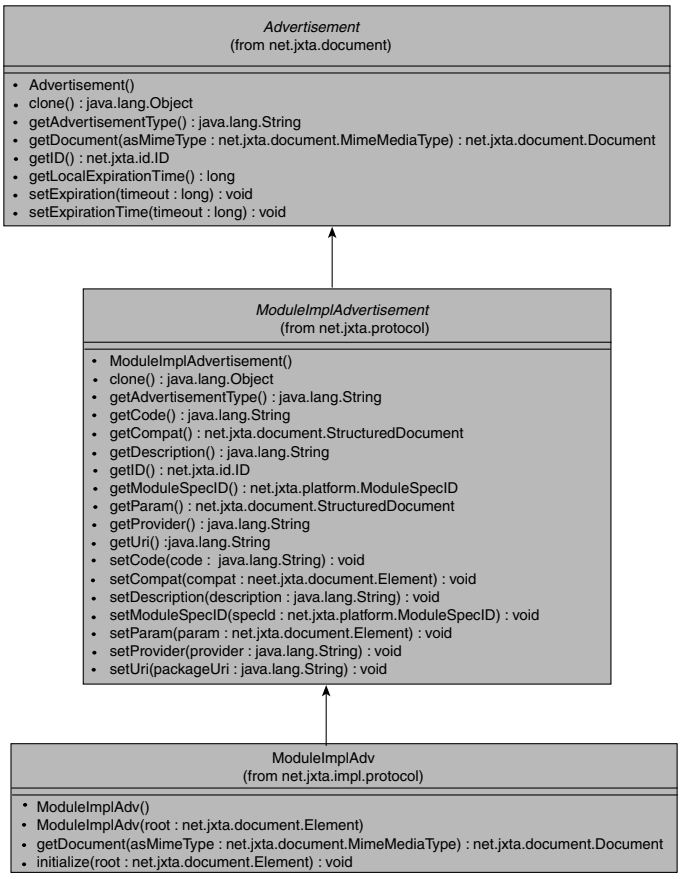

Figure 10.3 The Module Implementation Advertisement classes.

The Module, Service, and Application Interfaces

The actual implementation of a module can take one of three forms: a module, a service, or an application. For the most part, the functionality offered by each is almost identical, as are the interfaces that describe them, as shown in Figure 10.4.

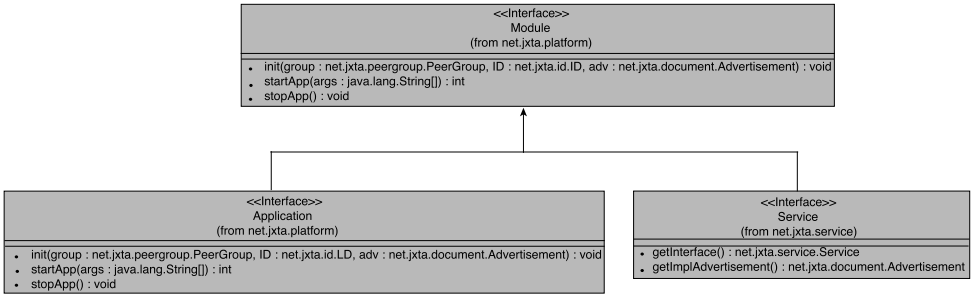

Figure 10.4 The `Module`, `Service`, and `Application` interfaces.

The `Module` and `Application` interfaces are identical, providing the `init`, `startApp`, and `stopApp` methods. As their names suggest, these methods are used to initialize, start, and stop the module. Probably the most important of these three is the `init` method:

```
public void init(PeerGroup group, ID assignedID, Advertisement implAdv)
    throws PeerGroupException;
```

This method provides the module with a `PeerGroup` that it can use to obtain services, such as the Discovery or Resolver services. In addition, an `ID` for the module within the group is provided to allow the module to uniquely identify itself within the peer group. This `ID` can be used by modules as the root of their service name space, and it is usually used as the handler name that the module registers with the Resolver service. The `implAdv` parameter is usually the `ModuleImplAdvertisement` that was used to instantiate the module, so it contains extra initialization parameters for the module in its `Parm` elements.

The `Service` interface adds only two additional methods: `getImplAdvertisement` and `getInterface`. The `getImplAdvertisement` method provides the Module Implementation Advertisement describing the service. The `getInterface` method returns another `Service` implementation that can be used to handle the `Service` implementation by proxy and protect the usage of the `Service`.

The Peer Group Lifecycle

To demonstrate the use of the JXTA reference implementation, the example code in this book has used one of two mechanisms to invoke the JXTA platform: command extensions running inside the JXTA Shell or a standalone application that invokes the platform directly. Each mechanism provided a way to access a `PeerGroup` object that allowed you to access services of a peer group.

In the case of the Shell, the examples started the platform using the `net.jxta.impl.peergroup.Boot` class:

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

When the example Shell command extension ran, a `PeerGroup` object was obtained using the Shell's environment variables, as shown in Listing 10.4.

Listing 10.4 **Obtaining the *PeerGroup* Object in a Shell Command**

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

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

When running the JXTA platform as a standalone application, the calling example applications have obtained a reference to a `PeerGroup` object using the following snippet of code:

```
net.jxta.peergroup.PeerGroup peerGroup =
    PeerGroupFactory.newNetPeerGroup();
```

Until now, no explanation has been given on why these mechanisms work or what goes on behind the scenes when either of these mechanisms is invoked. Each of these mechanisms is responsible for "bootstrapping" the JXTA platform, thereby preparing the platform to be used to perform P2P networking. This process revolves around instantiating two very special peer groups: the World Peer Group and the Net Peer Group. In JXTA, the peer group mechanism is implemented as a `Service` and therefore requires configuring the appropriate Module Implementation Advertisement. After this advertisement has been created, it is used to instantiate the Net Peer Group that allows the peer to communicate with other peers on the network.

Creating the World Peer Group

The first thing that the JXTA platform requires when bootstrapping is a *World Peer Group*, which is a peer group identified by a special Peer Group ID. The World Peer Group defines the basic capabilities of the peer, such as the services, endpoint protocol implementations, and applications that the peer will make available on the network.

Although each peer belongs to the World Peer Group, and the World Peer Group defines the endpoint protocol implementations supported by the peer, the World Peer Group itself can't be used to perform P2P networking. The World Peer Group is basically a template that can be used to either discover or generate a Net Peer Group instance. The Net Peer Group is a common peer group to peers in the network that allows all peers to communicate with each other.

In the reference implementation, the creation of the World Peer Group is managed by the `net.jxta.impl.peergroup.Platform` class. When bootstrapping the JXTA platform using either the `Boot` class or the `PeerGroupFactory.newNetPeerGroup` method, the `Platform` class is called to generate a Peer Advertisement and instantiate the World Peer Group. Note that the `Platform` class is simply a special implementation of `PeerGroup` configured to handle the bootstrapping process. A different implementation can be provided by changing the `PlatformPeerGroupClassName` property in the `config.properties` file in the `net.jxta.impl` package.

The configuration information for the endpoint protocol implementations and other services supported by the World Peer Group is extracted from a Peer Advertisement used to configure the World Peer Group. In the reference implementation, the Peer Advertisement is generated by the `Configurator` tool based on configuration input provided by the user. The Peer Advertisement uses the format given in Listing 10.5.

Listing 10.5 **The Peer Advertisement XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PA xmlns:jxta="http://jxta.org">
  <PID> . . . </PID>
  <GID> . . . </GID>
  <Name> . . . </Name>
  <Dbg> . . . </Dbg>
  <Desc> . . . </Desc>
  <Svc>
    <MCID> . . . </MCID>
    <Parm>
        . . .
    </Parm>
  </Svc>
</jxta:PA>
```

The parameters of the Peer Advertisement describe the fundamental information required for a remote peer to be capable of interacting with the peer:

- **PID**—A required element containing the Peer ID for the peer. The exact format of the ID used by JXTA isn't especially important for this discussion. The only important thing to note about the Peer ID at this point is that it incorporates the Peer Group ID in it. More information on the format of the ID used by the reference implementation can be found in the JXTA Protocols Specification.
- **GID**—An optional element containing the Peer Group ID of the peer group to which the peer described by this advertisement belongs.
- **Name**—An optional element containing a simple name for the peer. This string can be used in conjunction with the Discovery service to attempt to discover a particular peer; however, multiple peers may use the same `Name`. Only the Peer ID is guaranteed to uniquely identify a particular peer.
- **Dbg**—An optional element describing the debugging message level employed by the peer. This element is currently used only when configuring the peer while bootstrapping the platform. Currently accepted values for this element, from least explicit to most explicit, are `error`, `warn`, `info`, and `debug`.
- **Desc**—An optional element containing a description of the peer. As with the `Name` element, the contents of the `Desc` element are not necessarily unique among peers. This string can be used to perform discovery, with the same caveats as for the `Name` element.
- **Svc**—An optional element providing service configuration information. Note that multiple `Svc` elements may appear in the Peer Advertisement, each describing a different service. The format of the contents is unspecified by the Protocols Specification, and it is the responsibility of the `PeerGroup` implementation managing the bootstrapping process to know how to parse the `Svc` element's contents. The format expected by the reference implementation's `Configurator` class is a `MCID` element and a `Parm` element, explained next.
- **MCID**—The Module Class ID of the service that this `Svc` element is describing.
- **Parm**—The arbitrary parameters used to configure the service. The `Configurator` class understands only a few parameter formats, depending on the Module Class ID. The `Svc` parameters are mainly used to configure the peer's endpoint transports. Therefore, most parameters contain a Transport Advertisement containing configuration for the endpoint protocol implementation specified by the `MCID` element.

This configuration information is stored as a Peer Advertisement in a file called `PlatformConfig` in the current directory when the JXTA platform is started. Future attempts to bootstrap the platform from the same directory will use the same `PlatformConfig` file to automatically configure the peer.

After finding or creating the Peer Advertisement, the `Platform` class is responsible for generating a Peer Group Advertisement that will be used to instantiate a World Peer Group. A Peer Group Advertisement is described using the format in Listing 10.6.

Listing 10.6 **The Peer Group Advertisement XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PGA xmlns:jxta="http://jxta.org">
  <GID> . . . </GID>
  <MSID> . . . </MSID>
  <Name> . . . </Name>
  <Desc> . . . </Desc>
  <Svc>
     . . .
  </Svc>
</jxta:PGA>
```

The elements of the Peer Group Advertisement provide the following information:

- **`GID`**—A required element containing a Peer Group ID that uniquely identifies this group. For the World Peer Group, this ID is the same on all peers.
- **`MSID`**—A required element containing a Module Specification ID that identifies the module providing the implementation of the peer group module on the peer. Modules will be discussed in the section "Creating the Net Peer Group," later in this chapter.
- **`Name`**—An optional element containing a simple name for the peer group. This string is not necessarily unique.
- **`Desc`**—An optional element containing a description of the peer group. The `Desc` and `Name` elements, like their counterparts in the Peer Advertisement, can be used to discover a Peer Group Advertisement.
- **`Svc`**—An optional element describing a service provided by the peer group. Note that multiple `Svc` elements may appear in the Peer Group Advertisement, each describing a different service. The format of the contents of the `Svc` is again dependent on the `PeerGroup` implementation.

As part of generating Peer Group Advertisement for the World Peer Group, the `Platform` object creates a Module Implementation Advertisement for the group. The Module Implementation Advertisement for the group describes the services offered by the peer group.

The Module Implementation Advertisement is populated with a set of hard-coded advertisements for the core platform services: the Discovery, Resolver, Rendezvous, Peer Info, and Endpoint services. In addition, elements containing advertisements for the endpoint protocol implementations are inserted into the advertisement. When the `Platform` instantiates the World Peer Group using the generated Peer Group Advertisement, the services and protocols described by the Module Implementation Advertisement are loaded and initialized by the `Platform`.

The sequence of events leading to the creation of a World Peer Group is triggered when the `Platform` is invoked by either the `Boot` class or `PeerGroupFactory.newNetPeerGroup()`. Each of these methods invokes the `Platform` class and uses the hard-coded set of services to instantiate a World Peer Group. In some applications, it might not be desirable to load all the services hard-coded into the `Platform` class. In this case, you can create a Peer Group Advertisement yourself, use it to create a `PeerGroup` instance, and pass the resulting `PeerGroup` as a parameter to the other version of `PeerGroupFactory.newNetPeerGroup`:

```
public static PeerGroup newNetPeerGroup(PeerGroup pg)
    throws PeerGroupException
```

This version of `newNetPeerGroup` bypasses the creation of a World Peer Group using the `Platform` class, using the provided `PeerGroup` instance as the World Peer Group instead, and proceeds directly to the creation of the Net Peer Group.

Creating the Net Peer Group

After the World Peer Group has been created, the peer needs to instantiate the Net Peer Group. The Net Peer Group can describe additional characteristics about a peer, but most often it is simply a duplicate of the World Peer Group. Although the Net Peer Group can be discovered, the majority of peers using the reference implementation rely on the version of the Net Peer Group's Peer Advertisement hard-coded into the World Peer Group advertisement produced by the `Platform`.

So what exactly is the Net Peer Group? Basically, the Net Peer Group is the peer's starting point on the P2P network. All peers belong to a Net Peer Group, but not necessarily the same Net Peer Group. For example, an enterprise application might define its own Net Peer Group that peers instantiate

during the bootstrapping process. All members of the enterprise would be capable of using this Net Peer Group to communicate, but other JXTA peers wouldn't be capable of communicating with this network by default.

The World Peer Group is different from the Net Peer Group in that it is really only a configuration mechanism. The Net Peer Group is the peer group used to provide actual connectivity to the P2P network.

To generate the Net Peer Group, the `Platform` class hard-codes the `StartNetPeerGroup` application in `net.jxta.impl.peergroup` into the World Peer Group's Module Implementation Advertisement. This application is invoked by the platform and causes the `StartNetPeerGroup` class to discover or build the Net Peer Group's Peer Advertisement. Although the `StartNetPeerGroup` does provide a way for a user to choose to discover the Net Peer Group, this functionality is fairly hidden from the user. Usually, the `StartNetPeerGroup` builds a Peer Group Advertisement from the parent World Peer Group using a default Peer Group ID for the Net Peer Group.

After the Net Peer Group is instantiated, the services described by the peer group's Module Implementation Advertisement are started. In the standard Net Peer Group, this forces the core services to begin providing the Discovery, Resolver, Rendezvous, Peer Info, and Endpoint services. After these services are started, the peer is connected to the network and ready to interact with other peers. New peer groups can be created to segment the network space, using the Net Peer Group as a template for the set of services offered by the peer group.

In summary, the process of bootstrapping instantiates the World Peer Group, which comprises a set of services that are usually hard-coded into the JXTA platform implementation. After the World Peer Group has been instantiated, the peer uses the services of the World Peer Group to discover or generate a Net Peer Group instance. The Net Peer Group instance provides access to the P2P network and a set of core services used by the peer. After booting into the Net Peer Group, the peer can use the services offered by the Net Peer Group. The peer may also elect to create other peer groups with extra capabilities by using the Net Peer Group as a template for the set of core services offered by the peer group.

Working with Peer Groups

After the World and Net Peer Groups have been created, peers can communicate with each other using the core JXTA protocols. However, the Net Peer Group provides a common space where everyone in the P2P network can interact, a property that might not be suitable to all P2P applications. To allow

peers to group themselves in some meaningful way, peers can form their own peer groups, each providing its own set of services to members of the peer group.

Working with peer groups requires use of the `PeerGroup` interface, shown in Figure 10.5, defined in `net.jxta.peergroup`, and its implementations. The `StdPeerGroup` class defined in `net.jxta.impl.peergroup` provides the `PeerGroup` implementation used throughout the reference implementation.

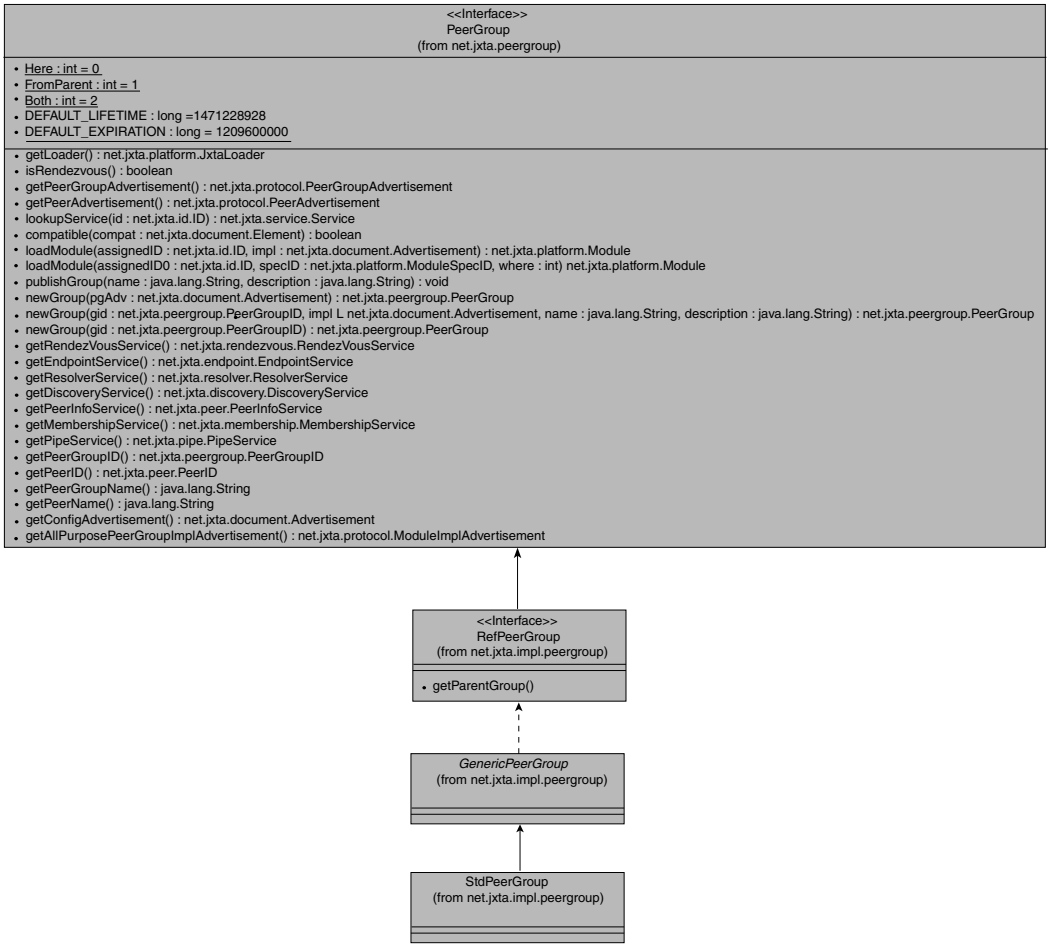

Figure 10.5 The `PeerGroup` interface and implementation classes.

The `PeerGroup` interface defines the standard Module Class IDs for the core JXTA services and Module Specification IDs for the reference implementations of those core services.

Creating a Peer Group

Creating a peer group isn't much different from using any of the other services that a peer group provides. To create a peer group, you only need to call one of the `newGroup` methods, shown in Listing 10.7, provided by the `PeerGroup` interface. Each of the different versions has a slightly different set of circumstances under which it should be invoked.

Listing 10.7 **Peer Group Creation Methods**

```
public PeerGroup newGroup(Advertisement pgAdv)
    throws PeerGroupException;newGroup(Advertisement);
public PeerGroup newGroup(PeerGroupID gid)
    throws PeerGroupException;
public PeerGroup newGroup(PeerGroupID gid, Advertisement impl,
    String name, String description)
        throws PeerGroupException;
```

The first version of `newGroup` uses a given Peer Group Advertisement to instantiate the peer group. This version is used to create a peer group using an existing Module Implementation Advertisement.

The second version of `newGroup` creates a new peer group using the given Peer Group ID. The version assumes that a Peer Group Advertisement with the corresponding Peer Group ID has already been published.

The final version of `newGroup` creates a new peer group using the given Peer Group ID, Module Implementation Advertisement, name, and description. If the given `PeerGroupID` is `null`, this method creates a new Peer Group ID for the new group.

The example in Listing 10.8 shows how to create a new peer group using the Net Peer Group and the `newGroup` method. This version simply makes a copy of the Net Peer Group's Module Implementation Advertisement and uses it to instantiate the new group. The `newGroup` method also publishes the Peer Group Advertisement for the new peer group in the parent peer group. The parent peer group is considered to be the peer group used to create the group through the call to `newGroup`.

Listing 10.8 **Source Code for *CreatePeerGroup.java***

```
package com.newriders.jxta.chapter10;

import java.util.Enumeration;

import net.jxta.discovery.DiscoveryService;

import net.jxta.exception.PeerGroupException;
```

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

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

import net.jxta.protocol.ModuleImplAdvertisement;
import net.jxta.protocol.PeerGroupAdvertisement;


/**
 * Create a new peer group and publish it.
 */
public class CreatePeerGroup
{
    /**
     * The Net Peer Group for the application.
     */
    private PeerGroup netPeerGroup = null;


    /**
     * Creates a new peer group using the Net Peer Group's
     * Module Implementation Advertisement as a template.
     *
     * @exception   Exception if an error occurs retrieving the
     *               copy of the Net Peer Group's Module
     *               Implementation Advertisement.
     */
    public void createPeerGroup() throws Exception
    {
        // The name and description for the peer group.
        String name = "CreatePeerGroup";
        String description =
            "An example peer group to test peer group creation";

        // Obtain a preformed ModuleImplAdvertisement to
        // use when creating the new peer group.
        ModuleImplAdvertisement implAdv =
            netPeerGroup.getAllPurposePeerGroupImplAdvertisement();

        // Create the Peer Group ID.
        PeerGroupID groupID = IDFactory.newPeerGroupID();
```

continues

Listing 10.8 **Continued**

```
        // Create the new group using the Peer Group ID,
        // advertisement, name, and description.
        PeerGroup newGroup = netPeerGroup.newGroup(
            groupID, implAdv, name, description);

        // Need to publish the group remotely only because
        // newGroup() handles publishing to the local peer.
        PeerGroupAdvertisement groupAdv =
            newGroup.getPeerGroupAdvertisement();
        DiscoveryService discovery =
            netPeerGroup.getDiscoveryService();
        discovery.remotePublish(groupAdv,
            DiscoveryService.GROUP);
    }

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

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

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

            // Create the group.
            creator.createPeerGroup();
```

```
            // Exit.
            System.exit(0);
        }
        catch (Exception e)
        {
            System.out.println("Error starting JXTA platform: "
                + e);
            System.exit(1);
        }
    }
}
```

The `CreatePeerGroup` example doesn't really do much of note, but it provides the first step toward creating a new peer group that provides a new service.

Joining a Peer Group

Instantiating a peer group from an advertisement is only the first step toward being able to interact with members of the peer group. Before a peer can interact with the group, it needs to join the peer group, a process that allows the peer to establish its identity within the peer group. This process allows peer groups to permit only authorized peers to join and interact with the peer group.

Each peer group has a membership policy that governs who can join the peer group. When a peer instantiates a peer group, the peer group's membership policy establishes a temporary identity for the peer, similar to the "nobody" identity in UNIX systems. This temporary identity exists for the sole purpose of allowing the peer to establish its identity by interacting with the membership policy. This interaction can involve the exchange of login information, exchange of public keys, or any other mechanism that a peer group's membership implementation uses to establish a peer's identity.

After a peer has successfully established its identity within the peer group, the membership policy provides the user with credentials. These credentials can then be used to provide verification of identity to other peers in the group when interacting with services offered by the peer group.

The membership policy for a peer group is implemented as the Membership service. The Membership service in the reference implementation is defined by `MembershipService`, shown in Figure 10.6, which is part of the `net.jxta.membership` package.

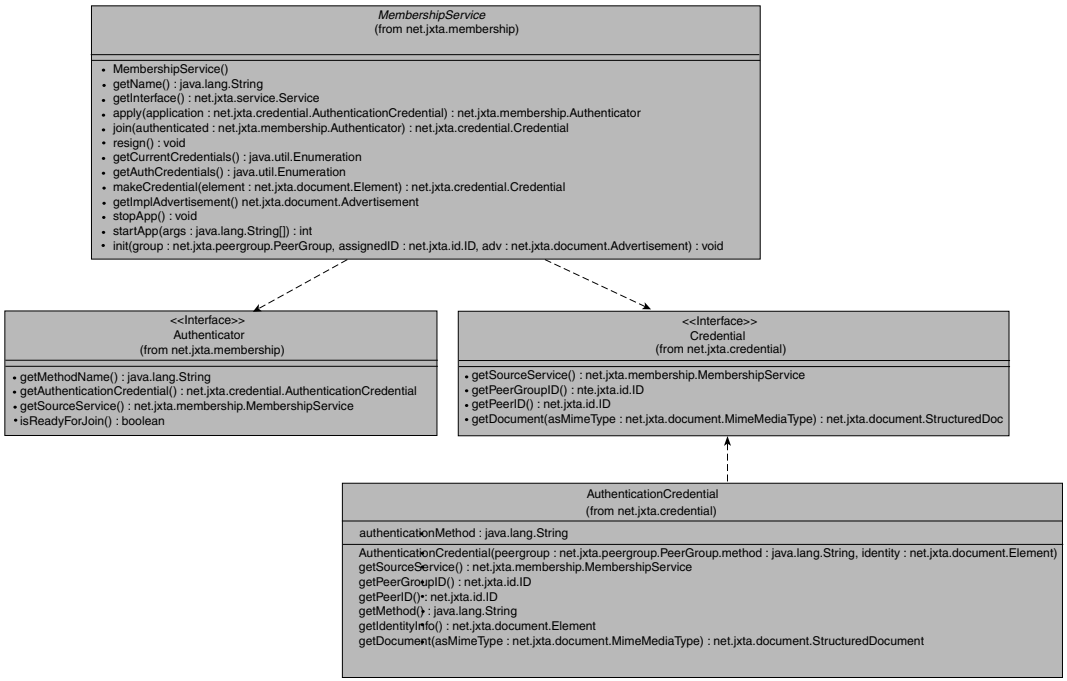

Figure 10.6 The Membership service and related classes.

In addition to the `MembershipService` class and its implementations, the reference implementation defines a `Credential` interface and an implementation called `AuthenticationCredential`. These classes, defined in `net.jxta.credential`, are used in conjunction with the `MembershipService` class to represent an identity and the access level associated with that identity.

Two steps are involved in establishing an identity within a peer group using the peer group's `MembershipService` instance:

1. **Applying for membership.** This involves calling the peer group's `MembershipService`'s `apply` method. The `apply` method takes an `AuthenticationCredential` argument, which specifies the authentication method and desired identity. The method returns an `Authenticator` implementation that the caller can use to authenticate with the peer group.
2. **Joining the group**. The peer must provide the `Authenticator` implementation with the information that it requires to authenticate. When the peer has completed authentication using the `Authenticator`, the `Authenticator`'s `isReadyForJoin` method returns `true`. The peer now calls the `MembershipService`'s `join` method, providing the `Authenticator` as an argument. The `join` method returns a `Credential` object that the peer can use to prove its identity to peer group services.

When applying for membership, the peer making the request must know the implementation of `Authenticator` to interact with the `Authenticator`. This is required because the `Authenticator` interface has no mechanism for the peer to interact with it. Using only the `Authenticator` interface, a peer can only determine whether it has completed the authentication process successfully and then proceed with joining the peer group.

The reference implementation currently provides two example `MembershipService` implementations in the `net.jxta.impl.membership` package: `NullMembershipService` and `PasswdMembershipService`. `NullMembershipService` provides a `MembershipService` that offers no real authentication and simply assigns whatever identity the peer requests. `PasswdMembershipService` provides a simple authentication based on login and "encrypted" passwords provided in the parameters in the advertisement for the Membership service. The "encryption" consists of a simple substitution cipher and thus is not practical for securing a real peer group.

Leaving a Peer Group

To leave a peer group, the peer simply calls the `resign` method on the `MembershipService` implementation for the peer group. This removes all authentication credentials from the `MembershipService` and reverts the peer to the "nobody" identity within the peer group.

The Current Membership Implementation

Unfortunately, the current implementations of `MembershipService` aren't especially useful. To provide proper authentication, a developer must develop a `MembershipService` of his own to manage the creation and validation of authentication credentials. In addition, the developer must provide a mechanism in his service to use the credentials and validate the credentials passed by other peers in requests to the service.

Although the Protocol Specification outlines the concept of an Access service whose responsibility it is to verify credentials passed with requests, no implementation is provided in the reference implementation. The Resolver service's Resolver Query and Response Messages support a `Credential` element, but the contents of the element are never verified. For now, it appears that it is the responsibility of a developer to define his own Access service implementation and use it to verify credentials passed to his custom peer group services.

As such, a developer currently needs only to instantiate a peer group and can skip the steps of applying for membership and joining the peer group. This will undoubtedly change in the future, but for now, you can safely ignore the Membership service.

Destroying a Peer Group

Many people ask, "How do I destroy a peer group that I created?" Unfortunately, there is no way to explicitly destroy a peer group after it has been created and its advertisement has been published. Although the `PeerGroup` instance on a particular peer might be destroyed, other peers on the network can still instantiate a `PeerGroup` object for the group as long as they can find the Peer Group Advertisement for the group. After the Peer Group Advertisement has been published, the peer group exists in the network until the advertisement expires.

If the Peer Group Advertisement was published to other peers using the default lifetime, the advertisement is removed from other peers' caches after two hours. However, if the Peer Group Advertisement was published locally using the default lifetime, it will not expire for a year.

One solution to this problem is to publish Peer Group Advertisements using a short lifespan. That way, the peer group will expire quickly if the advertisement isn't being used. However, doing this requires not using the `PeerGroup.publishGroup` method or the `PeerGroup.newGroup` method. By default, the `PeerGroup` reference implementation's `newGroup` method will call `publishGroup` to publish the Peer Group Advertisement. To publish the Peer Group Advertisement with a nondefault expiration or lifespan, you must manually create the `PeerGroup` instance and publish the Peer Group Advertisement using the following steps:

1. Create the `PeerGroupAdvertisement` instance using the `AdvertisementFactory.newAdvertisement` method, passing in the `String` obtained by calling the static `PeerGroupAdvertisement.getAdvertisementType` method.
2. Populate the fields of the `PeerGroupAdvertisement` instance, making sure to generate a new Peer Group ID using a call to `IDFactory`'s `newPeerGroupID` method.
3. Load the `PeerGroup` instance from the advertisement by calling the parent `PeerGroup`'s `loadModule` method, passing in the Peer Group ID from the Peer Group Advertisement and the Module Implementation Advertisement for the new Peer Group.
4. Publish the Peer Group Advertisement locally and remotely using the parent `PeerGroup` instance's `DiscoveryService` instance. This enables you to set the expiration and lifespan for the Peer Group Advertisement.

Creating a Service

Creating a service is a fairly simple task: Create a class that implements the `Service` interface. In addition to creating the `Service` implementation itself, other parts make up a good service design. A good service design separates, abstracts, and encapsulates the elements of the implementation in an object-oriented fashion.

The example service presented in the following sections extends one of the Resolver service examples presented in Chapter 5, "The Peer Resolver Protocol." The example given in that chapter showed how to use the Resolver service to create a `QueryHandler` that can handle a custom request that poses a basic math problem: What is the value of the given base raised to the given power? This example extends the basic functionality provided by that `QueryHandler` example and builds a full-fledged service module.

The Example Service Messages

To simplify the implementation of the example service, the example in Listing 10.9 reuses the `ExampleQueryMsg` and `ExampleResponseMsg` classes created in Chapter 5. These classes provide the functionality required to parse and format the XML used by the `QueryHandler` to send a query and receive a response.

Listing 10.9 **Source Code for *ExampleQueryMsg.java***

```
package com.newriders.jxta.chapter10;

import java.io.InputStream;
import java.io.StringWriter;

import java.util.Enumeration;

import net.jxta.document.Document;
import net.jxta.document.Element;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.TextElement;


/**
 * An example query message, which will be wrapped by a
 * Resolver query message to send the query to other peers.
```

continues

Listing 10.9 **Continued**

```
 * The query essentially asks a simple math question: "What
 * is the value of (base) raised to (power)?"
 */
public class ExampleQueryMsg
{
    /**
     * The base for query.
     */
    private double base = 0.0;

    /**
     * The power for the query.
     */
    private double power = 0.0;


    /**
     * Creates a query object using the given base and power.
     *
     * @param   aBase the base for the query.
     * @param   aPower the power for the query.
     */
    public ExampleQueryMsg(double aBase, double aPower)
    {
        super();

        this.base = aBase;
        this.power = aPower;
    }

    /**
     * Creates a query object by parsing the given input stream.
     *
     * @param       stream the InputStream source of the
     *              query data.
     * @exception   Exception if the message can't be parsed
     *              from the stream.
     */
    public ExampleQueryMsg(InputStream stream) throws Exception
    {
        StructuredTextDocument document = (StructuredTextDocument)
```

```
            StructuredDocumentFactory.newStructuredDocument(
                new MimeMediaType("text/xml"), stream);

        Enumeration elements = document.getChildren();

        while (elements.hasMoreElements())
        {
            TextElement element = (TextElement) elements.nextElement();

            if(element.getName().equals("base"))
            {
                base = Double.valueOf(element.getTextValue()).doubleValue();
                continue;
            }

            if(element.getName().equals("power"))
            {
                power = Double.valueOf(
                    element.getTextValue()).doubleValue();
                continue;
            }
        }
    }

    /**
     * Returns the base for the query.
     *
     * @return  the base value for the query.
     */
    public double getBase()
    {
        return base;
    }

    /**
     * Returns a Document representation of the query.
     *
     * @param        asMimeType the desired MIME type
     *               representation for the query.
     * @return       a Document form of the query in the
     *               specified MIME representation.
     * @exception    Exception if the document can't be created.
```

continues

Listing 10.9 **Continued**

```
     */
    public Document getDocument(MimeMediaType asMimeType)
        throws Exception
    {
        StructuredDocument document = (StructuredTextDocument)
            StructuredDocumentFactory.newStructuredDocument(
                asMimeType, "example:ExampleQuery");
        Element element;

        element = document.createElement("base",
            Double.toString(getBase()));
        document.appendChild(element);

        element = document.createElement("power",
            Double.toString(getPower()));
        document.appendChild(element);

        return document;
    }

    /**
     * Returns the power for the query.
     *
     * @return  the power value for the query.
     */
    public double getPower()
    {
        return power;
    }

    /**
     * Returns an XML String representation of the query.
     *
     * @return  the XML String representing this query.
     */
    public String toString()
    {
        try
        {
            StringWriter out = new StringWriter();
            StructuredTextDocument doc =
```

```
                (StructuredTextDocument) getDocument(
                    new MimeMediaType("text/xml"));
            doc.sendToWriter(out);

            return out.toString();
        }
        catch (Exception e)
        {
            return "";
        }
    }
}
```

`ExampleQueryMsg` is a simple class that wraps a base and power value for the exponentiation as an XML query that can be sent to another peer. The response to an `ExampleQueryMsg`, `ExampleResponseMsg`, contains the base and power values sent by the query, plus the result of the exponentiation. The source code for `ExampleResponseMsg` is shown in Listing 10.10.

Listing 10.10 **Source Code for *ExampleResponseMsg.java***

```
package com.newriders.jxta.chapter10;

import java.io.InputStream;
import java.io.StringWriter;

import java.util.Enumeration;

import net.jxta.document.Document;
import net.jxta.document.Element;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.TextElement;


/**
 * An example query response, which will be wrapped by a Resolver response
 * message to send the response to the query. The response contains the
 * answer to the simple math question posed by the query.
 */
```

continues

Listing 10.10 **Continued**

```
public class ExampleResponseMsg
{
    /**
     * The base from the original query.
     */
    private double base = 0.0;

    /**
     * The power from the original query.
     */
    private double power = 0.0;

    /**
     * The answer value for the response.
     */
    private double answer = 0;


    /**
     * Creates a response object using the given answer value.
     *
     * @param   anAnswer the answer for the response.
     */
    public ExampleResponseMsg(double aBase, double aPower, double anAnswer)
    {
        this.base = aBase;
        this.power = aPower;
        this.answer = anAnswer;
    }

    /**
     * Creates a response object by parsing the given input stream.
     *
     * @param        stream the InputStream source of the response data.
     * @exception    Exception if the message can't be parsed from the
     *               stream.
     */
    public ExampleResponseMsg(InputStream stream) throws Exception
    {
        StructuredTextDocument document = (StructuredTextDocument)
            StructuredDocumentFactory.newStructuredDocument(
```

```
            new MimeMediaType("text/xml"), stream);

    Enumeration elements = document.getChildren();

    while (elements.hasMoreElements())
    {
        TextElement element = (TextElement) elements.nextElement();

        if(element.getName().equals("answer"))
        {
            answer = Double.valueOf(
                element.getTextValue()).doubleValue();
            continue;
        }

        if(element.getName().equals("base"))
        {
            base = Double.valueOf(element.getTextValue()).doubleValue();
            continue;
        }

        if(element.getName().equals("power"))
        {
            power = Double.valueOf(
                element.getTextValue()).doubleValue();
            continue;
        }
    }
}

/**
 * Returns the answer for the response.
 *
 * @return  the answer value for the response.
 */
public double getAnswer()
{
    return answer;
}

/**
 * Returns the base for the query.
```

continues

Listing 10.10 **Continued**

```
     *
     * @return  the base value for the query.
     */
    public double getBase()
    {
        return base;
    }

    /**
     * Returns a Document representation of the response.
     *
     * @param        asMimeType the desired MIME type representation for
     *               the response.
     * @return       a Document form of the response in the specified
     *               MIME representation.
     * @exception    Exception if the document can't be created.
     */
    public Document getDocument(MimeMediaType asMimeType) throws Exception
    {
        Element element;
        StructuredDocument document = (StructuredTextDocument)
            StructuredDocumentFactory.newStructuredDocument(
                asMimeType, "example:ExampleResponse");

        element = document.createElement("base",
            Double.toString(getBase()));
        document.appendChild(element);

        element = document.createElement("power",
            Double.toString(getPower()));
        document.appendChild(element);

        element = document.createElement("answer", (
            new Double(getAnswer()).toString()));
        document.appendChild(element);

        return document;
    }

    /**
     * Returns the power for the query.
```

```
     *
     * @return  the power value for the query.
     */
    public double getPower()
    {
        return power;
    }

    /**
     * Returns an XML String representation of the response.
     *
     * @return  the XML String representing this response.
     */
    public String toString()
    {
        try
        {
            StringWriter out = new StringWriter();
            StructuredTextDocument doc = (StructuredTextDocument)
                getDocument(new MimeMediaType("text/xml"));
            doc.sendToWriter(out);

            return out.toString();
        }
        catch (Exception e)
        {
            return "";
        }
    }
}
```

The XML produced by these two classes, as well as the reasoning for encapsulating the classes, can be found in the original example in Chapter 5.

Creating a Listener Interface

The services throughout this book have employed listener objects to provide an elegant way for third-party developers to receive notification of arriving messages. This is a simple feature that the example service can add without too much difficulty.

To provide notifications, the service needs an interface for the listener objects. The code in Listing 10.11 provides a simple interface that the listener objects can implement to receive notification of a newly arrived `ExampleResponseMsg`.

Listing 10.11 **Source Code for *ExampleServiceListener.java***

```
package com.newriders.jxta.chapter10;

/**
 * An interface to encapsulate an object that listens for notification
 * from the ExampleService of newly arrived ExampleResponseMsg messages.
 */
public interface ExampleServiceListener
{
    /**
     * Process the newly arrived ExampleResponseMsg message.
     *
     * @param   answer the object encapsulating the notification event.
     */
    public void processAnswer(ExampleServiceEvent answer);
}
```

The `ExampleListener` interface defines only a single method, `processAnswer`, that the service calls to notify the listener. Although it is perhaps a bit unnecessary for this simple service, the `processAnswer` method takes an instance of the `ExampleServiceEvent` class, shown in Listing 10.12, as a parameter.

Listing 10.12 **Source Code for *ExampleServiceEvent.java***

```
package com.newriders.jxta.chapter10;

import java.util.EventObject;


/**
 * An object to encapsulate the event signaling the arrival of a
 * new ExampleResponseMsg at the ExampleService.
 */
public class ExampleServiceEvent extends EventObject
{
    /**
     * The response object that triggered this event.
```

```
     */
    private ExampleResponseMsg response = null;


    /**
     * Creates a new event object from the given source and
     * message object.
     *
     * @param   source the ExampleService source of the event.
     * @param   response the newly arrived ExampleResponseMsg message.
     */
    public ExampleServiceEvent(Object source, ExampleResponseMsg response)
    {
        super(source);
        this.response = response;
    }

    /**
     * Returns the response that triggered this event.
     *
     * @return  the newly arrived response message.
     */
    public ExampleResponseMsg getResponse()
    {
        return response;
    }
}
```

The `ExampleServiceEvent` serves only to wrap the arriving `ExampleResponseMsg`, which is perhaps overkill for such a simple service. But in a more elaborate service, the event object could provide other valuable information. For example, a more sophisticated service might require information about the exact time of the message's arrival, the endpoint protocol implementation used to receive the message, or the source of the response. All of this information, which is not a part of the message contents, could be added to the event object and thereby provided to the listener in addition to the received message itself.

Creating the Example Service Interface

Although the `Service` interface could be directly implemented by the example service's implementation class, it is better to separate the definition of the service from its implementation. By defining an interface for the example service, a third-party developer can define a different implementation that can be used transparently.

For the example service, only three pieces of functionality are required:

- The capability to register an `ExampleServiceListener` object with the service, allowing the listener to be notified of incoming messages
- The capability to send an `ExampleQueryMsg` to peers in the group without formulating the `ExampleQueryMsg` manually
- The capability to unregister an `ExampleServiceListener` object from the service, thus preventing the listener from receiving further message arrival notifications

The `ExampleService` interface shown in Listing 10.13 provides all of this functionality in its methods.

Listing 10.13 **Source Code for *ExampleService.java***

```
package com.newriders.jxta.chapter10;

import net.jxta.service.Service;


/**
 * An interface for the ExampleService. This interface defines the
 * operations that a developer can expect to use to manipulate the
 * ExampleService regardless of which underlying implementation of
 * the service is being used.
 */
public interface ExampleService extends Service
{
    /**
     * Add a listener object to the service. When new ExampleResponseMsg
     * responses arrive, the service will notify each registered listener.
     *
     * @param   listener the listener object to register with the service.
     */
    public void addListener(ExampleServiceListener listener);
```

```
    /**
     * Send a query to the network to determine the value of the given
     * base raised to the given power.
     *
     * @param   base the base for the exponentiation operation.
     * @param   power the exponent for the exponentiation operation.
     */
    public void findAnswer(double base, double power);

    /**
     * Remove a given listener object from the service. Once removed,
     * a listener will no longer be notified when a new ExampleResponseMsg
     * response arrives.
     *
     * @param   listener the listener object to unregister.
     */
    public void removeListener(ExampleServiceListener listener);
}
```

Notice that the interface doesn't implement the `Service` interface. This responsibility is left to the implementation of the `ExampleService` interface.

The *ExampleService* Implementation

The `ExampleService` implementation shown in Listing 10.14 provides the actual functionality provided by the service. It is responsible for registering with the Resolver service to accept queries from peers and managing the set of registered listeners.

Listing 10.14 **Source Code for *ExampleServiceImpl.java***

```
package com.newriders.jxta.chapter10;

import java.io.ByteArrayInputStream;
import java.io.IOException;

import java.util.Vector;

import net.jxta.document.Advertisement;

import net.jxta.exception.PeerGroupException;
import net.jxta.exception.NoResponseException;
```

continues

Listing 10.14 **Continued**

```
import net.jxta.exception.DiscardQueryException;
import net.jxta.exception.ResendQueryException;

import net.jxta.id.ID;

import net.jxta.impl.protocol.ResolverQuery;
import net.jxta.impl.protocol.ResolverResponse;

import net.jxta.peergroup.PeerGroup;

import net.jxta.protocol.ModuleImplAdvertisement;
import net.jxta.protocol.ResolverQueryMsg;
import net.jxta.protocol.ResolverResponseMsg;

import net.jxta.resolver.QueryHandler;
import net.jxta.resolver.ResolverService;

import net.jxta.service.Service;


/**
 * The implementation of the ExampleService interface. This service
 * builds on top of the Resolver service to provide the query
 * functionality.
 */
public class ExampleServiceImpl implements ExampleService, QueryHandler
{
    /**
     * The Module Implementation advertisement for this service.
     */
    private Advertisement implAdvertisement = null;

    /**
     * The handler name used to register the Resolver handler.
     */
    private String handlerName = null;

    /**
     * The set of listener objects registered with the service.
     */
    private Vector registeredListeners = new Vector();
```

```
/**
 * The peer group to which the service belongs.
 */
private PeerGroup myPeerGroup = null;

/**
 * The Resolver service used to send response messages.
 */
private ResolverService resolver = null;

/**
 * A unique query ID that can be used to track a query.
 */
private int queryID = 0;


/**
 * Add a listener object to the service. When new ExampleResponseMsg
 * responses arrive, the service will notify each registered listener.
 * This method is synchronized to prevent multiple threads from
 * altering the set of registered listeners simultaneously.
 *
 * @param   listener the listener object to register with the service.
 */
public synchronized void addListener(ExampleServiceListener listener)
{
    registeredListeners.addElement(listener);
}

/**
 * Send a query to the network to determine the value of the given
 * base raised to the given power.
 *
 * @param   base the base for the exponentiation operation.
 * @param   power the exponent for the exponentiation operation.
 */
public void findAnswer(double base, double power)
{
    // Make sure the service has been started.
    if (resolver != null)
    {
        // Create the query object using the given base and power.
```

continues

Listing 10.14 **Continued**

```
            ExampleQueryMsg equery = new ExampleQueryMsg(base, power);
            String localPeerId = myPeerGroup.getPeerID().toString();

            // Wrap the query in a Resolver Query Message.
            ResolverQuery query = new ResolverQuery(handlerName,
                "JXTACRED", localPeerId, equery.toString(), queryID++);

            // Send the query using the Resolver service.
                resolver.sendQuery(null, query);
        }
    }

    /**
     * Returns the advertisement for this service. In this case, this is
     * the ModuleImplAdvertisement passed in when the service was
     * initialized.
     *
     * @return  the advertisement describing this service.
     */
    public Advertisement getImplAdvertisement()
    {
        return implAdvertisement;
    }

    /**
     * Returns an interface used to protect this service.
     *
     * @return  the wrapper object to use to manipulate this service.
     */
    public Service getInterface()
    {
        // We don't really need to provide an interface object to protect
        // this service, so this method simply returns the service itself.
        return this;
    }

    /**
     * Initialize the service.
     *
     * @param       group the PeerGroup containing this service.
     * @param       assignedID the identifier for this service.
```

```
     * @param        implAdv the advertisement specifying this service.
     * @exception    PeerGroupException is not thrown ever by this
     *               implementation.
     */
    public void init(PeerGroup group, ID assignedID, Advertisement implAdv)
        throws PeerGroupException
    {
        // Save the module's implementation advertisement.
        implAdvertisement = (ModuleImplAdvertisement) implAdv;

        // Use the assigned ID as the Resolver handler name.
        handlerName = assignedID.toString();

        // Save a reference to the group of which that this service
        // is a part.
        myPeerGroup = group;
    }

    /**
     * Processes the Resolver query message and returns a response.
     *
     * @param       query the message to be processed.
     * @exception   IOException if the query can't be read.
     */
    public ResolverResponseMsg processQuery(ResolverQueryMsg query)
        throws IOException, NoResponseException, DiscardQueryException,
            ResendQueryException
    {
        ResolverResponse response;
        ExampleQueryMsg eq;
        double answer = 0.0;

        try
        {
            // Extract the query message.
            eq = new ExampleQueryMsg(
                new ByteArrayInputStream((query.getQuery()).getBytes()));
        }
        catch (Exception e)
        {
            throw new IOException();
        }
```

continues

Listing 10.14 **Continued**

```
        // Perform the calculation.
        answer = Math.pow(eq.getBase(), eq.getPower());

        // Create the response message.
        ExampleResponseMsg er = new ExampleResponseMsg(
            eq.getBase(), eq.getPower(), answer);

        // Wrap the response message in a Resolver Response Message.
        response = new ResolverResponse(handlerName, "JXTACRED",
            query.getQueryId(), er.toString());

        // Return the message so that the Resolver service can handle
        // sending it.
        return response;
    }

    /**
     * Process a Resolver Response Message.
     *
     * @param   response a response message to be processed.
     */
    public void processResponse(ResolverResponseMsg response)
    {
        ExampleResponseMsg er;
        ExampleServiceEvent event;


        try
        {
            // Extract the message from the Resolver response.
            er = new ExampleResponseMsg(
                new ByteArrayInputStream(
                    (response.getResponse()).getBytes()));

            // Create an event to send to the listeners.
            event = new ExampleServiceEvent(this, er);

            // Notify each of the registered listeners.
            if (registeredListeners.size() > 0)
            {
                ExampleServiceListener listener = null;
```

```
            for (int i = 0; i < registeredListeners.size(); i++)
            {
                listener = (ExampleServiceListener)
                    registeredListeners.elementAt(i);
                listener.processAnswer(event);
            }
        }
    }
    catch (Exception e)
    {
        // This is not the right type of response message, or
        // the message is improperly formed. Ignore the exception;
        // do nothing with the message.
    }
}

/**
 * Remove a given listener object from the service. Once removed,
 * a listener will no longer be notified when a new ExampleResponseMsg
 * response arrives.
 *
 * @param   listener the listener object to unregister.
 */
public synchronized void removeListener(ExampleServiceListener listener)
{
    registeredListeners.removeElement(listener);
}

/**
 * Start the service.
 *
 * @param   args the arguments to the service. Not used.
 * @return  0 to indicate the service started.
 */
public int startApp(String[] args)
{
    // Now that the service is being started, set the ResolverService
    // object to use to handle queries and responses.
    resolver = myPeerGroup.getResolverService();

    // Add ourselves as a listener using the unique constructed
    // handler name.
```

continues

Listing 10.14 **Continued**

```
            resolver.registerHandler(handlerName, this);

            return 0;
        }

        /**
         * Stop the service.
         */
        public void stopApp()
        {
            // Unregister ourselves as a listener.
            if (resolver != null)
            {
                resolver.unregisterHandler(handlerName);
            }
        }
    }
```

Note
The service implementation must have a zero-argument constructor to allow the platform to load the service properly when initializing a peer group configured to use the service implementation. In the `ExampleServiceImpl` class, no constructor is defined, so the Java compiler generates a zero-argument constructor by default.

The `init` method implementation simply stores the passed parameters for later use. The passed ID will be used later to register the service with the Resolver service, and the given peer group will be used to obtain access to the Resolver service. Although nothing prevents the `ExampleServiceImpl` from registering with the Resolver in the `init` method, that task is performed in the `startApp` method. The `init` method is called to prepare the service, but the service shouldn't begin handling queries until the `startApp` method is called. Hence, the service doesn't register with the Resolver service until `startApp` is called. Conversely, the `stopApp` method unregisters the service with the Resolver service to prevent the service from handling queries when it has been stopped.

Adding the *ExampleService* Implementation to a Peer Group

The most difficult part of creating a new service is not creating the `Service` implementation, but adding it to a peer group. Adding the service requires the creation of the Module Class, Module Specification, and Module

Implementation Advertisements describing the `Service` implementation. Usually it is preferred that the Module Class and Specification IDs be created beforehand so that their values are known for future reference. The simple application in Listing 10.15 generates the various required IDs and prints their values to the screen.

Listing 10.15 **Source Code for *GenerateID.java***

```
package com.newriders.jxta.chapter10;

import net.jxta.id.IDFactory;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupID;

import net.jxta.platform.ModuleClassID;
import net.jxta.platform.ModuleSpecID;


/**
 * A simple application to generate a Module Class ID, Module Specification
 * ID, Peer Group ID, and Module Specification ID based on the standard
 * peer group Module Class ID.
 */
public class GenerateID
{
    /**
     * Generates the IDs.
     *
     * @param   args the command-line arguments. Ignored by this app.
     */
    public static void main(String[] args)
    {
        // Create an entirely new Module Class ID.
        ModuleClassID classID = IDFactory.newModuleClassID();

        // Create a Module Specification ID based on the generated
        // Module Class ID.
        ModuleSpecID specID = IDFactory.newModuleSpecID(classID);

        // Create an entirely new Peer Group ID.
        PeerGroupID groupID = IDFactory.newPeerGroupID();
```

continues

Listing 10.15 **Continued**

```
        // Create a Module Specification ID based on the peer group
        // Module Class ID.
        ModuleSpecID groupSpecID = IDFactory.newModuleSpecID(
            PeerGroup.allPurposePeerGroupSpecID.getBaseClass());

        // Print out the generated IDs.
        System.out.println("Module Class ID: " + classID.toString());
        System.out.println("Module Spec ID: " + specID.toString());
        System.out.println("Peer Group ID: " + groupID.toString());
        System.out.println("Peer Group Module Spec ID: "
            + groupSpecID.toString());
    }
}
```

Although it is not essential, the `GenerateID` application also generates the Peer Group ID that will be used to create a peer group in the example. Creating a new peer group is a required part of adding a new service because the definition of the services offered by a peer group cannot change. The Module Implementation Advertisement associated with a peer group specifies which services the peer group offers. Because this advertisement is most likely cached throughout the network, allowing it to be changed would result in inconsistencies in the services offered by members of the peer group across the network. For the same reason that a Module Implementation Advertisement cannot be changed, a Peer Group Advertisement also cannot be altered. Therefore, offering a new service requires the creation of both a new Module Implementation Advertisement for the peer group and a new peer group that uses that Module Implementation Advertisement.

Because the example will have to create a new Module Implementation Advertisement, `GenerateID` creates the Module Specification ID that will be used for the new peer group's Module Implementation Advertisement. This ID is created using the Module Class ID of the standard peer group reference implementation.

With all those IDs generated, all that remains is to write an application such as Listing 10.16 that creates a new peer group that uses the new service.

Listing 10.16 **Source Code for *ExampleServiceTest.java***

```
package com.newriders.jxta.chapter10;

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

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

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

import java.net.MalformedURLException;
import java.net.UnknownServiceException;
import java.net.URL;

import java.util.Enumeration;
import java.util.Hashtable;

import net.jxta.discovery.DiscoveryService;

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.Element;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredTextDocument;

import net.jxta.exception.PeerGroupException;
import net.jxta.exception.ServiceNotFoundException;

import net.jxta.id.IDFactory;

import net.jxta.impl.peergroup.StdPeerGroupParamAdv;

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

continues

Listing 10.16 **Continued**

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

import net.jxta.platform.ModuleClassID;
import net.jxta.platform.ModuleSpecID;

import net.jxta.protocol.ModuleClassAdvertisement;
import net.jxta.protocol.ModuleImplAdvertisement;
import net.jxta.protocol.ModuleSpecAdvertisement;
import net.jxta.protocol.PeerGroupAdvertisement;


/**
 * An application to create a peer group, configure a new service for
 * the peer group, and then interact with other peers using that new
 * service.
 */
public class ExampleServiceTest implements ExampleServiceListener
{
    /**
     * The Module Class ID to use for the service.
     * YOU SHOULD REPLACE THIS WITH ONE YOU GENERATE
     * YOURSELF USING THE GenerateID APPLICATION!
     */
    private static final String refModuleClassID =
        "urn:jxta:uuid-128E938121DD4957B74B90EE27FDC61F05";

    /**
     * The Module Specification ID to use for the service.
     * YOU SHOULD REPLACE THIS WITH ONE YOU GENERATE
     * YOURSELF USING THE GenerateID APPLICATION!
     */
    private static final String refModuleSpecID =
        "urn:jxta:uuid-128E938121DD4957B74B90EE27FDC61FA385BCB"
        + "1BA504B0FA69F99FE84CDC25B06";

    /**
     * The Peer Group ID to use for the application.
     * YOU SHOULD REPLACE THIS WITH ONE YOU GENERATE
     * YOURSELF USING THE GenerateID APPLICATION!
     */
    private static final String refPeerGroupID =
```

```
    "urn:jxta:uuid-A87A7DD0762F47E88B2FB5452D47B3A802";

/**
 * The peer group Module Specification ID to use for the application.
 * YOU SHOULD REPLACE THIS WITH ONE YOU GENERATE
 * YOURSELF USING THE GenerateID APPLICATION!
 */
private static final String refPeerGroupSpec =
    "urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE00000001CB18295"
    + "F0DE94F99983AA2F00C1DE42F06";

/**
 * The Net Peer Group for the application.
 */
private PeerGroup netPeerGroup = null;

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

/**
 * The textfield for accepting the base input for the
 * exponentiation operation.
 */
private JTextField baseText = new JTextField(5);

/**
 * The textfield for accepting the power input for the
 * exponentiation operation.
 */
private JTextField powerText = new JTextField(5);

/**
 * The new group created by the application.
 */
private PeerGroup newGroup = null;


/**
 * Create the Module Class Advertisement for the service, using the
 * preconfigured ID in refModuleClassID.
```

continues

Listing 10.16 **Continued**

```
     *
     * @return       the generated Module Class Advertisement.
     * @exception    UnknownServiceException, MalformedURLException thrown
     *                if the refModuleClassID is invalid or malformed.
     */
    private ModuleClassAdvertisement createModuleClassAdv()
        throws UnknownServiceException, MalformedURLException
    {
        // Create the class ID from the refModuleClassID string.
        ModuleClassID classID = (ModuleClassID) IDFactory.fromURL(
            new URL((refModuleClassID)));

        // Create the Module Class Advertisement.
        ModuleClassAdvertisement moduleClassAdv =
            (ModuleClassAdvertisement)
                AdvertisementFactory.newAdvertisement(
                    ModuleClassAdvertisement.getAdvertisementType());

        // Configure the Module Class Advertisement.
        moduleClassAdv.setDescription(
            "A service to handle exponentiation math problems.");
        moduleClassAdv.setModuleClassID(classID);
        moduleClassAdv.setName("Exponentiator Class");

        // Return the advertisement to the caller.
        return moduleClassAdv;
    }

    /**
     * Create the Module Implementation Advertisement for the service,
     * using the specification ID in the passed in ModuleSpecAdvertisement
     * advertisement. Use the given ModuleImplAdvertisement to create the
     * compatibility element of the module impl specification.
     *
     * @param        groupImpl the ModuleImplAdvertisement of the parent
     *                peer group.
     * @param        moduleSpecAdv the source of the specification ID.
     * @return       the generated Module Implementation Advertisement.
     */
    private ModuleImplAdvertisement createModuleImplAdv(
        ModuleImplAdvertisement groupImpl,
```

```
    ModuleSpecAdvertisement moduleSpecAdv)
{
    // Get the specification ID from the passed advertisement.
    ModuleSpecID specID = moduleSpecAdv.getModuleSpecID();

    // Create the Module Implementation Advertisement.
    ModuleImplAdvertisement moduleImplAdv = (ModuleImplAdvertisement)
        AdvertisementFactory.newAdvertisement(
            ModuleImplAdvertisement.getAdvertisementType());

    // Configure the Module Implementation Advertisement.
    moduleImplAdv.setCode(
        "com.newriders.jxta.chapter10.ExampleServiceImpl");
    moduleImplAdv.setCompat(groupImpl.getCompat());
    moduleImplAdv.setDescription(
        "Reference Exponentiator implementation");
    moduleImplAdv.setModuleSpecID(specID);
    moduleImplAdv.setProvider("Brendon J. Wilson");

    // Return the advertisement to the caller.
    return moduleImplAdv;
}

/**
 * Create the Module Specification Advertisement for the service,
 * using the preconfigured ID in refModuleSpecID.
 *
 * @return       the generated Module Class Advertisement.
 * @exception   UnknownServiceException, MalformedURLException thrown
 *               if the refModuleSpecID is invalid or malformed.
 */
private ModuleSpecAdvertisement createModuleSpecAdv()
    throws UnknownServiceException, MalformedURLException
{
    // Create the specification ID from the refModuleSpecID string.
    ModuleSpecID specID = (ModuleSpecID) IDFactory.fromURL(
        new URL((refModuleSpecID)));

    // Create the Module Specification Advertisement.
    ModuleSpecAdvertisement moduleSpecAdv = (ModuleSpecAdvertisement)
        AdvertisementFactory.newAdvertisement(
            ModuleSpecAdvertisement.getAdvertisementType());
```

continues

Listing 10.16 **Continued**

```
        // Configure the Module Specification Advertisement.
        moduleSpecAdv.setCreator("Brendon J. Wilson");
        moduleSpecAdv.setDescription(
            "A specification for an exponentiation service.");
        moduleSpecAdv.setModuleSpecID(specID);
        moduleSpecAdv.setName("Exponentiator Spec");
        moduleSpecAdv.setSpecURI(
            "http://www.brendonwilson.com/projects/jxta");
        moduleSpecAdv.setVersion("1.0");

        // Return the advertisement to the caller.
        return moduleSpecAdv;
    }

    /**
     * Creates a peer group and configures the ExampleService
     * implementation to run as a peer group service.
     *
     * @exception   Exception, PeerGroupException if there is a problem
     *              while creating the peer group or the service
     *              advertisements.
     */
    public void createPeerGroup() throws Exception, PeerGroupException
    {
        // The name and description for the peer group.
        String name = "CreatePeerGroup";
        String description =
            "An example peer group to test peer group creation";

        // The Discovery service to use to publish the module and peer
        // group advertisements.
        DiscoveryService discovery = netPeerGroup.getDiscoveryService();

        // Obtain a preformed ModuleImplAdvertisement to use when creating
        // the new peer group. This is the Module Implementation
        // Advertisement of the Net Peer Group and contains all of the
        // services and applications already configured to run in that peer
        // group. Using this method simplifies the task of creating a new
        // peer group and configuring a new service.
        ModuleImplAdvertisement implAdv =
```

```
        netPeerGroup.getAllPurposePeerGroupImplAdvertisement();

    // Create the Module Class Advertisement.
    ModuleClassAdvertisement moduleClassAdv = createModuleClassAdv();

    // Create the Module Specification Advertisement.
    ModuleSpecAdvertisement moduleSpecAdv = createModuleSpecAdv();

    // Create the Module Implementation Advertisement.
    ModuleImplAdvertisement moduleImplAdv =
        createModuleImplAdv(implAdv,moduleSpecAdv);

    // Get the parameters for the peer group's Module Implementation
    // Advertisement to add our service.
    StdPeerGroupParamAdv params =
        new StdPeerGroupParamAdv(implAdv.getParam());

    // Get the services from the parameters.
    Hashtable services = params.getServices();

    // Add our service to the set of services.
    services.put(moduleClassAdv.getModuleClassID(), moduleImplAdv);

    // Set the services on the parameters, and set the parameters on
    // the implementation advertisement.
    params.setServices(services);
    implAdv.setParam((StructuredDocument) params.getDocument(
        new MimeMediaType("text", "xml")));

    // VERY IMPORTANT! You must change the Module Specification ID
    // for the implementation advertisement. If you don't, the new
    // peer group's Module Specification ID will still point to the
    // old specification, and the new service will not be loaded.
    implAdv.setModuleSpecID((ModuleSpecID) IDFactory.fromURL(
        new URL(refPeerGroupSpec)));

    // Publish the Module Class and Specification Advertisements.
    discovery.publish(moduleClassAdv, DiscoveryService.ADV);
    discovery.remotePublish(moduleClassAdv, DiscoveryService.ADV);
    discovery.publish(moduleSpecAdv, DiscoveryService.ADV);
    discovery.remotePublish(moduleSpecAdv, DiscoveryService.ADV);
    discovery.publish(implAdv, DiscoveryService.ADV);
```

continues

Listing 10.16 **Continued**

```
        discovery.remotePublish(implAdv, DiscoveryService.ADV);

        // Create the Peer Group ID.
        PeerGroupID groupID = (PeerGroupID) IDFactory.fromURL(
            new URL((refPeerGroupID)));

        // Create the new group using the group ID, advertisement, name,
        // and description.
        newGroup = netPeerGroup.newGroup(groupID, implAdv, name,
            description);

        // Need to publish the group remotely only because newGroup()
        // handles publishing to the local peer.
        PeerGroupAdvertisement groupAdv =
            newGroup.getPeerGroupAdvertisement();
        discovery.remotePublish(groupAdv, DiscoveryService.GROUP);
    }

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

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

        try
        {
            // Initialize the JXTA platform.
```

```
            test.initializeJXTA();

            // Create the group.
            test.createPeerGroup();

            // Show a GUI to accept input.
            test.showGUI();
        }
        catch (Exception e)
        {
            System.out.println("Error starting JXTA platform: " + e);
            System.exit(1);
        }
    }

    /**
     * The implementation of the ExampleServiceListener interface. This
     * allows us to display a message each time a message is received by
     * the ExampleService.
     *
     * @param   answer the event containing the newly arrived message.
     */
    public void processAnswer(ExampleServiceEvent event)
    {
        // Extract the response message from the event object.
        ExampleResponseMsg er = event.getResponse();

        // Print out the answer given in the response.
        String answer = "The value of " + er.getBase() + " raised to "
            + er.getPower() + " is: " + er.getAnswer();
        JOptionPane.showMessageDialog(null, answer, "Answer Received!",
            JOptionPane.INFORMATION_MESSAGE);
    }

    /**
     * Convenience method to find the service and use it to send
     * a query to other peers' ExampleService.
     *
     * @param   base the base for the exponentiation query.
     * @param   power the power for the exponentiation query.
     */
    private void sendMessage(String base, String power)
```

continues

Listing 10.16 **Continued**

```
{
    try
    {
        // Convert the input to numbers.
        double baseValue = Double.parseDouble(base);
        double powerValue = Double.parseDouble(power);

        // Find the service on the peer group.
        ModuleClassID classID = (ModuleClassID) IDFactory.fromURL(
            new URL((refModuleClassID)));
        ExampleService exponentiator =
            (ExampleService) newGroup.lookupService(classID);
        exponentiator.findAnswer(baseValue, powerValue);
    }
    catch (NumberFormatException e)
    {
        // Warn the user.
        JOptionPane.showMessageDialog(null, "The base and power must "
            + "both be numbers!", "Input Error!",
            JOptionPane.ERROR_MESSAGE);
    }
    catch (Exception e2)
    {
        // Warn the user.
        JOptionPane.showMessageDialog(null, "Error finding service!",
            "Service Error!", JOptionPane.ERROR_MESSAGE);
    }
}

/**
 * Displays a user interface to allow the user to send queries to
 * other peers.
 *
 * @exception   exceptions thrown only if the new service can't be
 *              found.
 */
private void showGUI() throws UnknownServiceException,
    MalformedURLException, ServiceNotFoundException
{
    JButton sendButton = new JButton("Send Message");
    JButton quitButton = new JButton("Quit");
```

```
JPanel sendPane = new JPanel();
JLabel baseLabel = new JLabel("Base:");
JLabel powerLabel = new JLabel("Power:");
Container pane = clientFrame.getContentPane();


// Populate the GUI frame.
sendPane.setLayout(new FlowLayout());
sendPane.add(baseLabel);
sendPane.add(baseText);
sendPane.add(powerLabel);
sendPane.add(powerText);
sendPane.add(sendButton);
sendPane.add(quitButton);
pane.setLayout(new BorderLayout());
pane.add(sendPane, BorderLayout.SOUTH);

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

            // Clear the text.
            baseText.setText("");
            powerText.setText("");
        }
    }
);

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

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

continues

Listing 10.16 **Continued**

```
                }
            }
        );

        // Find the new service on the peer group and add ourselves
        // as a listener.
        ModuleClassID classID = (ModuleClassID) IDFactory.fromURL(
            new URL((refModuleClassID)));
        ExampleService exponentiator =
            (ExampleService) newGroup.lookupService(classID);
        exponentiator.addListener(this);

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

To simplify the task of creating a Module Implementation Advertisement for a new peer group, a developer can use the `getAllPurposePeerGroupImplAdvertisement` on an existing peer group. This method provides a copy of the peer group's `ModuleImplAdvertisement` containing the parameters that specify the services offered by the peer group. In the reference implementation, these parameters can be manipulated via the `StdPeerGroupParamAdv` class provided in the `net.jxta.impl.peergroup` package.

The process of creating a new peer group with a new service can be confusing, so here are the essential steps that the `ExampleServiceTest` executes:

- **`createModuleClassAdv`**—This method creates the Module Class Advertisement for the example service, using a Module Class ID hard-coded in the `ExampleServiceTest`. This Module Class ID was generated using `GenerateID`. The Module Class Advertisement is configured with the Module Class ID, plus a simple name and description.
- **`createModuleSpecAdv`**—This method creates the Module Specification Advertisement for the example service, using a Module Specification ID hard-coded in the `ExampleServiceTest`. This Module Specification ID was generated using `GenerateID`. The Module Specification Advertisement is configured to provide version information on the new service and where to find a document describing the specification of the service.

- **`createModuleImplAdv`**—This method creates the Module Implementation Advertisement for the example service, using the same Module Specification ID used when creating the Module Specification Advertisement. The Module Implementation Advertisement is configured to provide information on the implementation of the service. This is where the `ExampleServiceImpl` code is bound to a Module Implementation Advertisement. To provide the same compatibility as other services on the peer, the generic implementation advertisement retrieved using the `getAllPurposePeerGroupImplAdvertisement` method is passed to this method. This advertisement is used as the source of the compatibility information configured on the Module Implementation Advertisement for the new service.

After the various module advertisements for the example service have been created, the Module Implementation Advertisement for the new service must be added to the Module Implementation Advertisement for the peer group. The `ExampleServiceTest` application performs the following steps to alter the generic peer group Module Implementation Advertisement returned by the `getAllPurposePeerGroupImplAdvertisement` method:

1. Extract the parameters from the generic peer group Module Implementation Advertisement using `getParam`, and create a `StdPeerGroupParams` object. This object deals with the format for the parameters used by the reference implementation of the `PeerGroup` interface.
2. Extract the parameter's `Hashtable` of services using `getServices`.
3. Add the new service implementation advertisement using the `put` method. In the reference implementation, services are added to the `Hashtable` using the service's class ID as a key.
4. Set the service `Hashtable` on the parameters using `setServices`.
5. Set the parameters on the peer group's Module Implementation Advertisement using `setParam`.
6. Change the Module Implementation Advertisement's Module Specification ID. This is a very important step! If the implementation's Module Implementation's Module Specification ID isn't changed, the new peer group will use the Module Specification ID of the peer group that provided the generic peer group Module Implementation Advertisement. When the new peer group is created, the platform will search for an implementation of the old module specification; therefore, the new service will never be loaded.

When those steps are completed, the new peer group can be created using the new peer group Module Implementation Advertisement. The new peer group's Module Implementation Advertisement causes the new service to be loaded and to start the new service.

Running the *ExampleServiceTest* Application

To run the `ExampleServiceTest` and see the example service in action, follow these steps:

1. Compile all the source code.
2. Place the resulting class files in a new directory.
3. Copy all the JXTA JAR files into this new directory.
4. Create a copy of this directory.
5. Start the `ExampleServiceTest` from the first directory by opening a command console, changing to the first directory, and executing this code:

```
java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;
instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;
log4j.jar;minimalBC.jar com.newriders.jxta.chapter10.ExampleServiceTest
```

The user interface for the application should appear. Start the `ExampleServiceTest` application from the second directory in the same way. After the user interface appears, you should be able to use the user interface to send a message via the example service between the two applications.

Summary

In this chapter, you've seen peer groups and peer group services and learned how they are related. As part of this discussion, this chapter explored how modules can be used within JXTA and how JXTA provides support for multiple versions and implementations of a module. Finally, the chapter demonstrated how to create a new service module and add it to a new peer group. In the next chapter, all the elements of the previous chapters are brought together in a sample application to demonstrate the power of JXTA.