3

JXTA Protocols

By Daniel Brookshier

In this chapter, you are going to read about the Java implementation of the JXTA protocols. We will highlight the important classes, interfaces, and functionality. The JXTA API is fairly large, not simple, and not always obvious. Consider this chapter an introduction to the API rather than a comprehensive treatment. We will cover important aspects of the JXTA API in more detail in later chapters. The Java version of JXTA is quite large, with almost three hundred Java files in the core platform. In the description here, we will use class diagrams that show relationships, the parent package, and sometimes the methods. Remember that JXTA is evolving, so there may be minor differences over time. With Sun Microsystems' acting stewardship, the changes at this level should be minor. Just in case, please check this book's Web page at `www.samspublishing.com` where we will track all the changes to JXTA from the time this book is published.

IN THIS CHAPTER

Protocol and API

The Java JXTA platform is a series of classes and methods for managing and transmitting application and control data between JXTA compatible peer platforms. These core services are used to create peer-to-peer applications.

One of the first concepts to cover is that JXTA was not initially defined as a Java API. JXTA was originally defined as a set of behaviors and messages. The messages were defined as XML documents with language and operating system independence. The Java version of JXTA is just one of many possible implementations of the JXTA protocols.

A *protocol* is a repeatable procedure for regulating data transmission between computers. There are implementations of the protocols written in Java, C, Perl, and others.

Each of these languages has a different API. The Java API covered in this book is the J2SE (Java 2 Standard Edition) version. There is also a J2ME (Java 2 Micro Edition) version for small devices like phones, PDAs, and other devices. Each API is written to be useful to its developers and does not need to match the Java reference platform in any way other than the JXTA protocol. Some versions, such as the JXTA for the J2ME platform, only implement certain part of the JXTA protocols.

The API can hide many of the details of a protocol. The differences between the Java JXTA API and the JXTA protocol are blurred in some areas and obvious in others. For example, the XML advertisements specified by the protocol are fairly well represented by Java classes and interfaces. Some actions, such as routing, are fairly well hidden from application programmers.

The key parts of the XJTA API are peer membership, pipes, discovery, and the resolver. Less used, but interesting, are the peer endpoint and peer information APIs. In addition, other APIs make up functionalities for rendezvous, gateways, and routers. Rendezvous, gateways, and routers are only of interest to the application developer because of the enhanced services they provide. This chapter covers some of their functionality because it does help to know where some of the mechanics reside.

JXTA Goals

The goal of JXTA is not to have Java everywhere, but peer-to-peer networking everywhere. The Java implementation of JXTA should be completely compatible with any other version, whether written in C, Pearl, or other popular language.

JXTA goals also include corporate and ISP acceptance. To that end, the platform can be configured to provide basic services that can be placed on a dedicated computer. The services can be controlled by the ISP or corporate network administrators, similar to how routers, firewalls, and proxy servers are used today.

Another goal of JXTA is to create a platform rather than an application. The JXTA platform aims to be application agnostic with services provided that can support a hopefully unlimited number of application types.

Finally, JXTA needs to be fast. Speed is a bit harder goal to master, especially for JXTA. Because JXTA is a platform, it is attempting to be all things to all applications. Because of this, the protocols are written with a general use in mind. Speed and efficiency can be measured via tests, but only use by applications will show how good the design is. P2P networking may seem simple, but the reality is that even the simplest system can have very complex behavior. JXTA has attempted to create a fast and efficient system by caching results, modularizing services, and allowing for specific helper services, such as routers.

Another goal that is related to the JXTA platform is to test the JXTA protocol. The platform is a test bed to refine the protocols and ensure that peers can interoperated both between peers based on the Java platform and other languages.

Finally, an important goal is to allow other developers to create applications on the platform. JXTA requires applications to succeed. Like Java, the platform is free to use as long as the open-source license is followed.

Now that we have a few goals, let's look at how the platform is designed. Note that you will not see everything we talked about in Chapter 2, "Overview of JXTA." We are at a higher level where applications can interact with JXTA services.

NOTE

Unlike the JXTA protocol specification, this chapter will focus on the Java implementation of the protocols. You should refer to the JXTA protocol specification and Chapter 2, "Overview of JXTA," for a more generic discussion.

JXTA Peer and Java

A *peer* is a node in the JXTA network. Each peer belongs to one or more groups and implements a set of services that allows other peers to interact with this peer, or to others of which the peer is aware. For most of the initial applications written, the peer is synonymous with the user, even though this is not ideal. Identity is left up to the implementer of the application. The peer is best thought of as a computer that does not care about identity. Keep this in mind when you write a JXTA application, and be sure to add some form of user identity.

A JXTA peer in the Java implementation is associated with one JVM. Having only one peer per device, such as a PC, is the normal scenario. You can start multiple JVMs to create multiple peers on a PC, but this is only really worth doing for debugging and experimentation. If you want to start multiple copies of JXTA applications, you need to do so in separate directories and use different communication ports to avoid the applications from conflicting. The key reason to use a separate directory is because of the cache management system.

The `cm` Directory

The cache management system is used to store information about the peer-to-peer (P2P) network. This information is primarily advertisements created by the peer or found on the network during discovery. The advertisements are stored as files with filenames that use the ID of the advertisement. Note that this is just the first version of the cache manager, and future versions may use a database instead of files.

The reason for such persistence is obvious when you consider there are hundreds of peers to interact with. Without local caching, you would need to contact a good portion of these peers to rebuild information required for your application.

The cache is required because it is very costly to accumulate advertisements. For example, to get a peer advertisement of a peer you would like to chat with, you may

need to pass through several rendezvous peers just to locate the peer. If you had to perform this operation every time, you would significantly reduce the performance of your application over time. The network would also be clogged with discovery messages from peers constantly rediscovering the information each time they started.

Each JVM instance has one associated peer because information is stored in the directory from which JXTA is launched. There are files in this rooted directory, along with various other directories and files.

One root directory created when the peer is started for the first time is the `cm` (cache management) directory. The content that is being managed are the advertisements that are both created locally and fetched from the P2P network. The role of the `cm` directory is to act as a local cache of these advertisements. The cache acts as a form of persistence between sessions. Without the cache, the advertisements would have to be reloaded from other peers.

Below the `cm` directory are group directories. For each group you join, there is a corresponding directory. There are two directories , which are always created because all peers belong to the World and Net groups. These directories are named `jxta-NetGroup` and `jxta-WorldGroup` for the Net and World groups, respectively.

As the peer joins new groups, new directories are added. Using the group ID, JXTA creates the directories. Each group directory contains information about advertisements discovered in the group and any other information about the group, such as membership and credentials.

NOTE

You can use the `cm` directory for monitoring the health of your peer and for debugging. If you are having problems with your messaging, the `cm` directory is very useful for debugging.

Looking at files in the group directories can help show that your application is connecting to the JXTA network. As you do remote discovery, peer and other advertisements will get written to these directories. Without writing code, you can examine the XML and learn about what is happening.

Another directory that can exist in the `cm` directory is the `HttpTransport` directory. This directory exists only if your peer is an HTTP gateway. The directory is used to manage messages from other peers that are using this peer as a middleman. Remember, from Chapter 2, that peers behind a firewall need a gateway that stores incoming messages.

Additional directories can also show under `cm` or other subdirectories (`tmp`, `public`, and `private` show up under group directories, but are not used). In future versions of JXTA, a database could be used instead of the file system. For now, using the file system is adequate for many applications.

PSE Directory

Another directory that appears where you run your peer is `pse`. The directory contains certificates, password files, and other information related to peer security.

Be careful not to modify any of these files. You can delete the `pse` directory if you want to start your peer security from scratch, but be careful about doing this. In the future, this directory may contain other data for the application that should not be deleted.

When the peer configuration tool cannot find the `pse` directory, it creates it. The system then asks for your login and password to create the appropriate files and initialize the security system.

The current implementation uses the `pse` information for secure pipes. In addition to pipes, there are additional uses for this information, such as logging into groups and signing messages.

Overview of the JXTA Protocols JAVA API

The JXTA protocols are based on XML messages. Each message is an XML document. The XML document defines its part in the communication and the data of the communication. These XML messages are passed between the peers to convey information or are exchanged as part of a longer communication with queries and responses. The sequencing of the XML messages, and the rules under which they are sent, completes the protocol.

JXTA for Java takes the obvious route to implement JXTA by mapping XML to classes and adding management, control, and the ability to extend a base advertisement to a more complex one by inheritance. This sounds like the system is well thought out, but there were and are a lot of growing pains.

Summary of the API

There are several base services that need to be performed in a peer-to-peer system. These services include discovery, membership, and communications. The JXTA protocols further break communications into pipe binding, endpoint, and resolver protocols. There is also a peer information protocol, which is similar to a network ping except that it can have more information about the peer.

Peer Group Modules, Services, and Applications

Each peer group has a set of services. There is a core set that is usually implemented that covers the JXTA protocols. Each service is a module, which is like a mini executable. Applications can also be a type of module. The UML diagram in Figure 3.1 shows these interfaces, their relationships, and the methods to implement them.

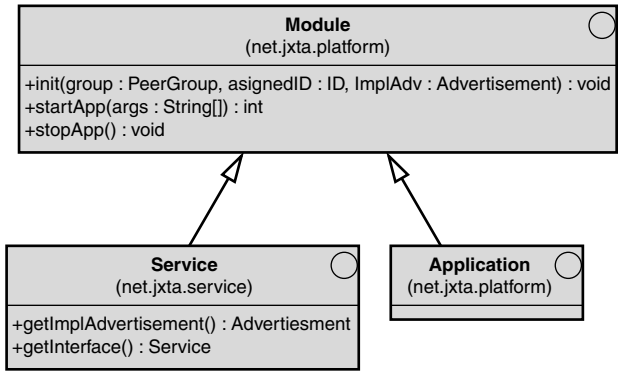

FIGURE 3.1 `Module`, `Service`, and `Application` interfaces.

The core services implemented by the reference platform are all derived from services. Each of these is displayed in Figure 3.2. Note that `MembershipService` is an abstract class and not an interface.

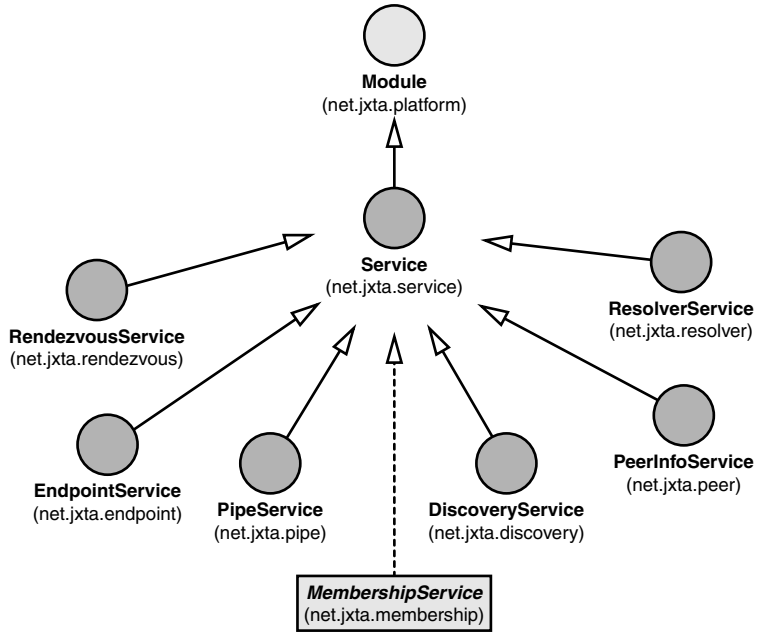

FIGURE 3.2 Core `PeerGroup` services and relationships to `Service` and `Module` interfaces.

Summary of Java API for JXTA Protocols

JXTA protocols are used to help peers discover, interact, and manage P2P applications. The protocols are not applications in themselves and require much more code to create something useful. The API hides a lot of detail about the P2P network, and its management that makes writing a JXTA application much easier than developing a P2P capable application from scratch. This section introduces the various APIs that we will discuss in detail later in the chapter.

Peer Discovery API

The peer discovery is an implementation of a searching mechanism with a local cache and the ability to forward requests.

The root of the discovery API is the `DiscoveryService` class. The `DiscoveryService` is obtained from the `PeerGroup` class, because discovery is always limited to its group.

Peer Resolver API

The resolver API is used by other APIs that need a request/response format. The resolver is accessed through the `ResolverService` interface.

The resolver should be thought of as a network-wide query. Instead of specifying a single peer, a group of peers is queried. An example of a use for this is the discovery service that passes a query to multiple rendezvous in search of answers.

Peer Information API

Peer information API is a way to request status information about a peer. The peer information API is accessed via the `PeerInfoService`.

Peer Membership Protocol

The peer membership API, like the discovery API, is only from the viewpoint and context of the peer group. The membership API is really in two parts—the membership authentication and credentialing. Credentialing is used in much of the messaging to prove that the peer is a valid member of the group, so the communications containing the credential are valid.

The membership protocol is accessed via the `MembershipService` abstract class. Note that this may be converted to an interface in a future version of the JXTA Java API.

Pipe Binding API

Pipe binding API is one of the more dynamic APIs. The reason is that the API is used for many different styles of pipe. The protocol is accessed via the `PipeService`.

It should be noted that the pipe service uses the resolver and the endpoint services. A module called `EndpointRouter` does the routing of pipes.

The `PipeService` interface does not define pipes, just the creation and management of pipes. Pipes are defined by implementing the `InputPipe` and `OutputPipe` interfaces.

Peer Endpoint API

The peer endpoint API is an API that is mostly invisible to JXTA application developers. The reason is that the endpoint API is really an implementation of a router. There are uncountable numbers of routers in use in corporations and the Internet that are just as invisible to the writers of browsers and other network software. However, this API can be used directly by applications that could be used to create applications that are more powerful. This API holds the key to accessing the transports available to other peer services, such as pipes and the resolver.

The key difference between a router and the endpoint router is that the routing is performed in the peer instead of a specialized piece of hardware and software. In the future, it is possible that there will be dedicated JXTA routers, but there is a great advantage to controlling your own destiny and routes. The API is probably less efficient than a dedicated router, but the endpoint router is built to route in some of the worst conditions caused by the mess of corporate LANs, firewalls, proxy servers, and NAT devices.

The router uses the resolver to query other peers for parts of the route. The endpoint protocols, such as TCP and HTTP, are defined and managed here too. The endpoint API is accessed with the `EndpointService` interface.

Where JXTA Applications Begin

JXTA applications need to be able to deal with the JXTA P2P network as the first thing they do. We call this *booting the peer platform*. This is very much like booting a computer on a network. The key difference here is that instead of a simple network, we are starting a peer in the JXTA network.

The JXTA platform is a group that implements the initial set of default behavior and protocols. The platform is also in the World peer group. The World group is the root of all other groups. Every peer is automatically added to the World group via the initialization of the platform.

Net Peer Group

The next thing that is done when starting a JXTA application is to load and join the Net peer group. The Net group is a specific group that is the default context for the peer. The World peer group has only limited capabilities and a very standard set of behavior. The Net peer group can be any group you desire.

The aim behind the separation of the root group is for supporting various devices. The JXTA's specification does not impose any kind of assumption about the peer capabilities, so they first join the basic world group (very easy to support) and then join other groups that may need to have more resources.

Most of the time, you will use the default Net group. The Net group is a placeholder to define a group if you need initial behavior that is different from the World group. The specific behavior includes different endpoints and knowledge of specific gateways or rendezvous. The Net group can also start monitoring services or even an initial application.

Another useful thing to do with a Net group is to specify a specific membership protocol. By placing the authentication in the Net group, you ensure that the peer is authenticated as soon as the platform boots. By becoming a valid member of the group, other peers can trust the peer. This trust can be used to prevent other peers from becoming a part of your P2P network. Normally this would only be done in a corporate network.

Starting peers in a group with secured membership is important for a P2P network that needs to be isolated. One reason for this is if all the cooperating peers are behind a firewall. Even though isolated, any other peer started behind the firewall could join. The new peer is free to interact with others on the corporate P2P. If the Net group does have a specific membership protocol, the peer booting into the network will be unable to interact with the other peers unless it joins with the specified membership.

Using a default membership may seem odd, but the simple fact is that all it takes is someone with a laptop and a JXTA peer running to compromise your P2P network.

The Peer

A peer is an identification of a specific instance of JXTA. The peer is associated with just about everything because it represents the identity of the platform. The concept of peer is similar to the way a computer is named on a LAN, except that the name is not guaranteed to be unique. As a way to make sure that peers are unique, there is a peer ID. The peer ID is generated like other IDs.

The concept of peer ID is used because there are multiple methods to reach a peer, so a fixed address or name is not that useful. Also, in the case of computers behind security barriers such as firewalls or NATs, the actual computer name or address is quite useless.

Peer ID

The peer ID is created during the initial configuration of the JXTA platform. The `Configurator` class creates a new `PeerAdvertisement` and calls the advertisement's `setPeerID` method with the ID built by the `IDFactory`. The following line shows how the factory is used. As you can see, the World group ID is used in the call. What this does is register the ID as a member of the World group:

```
IDFactory.newPeerID(PeerGroupID.worldPeerGroupID)
```

Remember that when a peer is a member of a group, the peer can be located via discovery by another peer in that group. All peers are members of the World group, so all peers can see each other from the World group discovery service. The reason that the world peer group is used is to associate the ID with the group to which it belongs.

Peer Classes

Peers are represented by a peer advertisement, which in the Java JXTA API is represented by the `PeerAdvertisement` class. In the Java implementation, the total implementation of a protocol is usually separated into two different packages. In this case, there is the `PeerAdvertisement` class that has methods that pretty much match the peer advertisement's XML specification. The `PeerAdv` class adds additional features used to complete the Java implementation. Figure 3.3 shows the `PeerAdvertisement`, its implementation, and associated ID classes.

NOTE

There are two different types of packages found in JXTA applications. These are an API and an implementation. The API is mostly abstract classes or interfaces. Some general functionality is provided in the API if the functionality is generic. Packages that are in the implementation have the platform specific code. Because the API is young, not all of the implementation is reflected by the API; so some parts of the implementation need to be accessed for some applications to work properly. In general, you should not need to use the implementation class explicitly unless it is a true extension of the API. This is true for the `GenericPeerGroup` class that adds the ability to look at child groups (discussed later in this chapter).

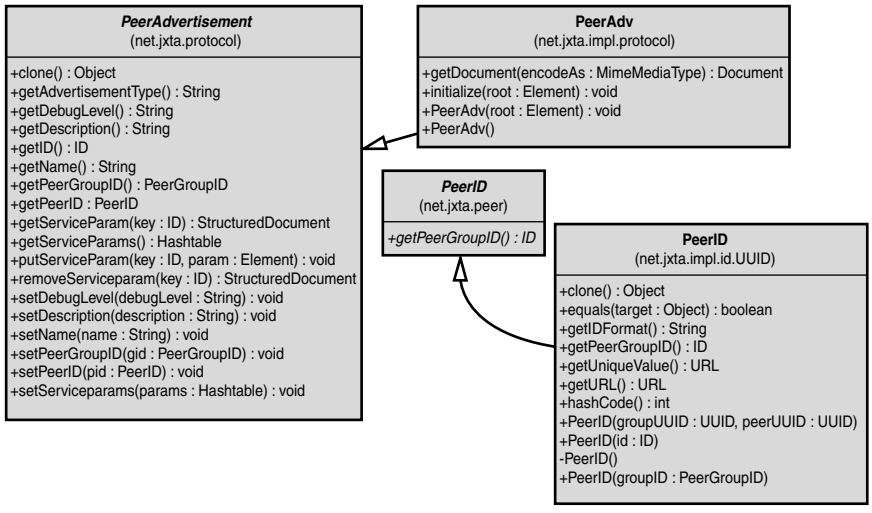

FIGURE 3.3 Peer classes used by JXTA applications.

Starting JXTA

Before we can start talking about any specific Java API for the JXTA protocols, you need to start JXTA. JXTA applications begin by joining the Net peer group. The Net group provides your default behavior and starts the JXTA services.

The following line of code is the usual way you start JXTA. The `PeerGroupFactory` is used to create a Net peer group. The Net group is a special group that is meant to be defaut or defined by a system administrator:

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

The variable `netPeerGroup` in this line represents a layering of two groups, the World group and the Net group. The services available are in the Net group but are mostly passed on to the World group. For example, if you use discovery services, the World peer group is used and all peers will be available.

Peer Discovery Protocol API

Peer discovery is one of the more important tasks performed by a JXTA application. Most of the time, the peers, groups, and other information is not known until a peer uses the discovery service. An example of this is the `shell` application. The `shell` discovers all of its information and has no starting information about the network.

Applications may have some built-in knowledge, such as well-known peers, peer groups, and other information. However, most peers know very little about what is available. A peer may also not know the peers or the data they manage until they are discovered.

In the Java API, advertisements that are discovered are stored locally. Because of the local cache, there are two types of discovery—local and remote. The remote discovery uses the resolver to find advertisements, while the local discovery uses the cache.

Classes in the Peer Discovery API

The discovery API is implemented as a service to peer groups. The API consists of the following key parts:

- `DiscoveryService`—This is the base interface that is used to access core functionality of the peer discovery protocol.
- `DiscoveryListener`—Listener interface used to wait for remote discovery messages.
- `DiscoveryEvent`—The event passed to the listener that contains information about the discovered advertisements.
- `DiscoveryResponseMsg`—The actual data payload that contains information about the discovered advertisements.
- `DiscoveryServiceImpl`—Implementation of the `DiscoveryService` interface.

A high-level UML class diagram of these classes is shown in Figure 3.4.

The discovery API uses other classes like `Cm`, which is an implementation of a content manager. The `Cm` class is used to manage JXTA advertisements discovered. `Cm` persists messages to search and retrieve advertisements. In addition, the discovery API uses other APIs implementing other JXTA protocols, such as the peer resolver protocol used to send discovery queries to other peers in the JXTA network.

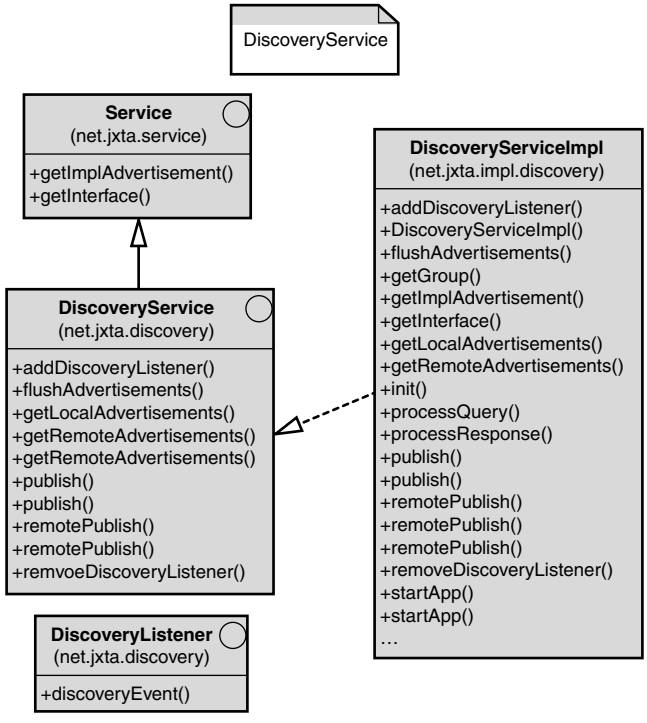

FIGURE 3.4 UML diagram of important classes in the Discovery Service API.

Accessing the Discovery Protocol

The discovery protocol takes place in the context of a peer group, so you need to have an instance of the `PeerGroup` interface. Peer groups are very important to discovery, because the group is a context that limits the scope of search. This means that if you are looking for peer advertisements, the only peers you should see are those that have joined your group. The same goes for subgroups, and all other advertisements created in the context of the group.

Now that you know that the group limits the discovery, you should understand that the World group is the first group you are given by the JXTA platform. The actual group you are in when you complete initialization of the platform it Net group, but World is the scope for discovery.

The Peer Group Interface

The peer group interface is a grab bag of methods that are associated with the information in the `PeerGroupAdvertisement` and other XML documents that describe features of the group. The interface is the primary access point to services. The services are either base services or services like the discovery service, which operate in the context of the group. In other words, if you access a service from a group, the service only works with peers in that group. This should be the preferred behavior of any service accessed from a specific group. The reason for isolation to the group is to limit the number of peers associated with actions enacted by peers that belong to the group.

The `PeerGroup` interface is a critical entry point to many of the services available in a peer group, so we will take a little more time to talk about the methods in the interface. Figure 3.5 shows the complete signature of the `PeerGroup` interface, and the following are summaries of the methods available in `PeerGroup` interface:

- `getLoader`—Returns the `JxtaLoader` object from which this peer group was created and launched. `JxtaLoader` is an extension of Java's `ClassLoader` class.
- `isRendezvous`—Returns a Boolean `true` if this instance of the group is a rendezvous peer.
- `getPeerGroupAdvertisement`—Returns the group's `PeerGroupAdvertisement`.
- `getPeerAdvertisement`—Returns the `PeerAdvertisement` for this peer as a member of the group.
- `lookupService`—Look up a service by its name ID. Returns the service registered by the name specified.
- `compatible`—Returns `true` if the given compatibility statement is loadable. This is required because a `PeerGroupAdvertisement` may be incompatible either by version or language that it supports.
- `loadModule`—Loads a given module. If the module is compatible and loadable, the object of the `Module` is created, initialized, and returned.
- `publishGroup`—Force publication of this group. Only useful if the group is being created from scratch and the `PeerGroup` advertisement has not been created beforehand. In such a case, the group must have been named and its description set before making the call.
- `newGroup`—Methods to create instantiate and initialize new groups.
- `getRendezVousService`—Returns the `RendezvousService` for the group.
- `getEndpointService`—Returns the `EndpointService` for this group.
- `getResolverService`—Returns the `ResolverService` for this group.

- `getDiscoveryService`—Returns the `DiscoveryService` object for this group.
- `getPeerInfoService`—Returns the `PeerInfoService` for this group.
- `getMembershipService`—Returns the `MembershipService` service for the group. Note that this is an instance of `NullMembershipService` or a customized implementation by default.
- `getPipeService`—Returns the `PipeService` for this group.
- `getPeerGroupID`—Gets the ID of this group.
- `getPeerID`—Gets the `PeerID` of this peer in the context of this group.
- `getPeerGroupName`—Gets the `Name` of this group.
- `getPeerName`—Gets the `Name` of this peer in this group.
- `getConfigAdvertisement`—Returns the configuration advertisement if any exist.
- `getAllPurposePeerGroupImplAdvertisement`—Gets an all-purpose `peerGroup ModuleImplAdvertisement` compatible with this group. The advertisement is initialized with all the data of a default peer group.

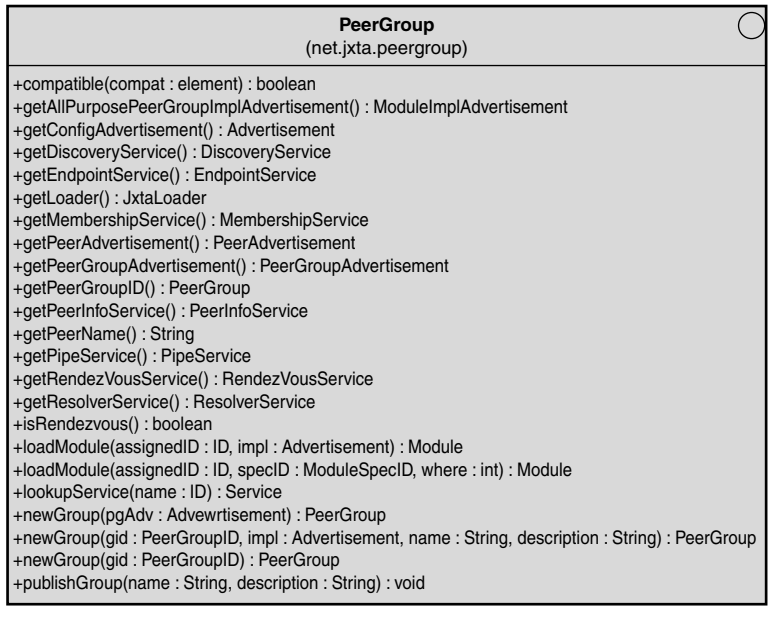

FIGURE 3.5 UML for PeerGroup interface.

Peer Group Inheritance

The classes and interfaces that make up the peer group API include several different layers used to isolate different aspects of the `PeerGroup` interface. Figure 3.6 shows the implementation and extensions of interface and classes to create the platform class.

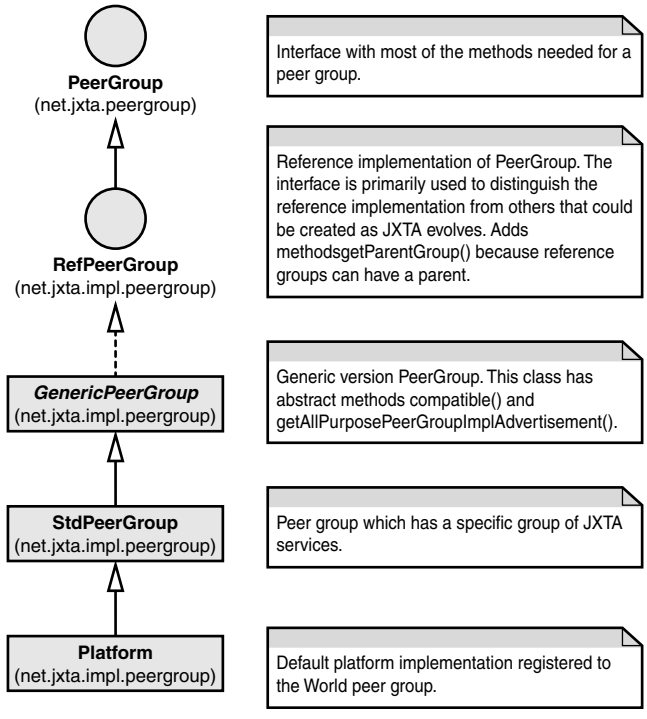

FIGURE 3.6 UML for PeerGroup interface.

When you first look at this stack, the reasons for so much abstraction is not immediately obvious. First, you have the PeerGroup interface, which gives you the core functionality as defined by the JXTA specification. The `RefPeerGroup` interface is primarily used to ensure that this implementation is isolated from any other. The primary reason you need this interface is to associate the reference implementation from any others that may be created. In addition, the interface adds the ability to discover a parent peer group if one exists.

The `GenericPeerGroup` is an abstract class that is used to further add to the reference platform for peer groups. In this class are additional methods used to ensure compatibility and add the ability to create a specific advertisement that is compatible with the reference group. This class also defines most of the static final constants used to access services.

The `StdPeerGroup` class is an abstraction with which you will most likely interact. The class implements most of the functionality required by a peer group.

The `Platform` class is the final concrete implementation of the PeerGroup and represents the World peer group. This class is instantiated as JXTA is started. The platform class is then used to for discovery and communications with other peers in the network. The platform group is also used as a parent to other groups.

Local Discovery

Local discovery is simply a method to search the locally cached advertisements in the `cm` directory. The following method of the `DiscoveryService` class does the work:

```
getLocalAdvertisements(int type, String attribute,String value)
```

The `getLocalAdvertisements` method will return an enumeration of base type `Advertisement`. The parameter type is an integer that corresponds to the constants in the `DiscoveryService` class shown in the following table.

Value	Name	Types of Document Returned
`0`	`PEER`	Peer advertisements
`1`	`GROUP`	Peer group advertisements
`2`	`ADV`	All other advertisements including peer and group

The document type constants are used for all of the following methods discussed for the `DiscoveryService` class.

The `attribute` parameter in `getLocalAdvertisement` is a string that matches a tag in the XML representing the `Advertisement`. Of course, the `value` parameter is the value of the contents between the tag, excluding leading and trailing spaces.

When the values for the attribute and the values are `null`, all of the specific type of advertisement will be retrieved.

The following is an example that searches for a peer named `"Mariann"`:

```
getLocalAdvertisements(DiscoveryService.PEER, "Name","Mariann")
```

The next example returns all advertisements (excluding Peer and Peer Group):

```
getLocalAdvertisements(DiscoveryService.ADV, null,nul)
```

The final example returns all the groups already known by the peer:

```
getLocalAdvertisements(DiscoveryService.GROUP, null,null)
```

Note that if there are no matches in the cache, an empty enumerator is returned.

One of the most important things to understand about local discovery is that it is based on remote discovery locating the advertisement first. If you don't run a remote search, you won't find any advertisements unless they are hardcoded by your implementation.

Advertisement Types

There are various advertisements returned. Figure 3.7 shows the many different types of advertisements and the classes that implement them. To access specific elements of these advertisements, you will need to cast the advertisement to the appropriate type. Note that there are API and implementation versions.

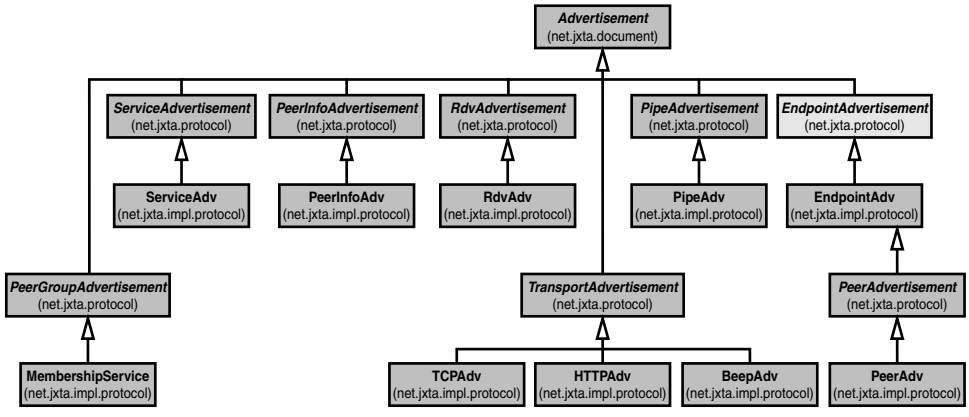

FIGURE 3.7 UML of `Advertisements` classes used in JXTA Java API.

Local Cache Created versus Remote Discovery

You should always run local discovery first. The reason is that remote discovery can take seconds to several minutes to locate advertisements specified by your search. By using the local cache, you make your applications run faster by avoiding network searches each time you need an advertisement.

Remote Discovery

Remote discovery is a process where one or more peers are queried for advertisements. Remote discovery is initiated by the `getRemoteAdvertisements` method found in `DiscoveryService`. The following method sends queries to rendezvous to search their local database for the advertisements that match the pattern specified:

```
int getRemoteAdvertisements( String peerid,
                             int type,
                             String attribute,
```

```
                                String value,
                                int threshold );
```

The parameters of this method are used as follows:

- `peerid`—A specific peer ID that created the advertisement. This parameter may be `null` to obtain advertisements from any peer.
- `type`—The type of advertisement (`PEER`, `GROUP`, `ADV`).
- `attribute`—An XML tag name that matches one in the advertisement. The attribute may be `null`, which selects all advertisements that match the type and the `peerid` parameters.
- `value`—The value of the contents of the tag enclosed by the tag specified by the attribute parameter. This value may be `null` if the attribute tag is `null`. If not `null`, the only advertisements returned are those that match the value.
- `threshold`—The upper limit of responses from one peer. This is an important variable, because the number of advertisements can be very large if the other parameters do not restrict the number of possible advertisements.

NOTE

The `threshold` parameter, in addition to limiting responses, is also a form of politeness in the P2P network. If the threshold is very high, you are utilizing a lot of resources from the rendezvous peer that is probably busy with other peers and its own operations. Remember that other peers share the discovery service.

This method is non-blocking. The method returns after threads begin the process of sending out queries to rendezvous. As answers are returned, the advertisements are stored in the local cache for later retrieval by the `getLocalAdvertisements` method.

This method is really only useful for populating the local cache periodically. The problem is that the peer really does not know when the advertisement has been found. The better method is the one described next that uses a listener to know exactly when an advertisement is found.

Remote Discovery with Listener

The next method is similar to the last, except that there is an additional parameter that passes in a listener. The `DiscoveryListener` interface is called each time a query response is returned from a rendezvous with an advertisement matching the other parameters in the method:

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

The `DiscoveryListener` interface has only one method `DiscoveryEvent`. The method is used to pass an `event` parameter of type `DiscoveryEvent`. You can see the complete spec for these in the UML diagram in Figure 3.8.

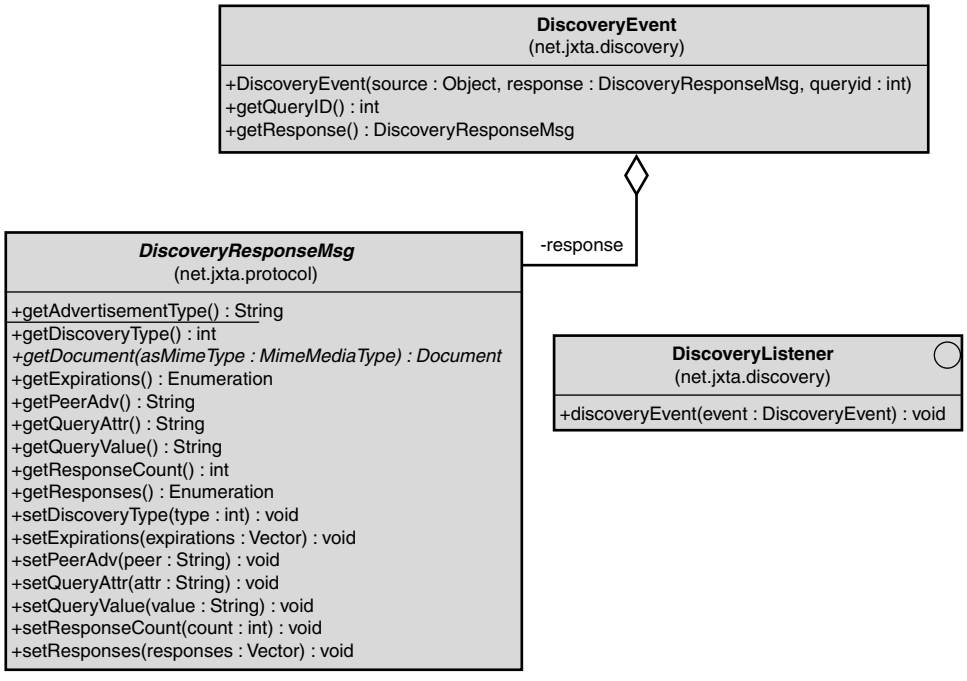

FIGURE 3.8 UML of `DiscoveryEvent`, `DiscoveryListener`, and `DiscoveryResponseMsg`.

The `DiscoveryEvent` object contains `DiscoveryResponseMsg`, which contains all of the advertisements returned from a specific rendezvous. Be careful not to call any of the set methods, because the methods are used for processing messages, not for manipulation by you. The only methods you should use in this class are the "get" methods. The `getResponses` method is the most important because it returns an

enumeration of the advertisements found. You can also get the original query information via `getQueryAttr`, `getQueryValue`, and `getDiscoveryType`. The advertisement returned by the `getPeerAdv` is for the rendezvous from which peer the advertisements originated. The `getDocument` and `getAdvertisementType` methods are of little value because they are just the XML the XML document type of the response message.

CAUTION

Do not use the setter methods of the `DiscoveryResponseMsg`. They are only to be used by the `DiscoveryService` when building the `DiscoveryResponseMsg` object. Using these methods does not guarantee changes to the XML representation of the advertisements returned by the `getResponses` method.

Advertisement Expirations

Advertisements have an expiration that is managed by JXTA. There are default expiration times, and you can override the lifetimes of advertisements when you publish with the `remotePublish` method and should not be set or overridden by other means.

Do not cache advertisements in application variables or use some other form of persistence, such as a database, to store advertisements. Only use the discovery mechanism and its cache. If you request an advertisement from the discovery cache when it has expired, you will not receive the advertisement in the results, and the current expired advertisement will be deleted. You should then rediscover the advertisement via the `getRemoteAdvertisement` method.

The publisher of an advertisement will automatically publish advertisements on a regular schedule as long as the peer is running. If the peer is shutdown, on restarting, it may publish the advertisements if the interval has expired. The only way to stop this is to flush the local version of the advertisement.

If a peer is not running, any advertisements that have propagated to other peers will eventually expire. If your peer is down longer than the expiration, the time to locate its published advertisements may be unnaturally long. If your peer is off for long periods, you should use a reasonable timeout when you first publish the advertisement.

The default expirations for locally created advertisements is one year in most cases, but some have longer defaults or none at all in the case of a peer advertisement. When advertisements are remotely published, the default is 2 hours, but this can vary depending on the value set during publishing. Note that you should not change the expiration value of the advertisement. You should set the expiration only via the `remotePublish` method.

Managing Advertisements

One of the critical management problems is handling advertisements that may be invalid. Part of the problem is that we just don't know when an advertisement is truly invalid. Either the peer or the member of a group is not currently on the JXTA network, or the advertisement is truly transient and the need for it is gone. The other problem that can occur is that by the time an advertisement is propagated to another peer, the need for it has expired or the peer that published it is already disconnected from the network.

To alleviate problems with invalid advertisement, you will need to manage how you treat a certain types of advertisements. You should start by avoiding communication with specific peers. This simply means that you write your application to rely on random peers in a group rather than a specific peer.

If you do need to contact a specific peer, you should create a mechanism to retry if there is a failure. In the case of multiple peers that are transient, you may want to use a specific peer that is more persistent to exchange information. For example, a set of PDA peers would use a set of relay peers to hold information until the target PDA is connected to the network. There are other possibilities; you just need to use a little imagination and some experimentation.

A big problem is a peer that is not operating or is disconnected from the network. You should probably characterize specific peers as being transient to avoid deleting advertisements that are probably still good. For example, a peer advertisement from a laptop is probably still valid, even though it cannot be contacted. Instead, the laptop may simply be disconnected from the network while moving to a new location. Your application should attempt to contact the laptop later.

If you are developing, try to use a very short lifetime for advertisements. You should also make sure that old advertisements are purged to ensure that your test peers are not using old data.

Removing or Flushing Old Advertisements

One of the important things to do in a JXTA application is housekeeping. The number of advertisements can get quite large if the P2P network is large. There is also the problem of invalid advertisements caused by development and testing. To ensure that the system is working correctly, you will need to delete advertisements. The mechanism is the `DiscoveryService` class with the following signature:

```
void flushAdvertisements( String id, int type ) throws IOException;
```

To use `flushAdvertisement`, you need to have both the ID and the type of advertisement. It probably seems logical that all you would need is the ID, but there is a good reason this is not true. Please remember that the content management system, used

to maintain advertisements, does so by the primary types of peer, peer group, and general advertisement. Consequently, you have to tell the system where to look for the advertisement by its type (`PEER`, `GROUP`, `ADV`), as well as the ID.

The best time to flush advertisements is when you have an advertisement that no longer works. You should be careful, because just because an advertisement fails does not mean that the advertisement is to blame. It is just as possible that the peers associated with the advertisement are not available. The good news, though, is that you can always search again for the advertisement. The bad news is that it may be the exact same one and the peer still not available. As a result, you may want to try an advertisement a few times over a set period before flushing the advertisement.

Remember also that advertisements have a built-in expiration. Expiration and the deletion is hidden, and there is no reason for you to flush advertisements if they are expired. Expiration and expected lifetime should still be considered when flushing advertisements. The reason is that those advertisements that are more volatile (shorter expirations) are more likely to be invalid. As a result, when dealing with pipes, you are more likely to get an invalid advertisement. Conversely, a peer group advertisement is less likely to be bad, because peer groups have a long lifetime.

Something else of note in this method is that the ID is a string and not a class. The reason for this is again related to the cache manager that stores advertisements by using the ID as a file name. In our example of this method, flushing the advertisement of a peer, we convert the peer ID to a string:

```
discovery.flushAdvertisements(padv.getPeerID().toString()
                              , Discovery.PEER);
```

Dealing with the Advertisements Discovered

One aspect about the way that JXTA was developed is that advertisements are not directly useable. To take advantage of an advertisement, you need to build another object that uses the advertisement as its initialization.

For each advertisement, you should find a class that accepts the ad in a constructor, `init`, or factory method. You will see many of these throughout this chapter and the rest of the book.

Also, not all of an advertisement is exposed by a get method. This is especially true for customized sections. To use the data, you need to call the `getDocument` method and retrieve the values of the tags that interest you.

Modifying Advertisements

Another slightly different aspect of advertisements is that they are not necessarily two-way. The fact that an advertisement has set methods does not guarantee that the

`getDocument` method will return an advertisement that is equivalent to the object. This may become true in the future, but for now, be sure to use the `AdvertisementFactory` class.

Group Services and Discovery

An important aspect of discovery is that because of its scooping of searches to its group, the items used by services in the group are also constrained by the group. What this means is that you should be able to search for any type of advertisement and expect that advertisement to be compatible to your group context and, thus, your services.

Finding Other Group Members

Another nice feature of group scoping is finding other users. When looking for advertisements of type `PEER`, you should only receive advertisements of peers that belong to the group from which you are searching. Be careful to remember that not all peers are visible at all times, and that sometimes peers change their name, ID, or both.

Peer Resolver Protocol API

The peer resolver protocol API can be thought of as rather misnamed. A resolver is classically defined for TCP/IP as a protocol for formatting requests to be sent to a domain name server to convert hostnames to an Internet address. However, a resolver is also defined as resolving a question, which is much closer to what the peer resolver protocol is actually used for. Simply, the JXTA resolver protocol is used to send a query to another peer and receive a response.

Simple P2P Messages, No Guarantees

The messages of the resolver are quite simple. They are also not guaranteed to reach their destinations nor are the results, if they exist, guaranteed to arrive back at the source of the query. The rendezvous may refuse or fail to transmit either message, or the answer may not exist. There is also no guarantee of an answer or even a notification that there is no answer.

Query Message

The query message is a standard wrapper of XML around a payload of information defined by the implementation. The implementation is specified by the handler name.

The wrapper contains a credential, the handler name, the source peer ID, a query ID, and the query. The DTD for the query is as follows:

```
<!ELEMENT ResolverQuery  (Credential,
                          HandlerName,
                          SrcPeerID,
                          QueryID,
                          Query)>

<!ELEMENT Credential  #PCDATA>
<!ELEMENT HandlerName #PCDATA>
<!ELEMENT SrcPeerID   #PCDATA>
<!ELEMENT QueryID     #PCDATA>
<!ELEMENT Query       #PCDATA>
```

Response Message

The response message is very similar to the query. The differences are in the credential and the response. The credential is the credential of the responding peer. Like the query message, the credential is the one created when the response peer joined the group. The credential could be checked to verify that the peer that answered the message was a valid member of the group.

The query ID in the response is identical to the query. This allows you to quickly match a query to a response if you perform multiple queries. Also, because queries can arrive out of order and may be repeated, the query ID can be used to order responses and consume duplicate messages.

The resolver finds your query handler using the specified handler name. The handler name should be the same as the query handler name. The handler interface has both a process query and a process response. The duality of the interface, and a single name, help promote the idea that all peers participate in both queries and responses.

Like the query, the payload is in a tag, this time called `Response`. The DTD for the XML is as follows:

```
<!ELEMENT ResolverResponse (Credential,
                            HandlerName,
                            QueryID,
                            Response)>

<!ELEMENT Credential  #PCDATA>
<!ELEMENT HandlerName #PCDATA>
```

```
<!ELEMENT QueryID     #PCDATA>
<!ELEMENT Response    #PCDATA>
```

Message Security

The credential is not a guarantee that the response message is from a valid peer. It would be very easy to copy a credential and masquerade as the response peer. To ensure that the message is not counterfeit, the message should either be encrypted or signed in a way that the signature contains verification of the peer, the message ID, and that the data has not been modified.

Resolver API Classes

The UML for key resolver classes can be seen in Figure 3.9. The classes and interfaces are as follows:

- `ResolverInterface`—A class that implements the resolver management. This class is the focal point of functionality for sending queries and distributing responses. The class implements the `ResolverService` and the `GenericResolver` interfaces.
- `ResolverService`—Interface that defines the interface to register and unregister query handler classes. The interface also extends the `GenericResolver` interface, which completes the resolver service by defining the methods used for sending queries and sending responses.
- `GenericResolver`—Interface that defines the methods used for sending queries and sending responses.
- `QueryHandler`—This is the interface to implement to handle queries and responses to queries.
- `ResolverQueryMsg`—This is the message that is sent to other peers for processing. The `ResolverQuery` is the default implementation and extends `ResolverQueryMsg`.
- `ResolverResponseMsg`—This is the message that is sent to other peers for processing. The `ResolverResponse` is the default implementation and extends `ResolverResponseMsg`.

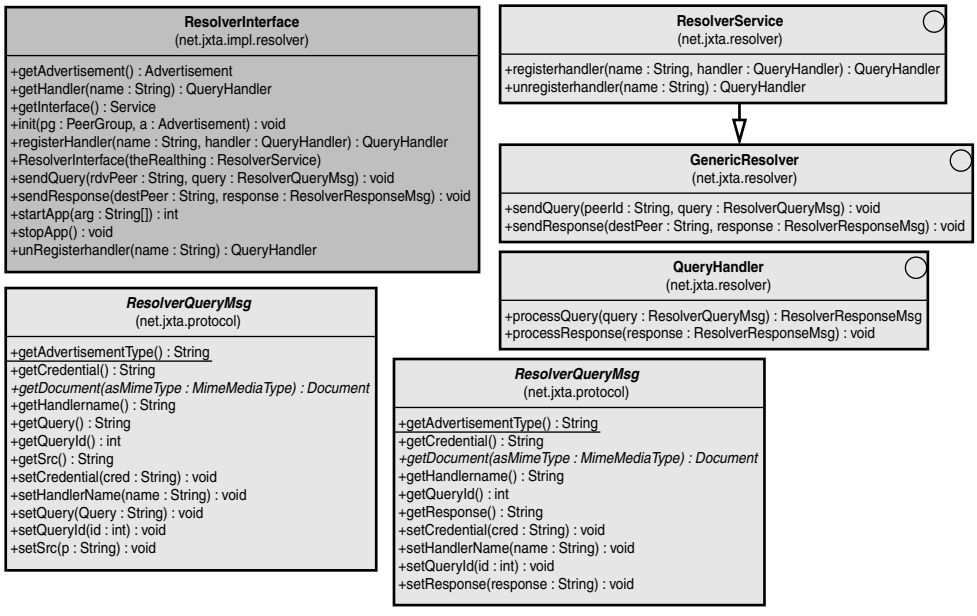

FIGURE 3.9 Key classes in the Peer Resolver API.

Coding and Starting the Handler

To create the handler, you simply implement the `QueryHandler` interface. The interface only has two methods—`processQuery` and `processResponse`. The `processQuery` method is called when a peer receives a query and `processResponse` is called when the query returns.

The `processQuery` Method

The `processQuery` method is, as we have just pointed out, used by a peer to process a query message. The method is called from within the resolver after the query is received. The query message is passed to this method for processing.

The `processQuery` method is used called on the responder peer. In other words, this method is only called if you receive a query message from another peer. The process response method of this interface is only called when a response is received from a peer you have queried.

The return from the method is the message to be sent back to the peer that asked the question. The signature of the call is as follows:

```
ResolverResponseMsg processQuery(ResolverQueryMsg query)
                                 throws NoResponseException,
                                        ResendQueryException,
                                        DiscardQueryException;
```

The exceptions thrown are quite important, because they result in very specific behavior of the resolver. Depending on the error, `processQuery` method throws `NoResponseException`, `ResendQueryException`, or a `DiscardQueryException`. The following are the exceptions that you can throw from the method and what the resolver will do in response:

- `NoResponseException`—Throw when you do not have a response, but the response peer is interested in getting an answer for itself. The resolver system will propagate the query and resend it. Note that the query will only be propagated if the peer is a rendezvous.
- `ResendQueryException`—Causes the resolver to resend the query.
- `DiscardQueryException`—The query is discarded and not forwarded from the peer. Remember that the response peer is not required to return a response.

NOTE

Java veterans may notice that the `processQuery` exceptions are used as a form of logic control. Expect this to change because the cost of processing an exception is very high.

Traversing a try...catch block costs nothing in most JVM implementations. When an exception is thrown, there are several operations that involve manipulations and examination of the stack, lookups of catch locations, and catch types. Although simple to use, exception handling is very expensive. Exceptions, which should be rare errors, are rarely seen, so the cost is for a single exception is not noticeable. This is not the case for processQuery method.

Look for this method to change in a future version.

Processing Responses

The `processResponse` method in the `QueryHandler` interface is a little simpler than the `processQuery` method because this is a termination point of the query. There are no exceptions to throw to drive resolver behavior. However, because this is a handler, be careful about allowing an exception to be thrown here. By throwing an exception, you can cause the resolver to cease functioning:

```
void processResponse(ResolverResponseMsg response);
```

Example `QueryHandler`

The query handler has two methods to be implemented. In Listing 3.1, the `processQuery` method adds a time stamp to the payload of the query message. The `processResponse` method simply prints the message that was received. Because you always have an answer (the time stamp) you never throw an exception.

LISTING 3.1 Sample implementation of a `QueryHandler`

```
class TestQueryHandler implements QueryHandler{
   protected String handlerName;
   protected String credential;
   protected SimpleDateFormat format   = new SimpleDateFormat (
                                         "MM, dd, yyyy hh:mm:ss.S");

   public TestQueryHandler(String handlerName, String credential){
      this.handlerName = handlerName;
      this.credential = credential;
   }// end of constructor TestQueryHandler()

   public void processResponse(ResolverResponseMsg response) {
      System.out.println("Received a response");
         String textDoc = response.getResponse();
         System.out.println(textDoc);
      }// end of processResponse()

   public ResolverResponseMsg processQuery(ResolverQueryMsg query)
                                        throws NoResponseException,
                                               ResendQueryException,
                                               DiscardQueryException,
                                               IOException {
      System.out.println("Received a query");
      System.out.println(((ResolverQuery)query).toString());
      StructuredTextDocument doc = null;
      String textDoc = query.getQuery();
      doc = (StructuredTextDocument)StructuredDocumentFactory
            .newStructuredDocument
            ( new MimeMediaType( "text/xml" )
            ,new ByteArrayInputStream(textDoc.getBytes()) );

      // Use the original payload and add the time
      Element e = null;
      long now = System.currentTimeMillis();
```

LISTING 3.1 Continued

```
      e = doc.createElement("timestamp 2",format.format( new Date(now)));
      doc.appendChild(e);
      // Return a generic response;
      ResolverResponseMsg message = null;
      String xml = serializeDoc(doc);
      message = new ResolverResponse( handlerName
                                    , credential
                                    , query.getQueryId()
                                    , xml);
      return (ResolverResponseMsg)message;
   }// end of processQuery()
}// end of class TestQueryHandler
```

Accessing and Setup of the Resolver

The resolver is found in your group. It is simple to just call the `getResolverService` method of the peer group you are using. The return type is a `ResolverService`, but actually you get an instance of `ResolverServiceImpl`. `ResolverServiceImpl` contains the actual methods used to submit the queries. The `ResolverService` interface specifies the adding and removal of query handlers.

The `GenericResolver` interface defines the methods for sending queries and responses. In `GenericResolver`, you really only use the method to initiate queries, while the method for sending responses is called by the internals of the resolver in response to the `QueryHander.processQuery` method:

```
ResolverServiceImpl resolver;
resolver = (ResolverServiceImpl)group.getResolverService();
TestQueryHandler handler = new  TestQueryHandler(handlerName,credential);
resolver.registerHandler(handlerName, handler);
```

Getting Ready—Finding a Rendezvous Peer

The resolver only operates in terms of rendezvous peers. Queries must begin at rendezvous. If your current peer is configured as a rendezvous, there is no problem. If your peer is not a rendezvous, you will need to know about a rendezvous.

Note that there is no guarantee that a specific peer will answer. The peer that answers is the peer that actually sends the response. If the specified rendezvous peer has the answer, it will reply and take no further action. If the peer does not have an answer and throws `NoResponseException` or `ResendQueryException`, the query is forwarded to another rendezvous for processing.

One way to look at this is to imagine a group of 100 people. Among the group are 10 areas that are rendezvous for each of 10 people. One person at the rendezvous is responsible for communicating with the other ten rendezvous. We will call these peers *rendezvous managers*. Peer 1 asks rendezvous manager 1 a question. If any of the other eight peers have communicated the answer or the manager knows the answer, it returns the response to peer 1. If manager 1 does not know the answer, it contacts all the rendezvous managers it knows about for an answer.

The manager does not need to know about all of the other managers, just enough that know of others, and so on. In other words, if there is a chain of relationships between the managers that connects all ten, all rendezvous are visible to the first peer. The topology here is that of the small world effect we talked about in Chapter 1, "What Is P2P."

Using the Resolver

Using the resolver is fairly simple, but there are several steps. Listing 3.2 covers just about everything you need to send a query.

NOTE

The code in Listing 3.2 is from a simple test of the resolver that can be found online at `www.sampspublishing.com`.

Create a document with the time, serialize that into an XML document, create a `ResolverQuery` object, and send it. As you can see, most of the effort is dedicated to building the message while the actual messaging is very simplistic.

LISTING 3.2 Example Used to Start a Query

```
// Create a document with the time
StructuredTextDocument doc = null;
doc = (StructuredTextDocument)
      StructuredDocumentFactory.newStructuredDocument(
         new MimeMediaType("text/xml"),"Pong");

Element e = null;

long now = System.currentTimeMillis();

e = doc.createElement("timestamp 1", format.format( new Date(now)) );
doc.appendChild(e);
String credential = "Sams";
// Create the message;
```

LISTING 3.2 Continued

```
ResolverQueryMsg message = null;
String xml = serializeDoc(doc);
message = new ResolverQuery( handlerName
                           , credential
                           , group.getPeerID().toString()
                           , xml
                           , 1);
System.out.println("Sending query");
// Note that the following may throw a RuntimeException
// if the peer is not found.
resolver.sendQuery(peerID, message);
```

Processing Responses

The rendezvous peer can operate in several different ways. First, the rendezvous can be given answers by other peers. This is what happens in the discovery process. When a peer does a remote publish of an advertisement, it publishes it to a rendezvous. Any remote query for advertisements in the discovery mechanism causes each rendezvous to look to see if it has that advertisement. If there is no copy of the advertisement, the query is forwarded to other peers.

The query could also cause the rendezvous to contact peers that are not rendezvous. This would occur in the handler. The peers that the rendezvous knows about could be contacted to get an answer. This rendezvous to peer mechanism takes place outside of the resolver mechanism. If none of these peers has an answer, the rendezvous can forward the query to another rendezvous that will repeat the process.

Remember that there is no direct addressing guaranteed in this process. You should use pipes if you want to get a response from a specific peer.

Removing a Handler

When you are no longer interested in receiving answers to queries, unregister your handler. This is very important to do as soon as possible because you could waste time on processing an answer multiple times. The following is the signature to unregister the handler. It simply takes the name of the handler you specified when you registered it:

```
QueryHandler unregisterHandler( String name )
```

Peer Information Protocol

The peer information protocol is a specific implementation of a peer resolver query. When a peer is first initialized, it publishes its `PeerAdvertisement` that is picked up by at least one rendezvous. When a request for a specific peer's information is requested, a query is made to locate the `PeerAdvertisement`. Recall that the `PeerAdvertisement` contains endpoint information for contacting the peer. Using the endpoint, the peer is contacted directly to obtain its peer information.

`PeerInfoService`

The `PeerInfoService` interface is very simple and very complete. To compare it to the `ResolverService`, the interface has more methods, but this is because the behavior is more constrained. The only actions are to locate peer information and manage information. You may recognize some of the functionality as being similar to the `DiscoveryService`. The two services are very similar and manage data in the same way using the content manager to store advertisements. In the `PeerInfoService`, the only advertisement that is cached is the `PeerInfoAdvertisement`. The UML for this advertisement is shown in Figure 3.10.

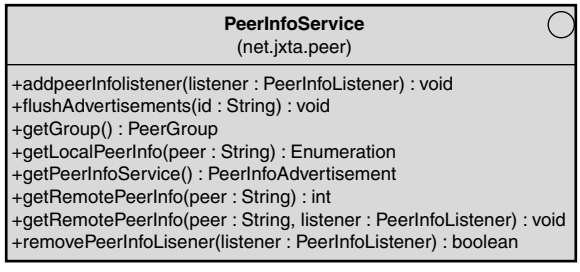

FIGURE 3.10 UML for `PeerInfoService`.

The `PeerInfoAdvertisement`

The `PeerInfoAdvertisement` is a bit different from the other advertisements in JXTA because it includes volatile data like uptime. The UML for the advertisement is shown in Figure 3.11.

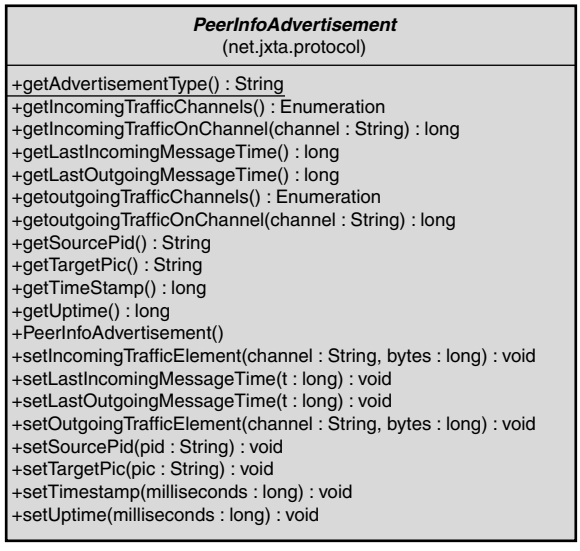

FIGURE 3.11 UML for `PeerInfoAdvertisement`.

The `PeerInfoListener`

The acquisition of a peer's info will vary according to how the information is finally accessed. Because you cannot predict the time it takes to get the peer info, the API uses a listener to notify the application when the information is available.

The interface of the listener is quite simple, with only one method. The `PeerInfoResponse` method passes a `PeerInfoEvent` to the listener. The `PeerInfoEvent` holds the peer info advertisement retrieved. The UML diagram in Figure 3.12 shows the specification for both the `PeerInfoListener` interface and the `PeerInfoEvent`.

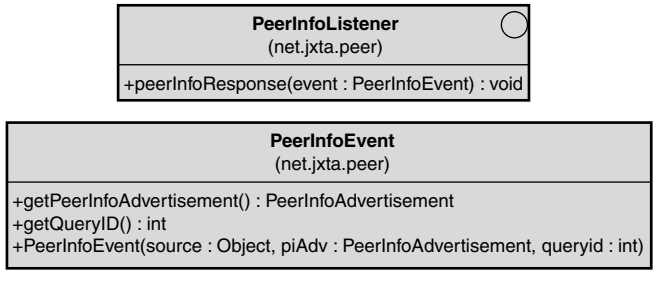

FIGURE 3.12 UML `PeerInfoListener` and `PeerInfoEvent`.

Flushing Advertisements

`PeerInfoAdvertisement`, like other advertisements, will automatically expire. However, you may want to remove peer info advertisements on a regular basis to ensure that the information is up to date. You should also remove the advertisements if your application accesses multiple peers because that may impact memory.

Why Use the Peer Info Protocol?

Of all the protocols, peer info probably seems the least useful in its current implentation. The protocol can help gauge the health of the network with uptime statistics and other information about messages. Overall, the protocol can provide interesting information, but not a lot.

You could write your own implementation of the protocol and use it in your custom groups. By implementing your own, the amount and richness of the data can be made to fit your requirements. The API may not lend its self to easy extension, so you are probably better off creating your own protocol.

Peer Membership Protocol

The peer membership protocol is a mechanism for joining peer groups. The protocol should not be confused with group creation and management except that when you create a group, you specify a specific implementation of this protocol.

What the membership protocol does is impose and verify specific requirements for a peer to join a group. In other words, the protocol makes sure you give the right answers to questions that identifies you as a valid group candidate before you are allowed to join a group. Alternatively, the implementation could look up other information or even ask other peers to vote. The possibilities are endless, and the protocol works as a framework to begin the process and allow the user to join once the requirements are met.

After you have successfully joined a group, you are issued a credential. The credential is an XML document that is used as a token of proof that you are a member of the group. The membership protocol does not keep track of users that have joined. In a peer environment, it is impractical to use a server for validating membership. The credential serves as a way to allow peers to recognize each other as valid members of the same peer group (see Chapter 6, “Working with Groups,” for a complete example of peer membership).

Membership Service

The `MembershipService` class is an abstract class that defines generic methods used for a concrete membership service class. Figure 3.13 shows the UML for this abstract class. This class is described in detail in Chapter 6 but the following is a short description of the methods:

- `getName`—Returns the name of the service. The name is often used to match the membership service with its authenticator.
- `getInterface`—Returns this object as a service.
- `apply`—Request the necessary `Authenticator` object to join the group based on a given policy. An `AuthenticationCredential` is provided to help select what version of `Authenticator` should be used and/or to initialize an `Authenticator` object. Returns an `Authenticator` object associated with the join. The `Authenticator` object has setter methods that must be properly set so that the authenticator returns a valid state to allow the peer into the group.
- `join`—Join the group. A valid `Authenticator` object from the apply method is provided. The result is a `Credential` object that can be used when needed to prove the peer is valid.
- `resign`—Method called to resign a peer from the group.
- `getCurrentCredentials`—Returns an enumeration of current credentials that identify the peer.
- `getAuthCredentials`—Returns an enumeration of the current `AuthenticationCredential` objects.
- `makeCredential`—Helper method used to create a credential from an XML element. This can be used to recreate a credential that was stored as XML.

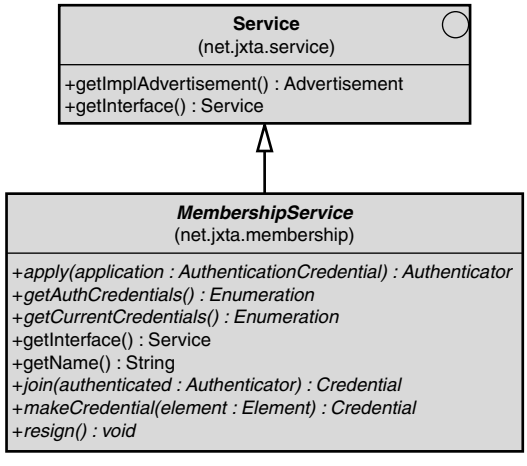

FIGURE 3.13 UML for `MembershipService` interface.

AuthenticationCredential

`AuthenticationCredential` provides the authentication method and an identity. In the UML diagram in Figure 3.14, the `getMethod` method of the `Authentication` class returns a string that is used to look up the authenticator. The identity information is derived by the implementation of the `Credential` interface methods.

One way to look at the `AuthenticationCredential` is as an initial introduction. Because of the object passed via a join, the membership service knows the peer, its current group, specific credential information, and the method with which the peer is attempting to join.

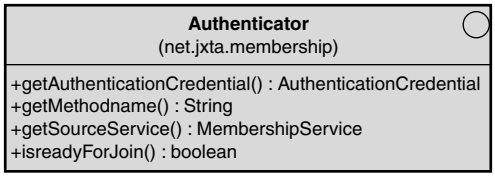

FIGURE 3.14 UML for `AuthenticationCredential` class.

Credential

After you have joined a group, you are issued a credential. The credential implements the credential interface that was shown in Figure 3.14. The credential is a custom implementation that can hold some type of data that is used to identify the peer to the group. Usually, the basic implementation is used because many groups are not concerned with security.

Null Membership Service

The UML in Figure 3.15 is an implementation of the Null membership service. The classes `NullMembershipService`, `NullCredential`, and `NullAuthenticator` are based on `MembershipService`, `Credential`, and `AuthenticationCredential`, respectively. The Null membership is the default membership for the world peer group and any sub-group unless specifically overridden by adding a custom membership service class.

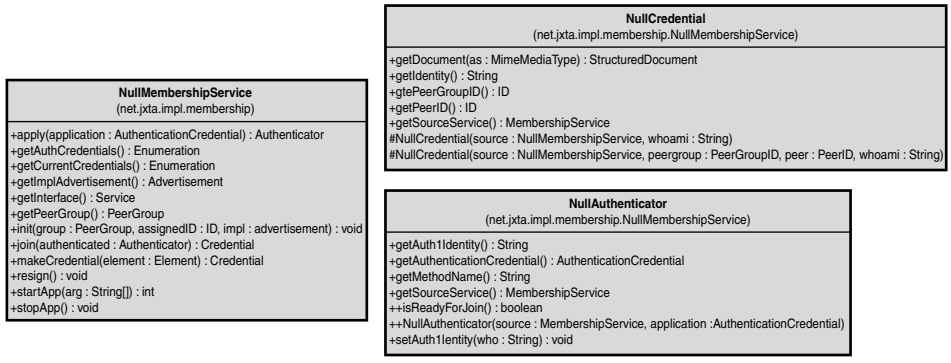

FIGURE 3.15 UML for `NullMembershipService`, `NullCredential`, and `NullAuthenticator` classes.

The Join Process

Joining a group via the membership protocol is a multi-step process. The steps are as follows:

1. Obtain or create an initial `AuthenticationCredential` object. The `AuthenticationCredential` constructor takes a credential document, the peer group, and the authentication method that the group uses.
2. Get the membership service from the group. Calling the `getMembershipService` method of the peer group you are going to join does this.
3. Call the apply method of the membership service with your authentication credential. This primes the membership service for step 5.
4. Fill in the appropriate data in the authenticator object.
5. Test the authenticator. You call the `isReadyForJoin` method of the authenticator and, if true, you can safely join the group.
6. Call the membership service method `join`.

The result of the join is a final credential that you can pass to other peers in messages. The group also contains this credential that will automatically be added to standard XML messages that require a credential.

Joining a Group

There are two different ways to join groups—via a null authenticator and with a custom group.

Null authentication is the default authenticator for the World group and the default for any new group. Null authentication does not care who you are and performs

only token processing to start the peer group normally. The following is a very simple join:

```
MembershipService membership;
membership = (MembershipService) newGroup.getMembershipService();
credential= membership.join( null );
```

Joining with a custom group is a little more difficult. The first step is to obtain the actual membership service from your group:

```
MembershipService membership;
membership = (MembershipService) newGroup.getMembershipService();
```

Now you create an authentication credential. The following example uses the group you are joining, the authentication method name, and an initial credential:

```
AuthenticationCredential authCred;
authCred = new AuthenticationCredential( newGroup
                                       , authenticationMethod
                                       , credentials );
```

Note that the authentication method string needs to match an authentication method supported by the membership service. The reason for specifying by name is so that different authenticators can be used in diffeereent languages.

Next, you obtain an authenticator by applying to the group. What this does is create an authenticator that is based on your credentials, the group, and the type of authentication you are requesting. Note that you could have multiple authentication methods and credentials represented by authentication credentials. In other words, an insurance group could be made up of patients and doctors. Patients could use a different credential and authentication method than a doctor who may need a different level of identification and processing. Here is an example of the `apply` method:

```
Authenticator authenticator = (Authenticator)membership.apply( authCred );
```

At this point, you now have an object that implements the `Authenticator` interface. The object is going to have specific methods to configure the authenticator that are specific to the requirements of the group, the method of authentication, and the initial credentials. The settings may be as simple as the username to a series of questions to a name and password.

The `Authenticator` interface includes a method called `isReadyForJoin`. The method is implemented to return `true` if the state of the custom data represents the requirements for a valid join. The following lines check the authenticator to see if it is valid. If you pass, you go ahead and join the group. The `isReadyForJoin` is also called during join. Make sure you check prior to joining to ensure proper handling of an invalid authenticator object:

```
if( authenticator.isReadyForJoin() ) {
   finalCredential= membership.join( authenticator );
}
```

If the `join` method fails because the authenticator returns `false` for the `isReadyForJoin` method or for some other reason, the `PeerGroupException` is thrown.

Final Credential

Notice in the prior example that the return value from the join is a custom implementation of a `Credential` object. The credential is used for all communications with other peers to prove that you have joined the group. The credential is used by the authenticator service to ensure that only valid peer group members can use services.

Renewing Membership

The protocol specification states that the membership protocol includes the ability to renew a membership. The initial version of the Java API does not include the ability to renew a membership. However, you can write your custom authentication credential that specifies a renewal authentication method and accept the final credential that was created after you joined the group.

Resigning Membership

Resignation from a group is simple. Just call the `resign` method of the `MembershipService` object. The credentials used to join the group are destroyed as a part of this process:

```
membership.resign();
```

Pipe Binding Protocol API

The pipe binding protocol is a mix of capabilities that includes the routing of connections and the interfaces for communicating between peers. In other words, the pipe binding protocol describes how we create connections between peers and how we send information.

The Pipe Binding API, as opposed to the JXTA protocol specification, goes a step further by adding the programmatic structure for your application to make connections to send and receive messages. In addition, the API works as a framework for extension to create different types of pipes and different control models.

What Are Pipes?

Pipes can be compared to sockets or streams, in other words, just a channel between computers to transfer data. The difference is that you assume that the P2P network

has a few things different from a normal network in that you assume that the connection is not necessarily direct and that there are many protocols that can be used. The *pipe binder* is a layer above the messy network that hides everything from the developer and adds capabilities that the network does not have via the base protocols alone.

One example of additional capabilities is the ability to cross a firewall. It is rather simple to use HTTP to cross a firewall to connect a server to a client. However, what about connecting two peers, each behind their own firewalls? Suddenly the problem is a little more difficult. Now imagine connecting a dozen clients together at the same time with each one behind its own firewall. Now the problems are almost impossible by normal means. But, by hiding the complexity in the protocol, along with using other JXTA protocols like the peer resolver and peer endpoint router, even the most convoluted interconnection of peers can be simply accessed with a simple pipe abstraction.

So, think of a pipe API as a second cousin to Java's stream API. There are a few differences, but many of the patterns you would normally use are the same. There are also a few extra functions to make our lives easier in a P2P network.

From our prior discussions, remember that there are different types of pipes. These can be asynchronous, encrypted, and other types. The API allows us to select the type we are requesting. Note that the default type of pipe is asynchronous and unidirectional.

The Pipe Service

The pipe service is obtained from your group via the following in the `PeerGroup` class:

```
PipeService getPipeService();
```

The class that is returned implements the `PipeService` interface. The UML diagram in Figure 3.16 shows the interface and the methods it contains. There are three types of methods in the interface—input pipe creation, output pipe creation, and a method to remove output pipe creation listeners (`OutputPipeListener`).

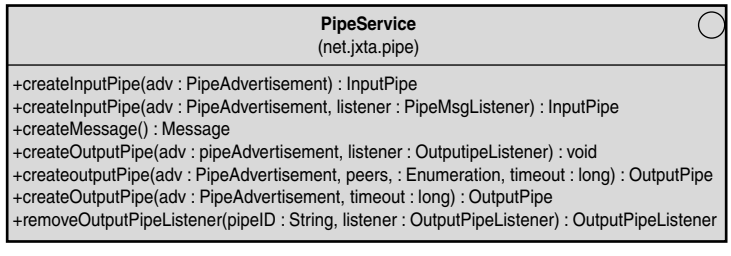

FIGURE 3.16 UML of `PipeService` interface.

PipeAdvertisement

Both input and output pipe creation methods use a pipe advertisement to create the pipe. The type of pipe is specified in the advertisement. The UML for the `PipeAdvertisement` is shown in Figure 3.17.

A pipe advertisement contains several pieces of information that will be used to find the pipe and to create the pipe. Specific implementations of the advertisement will contain additional information if required.

To find a pipe, the name is usually used, but the ID ensures that you have found a specific pipe.

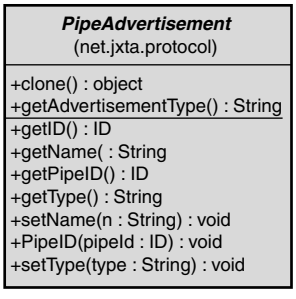

FIGURE 3.17 UML of `PipeAdvertisement` class.

The type of pipe is the most important for creation. There are many different types, and the initially supported types are propagate, secure, and non-blocking (these pipe types are explained fully in Chapter 2).

InputPipe

The `InputPipe` interface (shown in Figure 3.18) has a `close` method and two methods for receiving messages. The `poll` method waits for messages up to a certain timeout value. The `waitForMessage` method will wait indefinitely for a message.

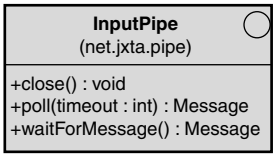

FIGURE 3.18 UML of `InputPipe` interface.

As you might guess, waiting for messages and polling for them will take a bit of work and possibly threads to process messages appropriately. The good news is that you

can use the `PipeMsgListener` to listen to messages. We will cover `PipeMsgListener` later in this section.

`OutputPipe`

The `OutputPipe` interface, shown in Figure 3.19, is simpler as the `InputPipe` interface. The only two methods are `close` and `send`.

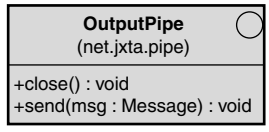

FIGURE 3.19 UML of `OutputPipe` interface.

One Way Pipes?

The initial representation of pipes are one way. The direction of a pipe flows from an output pipe on one peer to an input pipe on another peer. There are implementations of two way pipes, but they are really a wrapper around two pipes created in opposite directions.

The reason for pipes being one way is because of the nature of the P2P network. If the only pathway is via HTTP, full bi-directional capability is not easily created. Remember that HTTP is a response/request protocol. In fact, any bi-directional HTTP pipe is, by definition, a simulation.

Because many peers will be behind firewalls or other barriers, the most logical medium is HTTP. By creating an underlying protocol that can always be deconstructed into messages, the designers have ensured that most peers can be accessed. This also means that you can communicate to multiple peers using the same messages but different protocols. This may seem inefficient, but connectivity was preferred over speed.

The Pipe Process

Pipes are created in a specific order. The process is that the listener must open an input pipe before the talker creates an output pipe. If there is no input pipe available, an output pipe will fail because it cannot connect. This is very similar to a socket that needs to have a listener opened before another machine can connect to it.

The following is a sequence of events for a peer that waits for others to communicate with it:

1. Group advertises pipe advertisement.
2. Listener peer creates an input pipe from advertisement.

3. Talk peer creates output pipe addressed to the listener peer.
4. Talk peer sends message on pipe.
5. Listener peer receives message.

The following is the opposite:

1. Peer wanting information opens an input pipe.
2. Peer wanting information advertises the pipe.
3. Peer with data opens an output pipe to the input pipe.
4. Peer with output pipe sends data.

Connecting Pipes

There are two ways to connect pipes. First is a blind pipe and the second is a peer-addressed pipe. A blind pipe is no different from a stream opened on a port to accept the first caller to a server. A listener pipe is always blind and will accept a connection from any peer. Output pipes can be both blind and specifically addressed.

You may at first be uncomfortable with blind pipes, but they are not different from how we dealt with sockets in the traditional client server world. The one difference is that the resources are group based and not peer based in P2P. The peer you connect to in many P2P applications is not important. For example, a group of peers sharing storage or processing would not care who connected to what peer, only that they connected to a peer within the group that guarantees a resource.

Blind Output Pipe

Calling the following method in the pipe service creates a blind output pipe. The parameters are only the advertisement and a timeout that defines how long to wait for a connection. The timeout is the number of milliseconds to wait or, if `-1`, to wait forever:

```
OutputPipe createOutputPipe ( PipeAdvertisement adv
                            , long timeout)throws IOException;
```

This method will block until the pipe connects to an input pipe or the timeout occurs. If you look at the implementation, actual behavior will depend on the type of pipe in the pipe advertisement. For example, if the type is unspecified or if the advertisement type is `PipeService.UnicastType`, a non-blocking unicast pipe is created. Because there is no endpoint specified, the system just looks for the same pipe advertisement published by another peer and attempts a connection.

If the type of pipe is a propagate pipe, all peers in the group that have open input propagate pipes will simply listen for messages because you are not specifying a peer.

Propagate is much more a construct than a physical connection like the unicast pipe. There is not a specific search for peers with the same pipe, because messages are delivered by routing and not by a specific connection. The `OutputPipe` object returned would send messages to any peer that has an equivalent input pipe.

Remember that unicast and propagate pipes are unreliable. This means that you need to have code that accounts for dropped messages.

Blind Output Pipe with Listener

The method for creating a blind output pipe with a listener is a little different from one that directly returns a pipe. The key difference is that as peers are found, the listener is called with a new output pipe. The listener interface and the event class are shown in Figure 3.20.

The following is a simple interface that allows you to create output pipes as input pipes appear on other peers:

```
void createOutputPipe ( PipeAdvertisement adv
                      , OutputPipeListener listener)throws IOException;
```

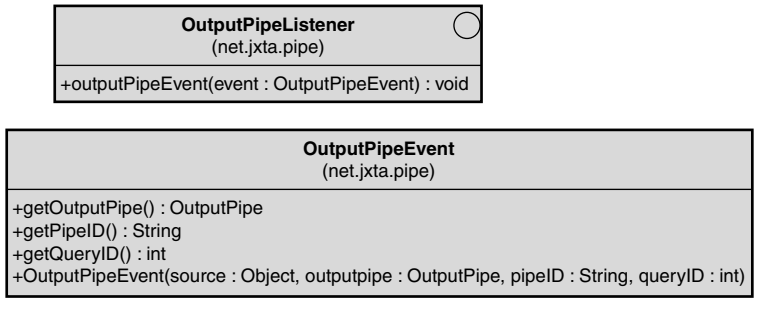

FIGURE 3.20 UML of `OutputPipeListener` interface and `OutputPipeEvent` class.

Addressed Output Pipe

With an addressed pipe, you are looking to connect with a specific peer. The `create` method will block until that peer is connected. If the peer cannot be found within a specific time, the creation fails.

The following method signature from the `PipeService` class adds an `Enumeration` of peers. Be careful to only pass in multiple peers if the pipe is a type of propagate pipe.:

```
OutputPipe createOutputPipe ( PipeAdvertisement adv
                            , Enumeration peers
                            , long timeout) throws IOException;
```

Blind Input Pipe

Input pipes are a bit different from output pipes because they are not addressed. In other words, they are always blind. The following method signature from the `PipeService` class returns an input pipe object. To receive a connection, you need to call the `waitForMessage` method that blocks until an output pipe on another peer attempts to connect.

```
InputPipe createInputPipe (PipeAdvertisement adv) throws IOException;
```

Blind Input Pipe with Listener

The signature for the next input pipe type allows you to specify a message listener. Note that this is not a listener for pipes but for messages. There is no need to process multiple output pipes connecting to the input pipe. The input pipe is only created to listen to one connection. If you want to accept messages on another peer, you need to create a new input pipe:

```
InputPipe createInputPipe ( PipeAdvertisement adv
                          , PipeMsgListener listener) throws IOException;
```

The UML of the listener and the listener event in Figure 3.21 shows how simple the system is. The listener's only method is `pipeMsgEvent`, which passes a `PipeMsgEvent` that, in turn, passes the `Message` object received.

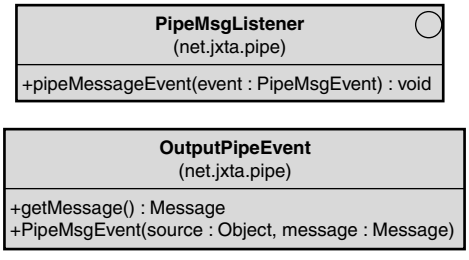

FIGURE 3.21 UML of `PipeMsgListener` interface and `PipeMsgEvent` class.

Unicast input pipes are simply waiting for messages sent to the propagate pipe's ID. For this reason, this method should return as soon as the pipe is initialized. Note that there may not ever be a corresponding propagate output pipe and the listener may never be called.

With the listener, there is no need to process messages by waiting. Instead, the listener is called every time a message is received. To remove a `PipeMsgListener`, close the input pipe.

CAUTION

The `pipeMsgEvent` method in your implementation of the `PipeMsgListener` must be thread safe. This method could easily become a bottleneck if the method is synchronized causing the thread to block for the entire run time of the method. Use `synchronized` blocks inside the method around shared objects. The method should be callable as soon as possible to process as many messages as possible.

Removing the Output Listener

To stop accepting requests for output pipes, you need to remove the listener. After the listener is removed, the output pipe is no longer available. The following is the signature for the `removeOutputPipeListener` method:

```
OutputPipeListener removeOutputPipeListener(String pipeID
                                            ,OutputPipeListener listener);
```

Messages

Messages are meant to be very simple containers of data. In the following lines, the message is created in the context of the pipe service. The data is created by adding a tag called `Test` and setting it with the raw bytes created from the `"Hello, World!"` string:

```
Message msg = peerGroup.getPipeService ().createMessage ();
msg.setBytes ("Test", "Hello, World!".getBytes ());
```

Bidirectional Pipes

The `BidirectionalPipeService` is an optional service that you can use to create bi-directional pipes. As we discussed when talking about one-way pipes, a bi-directional pipe is really two one-way pipes in complementary directions. The `BidirectionalPipeService` class and accompanying classes are shown in Figure 3.22.

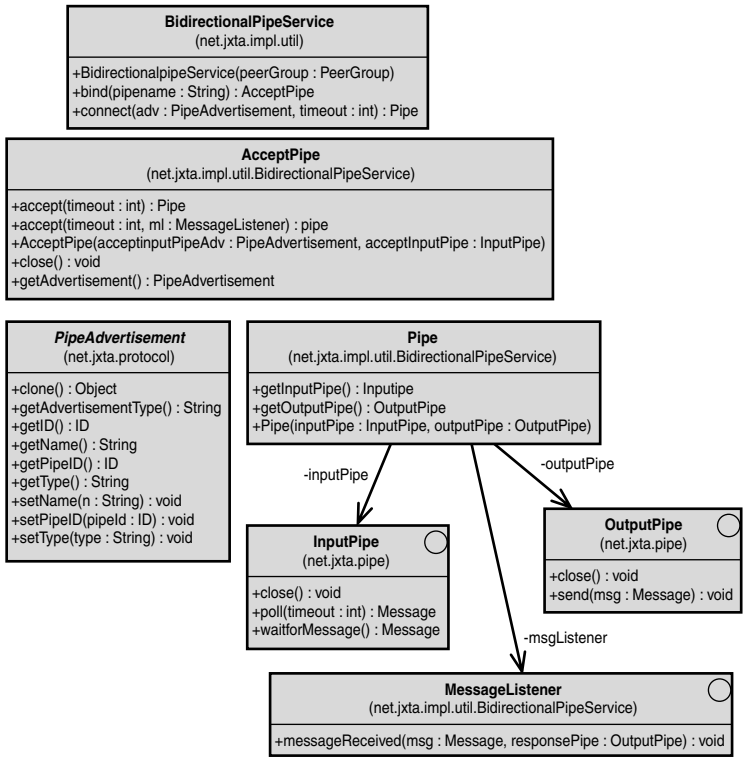

FIGURE 3.22 UML of `BidirectionalPipeService` and related classes and interfaces.

Wire Pipes (One-to-Many)

The `Wirepipe` class is used to create a pipe that will broadcast a copy of each message to each of a list of specified peers. Only peers that have accepted the pipe will receive messages.

Reliable Pipes

This `ReliablePipeService` class provides reliable message-delivery pipes. Each of the messages will be received in the order that they were sent, and the message is guaranteed to reach its destination. Reliable pipes can also be encrypted.

Peer Endpoint Protocol

Peer endpoint routing is not really a public API. Endpoint routing is used to enable pipes or simplistic messaging, such as that found in the peer info, peer resolver, and pipe binding protocols.

The `EndpointService` class provides a front-end API and environment to all endpoint protocols. Applications can use the Endpoint API directly to control or examine the topology of the JXTA network. The Endpoint API would normally be used to help implement a new endpoint protocol.

We will only touch on a few aspects of this protocol. The more interesting functionality to most JXTA developers is how routes are formed.

Use in Applications

In Chapter 2, we talked about the many reasons for routing in JXTA. You do not need to worry about using the actual protocol in your application. For example, the pipe binding protocol uses the Peer endpoint protocol to build routes. As a result, there is no need to supply the pipe with a route, just the other endpoint.

There are reasons for using the protocol directly if you are writing your own implementation of a pipe or other method of P2P communication. Knowing the protocol is also very important for understanding how routes are built.

Accessing the Service

To access the endpoint service, you get the service from your peer group as in the following line:

```
EndpointService getEndpointService();
```

The `EndpointService` interface (shown in Figure 3.23) contains multiple and very useful methods:

- `addEndpointProtocol`—This method allows us to add a new type of communication protocol.
- `addFilterListener`—Used to add a listener to filter endpoint messages.
- `addListener`—Used to listen for messages to a specific endpoint address.
- `demux`—Used to de-multiplex a message. Given an incoming message this method calls the appropriate listener by the destination returned by the `getDestAddress()` method of the message.

- `getEndpointProtocolByName`—This is a convenience method used to fetch an endpoint protocol.
- `getEndpointProtocol`—Returns an enumeration of endpoint protocols.
- `getGroup`—Returns the group of which this service is a member.
- `GetMessenger`—Creates an `EndpointMessenger` given an `EndpointAddress`. The messenger is used to send messages to the specific address defined by the `EndpointAddress`.
- `newEndpointAddress`—Creates a new endpoint at a specific URI.
- `newMessage`—Creates a message to be used to send a message to an endpoint.
- `ping`—Used to determine if a peer exists.
- `propagate`—Used to continue sending the message to other peers.
- `removeEndpointProtocol`—Uses to remove a protocol that is no longer valid.
- `removeEndpointListener`—Used to remove the endpoint listener.
- `removeEndpointFilterListener`—Used to remove the endpoint filter listener.

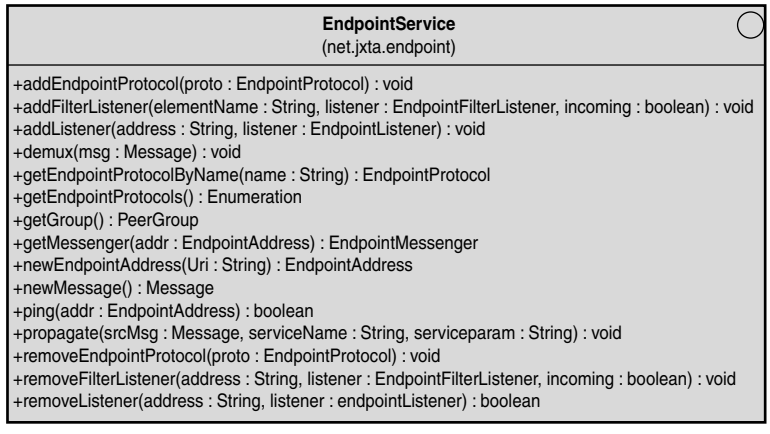

FIGURE 3.23 UML of `EndpointService` interface.

Filter Listener

The filter is a fun and useful part of the endpoint API. What the system allows you to do is snoop on any of the messages in the system. You simply specify an XML tag name, the listener, and a Boolean that specifies whether you are interested in oncoming or outgoing messages.

The following is the method from the `EndpointService` interface:

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

There is also an equivalent `remove` method:

```
void removeFilterListener( String address
                                   , EndpointFilterListener listener
                                   , Boolean incoming);
```

The listener is easy to use, as you can see by the UML diagram in Figure 3.24. The method gives you the message, the source address, and the destination address. Of course, the purpose of the listener is to filter messages. Consequently, the return value is a message. Normally, you would either modify the message or return a `null` if the message is to be blocked.

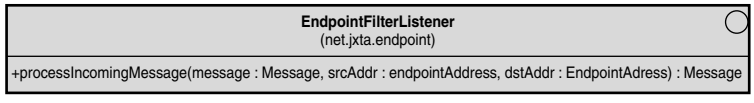

FIGURE 3.24 UML of `EndpointFilterListener` interface.

`EndpointRouterMessage`

One of the more interesting messages to listen for with the `EndpointFilterListener` is the `EndpointRouterMessage`. The message is used to build a path between gateway peers to connect endpoints. By looking for these messages, you can determine the route.

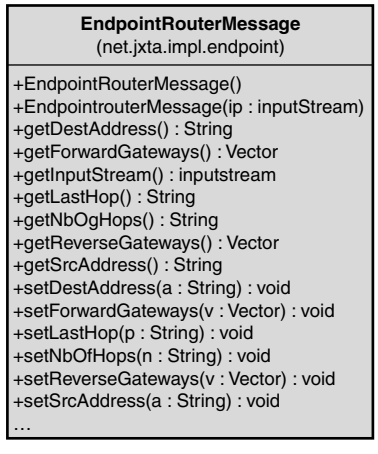

FIGURE 3.25 UML of `EndpointRouterMessage` interface.

The messages that are passed between peers are the `EndpointRouterQuery`, `EndpointRouterAnswer`, and the `EndpointRoute` messages. Note that the `EndpointRouter` class is the source of these messages.

The query is used to request a route, and the answer is a response that contains a route to the destination peer. Note that there is no guarantee that there is a route to the peer, and you may not get the answer message.

The DTD specification for the `EndpointRouterQuery` message is as follows:

```
<!ELEMENT EndpointRouterQuery   (Credential,
                                 Dest,
                                 Cached)>
<!ELEMENT Credential    #PCDATA>
<!ELEMENT Dest          #PCDATA>
<!ELEMENT Cached        #PCDATA>
```

The DTD specification for the `EndpointRouterAnswer` message is as follows:

```
<!ELEMENT EndpointRouterAnswer  (Credential,
                                 Dest,
                                 RoutingPeer,
                                 RouterPeerAdv,
                                 Gateway)>
<!ELEMENT Credential      #PCDATA>
<!ELEMENT Dest            #PCDATA>
<!ELEMENT RoutingPeer     #PCDATA>
<!ELEMENT RouterPeerAdv   (PeerAdvertisement)>
<!ELEMENT Gateway         #PCDATA>
```

The DTD specification for the `EndpointRoute` message is as follows:

```
<!ELEMENT EndpointRoute   (Src,
                           Dest,
                           TTL,
                           Gateway)>
<!ELEMENT Src        #PCDATA>
<!ELEMENT Dest       #PCDATA>
<!ELEMENT TTL        #PCDATA>
<!ELEMENT Gateway    #PCDATA>
```

ping

Another useful method is ping. The method, shown next, simply takes an endpoint address and returns a Boolean value of true if the address can be reached:

```
boolean ping (EndpointAddress addr);
```

The ping method is not as useful as the ping you are used to in normal networking. This ping does not return any useful information; it only lets you know that can reach the peer. The actual processing may also be much different. For example, the ping method may return true if the peer knows about the peer in some endpoint protocols. In HTTP, you do not ping the client because you really can't do so. Remember that the HTTP client must initiate the connection. If you did this, you have no reasonable guarantee that the client will check the gateway any time soon. So, for the sake of time and effort, you assume you can talk to an HTTP endpoint if you find its gateway.

Given how an HTTP ping works, the quality of the ping method may seem suspect compared to a traditional network ping that always connects to a machine if successful. However, the usefulness of ping is that it is implemented by each of the protocols. This essentially gives you a transmission protocol-independent ability to test for an endpoint. In JXTA, this is the primary goal, so other aspects, such as the actual connection or timing, is less important. Simply put, ping is less a network ping than a test for existence according to the protocol implemented by the endpoint.

Depending on the protocol of the endpoint, you may be able to sniff a returning ping message with the filter listener. You should look at the specific implementation of the protocol to understand what the message looks like.

If you are looking for a traditional ping, you will need to implement it yourself. Your ping will only work with peers in your group with your ping code.

EndpointMessenger

One method of the EndpointService interface that is very useful for applications is getMessenger. The getMessenger method takes an endpoint and returns an object that implements an EndpointMessenger interface. The EndpointMessenger (UML seen in Figure 3.26), has two methods—close and sendMessage. Messages sent via the sendMessage method are processed by an EndpointListener that will be covered in a moment.

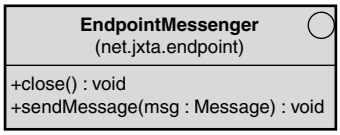

FIGURE 3.26 UML for `EndpointMessenger` interface.

The EndpointMessenger should seem very familiar to you from the previous discussion of output pipes. The simple fact is that a pipe can use the `EndpointMessenger` code implemented by the endpoint.

To get an endpoint messenger, you must first create an endpoint address. You do this with the following method in the `EndpointService`:

```
EndpointAddress newEndpointAddress (String uri);
```

The URI in this case is a unique string. The URI consists of a unique identifier that should be distinguishable between groups and services. For example, you could use the peer group ID and a service name like `Message Queue`. You could further refine the address by concatenating the peer group id, the service name, and a third parameter, such as a sequence number. In this way, you could create multiple addresses on a single endpoint. This is similar to how you would implement a type of pipe and create multiple pipe connections.

To retrieve the messenger, you supply the `EndpointAddress` object you created with the `newEndpointAddress` method, as seen by the following method signature from the `EndpointService` interface:

```
EndpointMessenger getMessenger (EndpointAddress addr) throws IOException;
```

Where Is the Router?

You might ask, if this is the endpoint router, where is the router? The router is deep in the bowels of this API and is actually a part of the platform as a module. The `getMessenger` method eventually results in the execution of methods to obtain a route between peers. The UML for the `EndpointRouter` is shown in Figure 3.27.

You can access the `EndpointRouter` by calling the `loadModule` method on your peer group. The following is an example that creates a router for a peer group:

```
EndpointRouter router;
router = peerGroup.loadModule( "net.jxta.impl.endpoint.EndpointRouter"
                             , Platform.refRouterProtoSpecID
                             , PeerGroup.Both);
```

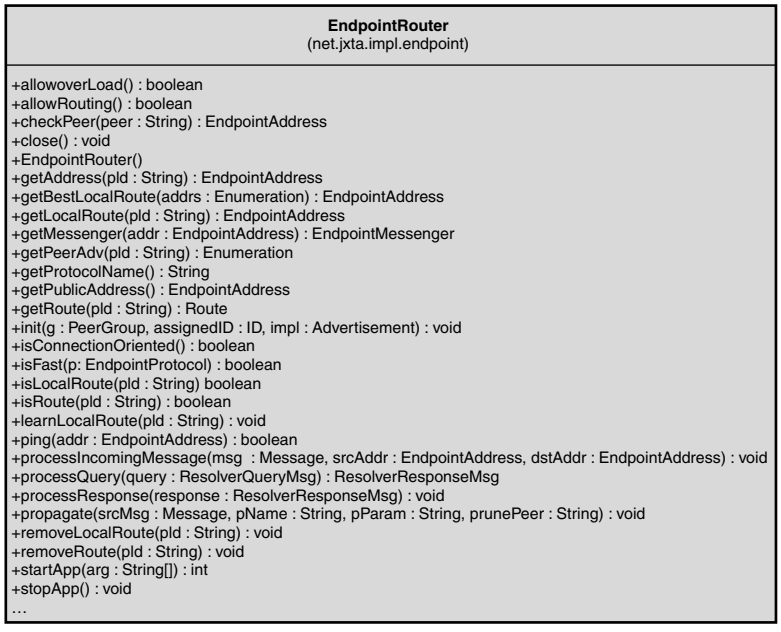

FIGURE 3.27 UML of `EndpointRouter` class.

NOTE

Note that there is no need to access the router in your code. The router is access via other classes in the JXTA API.

Endpoint Listener

There is another listener interface you can use. The `addListener` method in the `EndpointService` interface adds an incoming message listener to a specifically named address that is usually formed by concatenating the name of the invoking service and a parameter unique to that service across all groups.

The listener is called when you send a message through an `EndpointMessenger` to an `EndpointAddress` constructed with matching `serviceName` and `serviceParam`.

The `address` parameter is really the same string used to create the `EndpointAddress`. Although the parameters are called two different things, `uri` and `address`, they are the same thing (expect the parameter names to be one or the other in a future version).

The following is the method signature from the `EndpointService` interface:

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

The method may throw an `IllegalArgumentException` if the address is deemed incorrect by the protocol.

The `EndpointListener` interface is shown in Figure 3.28.

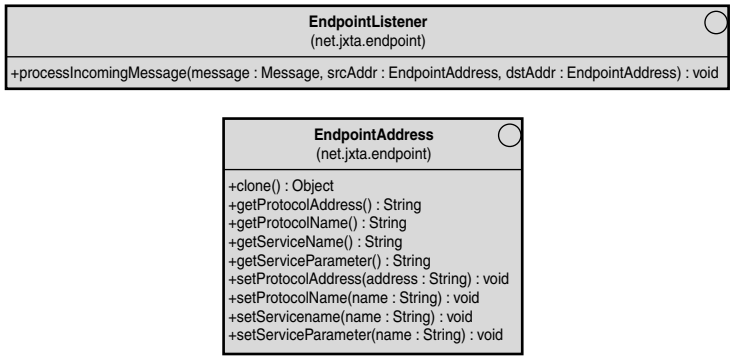

FIGURE 3.28 UML of `EndpointListener` interface and `EndpointAddress` class used to process endpoint messages.

Summary

The key areas of the JXTA API are the peer group, discovery, and pipes. In addition, the peer resolver API can be used for generic queries. Other APIs may be useful, but are less frequently used in most applications. You can also debug by using the listener interfaces of discovery, resolver, and endpoint APIs. We did not cover the authentication service in the peer group API because it had not been implemented.

You now have an overview of the JXTA core API. We have covered a lot of ground. In the following chapters, we will cover much of the API in more depth, in the context of applications and advanced examples.