Steal This Book!

Yes, you read that right. Steal this book. For free.

Nothing. Zero. Zilch. Nadda. Zip.

Undoubtedly you're asking yourself, "Why would he give away a book he probably spent six grueling months writing? Is he crazy?"

The answer…is yes. Sort of. I know that every day you're faced with hundreds of computer titles, and a lot of them don't deliver the value or the information you need. So here's the deal: I'll give you this book (or this chapter, if you've only downloaded part of the book) for free provided you do me a couple favors:

1. **Send this book to your friends:** No, not your manager. Your "with it" computer friends who are looking for the next Big Thing. JXTA is it. Trust me. They want to know about it.

2. **Send a link to the book's web site:** Maybe the book is too big to send. After all, not everyone can have a fibre optic Internet connection installed in their bedroom. The site, at **www.brendonwilson.com/projects/jxta**, provides chapter-sized PDFs for easy downloading by the bandwidth-challenged.

3. **Visit the book's web site:** Being a professional developer, you probably have Carpal Tunnel Syndrome and shudder at the idea of typing in example source code. Save yourself the trouble. Go to **www.brendonwilson.com/projects/jxta** and download the source code. And while you're there, why not download some of the chapters you're missing?

4. **Buy the book:** You knew there had to be a catch. Sure, the book's PDFs are free, but I'm hoping that enough of you like the book so much that you have to buy a copy. Either that, or none of you can stand to read the book from a screen (or, worse yet, print it all out <shudder>) and resort to paper to save what's left of your eyesight. The book is available at your local bookstore or from **Amazon.com** (at a **handsome discount**, I might add).

I now return to your regularly scheduled program: enjoy the book!

6

The Rendezvous Protocol

In Chapter 5, "The Peer Resolver Protocol," you learned that the Resolver service provides the foundation for the Discovery's service's capability to query remote peers and respond to queries from remote peers. Just as the Discovery service relies on the capabilities of the Resolver service, the Resolver service relies on the capabilities of another service: the Rendezvous service. The Rendezvous service is responsible not only for allowing a user to propagate messages to other peers via a rendezvous peer, but also for providing rendezvous peer services to other peers on the network.

This chapter explains the Rendezvous Protocol (RVP) that simple peers use to connect to rendezvous peers to propagate messages to other peers on their behalf. As you'll see, the Rendezvous service implementation of the RVP has a dual role, providing a unified API for propagating messages, independent of whether a peer is configured to act as a rendezvous peer.

Introducing the Rendezvous Protocol

Chapter 2, "P2P Concepts," introduced the concept of a rendezvous peer, a peer used to propagate messages within a peer group on another peer's behalf. In JXTA, a rendezvous peer provides simple peers in private networks with the capability to broadcast messages to other members of a peer group outside the private network. This functionality is independent of the underlying network transport, allowing message propagation over transports that don't support multicast or broadcast capabilities.

Before a peer can use a rendezvous peer to propagate messages, it must connect to the rendezvous peer and obtain a lease. A lease specifies the amount of time that the peer requesting a connection to the rendezvous peer is allowed to use the rendezvous peer before it must renew the connection lease. To handle the interactions required to provide this functionality, the RVP defines three message formats:

- **Lease Request Message**—A message format used by a peer to request a connection lease to the rendezvous peer
- **Lease Granted Message**—A message format used by the rendezvous peer to approve a peer's Lease Request Message and provide the length of the lease
- **Lease Cancel Message**—A message format used by a peer to disconnect from the rendezvous peer

Unlike previous protocols, these messages are not specifically defined in terms of XML; instead, they are defined in terms of message elements. As in XML, message elements consist of a name and the contents of the element, and they can be nested. These message elements are used by the Endpoint service, discussed in Chapter 9, "The Endpoint Routing Protocol," to render messages into a format suitable for transmission over a specific network transport. Although the Endpoint service can render these message elements into XML, in most cases, it is more efficient to render the message elements into a more compact binary representation for transmission.

To connect with a rendezvous peer, a peer uses the sequence of messages shown in Figure 6.1.

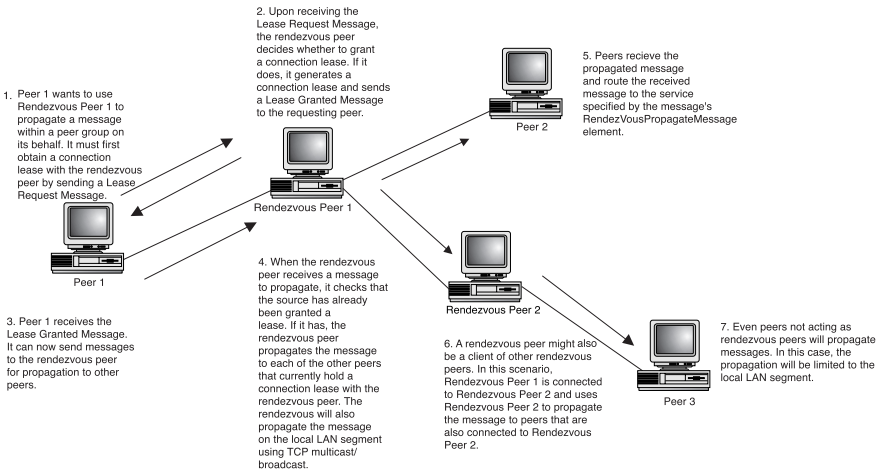

Figure 6.1 Exchange of RVP messages.

Of course, before a peer can even begin the process of connecting to a rendezvous peer, it must discover the rendezvous peer by finding its Rendezvous Advertisement. After a rendezvous peer has been discovered, the peer sends requests to the rendezvous peer using the Endpoint service, addressing requests using `JxtaPropagate` as the service name and the ID of the peer group for which the peer is requesting rendezvous services as the service parameter. Endpoint service names and parameters are detailed in Chapter 9's explanation of the Endpoint service.

The Rendezvous Advertisement

Peers that want to act as a rendezvous peer announce their capabilities to the network by publishing a Rendezvous Advertisement, as shown in Listing 6.1.

Listing 6.1 **The Rendezvous Advertisement XML**

```
<?xml version="1.0"?>
<jxta:RdvAdvertisement xmlns:jxta="http://jxta.org">
  <RdvGroupId> . . . </RdvGroupId>
  <RdvPeerId> . . . </RdvPeerId>
  <Name> . . . </Name>
</jxta:RdvAdvertisement>
```

The Rendezvous Advertisement provides all the details that a peer needs to find a rendezvous peer to use to propagate messages on its behalf:

- **`RdvGroupId`**—A required element containing the ID of the peer group to which the peer is providing Rendezvous services.
- **`RdvPeerId`**—A required element containing the ID of the peer providing Rendezvous services to the specified peer group.
- **`Name`**—An optional element containing a symbolic name for the rendezvous peer. This name could be used by other peers to search for the rendezvous peer.

As shown in Figure 6.2, in the reference implementation, the Rendezvous Advertisement is represented by the `RdvAdvertisement` abstract class in the `net.jxta.protocol` package and is implemented by the `RdvAdv` class in `net.jxta.impl.protocol`.

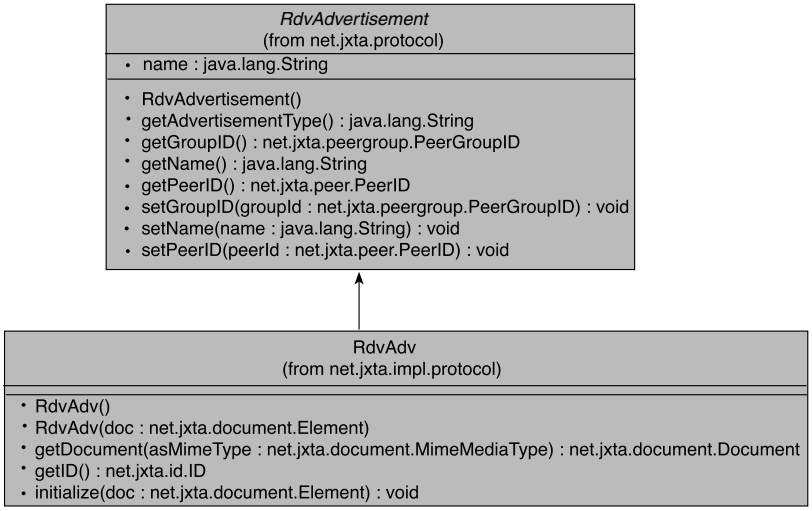

Figure 6.2 The Rendezvous Advertisement classes.

A peer can find rendezvous peers by sending a Discovery Query Message for Rendezvous Advertisements. To use the Discovery service in the reference implementation to search for Rendezvous Advertisements for a specific peer group, use this code:

```
discovery.getRemoteAdvertisements(null, 2, "RdvGroupId",
    currentGroup.getPeerGroupID().toString(),
    threshold, aListener);
```

This query searches for advertisements (type = 2) that match the attribute `RdvGroupId` to the value of the given Peer Group ID. The responses to the Discovery Query Message are passed to the `DiscoveryListener` instance, `aListener`, for processing. The `DiscoveryService` instance used here is the Discovery service of the parent peer group used to create the peer group associated with the rendezvous peer.

Lease Request Message

When a peer has discovered a Rendezvous Advertisement and the rendezvous peer's corresponding Peer Advertisement, a peer can connect to the rendezvous peer and request a connection lease. If the rendezvous peer grants the request, the rendezvous peer adds the peer to its set of authorized peers. These authorized peers are allowed to use the rendezvous peer to propagate messages to other peers that are also connected to the rendezvous peer.

To request a connection lease from a rendezvous peer, a peer sends its own Peer Advertisement as the contents of a message element named `jxta:Connect`, as detailed in Table 6.1.

Table 6.1 **The Lease Request Message**

Element Name	Element Content
`jxta:Connect`	The Peer Advertisement of the peer requesting a connection lease from the rendezvous peer.

The Peer Advertisement content of the message element always is rendered as XML, independent of how the Endpoint service renders the message element. For example, the Endpoint service could render the Lease Grant Message as an XML message:

```
<jxta:Connect>
  <jxta:PA xmlns:jxta="http://jxta.org">
    . . .
  </jxta:PA>
</jxta:Connect>
```

The Endpoint service could even compress this string to produce a pure binary representation of the message element. However, regardless of how the message element is rendered, the message element's Peer Advertisement itself always is an XML document.

Lease Granted Message

If the rendezvous peer approves the peer's request for a connection lease, the requesting peer is added to the rendezvous peer's set of connected peers. The rendezvous peer responds to the requesting peer with a set of message elements collectively called the Lease Granted Message. The Lease Granted Message contains the rendezvous peer's Peer ID and a lease time, and it might contain the rendezvous peer's Peer Advertisement, as detailed in Table 6.2.

Table 6.2 **The Lease Granted Message**

Element Name	Element Content
`jxta:RdvAdvReply`	An optional message element containing the Peer Advertisement of the rendezvous peer granting the lease
`jxta:ConnectedPeer`	A required message element containing the Peer ID of the rendezvous peer granting the lease
`jxta:ConnectedLease`	A required message element containing a string representation of the lease time, in milliseconds

The lease time specifies the amount of time, in milliseconds, before a connected peer is removed from the rendezvous peer's set of connected peers. Peers that are connected to the rendezvous peer receive messages propagated by the rendezvous peer on behalf of other peers. Peers that are also located on the same LAN segment as the rendezvous peer receive the propagated message via TCP multicast. If a peer is located on the same LAN segment as the rendezvous peer, it receives a propagated message twice, once via direct communication by the rendezvous peer and once via TCP multicast.

Lease Cancel Message

When a peer no longer wants to use a rendezvous peer, it can cancel its connection lease, thereby removing itself from the rendezvous peer's set of connected peers. After it is removed, a peer can no longer use the rendezvous peer to propagate messages, nor will it receive messages propagated by the rendezvous peer on another peer's behalf.

To cancel the connection lease, a peer sends a message containing a `jxta:Disconnect` message element, as detailed in Table 6.3.

Table 6.3 **The Lease Cancel Message**

Element Name	Element Content
`jxta:Disconnect`	The Peer Advertisement of the peer requesting removal from the rendezvous peer's set of connected peers

The rendezvous peer removes the peer from its list of connected peers but does not provide any response to the peer.

Controlling Message Propagation

In Chapter 2, I noted the possibility for propagation to result in loopbacks, messages propagating infinitely between peer and rendezvous peers connected in a closed loop. As detailed in Chapter 2, loopback can be prevented by using a Time To Live (TTL) value that gets decremented each time a message is propagated. When the TTL reaches 0, the message is no longer propagated.

The RVP defines a message element to hold information that allows a rendezvous peer to detect loopback and discard the duplicate message. The contents of the message element include a RendezVous Propagate Message document, as shown in Listing 6.2.

Listing 6.2 **The RendezVous Propagate Message XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:RendezVousPropagateMessage>
  <MessageId> . . . </MessageId>
  <DestSName> . . . </DestSName>
  <DestSParam> . . . </DestSParam>
  <TTL> . . . </TTL>
  <Path> . . . </Path>
</jxta:RendezVousPropagateMessage>
```

The contents of the RendezVous Propagate Message provide details about the service to which the message should be propagated and where the message has already been propagated:

- **`MessageId`**—A required element containing a unique identifier for the message being propagated. In the reference implementation, this is simply the time, in milliseconds, since the epoch, when the message is initially propagated. The reference implementation assumes the likelihood that two messages are being propagated within the same peer group at the same time and are sufficiently small to make this value unique.

- **`DestSName`**—A required element containing the name of the destination service for the propagated message.
- **`DestSParam`**—A required element containing the parameters for the destination service for the propagated message.
- **`TTL`**—A required element containing the propagated message's current TTL. The rendezvous peer discards the message if the message's TTL is `0`.
- **`Path`**—An optional element containing the Peer ID of a peer that the message being propagated has already visited. There can be more than one `Path` element, each specifying a waypoint in the message's propagation path. The rendezvous peer does not propagate a message to any peer that is contained in any of the RendezVous Propagate Message's `Path` elements.

For a message being propagated, the RendezVous Propagate Message is added to a message element with a name consisting of the concatenation of `RendezVousPropagate` and the ID of the peer group for which the rendezvous peer is providing Rendezvous services.

The Rendezvous Service

As shown in Figure 6.3, the Rendezvous service provides the implementation of the RVP, providing the functionality both to run a rendezvous peer and to propagate a message using a Rendezvous peer.

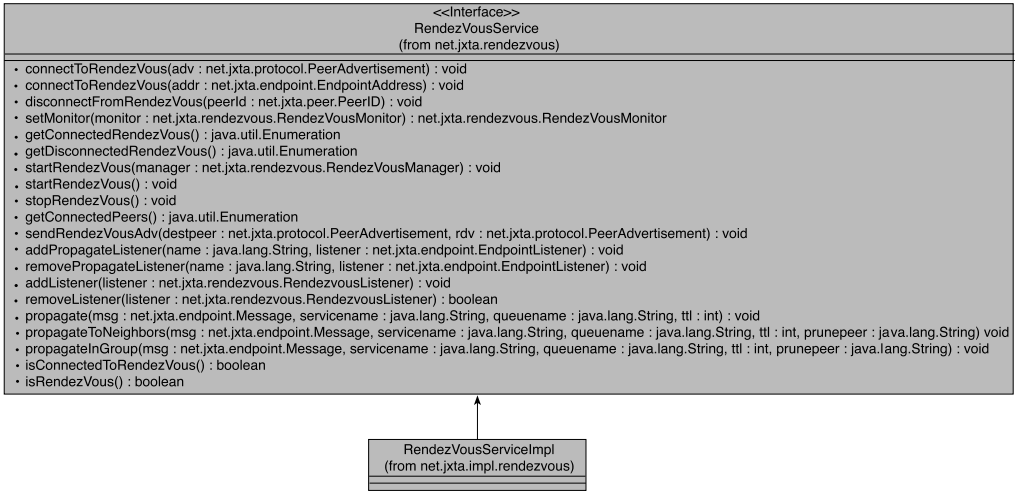

Figure 6.3 The Rendezvous service interfaces and classes.

When not configured to act as a rendezvous peer, a peer can use its Rendezvous service to propagate messages within a peer group using rendezvous peers to which it is connected. The peer can also use the Rendezvous service to propagate messages to peers in the same peer group using network transports that support multicasting within the LAN segment. When configured as a rendezvous peer, the Rendezvous service has the additional capability to propagate messages to other rendezvous and simple peers in the peer group on behalf of its set of connected peers.

Propagating Messages

The Rendezvous service's main functionality is to allow a peer to propagate messages to other peers on the network. This functionality is augmented when a Rendezvous service is configured to provide rendezvous peer services to other peers in the peer group. Regardless of whether a Rendezvous service is configured to provide rendezvous peer services, the `RendezVousService` interface provides three methods for propagating messages, as shown in Listing 6.3.

Listing 6.3 **The *RendezVousService* Message Propagation Methods**

```
public void propagate (Message msg, String serviceName,
    String serviceParam, int defaultTTL) throws IOException;
public void propagateInGroup (Message msg, String serviceName,
    String serviceParam, int defaultTTL, String prunePeer)
        throws IOException;
public void propagateToNeighbors (Message msg, String serviceName,
    String serviceParam, int defaultTTL, String prunePeer)
        throws IOException;
```

Each method provides a slightly different way of propagating a message to other peers in the peer group. All methods have the following parameters in common:

- **`msg`**—The `Message` object to be propagated to other peers.
- **`serviceName`**—A unique service name that identifies the service on the remote peer that is responsible for handling the `Message` object. In the reference implementation, this is set to the Module Class ID for the service. Modules and module classes are discussed in Chapter 10, "Peer Groups and Services."
- **`serviceParam`**—A parameter providing a name for a message queue. The service can use this parameter to route the handling of a message to the appropriate service instance.

- **defaultTTL**—A default Time To Live (TTL) value to be used when sending the Message. This TTL is used only if the Message doesn't currently have a TTL value. Otherwise, the propagation methods handle decrementing the Message's TTL. In the reference implementation, the value passed as a default TTL can't be greater than the maximum TTL value of 10. If the default TTL passed is larger than the maximum, it is set to the maximum TTL.

Both propagateInGroup and propagateToNeighbors take an extra argument, prunePeer, that specifies the Peer ID of a peer that should not be included in the propagation. The current reference implementation ignores this argument.

Each of the propagation methods has a slightly different purpose:

- **propagateToNeighbors**—This method propagates the Message to peers on the local network. In the reference implementation, a neighbor is a peer on the network that the rendezvous can communicate with directly, without going through a router peer. This method relies on the Endpoint service to broadcast the message to peers on the local LAN segment using available network transports.
- **propagateInGroup**—This method propagates the Message to all peers in the peer group. This method duplicates the functionality of propagateToNeighbors but also propagates the given Message to each rendezvous peer with which the peer has a connection lease. If the local peer is configured to act as a rendezvous peer, this method also propagates the given Message to each peer that has a connection lease with the local peer.
- **propagate**—The documentation for the Java implementation gives the same description for this method as propagateInGroup. However, the reference implementation uses this method as a convenience method: When the passed defaultTTL is 1, propagate calls propagateToNeighbors. Otherwise, propagate calls propagateInGroup.

The propagation methods are all responsible not only for setting the Message's TTL, but also for adding a message element containing the RendezVous Propagate Message to prevent against loopback.

Receiving Propagated Messages

The Rendezvous service is not responsible for blindly repropagating messages that it receives; instead, it serves only to propagate a message one network hop to another peer. It is the responsibility of a service on the peer to decide

whether to repropagate the message. For example, the `ResolverService` implementation described in Chapter 5 repropagates a message only if the `QueryHandler` implementation's `processQuery` method doesn't throw a `DiscardQueryException`.

To allow a service to listen for propagated messages and decide whether to repropagate the message, the `RendezVousService` interface defines the `addPropagateListener` method:

```
public void addPropagateListener(String serviceNamePlusParameters,
    EndpointListener listener)
        throws IOException;
```

The `addPropagateListener` method registers an instance of `EndpointListener`, an interface described in Chapter 9, with the `RendezVousService` instance. The listener object is notified via its `processIncomingMessage` method when the Endpoint service receives a propagated message that matches the service name and parameters passed to `addPropagateListener`.

When an `EndpointListener` instance is notified of the arrival of a propagated message, it can repropagate the message by invoking the `propagateInGroup` method on a `RendezVousService` instance. The `RendezVousService` implementation handles updating the message's RendezVous Propagate Message with new TTL and path information.

When notification of propagated messages is no longer required, the `EndpointListener` instance can be unregistered from the `RendezVousService` using the `removePropagateListener` method:

```
public void removePropagateListener(String serviceNamePlusParameters,
    EndpointListener listener)
        throws IOException;
```

To remove a listener object successfully, the parameters passed to `removePropagateListener` must match those used to register the listener using `addPropagateListener`.

Connecting to and Disconnecting from Rendezvous Peers

The process of obtaining or cancelling a connection lease with a rendezvous peer is conducted entirely via the `RendezVousService` interface. To obtain a connection lease with a remote rendezvous peer, a peer invokes this code:

```
public void connectToRendezVous (PeerAdvertisement adv) throws IOException;
```

Instead of using a Peer Advertisement, a peer can use an `EndpointAddress` to specify the remote rendezvous peer from which to obtain a connection lease:

```
public void connectToRendezVous (EndpointAddress addr) throws IOException;
```

The `EndpointAddress` is an abstraction of a network location that can be either network transport-neutral or transport-specific. Endpoint addresses are discussed in Chapter 9. When a peer has obtained a connection lease, the peer can cancel the lease granted by a remote rendezvous peer using this line:

```
public void disconnectFromRendezVous (PeerID peerID);
```

Cancelling the lease with a rendezvous peer requires the Peer ID of the rendezvous peer.

The *RendezvousListener* and *RendezvousEvent* Classes

The Rendezvous service provides a `RendezvousListener` interface, as shown in Figure 6.4, that developers can implement to monitor the Rendezvous service. Registered `RendezvousListener` objects are notified when the Rendezvous service connects and disconnect from a rendezvous peer and when client peers connect or disconnect.

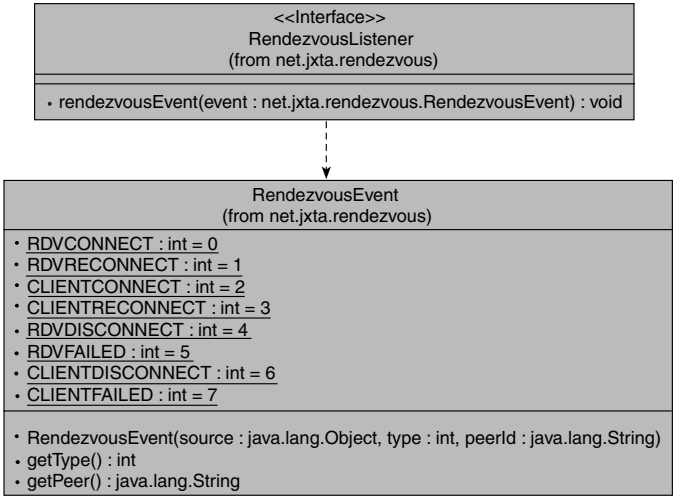

Figure 6.4 The `RendezvousListener` and `RendezvousEvent` classes.

The `RendezvousEvent` represents events fired by the `RendezVousService` when it is either acting as a client to another rendezvous peer or acting as a rendezvous peer to a client peer. The `RendezvousEvent.getType` method returns one of several possible values to inform the `RendezvousListener` what event has transpired. Type names starting with *CLIENT* indicate events triggered by the Rendezvous service handling a client peer message. Type names starting with

RDV indicate events triggered by the Rendezvous service receiving a response to messages sent to a remote rendezvous peer. In total, eight possible values are returned by `RendezvousEvent.getType`:

- **`CLIENTCONNECT`**—The Rendezvous service has successfully processed a client peer's Connect request.
- **`CLIENTDISCONNECT`**—The Rendezvous service has successfully processed a client peer's Disconnect request.
- **`CLIENTRECONNECT`**—This is not currently used in the reference implementation. It indicates that the Rendezvous service has successfully processed a client peer's Connect request. In this case, the client peer was already connected to the rendezvous peer but is connecting to renew its lease with the rendezvous peer. Most likely, this will be used when the lease time is used correctly.
- **`CLIENTFAILED`**—This also is not currently used in the reference implementation. It indicates that the Rendezvous service has unsuccessfully processed a client's Connect request.
- **`RDVCONNECT`**—The Rendezvous service, acting as a client peer, has received a response to its Connect request indicating that it is now connected to a rendezvous peer.
- **`RDVDISCONNECT`**—The Rendezvous service has successfully disconnected from a remote rendezvous peer. This event is not fired as a result of a response from the rendezvous peer, but it is fired immediately after the Disconnect request is sent to the rendezvous peer.
- **`RDVRECONNECT`**—This is not currently used in the reference implementation. It indicates that the Rendezvous service has received a response from a rendezvous peer confirming the success of a Connect request. In this case, the client peer was already connected to the rendezvous peer, but it sent a Connect to renew its lease with the rendezvous peer. Most likely, this will be used when the lease time is used correctly.
- **`RDVFAILED`**—This is not currently used in the reference implementation. The Rendezvous service has received a response indicating that its Connect request failed.

Implementations of `RendezvousListener` can be added to and removed from the `RendezVousService` instance using the `addListener` and `removeListener` methods. The methods operate in a similar fashion to `DiscoveryService`'s `addDiscoveryListener` and `removeDiscoveryListener` methods.

Support Classes Used by the Rendezvous Service

The `RendezVousService` relies on several other interfaces to abstract the task of managing peers' requests to obtain or cancel a connection lease. Each `RendezVousService` instance relies on an implementation of the `RendezVousManager` interface, as shown in Figure 6.5, to handle a client peer's request for a connection lease.

Figure 6.5 The `RendezVousManager` interface.

The `requestConnection` method processes the Peer Advertisement passed in the request and returns the lease time (in milliseconds). In the current reference implementation, the default lease time is 30 minutes, although this time is not currently used, as previously mentioned. A lease time of `0` indicates that the `RendezVousService` instance should not add the client to its set of connected client peers. A negative lease time indicates an infinite lease on the connection to the rendezvous peer.

Unlike the `RendezvousListener` interface, a `RendezVousService` instance has only one `RendezVousManager` instance. The `RendezVousManager` instance is initialized only when the `RendezVousService` is configured to act as a rendezvous for other peers. This `RendezVousManager` instance is passed to the `RendezVousService.startRendezVous` method used to start the Rendezvous service operating as rendezvous peer.

After the rendezvous peer is started using `startRendezVous`, the `RendezVousManager` instance can't be changed. Instead, the `RendezVousService` instance must be stopped using `stopRendezVous` and restarted using a different `RendezVousManager` instance. Stopping and starting the `RendezVousService` instance affects only the instance's operation as a rendezvous peer. The portion of `RendezVousService` instance responsible for allowing the local peer to propagate messages using other rendezvous peers is unaffected by `startRendezVous` and `stopRendezVous`.

Another support interface, `RdvMonitor`, provides functionality that is used when the Rendezvous service is acting as a client to a remote rendezvous peer. `RdvMonitor`'s methods, shown in Figure 6.6, are invoked when a client peer successfully obtains or cancels a connection lease with a rendezvous peer. A `RendezVousService` instance has only a single `RdvMonitor` instance.

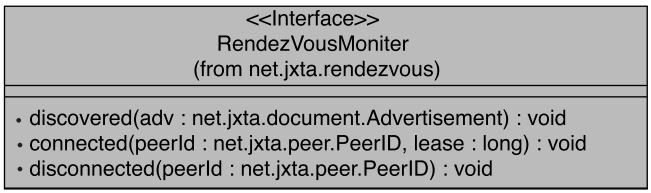

Figure 6.6 The `RendezVousMonitor` interface.

When a peer receives a response indicating that a rendezvous peer has granted a connection lease, the `RdvMonitor.connected` method is invoked by the `RendezVousService` instance. The `connected` method accepts the rendezvous peer's Peer ID and the lease time for the connection. In the reference implementation, the `connected` method adds a local Rendezvous Advertisement for the remote peer and starts a thread to handle renewing the lease.

When a peer cancels a connection lease with a rendezvous peer, the `RendezVousService` instance invokes the `disconnected` method. Currently, the reference implementation of `RdvMonitor` doesn't do anything in the `disconnected` method.

The `RdvMonitor` interface defines one other method, `discovered`, which is invoked by the `RendezVousService` instance to provide an advertisement for other rendezvous peers. When the `RendezVousService`'s `sendRendezVousAdv` method is called, the peer sends a message containing a message element named `jxta:RdvAdv` that contains a Peer Advertisement. This Peer Advertisement is for a rendezvous peer that the `RendezVousService` instance wants to publish to other peers. When a peer's `RendezvousService` receives a message containing a `jxta:RdvAdv` message element, the service's `RdvMonitor` instance has its `discovered` method invoked. This feature can be used to distribute the load away from a particular rendezvous peer. Currently, the reference implementation simply publishes the advertisement locally when `discovered` is called.

Unlike `RendezVousManager`, the `RdvMonitor` instance can be set using the `RendezVousService.setMonitor` method.

Other Useful *RendezVousService* Methods

The `RendezVousService` interface defines several other useful methods:

- **`getConnectedPeers`**—Returns an `Enumeration` of `ID`s of all the client peers currently connected to the rendezvous peer. When a peer is not acting as a rendezvous peer, the `Enumeration` is empty.
- **`getConnectedRendezVous`**—Returns an `Enumeration` of `ID`s of all the rendezvous peers to which the peer is connected. This method returns results regardless of whether the peer is operating as a rendezvous peer.

- **`getDisconnectedRendezVous`**—Returns an `Enumeration` of `ID`s of all rendezvous peers to which the peer has failed to connect.
- **`isConnectedToRendezVous`**—Returns `true` if the peer is currently connected to at least one rendezvous peer.
- **`isRendezVous`**—Returns `true` if the peer is providing rendezvous peer services to other client peers in Rendezvous service's peer group.

Maintaining Rendezvous Connections

To ensure that a peer receives propagated messages, the peer must maintain its connection lease to a number of rendezvous peers. Without maintaining and renewing the connection lease, a peer risks not receiving propagated messages from members of its peer group that are not part of its local LAN segment.

To address this issue, the reference implementation provides the `RendAddrCompactor` class in `net.jxta.impl.rendezvous`. The `RendAddrCompactor` class runs a thread that regularly uses the Discovery service to find Rendezvous Advertisements and maintain connections to rendezvous peers. The `RendAddrCompactor` thread tries to discover and obtain a connection lease with up to three rendezvous peers, thereby attempting to guarantee connectivity with peer group members outside the local LAN segment.

Summary

In this chapter, you saw how the Rendezvous service allows peers to propagate messages to other peers within a peer group. You also learned that the Rendezvous service provides the capability for a peer to act as a rendezvous peer and propagate messages on behalf of other peers in a peer group. Developers can use the Rendezvous service not only to send messages, but also to provide their own custom functionality when client peers connect to the Rendezvous service to obtain rendezvous peer services.

In the next chapter, you explore the Peer Information Protocol, which is a protocol that monitors peers and obtains peer status information. The Peer Information Protocol allows a peer to gather information about a remote peer that it can use to determine the suitability of the peer for performing a task.