CHAPTER 25

Activation

Many of the examples in earlier chapters use RMI/Jeri proxies for services. These services live within a server whose principal task is to keep the service alive and registered with lookup services. If the server fails to renew leases, then lookup services will eventually discard the proxy; if the server fails to keep itself and its service alive, then the service will not be available when a client wants to use it.

This results in a server and a service that most of the time will be idle, probably swapped out to disk, but still using virtual memory. Java memory requirements on the server side can be enormous. From JDK 1.2 onward, an extension to RMI called *activation* allows an idle object to be in a "dormant" state and brought to life when needed. In this way, the object does not occupy virtual memory while idle. Of course, another process needs to be alive to restore such objects, and RMI supplies the daemons `rmid` (in Jini 1.2) and `phoenix` (in Jini 2.0) to manage this. In effect, `rmid/phoenix` acts as another virtual memory manager as it stores information about dormant Java objects in its own files and restores them from there as needed.

There are serious limitations to `rmid` and `phoenix`: they are Java program themselves, and when running they also use enormous amounts of memory. So it only makes sense to use them when you expect to be running a number of largely idle services on the same machine. When a service is brought to life, or *activated*, a new JVM may be started to run the object. This again increases memory use.

If memory use is the only concern, then a variety of other systems, such as `echidna`, which run multiple applications within a single JVM, may be adequate to solve memory issues. However, RMI activation is also designed to work with distributed objects, and it allows JVMs to hold remote references to objects that are no longer active. Instead of throwing a remote exception on trying to access these objects, the activation system tries to resurrect the object using `rmid` or `phoenix` to give a valid (and new) reference. Of course, if it fails to do this, it will throw an exception anyway.

The standard RMI activation system is supported by Jini 2.0, in the same way as it supports JRMP. But with the advent of Jeri, Jini 2.0 has a new version of activation with the `phoenix` activation server, which we'll cover in the next section. In the rest of the chapter, we'll look at how to build and servers and services that use the activation system. We also look at more subtle issues such as nonlazy activation and how a system that is "reborn" each time can save state. If a service uses activation, then it will probably not be present to renew leases or to be discovered. We look at two additional Jini services, a `LeaseRenewalService` and a `LookupDiscovertService`, which can overcome these problems.

The phoenix Activation Server

phoenix replaces rmid in Jini 2.0. It comes in a variety of versions, depending on the protocol it supports. For example, if the services use Jeri, then phoenix should be configured to use Jeri also. Similarly, if the services use JRMP, then so should phoenix.

phoenix can be started by using the ServiceStarter or by shell scripts/batch files. Example scripts are given in the Jini distribution under the source/vob/jive/src/com/sun/jini/example/hello/scripts/ directory. For example, here is the shell script jeri-phoenix.sh, which starts the Jeri version of phoenix under Unix:

```
host=`hostname`
java -Djava.security.manager=                                          \
     -Djava.security.policy=config/phoenix.policy                      \
     -Djava.rmi.server.codebase=http://$host:8080/phoenix-dl.jar       \
     -DserverHost=$host                                                \
     -jar lib/phoenix.jar                                              \
     config/jeri-phoenix.config
```

And here is the batch file jrmp-phoenix.bat, which starts the Jeri version of phoenix under Windows:

```
java -Djava.security.manager= ^
     -Djava.security.policy=config\phoenix.policy ^
     -Djava.rmi.server.codebase=http://%computername%:8080/phoenix-dl.jar ^
     -DserverHost=%computername% ^
     -jar lib\phoenix.jar ^
     config\jrmp-phoenix.config
```

Each script file references a configuration script. A typical script such as config/jeri-phoenix.config contains the following:

```
com.sun.jini.phoenix {
    persistenceDirectory = "lib${/}phoenix-log";
    groupConfig = new String[] { "config${/}jeri-phoenix-group.config" };
}
```

which states the directory to store the activation log files and also a group configuration file, such as jeri-phoenix-group.config.

This file defines the protocol that will be used by phoenix (here, Jeri):

```
import com.sun.jini.phoenix.AccessILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
com.sun.jini.phoenix {
    instantiatorExporter =
        new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                              new AccessILFactory());
}
```

There is a little trap in running `phoenix`: it will create a new virtual machine for each different group. This new virtual machine will require several files, such as `phoenix-init.jar`, in its classpath. So it is not enough to specify `phoenix.jar` for `phoenix`—the `phoenix-init.jar` file must be in the classpath for any virtual machines created by `phoenix`, and this must be done for each activatable service. An alternative to explicitly setting the classpath is to copy the `phoenix-init.jar` file to the Java `jre lib` directory (as you probably did with `jsk-policy.jar`), but this approach is not really recommended. If the classpath is not set up, then when `phoenix` starts a new JVM, you will see errors such as the following:

```
Group-01: class not found ActivationInitGroup
```

The Sun documentation recommends including `sharedvm.jar` in the classpath, and the directory for this `.jar` file should also contain `phoenix-init.jar` and `jsk-platform.jar`.

A Service Using Activation

The major concepts in activation are the activatable object itself (which extends `java.rmi.activation.Activatable`) and the environment in which it runs, an `ActivationGroup`. A JVM may have an activation group associated with it. If an object needs to be activated and there is already a JVM running its group, then it is restarted within that JVM. Otherwise, a new JVM is started. An activation group may hold a number of cooperating objects.

In this section, we'll look in turn at building a service, building a server, and how to run these using the activation system.

Service

Making an object into an activatable object requires registering the object with the activation system by exporting it and using a special two-argument constructor that will be called when the object needs to be reconstructed. The constructor looks like this:

```
public ActivatableImpl(ActivationID id, MarshalledObject data)
    throws RemoteException {
    ...
}
```

■ **Note** The use of the marshalled data is discussed later in this chapter.

There is an important conceptual change from nonactivatable services. In a nonactivatable service, the server is able to create the service. In an activation system, the original server could have terminated and will not be available to start the service; instead, the activation server is responsible for starting the service. But the service still has to be exported, and it can't rely on the activation server to do that (e.g., it would have no knowledge of the protocol, such as Jeri, JRMP, or IIOP), so the service has to export itself. That is, within the constructor, the service must find an exporter and export itself. This is a change from "standard" activation as

used in Jini 1.2, where many things were hidden from the programmer and it was not necessary to pay attention to the exporter.

That change in turn raises a problem: in a nonactivatable service, the server creates the service, gets a proxy by exporting the service, and then does things like register the proxy with lookup services. But if the export operation is buried within the service constructor, then a server cannot readily access it. This is the role of the `ProxyAccessor` interface: it supplies a method that a server can call on the service to give the proxy. Unless the service can do everything itself, it will usually need to implement this interface. (An exception to this occurs when the service is its own proxy; it just needs to be `Serializable` in that case.)

With these changes in place, the file classifier becomes

```
package activation;
import net.jini.export.*;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.tcp.TcpServerEndpoint;
import net.jini.activation.ActivationExporter;
import net.jini.jrmp.JrmpExporter;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;
import net.jini.export.ProxyAccessor;
import common.MIMEType;
import common.FileClassifier;
import rmi.RemoteFileClassifier;
import java.rmi.Remote;
/**
 * FileClassifierImpl.java
 */
public class FileClassifierImpl implements RemoteFileClassifier,
                                           ProxyAccessor {
    private Remote proxy;
    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        if (fileName.endsWith(".gif")) {
            return new MIMEType("image", "gif");
        } else if (fileName.endsWith(".jpeg")) {
            return new MIMEType("image", "jpeg");
        } else if (fileName.endsWith(".mpg")) {
            return new MIMEType("video", "mpeg");
        } else if (fileName.endsWith(".txt")) {
            return new MIMEType("text", "plain");
        } else if (fileName.endsWith(".html")) {
            return new MIMEType("text", "html");
        } else
            // fill in lots of other types,
            // but eventually give up and
            return new MIMEType(null, null);
    }
```

```
    public FileClassifierImpl(ActivationID activationID, MarshalledObject data)
        throws java.rmi.RemoteException {
        Exporter exporter =
            new ActivationExporter(activationID,
                        new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                              new BasicILFactory(),
                                              false, true));

        proxy = (Remote) exporter.export(this);
    }
    // Implementation for ProxyAccessor
    public Object getProxy() {
        return proxy;
    }
} // FileClassifierImpl
```

This listing makes explicit use of an exporter. Later we'll consider how managing the exporter could be done using a configuration.

Server

The server doesn't actually start the service—that is the task of a process such as `phoenix`. The server has to set up the parameters for the service so that `phoenix` will know how to handle it. These parameters may include the following:

- The activation group(s) the service will belong to.
- The security policy to run services in a particular activation group.
- The classpath for `phoenix` to run the service in a new JVM. Note that this classpath cannot be one that is relative to the server, since it will be used by `phoenix`.
- The codebase for the client to find the service (needed if the service registers itself with lookup services).

A service is run within an activation group. When a group is run in a new virtual machine, it may need explicit command-line options (such as setting the classpath or the stack size) and properties (such as a security policy). Of course, properties can be set as command-line arguments, too, but Java allows them to be set separately. For example, the command-line arguments can be set as follows:

```
String[] options = {"-classpath",
                    "activation.FileClassifierServer.jar"};
CommandEnvironment commEnv =
                    new CommandEnvironment(null, options);
```

The group parameters are set using an `ActivationGroupDesc`, which takes both a `Properties` list and a `CommandEnvironment`:

```
String[] options = {"-classpath",
                    "activation.FileClassifierServer.jar"};
ActivationGroupDesc.CommandEnvironment commEnv =
           new CommandEnvironment(null, options);
Properties props = new Properties();
props.put("java.security.policy",
          SECURITY_POLICY_FILE);
props.put("java.rmi.server.codebase",
          "http://192.168.1.13/classes/activation.FileClassifierServer-dl.jar");
ActivationGroupDesc group = new ActivationGroupDesc(props, commEnv);
```

■ **Note** Although the classpath shown references only the classes required for the server, in practice you may need to add more. For example, `phoenix` requires `phoenix-init.jar`, and other Jini class files may be required, too. The easiest way to work out what is required is to run the server and observe what `phoenix` complains about. Alternatively, include `sharedvm.jar`, which points to all likely `.jar` files that may be required by Sun's tools.

The next steps are to register the group and get a group ID from that. Then an activation description for the service is constructed that includes the group ID and the name of the service's class file. (Two other parameters are discussed later.) This service can then be registered with `phoenix`. At this point, the service is with `phoenix`, but it has not been initialized, so its constructor has not been called and there is no proxy for it. This means the service cannot yet be registered with a lookup service and cannot yet be found by any client. How, then, can you force it to be constructed? At this point, the server has an activation ID for the service from the registration. It uses this to ask `phoenix` to activate the service with the `activate()` method. The code looks like this:

```
ActivationGroupDesc group = new ActivationGroupDesc(props, commEnv);
ActivationGroupID groupID  = actSys.registerGroup(group);
ActivationGroup.createGroup(groupID, group, 0);
String codebase = "...";
MarshalledObject data = null;
ActivationDesc desc = null;
desc = new ActivationDesc(groupID,
                          "activation.FileClassifierImpl",
                           codebase, data, true);
ActivationID aid =  actSys.registerObject(desc);
Remote proxy = (Remote) aid.activate(true);
```

The server now has a proxy that it can register with lookup services. The server can terminate, since any calls on the service will be handled by `phoenix`, which will construct the service whenever a call to that service is made by a client. (I'll address later how the registration with lookup services is kept alive. If this server terminates, then it cannot do any lease renewals.)

The file classifier server that uses an activatable service is as follows:

```
package activation;
//import rmi.RemoteFileClassifier;
import java.rmi.Remote;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
import java.rmi.RMISecurityManager;
import java.rmi.MarshalledObject;
import java.rmi.activation.ActivationDesc;
import java.rmi.activation.ActivationGroupDesc;
import java.rmi.activation.ActivationGroupDesc.CommandEnvironment;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationGroup;
import java.rmi.activation.ActivationGroupID;
import java.rmi.activation.ActivationSystem;
import java.rmi.activation.ActivationID;
import java.util.Properties;
import java.rmi.activation.UnknownGroupException;
import java.rmi.activation.ActivationException;
import java.rmi.RemoteException;
/**
 * FileClassifierServer.java
 */
public class FileClassifierServer implements DiscoveryListener {
    static final protected String SECURITY_POLICY_FILE =
        "/home/httpd/html/java/jini/tutorial/policy.all";
    static final protected String CODEBASE =
        "http://192.168.1.13/classes/activation.FileClassifierServer-dl.jar";

    // protected FileClassifierImpl impl;
    protected Remote proxy;

    public static void main(String argv[]) {
        new FileClassifierServer(argv);
        // stick around while lookup services are found
        try {
            Thread.sleep(100000L);
        } catch(InterruptedException e) {
            // do nothing
        }
        // the server doesn't need to exist anymore
        System.exit(0);
```

```
    }
    public FileClassifierServer(String[] argv) {
        // install suitable security manager
        System.setSecurityManager(new RMISecurityManager());
        ActivationSystem actSys = null;
        try {
            actSys = ActivationGroup.getSystem();
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }
        // Install an activation group
        String[] options = {"-classpath",
                            "activation.FileClassifierServer-act.jar:phoenix-
init.jar:jini-ext.jar"};
        CommandEnvironment commEnv =
            new CommandEnvironment(null, options);
        Properties props = new Properties();
        props.put("java.security.policy",
          SECURITY_POLICY_FILE);
        ActivationGroupDesc group = new ActivationGroupDesc(props, commEnv);
        ActivationGroupID groupID = null;
        try {
            groupID = actSys.registerGroup(group);
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }
        String codebase = CODEBASE;
        MarshalledObject data = null;
        ActivationDesc desc = null;
        desc = new ActivationDesc(groupID,
                                  "activation.FileClassifierImpl",
                                  codebase, data, true);
        ActivationID aid = null;
        try {
            aid = actSys.registerObject(desc);
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }
```

```
        try {
            proxy = (Remote) aid.activate(true);
        } catch(UnknownGroupException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        }

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }
        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {
        ServiceRegistrar[] registrars = evt.getRegistrars();
        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];
            // export the proxy service
            ServiceItem item = new ServiceItem(null,
                                               proxy,
                                               null);
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.print("Register exception: ");
                e.printStackTrace();
                // System.exit(2);
                continue;
            }
            try {
                System.out.println("service registered at " +
                                   registrar.getLocator().getHost());
            } catch(Exception e) {
            }
        }
    }
```

```
    public void discarded(DiscoveryEvent evt) {
    }
} // FileClassifierServer
```

Running the Service

The service and the server must be compiled as usual. Nonactivatable services just require classes for the client and for the server. For activatable services, it is more complex: classes are required for the client, for the start-up server, and for `phoenix`.

The classes that are required by the client must be copied to an HTTP server. In this case, it is only the class file `rmi/RemoteFileClassifier.class`, if a protocol such as Jeri is used with proxy generation at runtime. If JRMP was used, the `rmic` compiler would need to be run on `activation/FileClassifierImpl.class` and the resultant proxy would also need to be copied to the HTTP server.

The classes needed by the start-up server are the file classifier server and the classes it needs. This gets a bit tricky. The server doesn't actually create the service at any time, so it doesn't need the class file for `FileClassifierImpl`. But when it activates the service, `phoenix` will create the service and return a proxy for it. This proxy will implement `RemoteFileClassifier`. So the server will need the class files to support a `RemoteFileClassifier` even though it doesn't explicitly create one. The files could be either in the server's classpath or in its codebase. This server uses the codebase as information in the proxy when it registers the service with a lookup service, so you don't want to put extra stuff in there for downloading to a client. Instead, the class files may be better off in the server's classpath.

- `common/MIMEType.class`
- `common/FileClassifier.class`
- `rmi/RemoteFileClassifier.class`
- `activation/FileClassifierServer.class`

Finally, the classes needed by `phoenix` are `FileClassifierImpl` and the classes it depends on, but not the start-up server:

- `common/MIMEType.class`
- `common/FileClassifier.class`
- `rmi/RemoteFileClassifier.class`
- `activation/FileClassifierImpl.class`

Before starting the service provider, a `phoenix` process must be set running on the same machine as the service provider. An HTTP server must be running on a machine as specified by the `codebase` property on the service. The service provider can then be started. This will register the service with `phoenix` and copy a proxy object to any lookup services found. The server can then terminate (as mentioned earlier, this causes the service's lease to expire, but techniques to handle this are described later).

In summary, typically three processes are involved in getting an activatable service running:

- The service provider, which specifies information about the service to `phoenix`.
- `phoenix`, which must be running on the same machine as the service provider and must be started before the service provider. It creates the service on demand.
- An HTTP server, which can be on a different machine and is pointed to by the codebase.

While the service remains registered with lookup services, clients can download its proxy. The service will be created on demand by `phoenix`. You need to run the server only once, since `phoenix` keeps information about the service in it own log files.

An Ant file to build, deploy, and run the service (but not `phoenix`) is `activation.FileClassifierServer.xml`:

```
<project name="activation.FileClassifierServer" default="usage">
    <!-- Inherits properties
         jini.home
         jini.jars
         src
         dist
         build
         httpd.classes
      -->
    <!-- files for this project -->
    <property name="src.files"
              value="common/MIMEType.java,
                     common/FileClassifier.java,
                     rmi/RemoteFileClassifier.java,
                     activation/FileClassifierImpl.java
                     activation/FileClassifierServer.java
                    "/>
    <property name="class.files"
              value="
                     common/MIMEType.class,
                     common/FileClassifier.class,
                     activation/FileClassifierServer.class
                    "/>
    <property name="class.files.dl"
              value="
                     rmi/RemoteFileClassifier.class
                    "/>
    <property name="class.files.act"
              value="common/MIMEType.class,
                     common/FileClassifier.class,
                     rmi/RemoteFileClassifier.class,
                     activation/FileClassifierImpl.class
                    "/>
    <!-- <property name="no-dl" value="false"/> -->
    <!-- derived names - may be changed -->
```

```
    <property name="jar.file"
              value="${ant.project.name}.jar"/>
    <property name="jar.file.dl"
              value="${ant.project.name}-dl.jar"/>
    <property name="jar.file.act"
              value="${ant.project.name}-act.jar"/>
    <property name="main.class"
              value="${ant.project.name}"/>
   <property name="codebase"
              value="http://${localhost}/classes/${jar.file.dl}"/>
    <property name="jini.jars.start" value="${jini.jars}:${jini.home}/lib/
start.jar"/>
    <!-- targets -->
    <target name="all" depends="compile"/>
    <target name="compile">
        <javac destdir="${build}" srcdir="${src}"
               classpath="${jini.jars.start}"
               target="1.2"
               includes="${src.files}">
        </javac>
    </target>
    <target name="dist" depends="compile"
            description="generate the distribution">
        <jar jarfile="${dist}/${jar.file}"
             basedir="${build}"
             includes="${class.files}"/>
        <jar jarfile="${dist}/${jar.file.act}"
             basedir="${build}"
             includes="${class.files.act}"/>
        <antcall target="dist-jar-dl"/>
    </target>
    <target name="dist-jar-dl" unless="no-dl">
        <jar jarfile="${dist}/${jar.file.dl}"
             basedir="${build}"
             includes="${class.files.dl}"/>
    </target>
    <target name="build" depends="dist,compile"/>
    <target name="run" depends="build">
        <java classname="${main.class}"
              fork="true"
              classpath="${jini.jars}:${dist}/${jar.file}">
             <jvmarg value="-Djava.security.policy=${res}/policy.all"/>
             <jvmarg value="-Djava.rmi.server.codebase=${codebase}"/>
        </java>
    </target>
    <target name="deploy" depends="dist" unless="no-dl">
        <copy file="${dist}/${jar.file.dl}"
```

```
            todir="${httpd.classes}"/>
        <copy file="${dist}/${jar.file}"
            todir="${httpd.classes}"/>
    </target>
</project>
```

Nonlazy Services

The services just discussed are *lazy* services, meaning they are activated on demand when their methods are called. This reduces memory use at the expense of starting up a new JVM when required. Some services need to be continuously alive, but can still benefit from the log mechanism of `phoenix`. If `phoenix` crashes and is restarted, or the machine is rebooted and `phoenix` restarts, then it is able to use its log files to restart any "active" services registered with it, as well as restore lazy services on demand. Putting even active services under the activation system can help programmers avoid messing around with boot configuration files, by just ensuring that `phoenix` is started on reboot.

Maintaining State

An activatable object is created afresh each time a method is called on it, using its two-argument constructor. As a result, the object is created in the same state on each activation. However, method calls on objects (apart from `get...()` methods) usually result in a change of state of the object. Activatable objects will need some way of reflecting this change on each activation, and this is typically done by saving and restoring state using a disk file.

When an object is activated, one of the parameters passed to it is a `MarshalledObject` instance. This is the same object that was passed to the activation system in the `ActivationDesc` parameter to `ActivationSystem.registerObject()`. This object does not change between different activations, so it cannot hold changing state, but only data that is fixed for all activations. A simple use for `MarshalledObject` is to hold the name of a file that can be used for state. Then on each activation, the object can restore state by reading stored information, and on each subsequent method call that changes state, the information in the file can be overwritten.

The mutable file classifier, which was discussed in Chapter 16, can be sent `addType()` and `removeType()` messages. It begins with a given set of MIME type/file extension mappings. State here is very simple: it just stores all the file extensions and their corresponding MIME type in a `Map`. If you turn this service into an activatable object, you store the state by just storing the map. The state can be saved to disk using `ObjectOutputStream.writeObject()` and retrieved by `ObjectInputStream.readObject()`. More complex cases might require more complex storage methods.

The very first time a mutable file classifier starts on a particular host, it should build its initial state file. A variety of methods that can be used to achieve this. For example, if the state file does not exist, then the first activation could detect this and construct the initial state at that time. Alternatively, a method such as `init()` could be defined, to be called once after the object has been registered with the activation system.

The "normal" way of instantiating an object—through a constructor—doesn't work too well with activatable objects. If a constructor for a class doesn't start by calling another constructor with `this(...)` or `super(...)`, then the no-argument superclass constructor `super()` is called. But the class `Activatable` doesn't have a no-args constructor, so you can't

subclass from `Activatable` *and* have a constructor such as `FileClassifierMutable(String stateFile)` that doesn't use the activation system. You can avoid this problem by not inheriting from `Activatable`, and register explicitly with the activation system with the following, for example:

```
public FileClassifierMutable(ActivationID id,
          MarshalledObject data) throws java.rmi.RemoteException {
          Activatable.exportObject(this, id, 0); // continue with
          instantiation
```

This is a bit clumsy in use, as you create an object solely to build up initial state, and then discard it because the activation system will re-create it on demand.

The technique adopted in this example is to create initial state if the attempt to restore state from the state file fails for any reason as the object is activated. This is done in the `restoreMap()` method, which is called from the constructor `FileClassifierMutable (ActivationID id, MarshalledObject data)`. The name of the file is extracted from the marshalled object passed in as a parameter.

```
package activation;
import java.io.*;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;
import java.rmi.Remote;
import java.rmi.activation.ActivationID;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.EventRegistration;
import java.rmi.RemoteException;
import net.jini.core.event.UnknownEventException ;
import net.jini.export.ProxyAccessor;
import net.jini.export.*;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.tcp.TcpServerEndpoint;
import net.jini.activation.ActivationExporter;
import javax.swing.event.EventListenerList;
import common.MIMEType;
import common.MutableFileClassifier;
import mutable.RemoteFileClassifier;
import java.util.Map;
import java.util.HashMap;
/**
 * FileClassifierMutable.java
 */
public class FileClassifierMutable implements RemoteFileClassifier,
                                               ProxyAccessor {
    private Remote proxy;
```

```
/**
 * Map of String extensions to MIME types
 */
private Map map = new HashMap();
/**
 * Permanent storage for the map while inactive
 */
private String mapFile;
/**
 * Listeners for change events
 */
private EventListenerList listenerList = new EventListenerList();
public MIMEType getMIMEType(String fileName)
    throws java.rmi.RemoteException {
    System.out.println("Called with " + fileName);
    MIMEType type;
    String fileExtension;
    int dotIndex = fileName.lastIndexOf('.');
    if (dotIndex == -1 || dotIndex + 1 == fileName.length()) {
        // can't find suitable suffix
        return null;
    }
    fileExtension= fileName.substring(dotIndex + 1);
    type = (MIMEType) map.get(fileExtension);
    return type;
}
public void addType(String suffix, MIMEType type)
    throws java.rmi.RemoteException {
    map.put(suffix, type);
    fireNotify(MutableFileClassifier.ADD_TYPE);
    saveMap();
}
public void removeType(String suffix)
    throws java.rmi.RemoteException {
    if (map.remove(suffix) != null) {
        fireNotify(MutableFileClassifier.REMOVE_TYPE);
        saveMap();
    }
}
public EventRegistration addRemoteListener(RemoteEventListener listener)
    throws java.rmi.RemoteException {
    listenerList.add(RemoteEventListener.class, listener);
    return new EventRegistration(0, this, null, 0);
}
// Notify all listeners that have registered interest for
// notification on this event type.  The event instance
// is lazily created using the parameters passed into
// the fire method.
```

```
protected void fireNotify(long eventID) {
    RemoteEvent remoteEvent = null;

    // Guaranteed to return a non-null array
    Object[] listeners = listenerList.getListenerList();

    // Process the listeners last to first, notifying
    // those that are interested in this event
    for (int i = listeners.length - 2; i >= 0; i -= 2) {
        if (listeners[i] == RemoteEventListener.class) {
            RemoteEventListener listener = (RemoteEventListener) listeners[i+1];
            if (remoteEvent == null) {
                remoteEvent = new RemoteEvent(this, eventID,
                                              0L, null);
            }
            try {
                listener.notify(remoteEvent);
            } catch(UnknownEventException e) {
                e.printStackTrace();
            } catch(RemoteException e) {
                e.printStackTrace();
            }
        }
    }
}
/**
 * Restore map from file.
 * Install default map if any errors occur
 */
public void restoreMap() {
    try {
        FileInputStream istream = new FileInputStream(mapFile);
        ObjectInputStream p = new ObjectInputStream(istream);
        map = (Map) p.readObject();

        istream.close();
    } catch(Exception e) {
        e.printStackTrace();
        // restoration of state failed, so
        // load a predefined set of MIME type mappings
        map.put("gif", new MIMEType("image", "gif"));
        map.put("jpeg", new MIMEType("image", "jpeg"));
        map.put("mpg", new MIMEType("video", "mpeg"));
        map.put("txt", new MIMEType("text", "plain"));
        map.put("html", new MIMEType("text", "html"));

        this.mapFile = mapFile;
```

```
            saveMap();
        }
    }
    /**
     * Save map to file.
     */
    public void saveMap() {
        try {
            FileOutputStream ostream = new FileOutputStream(mapFile);
            ObjectOutputStream p = new ObjectOutputStream(ostream);
            p.writeObject(map);
            p.flush();
            ostream.close();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
    public FileClassifierMutable(ActivationID activationID,
                                 MarshalledObject data)
        throws java.rmi.RemoteException {
        Exporter exporter =
            new ActivationExporter(activationID,
                        new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                              new BasicILFactory(),
                                              false, true));

        proxy = (Remote) exporter.export(this);
        try {
            mapFile = (String) data.get();
        } catch(Exception e) {
            e.printStackTrace();
        }
        restoreMap();
    }
    // Implementation for ProxyAccessor
    public Object getProxy() {
        return proxy;
    }
} // FileClassifierMutable
```

The difference between the server for this service and the previous one is that you now have to prepare a marshalled object for the state file and register it with the activation system. Here the file name is hard-coded, but it could be given as a command-line argument (like services such as `reggie` do). I provide only the section of code relating to the marshalled object as that is all that changes from the previous server.

```
static final protected String CODEBASE =
        "http://192.168.1.13/classes/activation.FileClassifierServer-dl.jar";
    static final protected String LOG_FILE = "/tmp/file_classifier";
        String codebase = CODEBASE;
        MarshalledObject data = null;
        // set a log file for the service
        try {
            data = new MarshalledObject(LOG_FILE);
        } catch(java.io.IOException e) {
            e.printStackTrace();
        }
        ActivationDesc desc = null;
        desc = new ActivationDesc(groupID,
                                  "activation.FileClassifierImpl",
                                  codebase, data, true);
```

An Ant file for this server is `activation.FileClassifierServerMutable.xml`. It differs from the previous Ant file in the files used, so only these are given:

```
    <!-- files for this project -->
    <property name="src.files"
              value="common/MIMEType.java,
                     common/FileClassifier.java,
                     rmi/RemoteFileClassifier.java,
                     mutable/RemoteFileClassifier.java,
                     activation/FileClassifierMutable.java
                     activation/FileClassifierServerMutable.java
                    "/>
    <property name="class.files"
              value="
                     common/MIMEType.class,
                     common/FileClassifier.class,
                     activation/FileClassifierServerMutable.class
                    "/>
    <property name="class.files.dl"
              value="
                     rmi/RemoteFileClassifier.class
                    "/>
    <property name="class.files.act"
              value="common/MIMEType.class,
                     common/FileClassifier.class,
                     rmi/RemoteFileClassifier.class,
                     activation/FileClassifierMutable.class
                    "/>
```

The example presented here uses a simple way to store state. Sun uses a far more complex system in many of its services, such as `reggie`: a *reliable log*, in package `com.sun.jini.reliableLog`. Note that this package is not a part of standard Jini, so it may change or even be

removed in later versions of Jini, but there is nothing to stop you from using it if you need a robust storage mechanism.

Using a Configuration

The service implementations shown in the chapter so far have hard-coded the protocol Jeri. In general this is not a good idea, as a runtime configuration should specify this. The code to find an exporter should be handled by looking in a configuration, as shown in Chapter 19.

The start-up server will see the configuration file, typically a file name, as a command-line parameter. Previously for nonactivatable services, the server was able to extract the exporter directly from the configuration and use it to export the service. But as you have seen, it is now the responsibility of the service itself to define and use an exporter. The problem is how to get the command-line parameters from the start-up server into the service's constructor.

This problem can be solved by using the marshalled data discussed in the last section, but instead of using it for state, we can place the command-line arguments from the server into the marshalled data and so pass the configuration into the client.

The changes to the service are to add in configuration code to the constructor:

```
package activation;
import net.jini.export.*;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.tcp.TcpServerEndpoint;
import net.jini.activation.ActivationExporter;
import net.jini.jrmp.JrmpExporter;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;
import net.jini.export.ProxyAccessor;
import net.jini.config.Configuration;
import net.jini.config.ConfigurationException;
import net.jini.config.ConfigurationProvider;
import common.MIMEType;
import common.FileClassifier;
import rmi.RemoteFileClassifier;
import java.rmi.Remote;
/**
 * FileClassifierConfig.java
 */
public class FileClassifierConfig implements RemoteFileClassifier,
                                             ProxyAccessor {
    private Remote proxy;
    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        if (fileName.endsWith(".gif")) {
            return new MIMEType("image", "gif");
        } else if (fileName.endsWith(".jpeg")) {
            return new MIMEType("image", "jpeg");
```

```
            } else if (fileName.endsWith(".mpg")) {
                return new MIMEType("video", "mpeg");
            } else if (fileName.endsWith(".txt")) {
                return new MIMEType("text", "plain");
            } else if (fileName.endsWith(".html")) {
                return new MIMEType("text", "html");
            } else
                // fill in lots of other types,
                // but eventually give up and
                return new MIMEType(null, null);
        }
        public FileClassifierConfig(ActivationID activationID, MarshalledObject data)
            throws java.rmi.RemoteException {
            // The marshalled object should be an array of strings
            // holding a configuration
            String[] args = null;
            try {
                args = (String[]) data.get();
            } catch(Exception e) {
                // empty
            }
            Exporter defaultExporter =
                new ActivationExporter(activationID,
                             new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                                   new BasicILFactory(),
                                                   false, true));
            Exporter exporter = defaultExporter;
            try {
                Configuration config = ConfigurationProvider.getInstance(args);
                exporter = (Exporter) config.getEntry( "JeriExportDemo",
                                                       "exporter",
                                                       Exporter.class);
            } catch(ConfigurationException e) {
                // empty
            }
            proxy = (Remote) exporter.export(this);
        }
        // Implementation for ProxyAccessor
        public Object getProxy() {
            return proxy;
        }
    } // FileClassifierConfig
```

The start-up server marshalls the command-line arguments and passes them into the activation description. I provide only the code showing the use of the marshalled object:

```
    static final protected String CODEBASE =
        "http://192.168.1.13/classes/activation.FileClassifierServer-dl.jar";
    String codebase = CODEBASE;
    MarshalledObject data = null;
    // marshall the command-line args for the service
    try {
        data = new MarshalledObject(argv);
    } catch(IOException e) {
        e.printStackTrace();
        System.exit(1);
    }
    ActivationDesc desc = null;
    desc = new ActivationDesc(groupID,
                              "activation.FileClassifierImpl",
                              codebase, data, true);
```

An Ant file for this server is `activation.FileClassifierServerConfig.xml`. It differs from the Ant file given earlier only in the files used.

```
<!-- files for this project -->
<property name="src.files"
          value="common/MIMEType.java,
                 common/FileClassifier.java,
                 rmi/RemoteFileClassifier.java,
                 activation/FileClassifierConfig.java
                 activation/FileClassifierServerConfig.java
                "/>
<property name="class.files"
          value="
                 common/MIMEType.class,
                 common/FileClassifier.class,
                 activation/FileClassifierServerConfig.class
                "/>
<property name="class.files.dl"
          value="
                 rmi/RemoteFileClassifier.class
                "/>
<property name="class.files.act"
          value="common/MIMEType.class,
                 common/FileClassifier.class,
                 rmi/RemoteFileClassifier.class,
                 activation/FileClassifierConfig.class
                "/>
```

LeaseRenewalService

Activatable objects are one example of services that are not continuously alive. Mobile services, such as those exist on mobile phones, are another. These services are brought to life on demand (as activatable objects), or join the network on occasion. These services raise a number of problems, one of which we skirted around in the last section: how do you renew leases when the object is not alive?

Activatable objects are brought back to life when methods are invoked on them. The expiration of a lease does not cause any methods to be invoked. There is no "lease-expiring event" generated that could cause a listener method to be invoked, either. It is true that a `ServiceRegistrar` such as `reggie` will generate an event when a lease changes status, but this is a "service removed" event rather than a "service about to be removed" event—it is too late.

If a server is alive, then it can use a `LeaseRenewalManager` to keep leases alive, but first, the renewal manager works by sleeping and waking up just in time to renew the leases, and second, if the server exits, then no `LeaseRenewalManager` will continue to run.

Jini supplies a lease renewal service that partly avoids these problems. Since it runs as a service, it has an independent existence; it does not depend on the server for any other service. It can act like a `LeaseRenewalManager` in keeping track of leases registered with it and renewing them as needed. In general, it can keep leases alive without waking the service itself, which can slumber until activated by clients calling methods.

But how long should the `LeaseRenewalService` keep renewing leases for a service? The `LeaseRenewalManager` utility has a simple solution: keep renewing while the server for that service is alive. If the server dies, taking down a service, then it will also take down the `LeaseRenewalManager` running in the same JVM, so leases will expire as expected after an interval.

This mechanism won't work for `LeaseRenewalService`, however, because the managed service can disappear without the `LeaseRenewalService` knowing about it. So the lease renewal must be done on a leased basis itself! The `LeaseRenewalService` will renew leases for a service only for a particular amount of time, specified by a lease. The service will still have to renew its lease, even though it is with a `LeaseRenewalService` instead of a bunch of lookup services. The lease granted by this service should be of a much longer duration than those granted by the lookup services for this to be of value.

Activatable services can only be woken by calling one of their methods. The `LeaseRenewalService` accomplishes this by generating renewal events *in advance* and calling a `notify()` method on a listener. If the listener is the activatable object, the `LeaseRenewalService` will wake it up so that it can perform the renewal. If the `phoenix` process managing the service has died or is unavailable, then the event will not be delivered and the `LeaseRenewalService` can remove this service from its renewal list.

This approach is not quite satisfactory for other types of "dormant" services such as might exist on mobile phones, since there is no equivalent of `phoenix` to handle activation. Instead, the mobile phone service might determine that it will connect once a day and renew the lease, as long as the `LeaseRenewalService` agrees to keep the lease for at least a day. This is still negotiable, in that the service asks for a duration and the `LeaseRenewalService` replies with a value that might not be so long. Still, it should be better than dealing with the lookup services, which may ask for renewals as often as every five minutes.

In the sections that follow, we'll discuss the `norm` service, Sun's implementation of a `LeaseRenewalService`.

Then we look at how a client can use this service to renew leases.

The norm Service

Jini 1.1 supplied an implementation of `LeaseRenewalService` called `norm`. This was a nonlazy activatable service that required `rmid` to be running. In Jini 2.0, it has been extended to be much more flexible and is controlled by various configurations:

- JRMP
  - Transient
  - Persistent
  - Activatable (requires an activation server such as `phoenix`)
- Jeri
  - Transient
  - Persistent
  - Activatable (requires an activation server such as `phoenix`)

■ **Note** These options are all documented in the Jini documentation `doc/api/com/sun/jini/norm/package-summary.html#examples`.

For example, use this to run the transient Jeri version for suitable values of `config_dir` and `install_dir`:

```
java -Djava.security.policy=config_dir/jsk-all.policy \
     -jar install_dir/lib/start.jar \
     config_dir/start-transient-norm.config
```

The policy file could contain the following:

```
grant codebase "file:install_dir/lib/*" {
    permission java.security.AllPermission;
};
```

The `start-transient-norm.config` file should contain this:

```
import com.sun.jini.start.NonActivatableServiceDescriptor;
import com.sun.jini.start.ServiceDescriptor;
com.sun.jini.start {
    private static codebase = "http://your_host:http_port/norm-dl.jar";
    private static policy = "config_dir/jsk-all.policy";
```

```
    private static classpath = "install_dir/lib/norm.jar";
    private static config = "config_dir/transient-norm.config";
    static serviceDescriptors = new ServiceDescriptor[] {
        new NonActivatableServiceDescriptor(
            codebase, policy, classpath,
            "com.sun.jini.norm.TransientNormServerImpl",
            new String[] { config })
    };
}
```

This file points to the `transient-norm.config` file, which in turn contains the following:

```
com.sun.jini.norm {
    initialLookupGroups = new String[] { "your.group" };
}
```

Note that there is no mention of Jeri in any of these files—presumably it is a default (the JRMP version contains a definition of `serverExporter` as a JRMPExporter).

The `norm` service will maintain a set of leases for a period of up to two hours. The `reggie` lookup service grants leases for only five minutes, so that using this service increases the amount of time between renewing leases by a factor of over twenty.

Using the LeaseRenewalService

The `norm` service exports an object of type `LeaseRenewalService` that is defined by the interface:

```
package net.jini.lease;
public  interface LeaseRenewalService {
    LeaseRenewalSet createLeaseRenewalSet(long leaseDuration)
          throws java.rmi.RemoteException;
}
```

The `leaseDuration` is a requested value in milliseconds for the lease service to manage a set of leases. The lease service creates a lease for this request, and in order for it to continue to manage the set beyond the lease's expiration, the lease must be renewed before expiration. Because the service may be inactive around the time of expiration, the `LeaseRenewalSet` can be asked to register a listener object that will receive an event containing the lease, which will activate a dormant listener so that it can renew the lease in time. If the lease for the `LeaseRenewalSet` is allowed to lapse, then eventually all the leases for the services it was managing will also expire, making the services unavailable.

The `LeaseRenewalSet` returned from `createLeaseRenewalSet` has interfaces including the following:

```
package net.jini.lease;
public interface LeaseRenewalSet {
    public void renewFor(Lease leaseToRenew,
                         long membershipDuration)
                throws RemoteException;
    public EventRegistration setExpirationWarningListener(
                         RemoteEventListener listener,
```

```
                        long minWarning,
                        MarshalledObject handback)
                throws RemoteException;
    ....
}
```

The `renewFor()` method adds a new lease to the set being looked after. The `LeaseRenewalSet` will keep renewing the lease until either the requested `membershipDuration` expires or the lease on the whole `LeaseRenewalSet` expires (or an exception happens, such as a lease being refused).

Setting an expiration warning listener means that its `notify()` method will be called at least `minWarning` milliseconds before the lease for the set expires. The event argument to this will actually be an `ExpirationWarningEvent`:

```
package net.jini.lease;
public class ExpirationWarningEvent extends RemoteEvent {
    Lease getLease();
}
```

This allows the listener to get the lease for the `LeaseRenewalSet` and (probably) renew it. A simple activatable class that can renew the lease is as follows:

```
/**
 * @version 1.1
 */
package activation;
import java.rmi.Remote;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.lease.Lease;
import net.jini.lease.ExpirationWarningEvent;
import net.jini.export.*;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.tcp.TcpServerEndpoint;
import net.jini.activation.ActivationExporter;
public class RenewLease implements RemoteEventListener,
                                   ProxyAccessor  {

    private Remote proxy;
    public RenewLease(ActivationID activationID, MarshalledObject data)
        throws java.rmi.RemoteException {
        Exporter exporter =
            new ActivationExporter(activationID,
                        new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                              new BasicILFactory(),
                                              false, true));
```

```
            proxy = (Remote) exporter.export(this);
        }

        public void notify(RemoteEvent evt) {
            System.out.println("expiring... " + evt.toString());
            ExpirationWarningEvent eevt = (ExpirationWarningEvent) evt;
            Lease lease = eevt.getRenewalSetLease();
            try {
                // This is short, for testing. Try 2+ hours
                lease.renew(20000L);
            } catch(Exception e) {
                e.printStackTrace();
            }
            System.out.println("Lease renewed for " +
                               (lease.getExpiration() -
                                System.currentTimeMillis()));
        }
        public Object getProxy() {
            return proxy;
        }
    }
```

The server will need to register the service and export it as an activatable object. This is done in exactly the same way as in the first example of this chapter. In addition, it will need to

1. Register the lease listener (such as the previous `RenewLease`) with the activation system as an activatable object.
2. Find a `LeaseRenewalService` from a lookup service.
3. Register all leases from lookup services with the `LeaseRenewalService`. Since it may find lookup services before it finds the renewal service, it will need to keep a list of lookup services found before finding the service, in order to register them with it.

```
package activation;
import rmi.RemoteFileClassifier;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.lease.Lease;
import net.jini.lease.LeaseRenewalService;
import net.jini.lease.LeaseRenewalSet;
```

```
import java.rmi.RMISecurityManager;
import java.rmi.MarshalledObject;
import java.rmi.activation.ActivationDesc;
import java.rmi.activation.ActivationGroupDesc;
import java.rmi.activation.ActivationGroupDesc.CommandEnvironment;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationGroup;
import java.rmi.activation.ActivationGroupID;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;
import java.util.Properties;
import java.util.Vector;
import java.rmi.activation.UnknownGroupException;
import java.rmi.activation.ActivationException;
import java.rmi.RemoteException;
/**
 * FileClassifierServer.java
 */
public class FileClassifierServerLease
    implements DiscoveryListener {
    static final protected String SECURITY_POLICY_FILE =
        "/home/jan/projects/jini/doc/policy.all";
    // Don't forget the trailing '/'!
    static final protected String CODEBASE = "http://localhost/classes/";

    protected RemoteFileClassifier proxy;
    protected RemoteEventListener leaseProxy;
    // Lease renewal management
    protected LeaseRenewalSet leaseRenewalSet = null;
    // List of leases not yet managed by a LeaseRenewalService
    protected Vector leases = new Vector();
    public static void main(String argv[]) {
        new FileClassifierServerLease(argv);
        // stick around while lookup services are found
        try {
            Thread.sleep(10000L);
        } catch(InterruptedException e) {
            // do nothing
        }
        // the server doesn't need to exist anymore
        System.exit(0);
    }
    public FileClassifierServerLease(String[] argv) {
        // install suitable security manager
        System.setSecurityManager(new RMISecurityManager());
        // Install an activation group
        Properties props = new Properties();
```

```
props.put("java.security.policy",
        SECURITY_POLICY_FILE);
ActivationGroupDesc.CommandEnvironment ace = null;
ActivationGroupDesc group = new ActivationGroupDesc(props, ace);
ActivationGroupID groupID = null;
try {
    groupID = ActivationGroup.getSystem().registerGroup(group);
} catch(RemoteException e) {
    e.printStackTrace();
    System.exit(1);
} catch(ActivationException e) {
    e.printStackTrace();
    System.exit(1);
}

try {
    ActivationGroup.createGroup(groupID, group, 0);
} catch(ActivationException e) {
    e.printStackTrace();
    System.exit(1);
}

String codebase = CODEBASE;
MarshalledObject data = null;
ActivationDesc desc = null;
ActivationDesc descLease = null;
try {
    desc = new ActivationDesc("activation.FileClassifierImpl",
                                        codebase, data);
    descLease = new ActivationDesc("activation.RenewLease",
                                        codebase, data);
} catch(ActivationException e) {
    e.printStackTrace();
    System.exit(1);
}
try {
    proxy = (RemoteFileClassifier) Activatable.register(desc);
    leaseProxy = (RemoteEventListener) Activatable.register(descLease);
} catch(UnknownGroupException e) {
    e.printStackTrace();
    System.exit(1);
} catch(ActivationException e) {
    e.printStackTrace();
    System.exit(1);
} catch(RemoteException e) {
    e.printStackTrace();
    System.exit(1);
```

```
        }

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }
        discover.addDiscoveryListener(this);
    }
    public void discovered(DiscoveryEvent evt) {
        ServiceRegistrar[] registrars = evt.getRegistrars();
        RemoteFileClassifier service;
        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];
            // export the proxy service
            ServiceItem item = new ServiceItem(null,
                                               proxy,
                                               null);
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.print("Register exception: ");
                e.printStackTrace();
                // System.exit(2);
                continue;
            }
            try {
                System.out.println("service registered at " +
                                   registrar.getLocator().getHost());
            } catch(Exception e) {
            }
            Lease lease = reg.getLease();
            // if we have a lease renewal manager, use it
            if (leaseRenewalSet != null) {
                try {
                    leaseRenewalSet.renewFor(lease, Lease.FOREVER);
                } catch(RemoteException e) {
                    e.printStackTrace();
                }
            } else {
                // add to the list of unmanaged leases
                leases.add(lease);
                // see if this lookup service has a lease renewal manager
                findLeaseService(registrar);
```

```
            }
        }
    }
    public void findLeaseService(ServiceRegistrar registrar) {
        System.out.println("Trying to find a lease service");
        Class[] classes = {LeaseRenewalService.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);
        LeaseRenewalService leaseService = null;
        try {
            leaseService = (LeaseRenewalService) registrar.lookup(template);
        } catch(RemoteException e) {
            e.printStackTrace();
            return;
        }
        if (leaseService == null) {
            System.out.println("No lease service found");
            return;
        }
        try {
            // This time is unrealistically small - try 10000000L
            leaseRenewalSet = leaseService.createLeaseRenewalSet(20000);
            System.out.println("Found a lease service");
            // register a timeout listener
            leaseRenewalSet.setExpirationWarningListener(leaseProxy, 5000,
                                                         null);
            // manage all the leases found so far
            for (int n = 0; n < leases.size(); n++) {
                Lease ll = (Lease) leases.elementAt(n);
                leaseRenewalSet.renewFor(ll, Lease.FOREVER);
            }
            leases = null;
        } catch(RemoteException e) {
            e.printStackTrace();
        }
        Lease renewalLease = leaseRenewalSet.getRenewalSetLease();
        System.out.println("Lease expires in " +
                           (renewalLease.getExpiration() -
                            System.currentTimeMillis()));
    }
    public void discarded(DiscoveryEvent evt) {
    }
} // FileClassifierServerLease
```

An Ant file to build, deploy, and run the service is `activation.FileClassifierServerLease.xml`. It only differs from the previous Ant files in the files used.

```
<property name="src.files"
          value="
                 common/MIMEType.java,
                 common/FileClassifier.java,
                 rmi/RemoteFileClassifier.java,
                 activation/FileClassifierImpl.java
                 activation/RenewLease.java
                 activation/FileClassifierServerLease.java
                "/>
<property name="class.files"
          value="
                 common/MIMEType.class,
                 common/FileClassifier.class,
                 activation/FileClassifierServerLease.class
                "/>
<property name="class.files.dl"
          value="
                 rmi/RemoteFileClassifier.class
                "/>
<property name="class.files.act"
          value="common/MIMEType.class,
                 common/FileClassifier.class,
                 rmi/RemoteFileClassifier.class,
                 activation/RenewLease.class
                 activation/FileClassifierImpl.class
                "/>
```

In order to run the server, the following need to be running:

- `reggie` to run as a lookup service.
- `phoenix` to act as an activation server for the `FileClassifier` service and also for the `RenewLease` service.
- `norm` as a lease renewal service. Each lease will be registered with this service, and it will have the `RenewLease` as listener for expiration events.

The server starts, finds lookup services, and registers the service with each of them. Each lease that it gets is also registered with the lease renewal service, and the listener is also registered. The server then terminates. The lease renewal service renews leases with the lookup service. When the lease renewal set is about to expire, it wakes up the lease renewal listener, which renews the set. Note that since the listener is activatable, this "wake-up" is performed by the activation server `phoenix`. Trace messages from the listener thus appear in whatever window the activation server is run from.

LookupDiscoveryService

It is easy enough for a server to discover all of the lookup services within reach at the time it is started, using `LookupDiscovery`. While the server continues to stay alive, any new lookup

services that start will also be found by LookupDiscovery. But if the server terminates, which it will for activatable services, then these extra lookup services will probably never be found. This results in the service not being registered with them, which could mean in turn that clients may not find it. This is analogous to leases not being renewed if the server terminates.

In Jini 1.1, there is a LookupDiscoveryService that can be used to continuously monitor the state of lookup services. It will monitor these on behalf of a service that will most likely want to register with each new lookup service as it starts. If the service is an activatable one, the server that would have done this will have terminated, as its role would have just been to register the service with phoenix.

When there is a change to lookup services, the LookupDiscoveryService needs to notify an object about this by sending it a remote event (actually of type RemoteDiscoveryEvent). But again, we do not want to have a process sitting around waiting for such notification, so the listener object will probably also be an activatable object.

The LookupDiscoveryService interface has the following specification:

```
public interface LookupDiscoveryService {
    LookupDiscoveryRegistration register(String[] groups,
                                         LookupLocator[] locators,
                                         RemoteEventListener listener,
                                         MarshalledObject handback,
                                         long leaseDuration);
}
```

Calling the register() method will begin a multicast search for the groups and a unicast lookup for the locators. The registration is leased and will need to be renewed before expiration (a lease renewal service can be used for this). Note that the listener *cannot* be null—this is simple sanity checking, because if the listener were null, then the service could never do anything useful!

A lookup service in one of the groups can start or terminate, or it can change its group membership in such a way that it now does (or doesn't) meet the group criteria. A lookup service in the locators list can also start or stop. Any of these changes will generate RemoteDiscoveryEvent events and call the notify() method of the listener. The event interface includes the following:

```
package net.jini.discovery;
public interface RemoteDiscoveryEvent {
    ServiceRegistrar[] getRegistrars();
    boolean isDiscarded();
    ...
}
```

The list of registrars is the set that triggered the event. The isDiscarded() method is used to check if it is a "discovered" lookup service or a "discarded" lookup service. An initial event is not posted when the listener is registered; the set of lookup services that are initially found can be retrieved from the LookupDiscoveryRegistration object returned from the register() method, by its getRegistrars()method.

The fiddler Service

The Jini 1.1 release includes an implementation of the lookup discovery service called `fiddler`. This service has been modified in Jini 2.0 to be more flexible. It can be run in three modes, using either Jeri (the default) or JRMP:

- Transient
- Persistent
- Activatable (requires `phoenix` to be running)

Information about how to run `fiddler` in each mode is given in the Jini download, under `file:///usr/local/jini2_0/doc/api/com/sun/jini/fiddler/package-summary.html`.

To run `fiddler` in transient mode using Jeri over TCP, execute a command line such as the following:

```
java \
  -Djava.security.manager= \
  -Djava.security.policy=fiddler-start-transient.policy \
    -jar jini_install_dir/lib/start.jar \
           config/fiddler-start-transient.config
```

where `fiddler-start-transient.policy` could be the same as `policy.all`.

The contents of `fiddler-start-transient.config` could be as follows:

```
import com.sun.jini.start.NonActivatableServiceDescriptor;
import com.sun.jini.start.ServiceDescriptor;
com.sun.jini.start {
    private static serviceCodebase   =
            new String("http://myHost:8080/fiddler-dl.jar");
    private static servicePolicyFile =
            new String("example_install_dir${/}policy${/}jeri-transient-fiddler.pol-
icy");
    private static serviceClasspath  =
            new String("jini_install_dir${/}lib${/}fiddler.jar");
    private static serviceImplName   =
            new String("com.sun.jini.fiddler.TransientFiddlerImpl");
    private static serviceConfig     =
            new String("example_install_dir${/}config${/}jeri-transient-fiddler.con-
fig");
    private static serviceArgsArray  = new String[] { serviceConfig };
    private static nonActivatableServiceDescriptor =
                  new NonActivatableServiceDescriptor(serviceCodebase,
                                                      servicePolicyFile,
                                                      serviceClasspath,
                                                      serviceImplName,
                                                      serviceArgsArray);
    static serviceDescriptors =
                new ServiceDescriptor[] { nonActivatableServiceDescriptor };
}//end com.sun.jini.start
```

The configuration file `jeri-transient-fiddler.config` would contain this:

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
com.sun.jini.fiddler {
    private invocationLayerFactory = new BasicILFactory();
    serverExporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                           invocationLayerFactory,
                                           false,
                                           true);
    initialLookupGroups = new String[] {"myGroup.myCompany.com"};
}//end com.sun.jini.fiddler
```

Using the LookupDiscoveryService

An activatable service can make use of a lease renewal service to look after the leases for lookup services discovered. It can find these lookup services by means of a lookup discovery service. The logic to manage these two services could be a little tricky, as we attempt to find two different services. We can simplify for this example by just doing a sequential search using a `ServiceDiscoveryManager`.

While lease management can be done by the lease renewal service, the lease renewal set will also be leased and will need to be renewed on occasion. The lease renewal service can call an activatable `RenewLease` object to do this, as in the last section.

The lookup discovery service is also a leased service—it will only report changes to lookup services while its *own* lease is current. So the lease from this service will have to be managed by the lease renewal service, in addition to the leases for any lookup services discovered.

The primary purpose of the lookup discovery service is to call the `notify()` method of some object when information about lookup services changes. This object should also be an activatable object. We define a `DiscoveryChange` object with the `notify()` method to handle changes in lookup services. If a lookup service has disappeared, we don't worry about it. If a lookup service has been discovered, we want to register the service with it, and then manage the resultant lease. This means that the `DiscoveryChange` object must know both the service to be registered and the lease renewal service. This is static data, so these two objects can be passed in an array of two objects as the `MarshalledObject` to the activation constructor. The class itself can be implemented as follows:

```
package activation;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.lease.Lease;
import net.jini.lease.ExpirationWarningEvent;
import net.jini.core.lookup.ServiceItem;
```

```
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.lease.LeaseRenewalSet;
import net.jini.discovery.RemoteDiscoveryEvent;
import java.rmi.RemoteException;
import  net.jini.discovery.LookupUnmarshalException;
import rmi.RemoteFileClassifier;
public class DiscoveryChange extends Activatable
    implements RemoteEventListener {
    protected LeaseRenewalSet leaseRenewalSet;
    protected RemoteFileClassifier service;

    public DiscoveryChange(ActivationID id, MarshalledObject data)
        throws java.rmi.RemoteException {
        super(id, 0);
        Object[] objs = null;
        try {
            objs = (Object []) data.get();
        } catch(ClassNotFoundException e) {
            e.printStackTrace();
        } catch(java.io.IOException e) {
            e.printStackTrace();
        }
        service = (RemoteFileClassifier) objs[0];
        leaseRenewalSet= (LeaseRenewalSet) objs[1];
    }

    public void notify(RemoteEvent evt) {
        System.out.println("lookups changing... " + evt.toString());
        RemoteDiscoveryEvent revt = (RemoteDiscoveryEvent) evt;
        if (! revt.isDiscarded()) {
            // The event is a discovery event
            ServiceItem item = new ServiceItem(null, service, null);
            ServiceRegistrar[] registrars = null;
            try {
                registrars = revt.getRegistrars();
            } catch(LookupUnmarshalException e) {
                e.printStackTrace();
                return;
            }
            for (int n = 0; n < registrars.length; n++) {
                ServiceRegistrar registrar = registrars[n];

                ServiceRegistration reg = null;
                try {
                    reg = registrar.register(item, Lease.FOREVER);
                    leaseRenewalSet.renewFor(reg.getLease(), Lease.FOREVER);
```

```
                } catch(java.rmi.RemoteException e) {
                    System.err.println("Register exception: " + e.toString());
                }
            }
        }
    }
}
```

The server must install an activation group, and then find activation proxies for the service itself and also for our lease renewal object. After this, it can use a `ClientLookupManager` to find the lease service, and register our lease renewal object with it. Now that it has a proxy for the service object, and also a lease renewal service, it can create the marshalled data for the lookup discovery service and register this with `phoenix`. Now we can find the lookup discovery service and register our discovery change listener, `DiscoveryChange`, with it. At the same time, we have to register the service with all the lookup services the lookup discovery service finds on initialization. This all leads to the following code:

```
package activation;
import rmi.RemoteFileClassifier;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.LookupDiscoveryService;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.discovery.LookupDiscoveryRegistration;
import net.jini.discovery.LookupUnmarshalException;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.event.RemoteEvent;
import net.jini.core.event.RemoteEventListener;
import net.jini.core.lease.Lease;
import net.jini.lease.LeaseRenewalService;
import net.jini.lease.LeaseRenewalSet;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lookup.ServiceDiscoveryManager;
import java.rmi.RMISecurityManager;
import java.rmi.activation.ActivationDesc;
import java.rmi.activation.ActivationGroupDesc;
import java.rmi.activation.ActivationGroupDesc.CommandEnvironment;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationGroup;
import java.rmi.activation.ActivationGroupID;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;
import java.rmi.activation.UnknownGroupException;
import java.rmi.activation.ActivationException;
```

```
import java.rmi.RemoteException;
import java.util.Properties;
import java.util.Vector;
/**
 * FileClassifierServerDiscovery.java
 */
public class FileClassifierServerDiscovery
    /* implements DiscoveryListener */ {
    private static final long WAITFOR = 10000L;
    static final protected String SECURITY_POLICY_FILE =
        "/home/jan/projects/jini/doc/policy.all";
    // Don't forget the trailing '/'!
    static final protected String CODEBASE = "http://localhost/classes/";

    protected RemoteFileClassifier serviceProxy;
    protected RemoteEventListener leaseProxy,
                                  discoveryProxy;
    // Services
    protected LookupDiscoveryService discoveryService = null;
    protected LeaseRenewalService leaseService = null;
    // Lease renewal management
    protected LeaseRenewalSet leaseRenewalSet = null;
    // List of leases not yet managed by a LeaseRenewalService
    protected Vector leases = new Vector();
    protected ServiceDiscoveryManager clientMgr = null;
    public static void main(String argv[]) {
        new FileClassifierServerDiscovery();
        // stick around while lookup services are found
        try {
            Thread.sleep(20000L);
        } catch(InterruptedException e) {
            // do nothing
        }
        // the server doesn't need to exist anymore
        System.exit(0);
    }
    public FileClassifierServerDiscovery() {
        // install suitable security manager
        System.setSecurityManager(new RMISecurityManager());
        installActivationGroup();
        serviceProxy = (RemoteFileClassifier)
                       registerWithActivation("activation.FileClassifierImpl", null);

        leaseProxy = (RemoteEventListener)
                       registerWithActivation("activation.RenewLease", null);
        initClientLookupManager();
        findLeaseService();
```

```
        // the discovery change listener needs to know the service
        // and the lease service
        Object[] discoveryInfo = {serviceProxy, leaseRenewalSet};
        MarshalledObject discoveryData = null;
        try {
            discoveryData = new MarshalledObject(discoveryInfo);
        } catch(java.io.IOException e) {
            e.printStackTrace();
        }
        discoveryProxy = (RemoteEventListener)
                         registerWithActivation("activation.DiscoveryChange",
                                                discoveryData);
        findDiscoveryService();
    }
    public void installActivationGroup() {
        Properties props = new Properties();
        props.put("java.security.policy",
                  SECURITY_POLICY_FILE);
        ActivationGroupDesc.CommandEnvironment ace = null;
        ActivationGroupDesc group = new ActivationGroupDesc(props, ace);
        ActivationGroupID groupID = null;
        try {
            groupID = ActivationGroup.getSystem().registerGroup(group);
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }

        try {
            ActivationGroup.createGroup(groupID, group, 0);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
    public Object registerWithActivation(String className, MarshalledObject data) {
        String codebase = CODEBASE;
        ActivationDesc desc = null;
        Object proxy = null;
        try {
            desc = new ActivationDesc(className,
                                      codebase, data);
        } catch(ActivationException e) {
            e.printStackTrace();
```

```
            System.exit(1);
        }
        try {
            proxy = Activatable.register(desc);
        } catch(UnknownGroupException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        } catch(RemoteException e) {
            e.printStackTrace();
            System.exit(1);
        }
        return proxy;
    }
    public void initClientLookupManager() {
        LookupDiscoveryManager lookupDiscoveryMgr = null;
        try {
            lookupDiscoveryMgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            clientMgr = new ServiceDiscoveryManager(lookupDiscoveryMgr,
                                                    new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
    public void findLeaseService() {
        leaseService = (LeaseRenewalService) findService(LeaseRenewalService.class);
        if (leaseService == null) {
            System.out.println("Lease service null");
        }
        try {
            leaseRenewalSet = leaseService.createLeaseRenewalSet(20000);
            leaseRenewalSet.setExpirationWarningListener(leaseProxy, 5000,
                                                         null);
        } catch(RemoteException e) {
            e.printStackTrace();
        }
    }
    public void findDiscoveryService() {
        discoveryService = (LookupDiscoveryService) findService(LookupDiscoverySer-
vice.class);
        if (discoveryService == null) {
```

```
            System.out.println("Discovery service null");
        }
        LookupDiscoveryRegistration registration = null;
        try {
            registration =
                discoveryService.register(LookupDiscovery.ALL_GROUPS,
                                          null,
                                          discoveryProxy,
                                          null,
                                          Lease.FOREVER);
        } catch(RemoteException e) {
            e.printStackTrace();
        }
        // manage the lease for the lookup discovery service
        try {
            leaseRenewalSet.renewFor(registration.getLease(), Lease.FOREVER);
        } catch(RemoteException e) {
            e.printStackTrace();
        }
        // register with the lookup services already found
        ServiceItem item = new ServiceItem(null, serviceProxy, null);
        ServiceRegistrar[] registrars = null;
        try {
            registrars = registration.getRegistrars();
        } catch(RemoteException e) {
            e.printStackTrace();
            return;
        } catch(LookupUnmarshalException e) {
            e.printStackTrace();
            return;
        }
        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
                leaseRenewalSet.renewFor(reg.getLease(), Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.println("Register exception: " + e.toString());
            }
        }
    }
    public Object findService(Class cls) {
        Class [] classes = new Class[] {cls};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);
        ServiceItem item = null;
```

```
        try {
            item = clientMgr.lookup(template,
                                    null, /* no filter */
                                    WAITFOR /* timeout */);
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
        if (item == null) {
            // couldn't find a service in time
            System.out.println("No service found for " + cls.toString());
            return null;
        }
        return item.service;
    }
} // FileClassifierServerDiscovery
```

To run this example, you need to perform the following steps:

1. Run the lookup service `reggie`.
2. Run the activation server `phoenix`.
3. Run the lease renewal service `norm`.
4. Run the lookup discovery service `fiddler`.
5. Run the server. This will terminate, hopefully after finding the services and registering the `DiscoveryChange` with the lookup discovery service, and register the leases for the service and the discovery service.

An Ant file to build, deploy, and run the service is `activation.FileClassifierServerDiscovery.xml`. It only differs from the previous Ant files in the files used.

```
    <!-- files for this project -->
    <property name="src.files"
              value="common/MIMEType.java,
                     common/FileClassifier.java,
                     rmi/RemoteFileClassifier.java,
                     activation/FileClassifierImpl.java
                     activation/FileClassifierServer.java
                    "/>
    <property name="class.files"
              value="
                     common/MIMEType.class,
                     common/FileClassifier.class,
                     activation/FileClassifierServer.class
                    "/>
    <property name="class.files.dl"
```

```
            value="
                   rmi/RemoteFileClassifier.class
                  "/>
    <property name="class.files.act"
            value="common/MIMEType.class,
                   common/FileClassifier.class,
                   rmi/RemoteFileClassifier.class,
                   activation/FileClassifierImpl.class
                  "/>
```

Summary

Some objects may not always be available, either because of mobility issues or because they are activatable objects. This chapter has dealt with activatable objects, and also with some of the special services that are needed to properly support these transient objects.