CHAPTER 20

Activation

MANY OF THE EXAMPLES IN EARLIER CHAPTERS use RMI proxies for services. These services subclass `UnicastRemoteObject` and 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 it; if it 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 will be idle most of the time, probably swapped out to disk, but still using virtual memory. In JDK 1.2, the memory requirements on the server side can be enormous (hopefully this will be fixed, but at the moment this is a severe embarrassment to Java and a potential threat to the success of Jini). In JDK 1.2, there is an extension to RMI called Activation, which allows an idle object to die and be recalled to life when needed. In this way, it does not occupy virtual memory while idle. Of course, a process needs to be alive to restore such objects, and RMI supplies a daemon `rmid` to manage this. In effect, `rmid` acts as another virtual memory manager because it stores information about dormant Java objects in its own files and restores them from there as needed.

There is a serious limitation to `rmid`: it is a Java program itself, and when running also uses enormous amounts of memory. So it only makes sense to use this technique when you expect to be running a number of largely idle services on the same machine. When a service is recalled to life, or activated, a new JVM may be started to run the object. This again increases memory use.

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is also designed to work with distributed objects and allows JVMs to hold remote references to objects that are no longer active.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

The principal place that this is used in the standard Jini distribution is with the `reggie` lookup service. `reggie` is an activatable service that starts, registers itself with `rmid`, and then exits. Whenever lookup services are required, `rmid` restarts `reggie` in a new JVM. Clients of the lookup service are unaware of this mechanism; they simply make calls on their proxy `ServiceRegistration` object and the Activation system looks after the rest. The main problem is for the system administrator—getting `reggie` to work in the first place!

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.

The next sections show how to create a service as an activatable object that starts life in a server that sets up the activation group. Issues related to activation, such as security and state maintenance, will also be discussed.

The Service

An activable object subclasses from `Activatable` and uses a special two-argument constructor that will be called when the object needs to be reconstructed. There is a standard implementation of this constructor that just calls the superclass constructor:

```
public ActivatableImpl(ActivationID id, MarshalledObject data)
    throws RemoteException {
    super(id, 0);
}
```

(The use of the marshalled object parameter is discussed later in the "Maintaining State" section). Adding this constructor is all that is normally needed to change a remote service (that implements `UnicastRemoteObject`) into an activatable service. For example, an activatable version of the remote file classifier described in Chapter 9 in the "RMI Proxy for FileClassifier" section is as follows:

```
package activation;

import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;

import common.MIMEType;
import common.FileClassifier;
import rmi.RemoteFileClassifier;

/**
```

```
 * FileClassifierImpl.java
 */

public class FileClassifierImpl extends Activatable
                                implements RemoteFileClassifier {

    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 id, MarshalledObject data)
        throws java.rmi.RemoteException {
        super(id, 0);
    }

} // FileClassifierImpl
```

Note that an activatable object cannot have a default no-args constructor to initialize itself, since this new constructor is required for the object to be constructed by the activation system.

The Server

The server needs to create an activation group for the objects to run in. The main issue involved here is to set a security policy file. There are two security policies in activatable objects: the policy used to create the server and export the service, and the policy used to run the service. The activation group sets a policy file for running methods of the service object. The policy file for the server is set using the normal `-Djava.security.policy=...` argument to start the server. After setting various parameters, the activation group is set for the JVM by `ActivationGroup.createGroup()`.

Remote objects that subclass `UnicastRemoteObject` are created in the normal way using a constructor on the server. Activatable objects are not constructed in the server but are instead registered with `rmid`, which will look after construction on an as-needed basis.

In order to create activatable objects, `rmid` needs to know the class name and the location of the class files. The server wraps these up in an `ActivationDesc`, and registers this with `rmid` by using `Activatable.register()`. This returns an RMI stub object that can be registered with lookup services using the `ServiceRegistrar.register()` methods. This is also a little different from subclasses of `UnicastRemoteObject`, which pass an object that is converted to a stub by the RMI runtime. The required actions, in point form, are as follows:

- A service creates a subclass of `UnicastRemoteObject` using its constructor.
- A subclass of `Activatable` is created by `rmid` using a special constructor.
- For a `UnicastRemoteObject` object, the server needs to know the class files for the class in its `CLASSPATH` and the client needs to know the class files for the stub from an HTTP server.
- For an `Activatable` object, `rmid` needs to know the class files from an HTTP server, the server must be able to find the stub files from its `CLASSPATH`, and the client must be able to get the stub files from an HTTP server.
- A server hands a `UnicastRemoteObject` object to the `ServiceRegistrar.register()`. This is converted to the stub by the RMI runtime.
- A server gets a stub for an `Activatable` object from `Activatable.register()`. This stub is given directly to `ServiceRegistrar.register()`.

Changes need to be made to servers that export activatable objects instead of unicast remote objects. The server in Chapter 9, in the "RMI Proxy for FileClassifier" section, creates a unicast remote object and exports its RMI proxy to lookup services by passing the remote object to the `ServiceRegistrar.register()` method. The changes for such servers to export activatable objects are as follows:

- An activation group has to be created with a security policy file.
- The service is not created explicitly but is instead registered with `rmid`.

- The return object from the registration is a stub that can be registered with lookup services.
- Leasing vanishes—the server just exits. The service will just expire after a while. See the "LeaseRenewalService" section, later in the chapter, for more details on how to keep the service alive.

The file classifier server using an activatable service would look like this:

```
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.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.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/jan/projects/jini/doc/policy.all";
    // Don't forget the trailing '/'!
```

```
static final protected String CODEBASE = "http://localhost/classes/";

// protected FileClassifierImpl impl;
protected RemoteFileClassifier stub;

public static void main(String argv[]) {
    new FileClassifierServer(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 FileClassifierServer(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;
        try {
            desc = new ActivationDesc("activation.FileClassifierImpl",
                                                 codebase, data);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }
System.out.println("Group ID " + ActivationGroup.currentGroupID().toString());
```

Delete this line - it was for debugging

```
        try {
            stub = (RemoteFileClassifier) 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);
        }

        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,
                                               stub,
                                               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 backend and the server must be compiled as usual, and in addition, an RMI stub object must be created for the service backend using the `rmic` compiler (in JDK 1.2, at least). The class files for the stub must be copied to somewhere where an HTTP server can deliver them to clients. This is the same as for any other RMI stubs.

There is an extra step that must be performed for Activatable objects: the activation server `rmid` must be able to reconstruct a copy of the service backend (the client must be able to reconstruct a copy of the service's stub). This means that `rmid` must have access to the class files of the service backend, either from an HTTP server or from the file system. In the previous server, the `codebase` property in the `ActivationDesc` is set to an HTTP URL, so the class files for the service backend must be accessible to an HTTP server. Note that `rmid` does not get the class files for a service backend from the `CLASSPATH`, but from the codebase of the service. The HTTP server need not be on the same machine as the service backend.

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

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

- The service provider, which specifies the location of class files in its codebase.
- `rmid`, which must be running on the same machine as the service provider and must be started before the service provider. It gets class files using the codebase of the service.
- 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 RMI stub. The service will be created on demand by `rmid`. You only need to run the server once, since `rmid` keeps information about the service in its own log files.

Security

The JVM for the service will be created by `rmid` and will be running in the same environment as `rmid`. Such things as the current directory for the service will be the same as for `rmid`, not from where the server ran. Similarly, the user ID for the service will be the user ID of `rmid`. This is a potential security problem in multi-user systems. For example, any user on a Unix system could write a service that attempts to read the shadow password file on the system, as an activatable service. Once registered with `rmid`, this same user could write a client that calls the appropriate methods on the service. If `rmid` is running in privileged mode, owned by the superuser of the system, then the service will run in that same mode and will happily read any file in the entire file system! For safety, `rmid` should probably be run using the user ID `nobody`, much like the recommendations for HTTP servers.

Some of the security issues with `rmid` have been addressed in JDK 1.3. These were discussed in Chapter 12, and they allow a security policy to be associated with each activatable service.

Non-Lazy Services

The types of services discussed in this chapter so far are "lazy" services, 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 logging mechanism of `rmid`. If `rmid` crashes and is restarted, or the machine is rebooted and `rmid` restarts, then the server is able to use its log files to restart any "active" services registered with it, as well as to restore "lazy" services on demand. By making services non-lazy and ensuring that `rmid` is started on reboot, you can avoid messing around with boot configuration files.

Maintaining State

An activatable object is created afresh each time a method is called on it, using its two-argument constructor. The default action, calling `super(id, 0)` will result in the object being 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 saving and restoring state using a disk file typically does this.

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 `Activation.register()`. This object does not change between different activations, so it cannot hold changing state, but only data, which is fixed for all activations. A simple use for it 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. On each subsequent method call that changes state, the information in the file can be overwritten.

The mutable file classifier example was discussed in Chapter 14—it could be sent `addType()` and `removeType()` messages. It begins with a given set of MIME type/ file extension mappings. State here is very simple; it is just a matter of storing all the file extensions and their corresponding MIME types in a `Map`. If we turn this into an activatable object, we store the state by just storing the map. This map can be saved to disk using `ObjectOutputStream.writeObject()`, and it can be retrieved by `ObjectInputStream.readObject()`. More complex cases might need more complex storage methods.

The very first time a mutable file classifier starts on a particular host, it should build its initial state file. There are a variety of methods that could be used. 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 very 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. However, 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 registering explicitly with the activation system, like this:

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

Nevertheless, this is a bit clumsy: you create an object solely to build up initial state, and then discard it because the activation system will recreate it on demand.

The technique we'll use here is to create initial state if the attempt to restore state from the state file fails for any reason when the object is activated. This is done in the `restoreMap()` method called from the constructor `FileClassifier-Mutable(ActivationID id, MarshalledObject data)`. The name of the file is extracted from the marshalled object passed in as parameter.

```
package activation;

import java.io.*;
import java.rmi.activation.Activatable;
import java.rmi.activation.ActivationID;
import java.rmi.MarshalledObject;

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 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 extends  Activatable
                                implements RemoteFileClassifier {

    /**
     * Map of String extensions to MIME types
     */
    protected Map map = new HashMap();

    /**
     * Permanent storage for the map while inactive
     */
    protected String mapFile;

    /**
     * Listeners for change events
     */
    protected 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 removeMIMEType(String suffix, MIMEType type)
    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 id, MarshalledObject data)
        throws java.rmi.RemoteException {
        super(id, 0);
        try {
            mapFile = (String) data.get();
        } catch(Exception e) {
            e.printStackTrace();
        }
        restoreMap();
    }
} // FileClassifierMutable
```

The difference between the server for this service and the last one is that we now have to prepare a marshalled object for the state file and register it with the activation system. Here the filename is hard-coded, but it could be given as a command line argument (as services such as `reggie` do).

```
package activation;

import mutable.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.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.util.Properties;

import java.rmi.activation.UnknownGroupException;
import java.rmi.activation.ActivationException;
import java.rmi.RemoteException;

/**
 * FileClassifierServerMutable.java
 */

public class FileClassifierServerMutable 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/";
    static final protected String LOG_FILE = "/tmp/file_classifier";

    // protected FileClassifierImpl impl;
    protected RemoteFileClassifier stub;

    public static void main(String argv[]) {
        new FileClassifierServerMutable(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 FileClassifierServerMutable(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;
        try {
            data = new MarshalledObject(LOG_FILE);
        } catch(java.io.IOException e) {
            e.printStackTrace();
        }

        ActivationDesc desc = null;
        try {
            desc = new ActivationDesc("activation.FileClassifierMutable",
                                                 codebase, data);
        } catch(ActivationException e) {
            e.printStackTrace();
            System.exit(1);
        }
System.out.println("Group ID " + ActivationGroup.currentGroupID().toString());
```
delete this debugging line

```
        try {
            stub = (RemoteFileClassifier) 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);
        }
        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,
                                               stub,
                                               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) {

    }
```

```
} // FileClassifierServerMutable
```

This example used a simple way of storing state. Sun uses a far more complex system in many of its services, such as `reggie`–a "reliable log" in the package `com.sun.jini.reliableLog`. However, this package is not a part of standard Jini, so it may change or even be removed in later versions of Jini. There is nothing to stop you from using it, though, if you need a robust storage mechanism.

LeaseRenewalService

Activatable objects are an example of services that are not continuously alive. Mobile services, such as those that will exist on mobile phones, are another. These services will be brought to life on demand (as activatable objects), or will join the network on occasion. These services raise a number of problems, and one was 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, and 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—at that point it is too late.

If a server is alive, then it can use a `LeaseRenewalManager` to keep leases alive, but there are two problems with this: 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 1.1 supplies a lease renewal service that partly solves these problems. Since it runs as a service, it has an independent existence that does not depend on the server for any other service. It can act like a `LeaseRenewalManager` in keeping track of leases registered with it, renewing them as needed. In general, it can keep leases alive without waking the service itself, which can slumber until it is activated by clients calling methods.

There is a small hiccup in this system, though: 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.

But this mechanism won't work for `LeaseRenewalService` 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, as specified by a lease. The service will still have to renew its lease, but with a `LeaseRenewalService` instead of a bunch of lookup services. The lease granted by this service will need to be of 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, this will wake it up so that it can perform the renewal. If the rmid 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 is not quite satisfactory for other types of "dormant" services, such as might exist on mobile phones, since there is no equivalent of rmid to handle activation. Instead, the mobile phone service might say 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.

The Norm Service

Jini 1.1 supplies an implementation of LeaseRenewalService called norm. This is a non-lazy Activatable service that requires rmid to be running. This is run with the following command

```
java -jar [setup_jvm_options] executable_jar_file
          codebase_arg  norm_policy_file_arg
          log_directory_arg
          [groups] [server_jvm] [server_jvm_args]
```

as in the following

```
java -jar \
        -Djava.security.policy=/files/jini1_1/example/txn/policy.all \
        /files/jini1_1/lib/norm.jar \
        http://`hostname`:8080/norm-dl.jar \
        /files/jini1_1/example/books/policy.all /tmp/norm_log
```

The first security file defines the policy that will be used for the server startup. The norm.jar file contains the class files for the norm service. This exports RMI stubs, and the class definitions for these are in norm-dl.jar. The second security file defines the policy file that will be used in the execution of the LeaseRenewal-Service methods. Finally, the log file is used to keep state, so that it can keep track of the leases it is managing.

The norm service will maintain a set of leases for a period of up to two hours. The reggie lookup service only grants leases for five minutes, so using this service increases the amount of time between renewing leases by a factor of over 20.

Using the LeaseRenewalService

The norm service exports an object of type LeaseRenewalService, which is defined by the following interface:

```
package net.jini.lease;

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

A server that wants to use the lease renewal service will first find this service and then call the createLeaseRenewal() method. The server requests a leaseDuration value, measured in milliseconds, for the lease service to manage a set of leases. The lease service creates a lease for this request, but the lease time may be less than the requested time (for norm, it is a maximum of two hours). In order for the lease service to continue to manage the set beyond the lease's expiry, the lease must be renewed before expiration. Since the service may be inactive at the time of expiry, the LeaseRenewalSet can be asked to register a listener object that will receive an event containing the lease. This will activate a dormant listener so that the listener 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 an interface 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 `membership-Duration` expires or the lease for the whole `LeaseRenewalSet` expires (or until an exception happens, like a lease being refused).

Setting an expiration warning listener means that the `notify()` method of the listener will be called at least `minWarning` milliseconds before the lease for the set expires. The event argument 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. Here is a simple activatable class that can renew the lease:

```
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;

public class RenewLease extends Activatable
    implements RemoteEventListener {

    public RenewLease(ActivationID id, MarshalledObject data)
        throws java.rmi.RemoteException {
        super(id, 0);
    }

    public void notify(RemoteEvent evt) {
        System.out.println("expiring... " + evt.toString());
        ExpirationWarningEvent eevt = (ExpirationWarningEvent) evt;
        Lease lease = eevt.getLease();
        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()));
    }
}
```

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 `FileClassifierServer` example of the first section of this chapter. In addition, it will need to do a few other things:

- It will need to register the lease listener with the activation system as an activatable object.
- It will need to find a `LeaseRenewalService` from a lookup service.
- It will need to 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.

Adding these additional requirements to the `FileClassifierServer` of the first section results in this server:

```
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;

/**
 * FileClassifierServerLease.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 stub;

    protected RemoteEventListener leaseStub;

    // 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 {
            stub = (RemoteFileClassifier) Activatable.register(desc);
            leaseStub = (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,
                                               stub,
                                               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(leaseStub, 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
```

LookupDiscoveryService

It is easy enough for a server to discover all of the lookup services within reach at the time it is started, by 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 activable services, then any new lookup services will probably never be found. This will result in the service not being registered with them, which could mean that clients may not find it. This is analogous to leases not being renewed if the server terminates.

Jini 1.1 supplies a service, the LookupDiscoveryService, that can be used to continuously monitor the state of lookup services. It will monitor them 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 registered the service will have terminated, as its role would have just been to register the service with `rmid`.

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:

```
package this line should read "package net.jini.discovery;"
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 unicast lookup for the `locators`. The registration is leased and will need to be renewed before expiring (a lease renewal service can be used for this). Note that the listener cannot be `null`–this is simple sanity checking, for if the listener was `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. These 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 whether the lookup service 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()`.

The Fiddler Service

The Jini 1.1 release includes an implementation of the lookup discovery service called `fiddler`. It is a non-lazy activatable service and is started much like other services, such as `reggie`:

I have messy layout here. Align "codebase" on line 2 with "-jar" on line 1. Place "[groups and locators]" after "log_directory_arg". The last two bits "[server_jvm] {server_jvm_args]" can be on the 4th line

```
java -jar [setup_jvm_options] executable_jar_file
          codebase_arg  fiddler_policy_file_arg
      log_directory_arg
          [groups and
    locators] [server_jvm] [server_jvm_args]
```

For example,

```
java -jar \
        -Djava.security.policy=/files/jini1_1/example/txn/policy.all \
        /files/jini1_1/lib/fiddler.jar \
        http://`hostname`:8080/norm-dl.jar \
        /files/jini1_1/example/books/policy.all /tmp/fiddler_log
```

Using the LookupDiscoveryService

An activatable service can make use of a lease renewal service to look after the leases for discovered lookup services. It can find these lookup services by means of a lookup discovery service. The logic that manages these two services is a little tricky.

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 shown in the preceding section of this chapter.

The lookup discovery service is also a leased service—it will only report changes to lookup services while its own lease is current. Therefore, 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 a `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 shown here:

```
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 the lease renewal object. After this, it can use a `ClientLookupManager` to find the lease service and register the 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 `rmid`. Then 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 server:

```
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.ClientLookupManager;

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 serviceStub;

    protected RemoteEventListener leaseStub,
                              discoveryStub;

    // 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 ClientLookupManager 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();
        serviceStub = (RemoteFileClassifier)
                    registerWithActivation("activation.FileClassifierImpl", null);

        leaseStub = (RemoteEventListener)
                     registerWithActivation("activation.RenewLease", null);

        initClientLookupManager();

        findLeaseService();

        // the discovery change listener needs to know the service and the lease
service
        Object[] discoveryInfo = {serviceStub, leaseRenewalSet};
        MarshalledObject discoveryData = null;
        try {
            discoveryData = new MarshalledObject(discoveryInfo);
        } catch(java.io.IOException e) {
            e.printStackTrace();
        }
        discoveryStub = (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);
        }
```

// the discovery change listener needs to know
// the service and the lease service

```
    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 stub = null;

    try {
        desc = new ActivationDesc(className,
                                        codebase, data);
    } catch(ActivationException e) {
        e.printStackTrace();
        System.exit(1);
    }

    try {
        stub = 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 stub;
}

public void initClientLookupManager() {
    LookupDiscoveryManager lookupDiscoveryMgr = null;
    try {
        lookupDiscoveryMgr =
            new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                        null /* unicast locators */,
                                        null /* DiscoveryListener */);
            clientMgr = new ClientLookupManager(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(leaseStub, 5000,
                                                         null);
        } catch(RemoteException e) {
            e.printStackTrace();
        }
    }

    public void findDiscoveryService() {
        discoveryService = (LookupDiscoveryService)
findService(LookupDiscoveryService.class);
        if (discoveryService == null) {
            System.out.println("Discovery service null");
        }
        LookupDiscoveryRegistration registration = null;
        try {
            registration =
                discoveryService.register(LookupDiscovery.ALL_GROUPS,
                                          null,
                                          discoveryStub,
                                          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) {
```

indent this to start under "(LookupDis..."

```
            e.printStackTrace();
        }

        // register with the lookup services already found
        ServiceItem item = new ServiceItem(null, serviceStub, 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
```

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.