CHAPTER 13

More Complex Examples

CHAPTER 8 LOOKED AT A SIMPLE JINI APPLICATION. In Chapter 9, some of the architectural choices for services were explored. There are, however, many other issues involved in building Jini services and clients.

This chapter delves into some of the more complex things that can happen with Jini applications. It covers issues such as the location of class files, multithreading, extending the matching algorithm used by Jini service locators, finding a service once only, and lease management. These are issues that can arise using the Jini components discussed so far. There are also further aspects to Jini that are explored in later chapters.

Where Are the Class Files?

Clients, servers, and service locators can use class files from a variety of sources. Which source they use can depend on the structure of the client and the server. This section looks at some of the variations that can occur.

Problem Domain

A service may require information about a client before it can (or will) proceed. For example, a banking service may require a user ID and a PIN number. Using the techniques discussed in earlier chapters, this could be done by the client collecting the information and calling suitable methods, such as `void setName(String name)` in the service (or more likely, in the service's proxy) running in the client, as shown here:

```
public class Client {
    String getName() {
        ...
        service.setName(...);
        ...
    };
}

class Service {
    void setName(String name) {
```

```
        ...
    };
}
```

A service may wish to have more control over the setting of names and passwords than this. For example, it may wish to run verification routines based on the pattern of keystroke entries. More mundanely, it may wish to set time limits on the period between entering the name and the password. Or it may wish to enforce some particular user interface to collect this information. In any case, the service proxy may perform some sort of input processing on the client side before communicating with the real service. The service proxy may need to find extra classes in order to perform this processing.

A standalone application that gets a user name might use a GUI interface as shown in Figure 13-1.

Figure 13-1. User interface for name entry

The implementation for this name entry user interface might look like this:

```
package standalone;

import java.awt.*;
import java.awt.event.*;

/**
 * NameEntry.java
 */

public class NameEntry extends Frame {

    public NameEntry() {
        super("Name Entry");
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {System.exit(0);}
            });

        Label label = new Label("Name");
        TextField name = new TextField(20);
```

```
        add(label, BorderLayout.WEST);
        add(name, BorderLayout.CENTER);
        name.addActionListener(new NameHandler());

        pack();
    }

    public static void main(String[] args) {

        NameEntry f = new NameEntry();
        f.setVisible(true);
    }
} // NameEntry

class NameHandler implements ActionListener {
    public void actionPerformed(ActionEvent evt) {
        System.out.println("Name was: " + evt.getActionCommand());
    }
}
```

The classes used in this implementation are these:

- A set of standard classes: `Frame`, `Label`, `TextField`, `ActionListener`, `ActionEvent`, `BorderLayout`, `WindowEvent`, and `System`
- A couple of new classes: `NameEntry` and `NameHandler`

At compile time and at runtime these will need to be accessible.

NameEntry Interface

A standalone application needs to have all the class files available to it. In a Jini system, we have already seen that different components may only need access to a subset of the total set of classes. The simple application just shown used a large set of classes. If this is used to form part of a Jini system then some parts of this application will end up in Jini clients, and some will end up in Jini services. Each of them will have requirements about which classes they have access to, and this will depend on how the components are distributed.

We don't want to be overly concerned about the program logic of what is done with the user name once it has been entered—the interesting part is the location

of the classes. All possible ways of distributing this application into services and clients will need an interface definition, which we can make as follows:

```
package common;

/**
 * NameEntry.java
 */

public interface NameEntry  {

    public void show();

} // NameEntry
```

Then the client can call upon an implementation to simply `show()` itself and collect information in whatever way it chooses.

> **NOTE** *We don't want to get involved here in the ongoing discussion about the most appropriate interface definition for GUI classes—this is taken up in Chapter 19.*

Naive Implementation

A simple implementation of this `NameEntry` interface is as follows:

```
package complex;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import com.sun.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
import com.sun.jini.lookup.ServiceIDListener;
import com.sun.jini.lease.LeaseRenewalManager;

/**
 * NameEntryImpl1.java
 */
```

```
public class NameEntryImpl1 extends Frame implements common.NameEntry,
                                        ActionListener, java.io.Serializable {

    public NameEntryImpl1() {
        super("Name Entry");
        /*
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {System.exit(0);}
            public void windowOpened(WindowEvent e) {}});
        */
        setLayout(new BorderLayout());
        Label label = new Label("Name");
        add(label, BorderLayout.WEST);
        TextField name = new TextField(20);
        add(name, BorderLayout.CENTER);
        name.addActionListener(this);

        // don't do this here!
        // pack();
    }

    /**
     * method invoked on pressing <return> in the TextField
     */
    public void actionPerformed(ActionEvent evt) {
        System.out.println("Name was: " + evt.getActionCommand());
    }

    public void show() {
        pack();
        super.show();
    }

} // NameEntryImpl1
```

This implementation of the user interface creates the GUI elements in the constructor. When exported, this entire user interface will be serialized and exported. The instance data isn't too big in this case (about 2,100 bytes), but that is because the example is small. A GUI with several hundred objects will be much larger. This is overhead, which could be avoided by deferring creation to the client side.

Figure 13-2 shows which instances are running in which JVM.

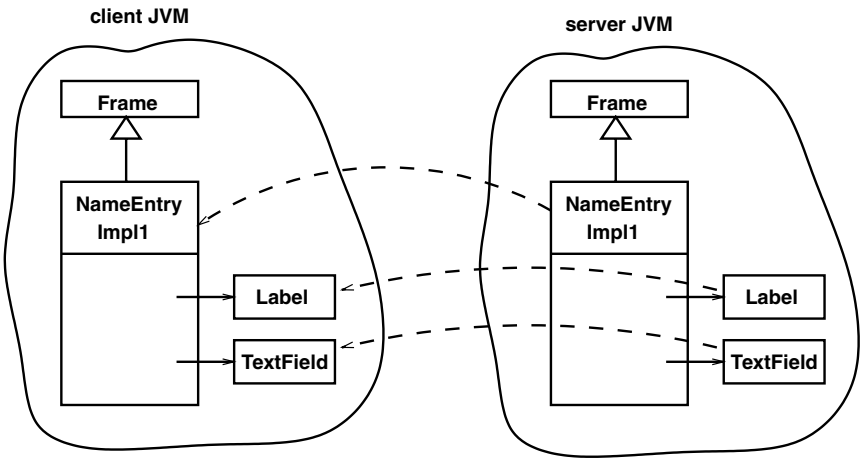

Figure 13-2. JVM objects for the naive implementation of the user interface

Another problem with this code is that it first creates an object on the server that has heavy reliance on environmental factors on the server. It then removes itself from that environment and has to reestablish itself on the target client environment.

On my current system, this dependence on environments shows up as a `TextField` complaining that it cannot find a whole bunch of fonts on my server. Of course, that doesn't matter because it gets moved to the client machine. (As it happens, the fonts aren't available on my client machine either, so I end up with two batches of complaint messages, from the server and from the client. I should only get the client complaints.) It could matter if the service died because of missing pieces on the server side that exist on the client.

What Files Need to Be Where?

The client needs to know the `NameEntry` interface class. This must be in its `CLASSPATH`.

The server needs to know the class files for

- `NameEntry`
- `Server1`
- `NameEntryImpl1`

These must be in its `CLASSPATH`.

The HTTP server needs to know the class files for `NameEntryImpl1`. This must be in the directory of documents for this server.

Factory Implementation

The second implementation minimizes the amount of serialized code that must be shipped around by creating as much as possible on the client side. We don't even need to declare the class as a subclass of `Frame`, because that class also exists on the client side. The client calls the `show()` interface method, and all the GUI creation is moved to there. Essentially, what is created on the server side is a factory object, and this object is moved to the client. The client than makes calls on this factory to create the user interface.

```
package complex;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import com.sun.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
import com.sun.jini.lookup.ServiceIDListener;
import com.sun.jini.lease.LeaseRenewalManager;

/**
 * NameEntryImpl2.java
 */

public class NameEntryImpl2 implements common.NameEntry,
                                       ActionListener, java.io.Serializable {

    public NameEntryImpl2() {
    }

    /**
     * method invoked on pressing <return> in the TextField
     */
    public void actionPerformed(ActionEvent evt) {
        System.out.println("Name was: " + evt.getActionCommand());
    }

    public void show() {
        Frame fr = new Frame("Name Entry");

        fr.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {System.exit(0);}
            public void windowOpened(WindowEvent e) {}});
```

```
        fr.setLayout(new BorderLayout());
        Label label = new Label("Name");
        fr.add(label, BorderLayout.WEST);
        TextField name = new TextField(20);
        fr.add(name, BorderLayout.CENTER);
        name.addActionListener(this);

        fr.pack();
        fr.show();
    }


} // NameEntryImpl2
```

Figure 13-3 shows which instances are running in which JVM.

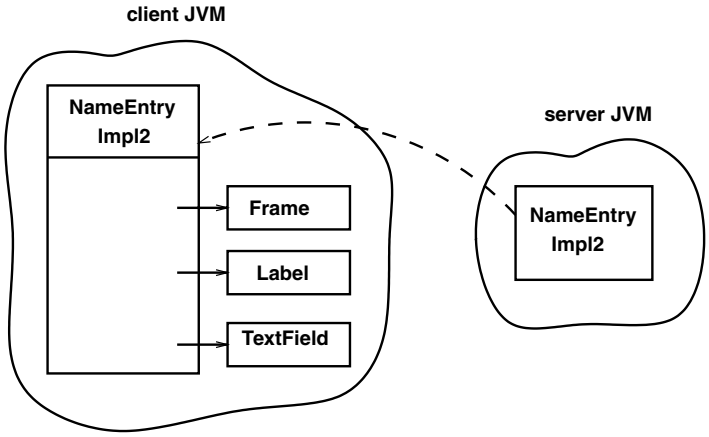

Figure 13-3. JVM objects for the factory implementation of the user interface

There are some standard classes that cannot be serialized: one example is the Swing `JTextArea` class (as of Swing 1.1). This has frequently been logged as a bug against Swing. Until this is fixed, the only way one of these objects can be used by a service is to create it on the client.

NOTE *Swing is the set of user interface classes introduced as part of the Java Foundation Classes in JDK 1.2*

What Files Need to Be Where?

For this implementation, the client needs to know the `NameEntry` interface class.

The server needs to know the class files for

- `NameEntry`
- `Server2`
- `NameEntryImpl2`
- `NameEntryImpl2$1`

The last class in the list is an *anonymous class* that acts as the `WindowListener`. The class file is produced by the compiler. In the naive implementation earlier in the chapter, this part of the code was commented out for simplicity.

The HTTP server needs to know the class files for

- `NameEntryImpl2`
- `NameEntryImpl2$1`

Using Multiple Class Files

Apart from the standard classes and a common interface, the previous implementations just used a single class that was uploaded to the lookup service and then passed on to the client. A more realistic situation might require the uploaded service to access a number of other classes that could not be expected to be on the client machine. That is, the server might upload an object from a single class to the lookup service and from there to a client. However, when the object runs, it needs to create *other* objects using class files that are not known to the client.

For example, the listener object of the last implementation could belong to a separate `NameHandler` class. The code looks like this:

```
package complex;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import com.sun.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
import com.sun.jini.lookup.ServiceIDListener;
```

```
import com.sun.jini.lease.LeaseRenewalManager;

/**
 * NameEntryImpl3.java
 */

public class NameEntryImpl3 implements common.NameEntry,
                                       java.io.Serializable {

    public NameEntryImpl3() {
    }

    public void show() {
        Frame fr = new Frame("Name Entry");

        fr.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {System.exit(0);}
            public void windowOpened(WindowEvent e) {}});

        fr.setLayout(new BorderLayout());
        Label label = new Label("Name");
        fr.add(label, BorderLayout.WEST);
        TextField name = new TextField(20);
        fr.add(name, BorderLayout.CENTER);
        name.addActionListener(new NameHandler());

        fr.pack();
        fr.show();
    }


} // NameEntryImpl3

class NameHandler implements ActionListener {
    /**
     * method invoked on pressing <return> in the TextField
     */
    public void actionPerformed(ActionEvent evt) {
        System.out.println("Name was: " + evt.getActionCommand());
    }
} // NameHandler
```

This version of the user interface implementation uses a `NameHandler` class that only exists on the server machine. When the client attempts to deserialize the

NameEntryImpl3 instance, it will fail to find this class and be unable to complete deserialization. How is this resolved? Well, in the same way as before, by making it available through the HTTP server.

Figure 13-4 shows which instances are running in which JVM.

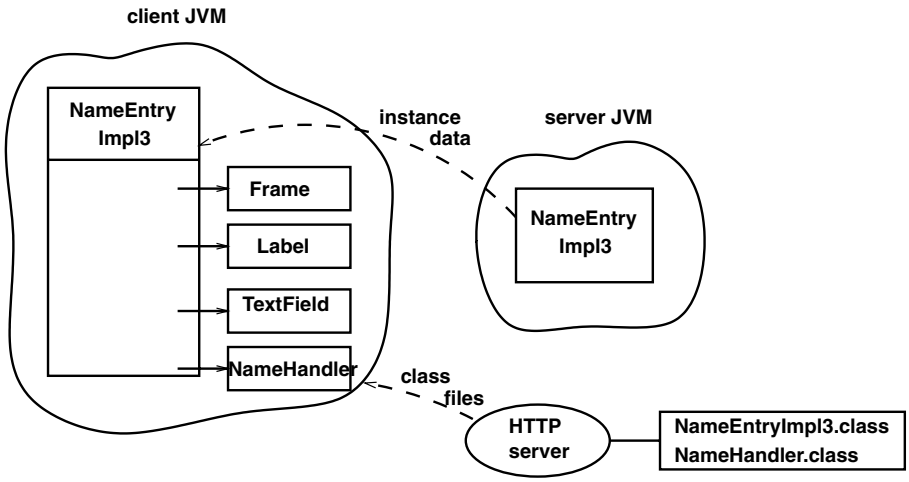

Figure 13-4. JVM objects for multiple class files implementation

What Files Need to Be Where?

The client needs to know the NameEntry interface class.

The server needs to know the class files for

- NameEntry
- Server3
- NameEntryImpl3
- NameEntryImpl3$1
- NameHandler

The NameHandler class file is another one produced by the compiler.

The HTTP server needs to know the class files for

- NameEntryImpl3
- NameEntryImpl3$1
- NameHandler

Running Threads from Discovery

The previous section looked at issues involving the location of classes in order to reduce network traffic and to improve the speed and responses of clients and services. Within a client or service, other techniques, such as multithreading, can also be used to improve responsiveness.

In all of the examples using explicit registration (such as those in Chapters 8 and 9), a single thread was used. That is, as a service locator was discovered, the registration process commenced in the same thread. This registration may take some time, and during this time, new lookup services may be discovered. To avoid the possibility of these new services timing out and being missed, all registration processing should be carried out in a separate thread, rather than possibly holding up the discovery thread.

Server Threads

Running another thread is not a difficult procedure. Basically we have to define a new class that extends `Thread`, and move most of the registration into its `run` method. This is done in the following version of the file classifier server, which is based on the server in Chapter 3 that uploads a complete service. In this version, the registration code is moved to a separate thread, which is implemented using an inner class:

```
package complex;

import complete.FileClassifierImpl;

import java.rmi.RMISecurityManager;
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 com.sun.jini.lease.LeaseRenewalManager;
// import com.sun.jini.lease.LeaseListener;
// import com.sun.jini.lease.LeaseRenewalEvent;
import net.jini.lease.LeaseRenewalManager;
import net.jini.lease.LeaseListener;
import net.jini.lease.LeaseRenewalEvent;
```

```
/**
 * FileClassifierServer.java
 */

public class FileClassifierServer implements DiscoveryListener,
                                             LeaseListener {

    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

    public static void main(String argv[]) {
        new FileClassifierServer();

        // keep server running forever to
        // - allow time for locator discovery and
        // - keep re-registering the lease
        Object keepAlive = new Object();
        synchronized(keepAlive) {
            try {
                keepAlive.wait();
            } catch(InterruptedException e) {
                // do nothing
            }
        }
    }

    public FileClassifierServer() {

        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];
```

```
            new RegisterThread(registrar).start();
        }
    }

    public void discarded(DiscoveryEvent evt) {

    }

    public void notify(LeaseRenewalEvent evt) {
        System.out.println("Lease expired " + evt.toString());
    }

    /**
     * an inner class to register the service in its own thread
     */
    class RegisterThread extends Thread {

        ServiceRegistrar registrar;

        RegisterThread(ServiceRegistrar registrar) {
            this.registrar = registrar;
        }

        public void run() {
            ServiceItem item = new ServiceItem(null,
                                               new FileClassifierImpl(),
                                               null);
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.println("Register exception: " + e.toString());
                return;
            }
            System.out.println("service registered");

            // set lease renewal in place
            leaseManager.renewUntil(reg.getLease(), Lease.FOREVER,
                                    FileClassifierServer.this);
        }
    }
} // FileClassifierServer
```

Join Manager Threads

If you use a `JoinManager` to handle lookup and registration, then it essentially does this for you: it creates a new thread to handle registration. Thus, the examples in Chapter 11 do not need any modification, as the `JoinManager` already uses the concepts of this section.

Client Threads

It is probably more important to use threads in the client than in the server, because the client will actually perform some computation (which may be lengthy) based on the service it discovers. Again, this is a simple matter of moving code into a new class that implements `Thread`. Doing this to the multicast client `TestFileClassifier` of Chapter 3 results in the following code:

```
package client;

import common.FileClassifier;
import common.MIMEType;

import java.rmi.RMISecurityManager;
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.ServiceTemplate;


/**
 * TestFileClassifierThread.java
 */

public class TestFileClassifierThread implements DiscoveryListener {

    public static void main(String argv[]) {
        new TestFileClassifierThread();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(10000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
```

```
    }
}

public TestFileClassifierThread() {
    System.setSecurityManager(new RMISecurityManager());

    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++) {
        System.out.println("Service found");
        ServiceRegistrar registrar = registrars[n];

        new LookupThread(registrar).start();
    }
}

public void discarded(DiscoveryEvent evt) {
    // empty
}

class LookupThread extends Thread {

    ServiceRegistrar registrar;

    LookupThread(ServiceRegistrar registrar) {
        this.registrar = registrar;
    }


    public void run() {
```

```
            Class[] classes = new Class[] {FileClassifier.class};
            FileClassifier classifier = null;
            ServiceTemplate template = new ServiceTemplate(null, classes,
                                                           null);

            try {
                classifier = (FileClassifier) registrar.lookup(template);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                return;
            }
            if (classifier == null) {
                System.out.println("Classifier null");
                return;
            }
            MIMEType type;
            try {
                type = classifier.getMIMEType("file1.txt");
                System.out.println("Type is " + type.toString());
            } catch(java.rmi.RemoteException e) {
                System.err.println(e.toString());
            }
        }
    }

} // TestFileClassifier
```

Inexact Service Matching

Suppose you have a printer service that prints at 30 pages per minute. A client wishes to find a printer that will print at least 24 pages per minute. How will this client find the service? The standard Jini pattern matching will either be for an exact match on an attribute or an ignored match on an attribute, so the only way a client can find this printer is to ignore the speed attribute and perform a later selection among all the printers that it sees.

We can define a simple printer interface that will allow us to print documents and also allow us to access the printer speed as follows:

```
package common;

import java.io.Serializable;
```

```
/**
 * Printer.java
 */

public interface Printer extends Serializable {

    public void print(String str);
    public int getSpeed();

} // Printer
```

I don't want to delve here into the complexities of building a real printer service. A "fake" printer implementation that takes its speed from a parameter in the constructor can be written as a complete uploadable service (see Chapter 3) as follows:

```
package printer;

/**
 * PrinterImpl.java
 */

public class PrinterImpl implements common.Printer, java.io.Serializable {

    protected int speed;

    public PrinterImpl(int sp) {
    speed = sp;
    }

    public void print(String str) {
    // fake stuff:
    System.out.println("I'm the " + speed + " pages/min printer");
    System.out.println(str);
    }

    public int getSpeed() {
    return speed;
    }

} // PrinterImpl
```

Printer implementations can be created and made available using server implementations of earlier chapters.

Given this, a client can choose a suitably fast printer in a two-step process:

1. Find a service using the lookup exact/ignore match algorithm.
2. Query the service to see if it satisfies other types of Boolean conditions.

The following program shows how you can find a printer that is "fast enough":

```
package client;

import common.Printer;

import java.rmi.RMISecurityManager;

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.ServiceTemplate;
import net.jini.core.lookup.ServiceMatches;

/**
 * TestPrinterSpeed.java
 */

public class TestPrinterSpeed implements DiscoveryListener {

    public TestPrinterSpeed() {

        System.setSecurityManager(new RMISecurityManager());

        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();
    Class[] classes = new Class[] {Printer.class};

    ServiceTemplate template = new ServiceTemplate(null, classes,
                                                   null);

    for (int n = 0; n < registrars.length; n++) {
        ServiceRegistrar registrar = registrars[n];
        ServiceMatches matches;

        try {
            matches = registrar.lookup(template, 10);
        } catch(java.rmi.RemoteException e) {
            e.printStackTrace();
            continue;
        }
        // NB: matches.totalMatches may be greater than matches.items.length
        for (int m = 0; m < matches.items.length; m++) {
            Printer printer = (Printer) matches.items[m].service;

            // Inexact matching is not performed by lookup()
            // we have to do it ourselves on each printer
            // we get
            int speed = printer.getSpeed();
            if (speed >= 24) {
                // this one is okay, use its print() method
                printer.print("fast enough printer");
            } else {
                // we can't use this printer, so just say so
                System.out.println("Printer too slow at " + speed);
            }
        }

    }
}

public void discarded(DiscoveryEvent evt) {
    // empty
}

public static void main(String[] args) {
```

```
        TestPrinterSpeed f = new TestPrinterSpeed();

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(10000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

} // TestPrinterSpeed
```

Matching Using Local Services

When a user connects their laptop into a brand-new network, they will probably know little about the environment they have joined. If they want to use services in this network, they will probably want to use general terms and have them translated into specific terms for this new environment. For example, the user may want to print a file on a nearby printer. In this situation, there is little likelihood that the new user knows how to work out the distance between themselves and the printers. However, a local service could be running which does know how to calculate physical distances between objects on the network.

Finding a "close enough" printer then becomes a matter of querying service locators both for printers and for a distance service. As each printer is found, the distance service can be asked to calculate the distance between itself and the laptop (or camera, or any other device that wants to print).

The complexity of the task to be done by clients is growing: a client has to find two sets of services, and when it finds one (a printer) invoke the other (the distance service). This calls for lookup processing to be handled in separate threads. In addition, as each locator is found, it may know about printers, it may know about distance services, it may know both, or it may know none! When the client starts up, it will be discovering these services in an arbitrary order, and the code must be structured to deal with this.

These are some of the cases that may arise:

- A printer may be discovered before any distance service has been found. In this case, the printer must be stored for later distance checking.

- A printer may be discovered after a distance service has been found. It can be checked immediately.

- A distance service is found after some printers have been found. This saved set of printers should be checked at this point.

In this problem, we only need to find one distance service, but possibly many printers. The client code given shortly will save printers in a `Vector`, and save a distance service in a single variable.

In searching for printers, we only want to find those that have location information. However, we do not want to match on any particular values. The client will have to use wildcard patterns in a location object. The location information of a printer will need to be retrieved along with the printer so it can be used. Therefore, instead of just storing printers, we need to store `ServiceItem` objects, which carry the attribute information as well as the objects.

Of course, for this to work, the client also needs to know where it is! This could be done, for example, by popping up a dialog box asking the user to locate themselves.

A client satisfying these requirements is given in the following program. (The location of the client is hard-coded into the `getMyLocation()` method for simplicity.)

```
package client;

import common.Printer;
import common.Distance;

import java.util.Vector;

import java.rmi.RMISecurityManager;
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.ServiceTemplate;
import net.jini.lookup.entry.Location;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceMatches;
import net.jini.core.entry.Entry;

/**
 * TestPrinterDistance.java
 */

public class TestPrinterDistance implements DiscoveryListener {

    protected Distance distance = null;
    protected Object distanceLock = new Object();
```

```
protected Vector printers = new Vector();

public static void main(String argv[]) {
    new TestPrinterDistance();

    // stay around long enough to receive replies
    try {
        Thread.currentThread().sleep(10000L);
    } catch(java.lang.InterruptedException e) {
        // do nothing
    }
}

public TestPrinterDistance() {
    System.setSecurityManager(new RMISecurityManager());

    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++) {
        System.out.println("Service found");
        ServiceRegistrar registrar = registrars[n];

        new LookupThread(registrar).start();
    }
}

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

```
class LookupThread extends Thread {

    ServiceRegistrar registrar;

    LookupThread(ServiceRegistrar registrar) {
        this.registrar = registrar;
    }


    public void run() {

        synchronized(distanceLock) {
            // only look for one distance service
            if (distance == null) {
                lookupDistance();
            }
            if (distance != null) {
                // found a new distance service
                // process any previously found printers
                synchronized(printers) {
                    for (int n = 0; n < printers.size(); n++) {
                        ServiceItem item = (ServiceItem) printers.elementAt(n);
                        reportDistance(item);
                    }
                }
            }
        }

        ServiceMatches matches = lookupPrinters();
        for (int n = 0; n < matches.items.length; n++) {
            if (matches.items[n] != null) {
                synchronized(distanceLock) {
                    if (distance != null) {
                        reportDistance(matches.items[n]);
                    } else {
                        synchronized(printers) {
                            printers.addElement(matches.items[n]);
                        }
                    }
                }
            }
        }
    }
```

```
/*
 * We must be protected by the lock on distanceLock here
 */
void lookupDistance() {
    // If we don't have a distance service, see if this
    // locator knows of one
    Class[] classes = new Class[] {Distance.class};
    ServiceTemplate template = new ServiceTemplate(null, classes,
                                                   null);

    try {
        distance = (Distance) registrar.lookup(template);
    } catch(java.rmi.RemoteException e) {
        e.printStackTrace();
    }

}

ServiceMatches lookupPrinters() {
    // look for printers with
    // wildcard matching on all fields of Location
    Entry[] entries = new Entry[] {new Location(null, null, null)};

    Class[] classes = new Class[1];
    try {
        classes[0] = Class.forName("common.Printer");
    } catch(ClassNotFoundException e) {
        System.err.println("Class not found");
        System.exit(1);
    }
    ServiceTemplate template = new ServiceTemplate(null, classes,
                                                   entries);
    ServiceMatches matches = null;
    try {
        matches =  registrar.lookup(template, 10);
    } catch(java.rmi.RemoteException e) {
        e.printStackTrace();
    }
    return matches;
}

/**
 * report on the distance of the printer from
 * this client
```

```
         */
        void reportDistance(ServiceItem item) {
            Location whereAmI = getMyLocation();
            Location whereIsPrinter = getPrinterLocation(item);
            if (whereIsPrinter != null) {
                int dist = distance.getDistance(whereAmI, whereIsPrinter);
                System.out.println("Found a printer at " + dist +
                                   " units of length away");
            }
        }

        Location getMyLocation() {
            return new Location("1", "1", "Building 1");
        }

        Location getPrinterLocation(ServiceItem item) {
            Entry[] entries = item.attributeSets;
            for (int n = 0; n < entries.length; n++) {
                if (entries[n] instanceof Location) {
                    return (Location) entries[n];
                }
            }
            return null;
        }
    }

} // TestFileClassifier
```

A number of services will need to be running. At least one distance service will be needed, implementing the interface `Distance`:

```
package common;

import net.jini.lookup.entry.Location;

/**
 * Distance.java
 */

public interface Distance extends java.io.Serializable {

    int getDistance(Location loc1, Location loc2);

} // Distance
```

The following is an example implementation of a distance service:

```
package complex;

import net.jini.lookup.entry.Location;

/**
 * DistanceImpl.java
 */

public class DistanceImpl implements common.Distance {

    public DistanceImpl() {

    }

    /**
     * A very naive distance metric
     */
    public int getDistance(Location loc1, Location loc2) {
        int room1, room2;
        try {
            room1 = Integer.parseInt(loc1.room);
            room2 = Integer.parseInt(loc2.room);
        } catch(Exception e) {
            return -1;
        }
        int value = room1 - room2;
        return (value > 0 ? value : -value);
    }

} // DistanceImpl
```

Earlier in this chapter we gave the code for `PrinterImpl`. A simple program to start up a distance service and two printers is as follows:

```
package complex;

import printer.PrinterImpl;
import printer.PrinterImpl;
import complex.DistanceImpl;

// import com.sun.jini.lookup.JoinManager;
import net.jini.lookup.JoinManager;
```

```
import net.jini.core.lookup.ServiceID;
// import com.sun.jini.lookup.ServiceIDListener;
// import com.sun.jini.lease.LeaseRenewalManager;
import net.jini.lookup.ServiceIDListener;
import net.jini.lease.LeaseRenewalManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.lookup.entry.Location;
import net.jini.core.entry.Entry;
import net.jini.discovery.LookupDiscoveryManager;

/**
 * PrinterServerLocation.java
 */

public class PrinterServerLocation implements ServiceIDListener {

    public static void main(String argv[]) {
        new PrinterServerLocation();

        // run forever
        Object keepAlive = new Object();
        synchronized(keepAlive) {
            try {
                keepAlive.wait();
            } catch(InterruptedException e) {
                // do nothing
            }
        }
    }

    public PrinterServerLocation() {

        JoinManager joinMgr = null;
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            // distance service
            joinMgr = new JoinManager(new DistanceImpl(),
                                      null,
                                      this,
                                      mgr,
                                      new LeaseRenewalManager());
```

```
            // slow printer in room 120
            joinMgr = new JoinManager(new PrinterImpl(20),
                                      new Entry[] {new Location("1", "120",
                                                                "Building 1")},
                                      this,
                                      mgr,
                                      new LeaseRenewalManager());

            // fast printer in room 130
            joinMgr = new JoinManager(new PrinterImpl(30),
                                      new Entry[] {new Location("1", "130",
                                                                "Building 1")},
                                      this,
                                      mgr,
                                      new LeaseRenewalManager());


        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public void serviceIDNotify(ServiceID serviceID) {
        System.out.println("got service ID " + serviceID.toString());
    }

} // PrinterServerLocation
```

Finding a Service Once Only

There may be many lookup services on the local network, perhaps specializing in certain groups of services. There could be many lookup services running on the Internet, which could act as global repositories of information. For example, there is a lookup service running at `www.jini.canberra.edu.au` that acts as a publicly available lookup service for those who wish to experiment with Jini. One may expect to find lookup services acting in a "portal" role, listing all of the public clock services, the real estate services, and so on.

A service will probably register with as many service locators as it can to improve its chances of being found. On the other hand, clients looking for a service may be content to find just a single suitable implementation, or may wish to

find all service implementations. This second case can cause some uniqueness problems: if a client finds every service that has been registered with multiple locators, then it will probably find the same service more than once.

Why is this a problem? Well, suppose the client wants to find all power drills in the factory and ask them to drill exactly one hole each. Or suppose it finds all backup services for the system, and wants each one to perform a single backup. In that case, it needs to know the identity of each service so that it can tell when it is getting a duplicate copy from another locator source. Otherwise, each drill might make six holes because the client got a copy of each drill from six service locators, or you might get six backups of the same data. Whenever a service can perform a non-idempotent service (i.e., one in which repeating the action has a different effect each time), then duplicate copies on the client side must be avoided.

Jini has a concept of a service being a "good citizen." This concept includes having a single identity across all lookup services, which allows clients to tell whether they have come across multiple copies of the same service or have encountered a different implementation of the service. The behavior on the part of services is contained in the Jini "Lookup Service" specification, and it hinges on the use of the `ServiceID`.

A `ServiceID` can be specified when registering a service with a service locator. If this is the first time this service has ever been registered, then the `ServiceID` should be `null`. The service locator will then generate a non-null `ServiceID` that can be used in future to identify this service. This object is specified to be unique, so that a service locator cannot generate the same `ServiceID` for two different services, and two different locators cannot generate the same `ServiceID`. This provides a unique identifier that can be used to identify duplicates.

The procedure for a service to follow when it is registering itself with service locators is as follows:

1. The very first time a service is registered, use `null` as the `serviceID` value of the `ServiceItem` in `ServiceRegistrar.register()`.

2. The returned `ServiceRegistration` has a `getServiceID()` method for retrieving the `ServiceID`. This `ServiceID` should then be used in any future registrations both with this service locator and with any others. This ensures that the service has a unique identity across all lookup services. It should be noted that `JoinManager` already does this, although this is not stated in its documentation. We have done this in earlier examples, such as the server in Chapter 8.

3. The client has a choice of two `lookup()` methods to use with its `ServiceRegistrar` object: the first just returns a single object, and the second returns an array of `ServiceMatches` objects. This second one is more useful

here, as it can give the array of `ServiceItem` objects, and the `ServiceID` can be extracted from there.

4. The client should maintain a list of service IDs that it has seen, and compare any new ones against it—then it can check whether a service is a new one or a previously seen one.

5. The service should save its ID in persistent storage so that if it dies and restarts, it can use the same ID—after all, it is the same service. (This may involve subtle considerations: it should only use the same `ServiceID` if it really is the same service. For example, if the service maintains state that is lost in a crash, then it isn't the same service!)

In Chapter 8 we gave the code for a multicast client that looked for a service, used it, and exited. A modified client that looks for all *unique* services and uses each one is as follows:

```
package unique;

import common.FileClassifier;
import common.MIMEType;

import java.rmi.RMISecurityManager;
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.ServiceTemplate;
import net.jini.core.lookup.ServiceMatches;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceID;

import java.util.Vector;

/**
 * TestFileClassifier.java
 */

public class TestFileClassifier implements DiscoveryListener {

    protected Vector serviceIDs = new Vector();

    public static void main(String argv[]) {
        new TestFileClassifier();
```

```
        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(10000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public TestFileClassifier() {
        System.setSecurityManager(new RMISecurityManager());

        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();
        Class [] classes = new Class[] {FileClassifier.class};
        FileClassifier classifier = null;
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);

        for (int n = 0; n < registrars.length; n++) {
            System.out.println("Service found");
            ServiceRegistrar registrar = registrars[n];
            ServiceMatches matches = null;
            try {
                 matches = registrar.lookup(template, 10);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                continue;
            }

            ServiceItem[] items = matches.items;
```

```
            for (int m = 0; m < items.length; m++) {
                ServiceID id = items[m].serviceID;
                if (serviceIDs.indexOf(id) != -1) {
                    // found a new serviceID - record it and use it
                    classifier = (FileClassifier) items[m].service;
                    if (classifier == null) {
                        System.out.println("Classifier null");
                        continue;
                    }

                    serviceIDs.add(id);

                    MIMEType type;
                    try {
                        type = classifier.getMIMEType("file1.txt");
                        System.out.println("Type is " + type.toString());
                    } catch(java.rmi.RemoteException e) {
                        System.err.println(e.toString());
                    }
                }
            }
        }
    }

    public void discarded(DiscoveryEvent evt) {
        // empty
    }
} // TestFileClassifier
```

Leasing Changes to a Service

Sometimes a service may allow changes to be made to its state by external (remote) objects. This happens all the time to service locators, which have services added and removed. A service may wish to behave in the same manner as the locators, and just grant a lease for the change. After the lease has expired, the service will remove the change. Such a situation may occur with file classification, where a new service that can handle a particular MIME type starts: it can register the file-name mapping with a file classifier service. However, the file classifier service will just time out the mapping unless the new service keeps it renewed.

The example in this section follows the "Granting and Handling Leases" section of Chapter 7. It gives a concrete illustration of that section, now that there is enough background to do so.

Leased FileClassifier

A dynamically extensible version of a file classifier will have methods to add and remove MIME mappings:

```
package common;

import java.io.Serializable;

/**
 * LeaseFileClassifier.java
 */

import net.jini.core.lease.Lease;

public interface LeaseFileClassifier extends Serializable {

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException;

    /*
     * Add the MIME type for the given suffix.
     * The suffix does not contain '.' e.g. "gif".
     * @exception net.jini.core.lease.LeaseDeniedException
     * a previous MIME type for that suffix exists.
     * This type is removed on expiration or cancellation
     * of the lease.
     */
    public Lease addType(String suffix, MIMEType type)
        throws java.rmi.RemoteException,
               net.jini.core.lease.LeaseDeniedException;

    /**
     * Remove the MIME type for the suffix.

     */
    public void removeType(String suffix)
        throws java.rmi.RemoteException;
} // LeaseFileClasssifier
```

The `addType()` method returns a lease. We shall use the landlord leasing system discussed in Chapter 7. The client and the service will be in different Java VMs, probably on different machines. Figure 13-5 gives the object structure on the service side.

This should be compared to Figure 7-3 where we considered the "foo" implementation of landlord leasing.

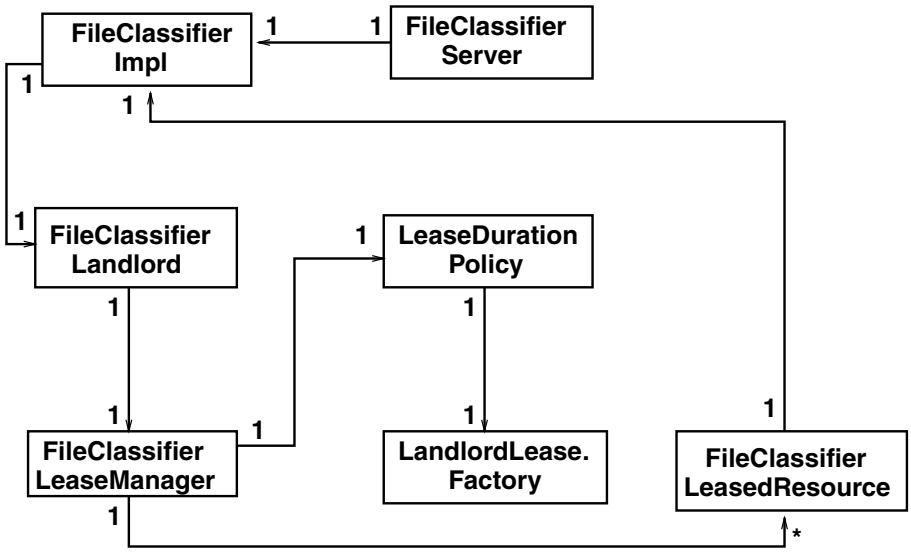

Figure 13-5. Class diagram for leasing on the server

On the client side, the lease object will be a copy of the lease created on the server (normally RMI semantics), but the other objects from the service will be stubs that call into the real objects on the service. This is shown in Figure 13-6.

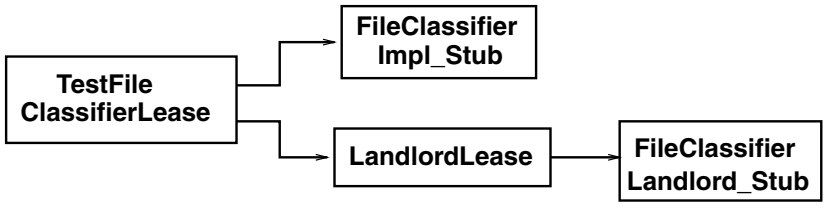

Figure 13-6. Class diagram for leasing on the client

The FileClassifier Leased Resource Class

The FileClassifierLeasedResource class acts as a wrapper around the actual resource, adding cookie and time expiration fields around the resource. It adds a unique cookie mechanism, in addition to making the wrapped resource visible.

```
/**
 * FileClassifierLeasedResource.java
 */
package lease;

import common.LeaseFileClassifier;
import com.sun.jini.lease.landlord.LeasedResource;

public class FileClassifierLeasedResource implements LeasedResource  {

    static protected int cookie = 0;
    protected int thisCookie;
    protected LeaseFileClassifier fileClassifier;
    protected long expiration = 0;
    protected String suffix = null;

    public FileClassifierLeasedResource(LeaseFileClassifier fileClassifier,
                                        String suffix) {
        this.fileClassifier = fileClassifier;
        this.suffix = suffix;
        thisCookie = cookie++;
    }

    public void setExpiration(long newExpiration) {
        this.expiration = newExpiration;
    }

    public long getExpiration() {
        return expiration;
    }
    public Object getCookie() {
        return new Integer(thisCookie);
    }

    public LeaseFileClassifier getFileClassifier() {
        return fileClassifier;
    }
```

```
    public String getSuffix() {
        return suffix;
    }
} // FileClassifierLeasedResource
```

The FileClassifier Lease Manager Class

The `FileClassifierLeaseManager` class is very similar to the code given for the `FooLeaseManager` in Chapter 7:

```
/**
 * FileClassifierLeaseManager.java
 */
package lease;

import java.util.*;
import common.LeaseFileClassifier;

import net.jini.core.lease.Lease;
import com.sun.jini.lease.landlord.LeaseManager;
import com.sun.jini.lease.landlord.LeasedResource;
import com.sun.jini.lease.landlord.LeaseDurationPolicy;
import com.sun.jini.lease.landlord.Landlord;
import com.sun.jini.lease.landlord.LandlordLease;
import com.sun.jini.lease.landlord.LeasePolicy;

public class FileClassifierLeaseManager implements LeaseManager {

    protected static long DEFAULT_TIME = 30*1000L;

    protected Vector fileClassifierResources = new Vector();
    protected LeaseDurationPolicy policy;

    public FileClassifierLeaseManager(Landlord landlord) {
        policy = new LeaseDurationPolicy(Lease.FOREVER,
                                         DEFAULT_TIME,
                                         landlord,
                                         this,
                                         new LandlordLease.Factory());
        new LeaseReaper().start();
    }

    public void register(LeasedResource r, long duration) {
```

```
            fileClassifierResources.add(r);
        }

        public void renewed(LeasedResource r, long duration, long olddur) {
            // no smarts in the scheduling, so do nothing
        }

        public void cancelAll(Object[] cookies) {
            for (int n = cookies.length; --n >= 0; ) {
                cancel(cookies[n]);
            }
        }

        public void cancel(Object cookie) {
            for (int n = fileClassifierResources.size(); --n >= 0; ) {
                FileClassifierLeasedResource r = (FileClassifierLeasedResource)
    fileClassifierResources.elementAt(n);
                if (!policy.ensureCurrent(r)) {
                    System.out.println("Lease expired for cookie = " +
                                       r.getCookie());
                    try {
                        r.getFileClassifier().removeType(r.getSuffix());
                    } catch(java.rmi.RemoteException e) {
                        e.printStackTrace();
                    }
                    fileClassifierResources.removeElementAt(n);
                }
            }
        }

        public LeasePolicy getPolicy() {
            return policy;
        }

        public LeasedResource getResource(Object cookie) {
            for (int n = fileClassifierResources.size(); --n >= 0; ) {
                FileClassifierLeasedResource r = (FileClassifierLeasedResource)
    fileClassifierResources.elementAt(n);
                if (r.getCookie().equals(cookie)) {
                    return r;
                }
            }
            return null;
        }
```

```
    class LeaseReaper extends Thread {
        public void run() {
            while (true) {
                try {
                    Thread.sleep(DEFAULT_TIME) ;
                }
                catch (InterruptedException e) {
                }
                for (int n = fileClassifierResources.size()-1; n >= 0; n--) {
                   FileClassifierLeasedResource r = (FileClassifierLeasedResource)
                                        fileClassifierResources.elementAt(n)
 ;
                    if (!policy.ensureCurrent(r)) {
                        System.out.println("Lease expired for cookie = " +
                                        r.getCookie()) ;
                        try {
                            r.getFileClassifier().removeType(r.getSuffix());
                        } catch(java.rmi.RemoteException e) {
                            e.printStackTrace();
                        }
                        fileClassifierResources.removeElementAt(n);

                    }
                }
            }
        }
    }

} // FileClassifierLeaseManager
```

The FileClassifier Landlord Class

The `FileClassifierLandlord` class is very similar to the `FooLandlord` in Chapter 7:

```
/**
 * FileClassifierLandlord.java
 */

package lease;

import common.LeaseFileClassifier;
```

```
import com.sun.jini.lease.landlord.*;
import net.jini.core.lease.LeaseDeniedException;
import net.jini.core.lease.Lease;
import java.rmi.server.UnicastRemoteObject;
import java.rmi.Remote;

public class FileClassifierLandlord extends UnicastRemoteObject implements Land-
lord, Remote {

    FileClassifierLeaseManager manager = null;

    public FileClassifierLandlord() throws java.rmi.RemoteException {
        manager = new FileClassifierLeaseManager(this);
    }

    public void cancel(Object cookie) {
        manager.cancel(cookie);
    }

    public void cancelAll(Object[] cookies) {
        manager.cancelAll(cookies);
    }

    public long renew(java.lang.Object cookie,
                      long extension)
        throws net.jini.core.lease.LeaseDeniedException,
               net.jini.core.lease.UnknownLeaseException {
        LeasedResource resource = manager.getResource(cookie);
        if (resource != null) {
            return manager.getPolicy().renew(resource, extension);
        }
        return -1;
    }

    public Lease newFileClassifierLease(LeaseFileClassifier fileClassifier,
                                        String suffixKey, long duration)
        throws LeaseDeniedException {
        FileClassifierLeasedResource r = new
FileClassifierLeasedResource(fileClassifier,
                                                                    suffixKey);
        return manager.getPolicy().leaseFor(r, duration);
    }

    public Landlord.RenewResults renewAll(java.lang.Object[] cookie,
```

```
                                        long[] extension) {
        return null;
    }
} // FileClassifierLandlord
```

Summary

Jini provides a framework for building distributed applications. Nevertheless, there is still room for variation in how services and clients are written, and some of these are better than others. This chapter has looked at some of the variations that can occur, and how to deal with them.