CHAPTER 15

More Complex Examples

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. Further aspects of Jini 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 a client and a 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, you could achieve this 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. This section explores what happens when the service proxy needs to find extra classes in order to perform this processing.

A stand-alone application to get a user name might use a GUI interface with the appearance of Figure 15-1.

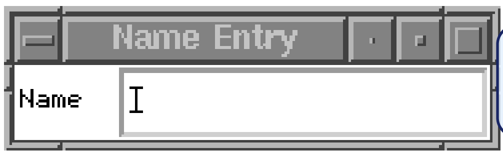

Figure 15-1. *User interface for name entry*

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 here are as follows:

- 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 classes will need to be accessible.

NameEntry Interface

A stand-alone 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 same will apply here, but it will critically depend on how the application is changed into a Jini system.

We don't want to be overly concerned about program logic of what is done with the user name once it has been entered—the interesting part is the location of 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 topic is taken up in Chapter 24.

Naive Implementation

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

```
package complex;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/**
 * NameEntryImpl1.java
 */
```

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

    public NameEntryImpl1() {
        super("Name Entry");
        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. The object is serializable, which means it will first be created in the server. When sent to the client, its data is serialized, so the *entire* user interface will be serialized and sent. The instance data isn't too big in this case (about 2,100 bytes), but that is because the example is small. Once it arrives at the client side, a copy will be constructed using this instance data and the class files, which will have been pulled down from a server. 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 15-2 shows which instances are running in which JVM.

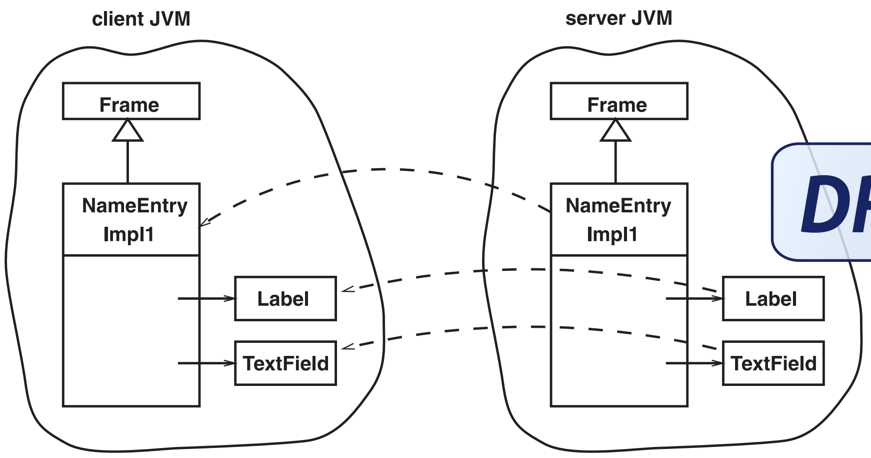

Figure 15-2. *JVM objects for the naive implementation of the user interface*

Another problem with this code is that it first creates an object on the server machine 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 get only 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 class files 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 then makes calls on this factory to create the user interface.

```
package complex;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/**
 * 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 15-3 shows which instances are running in which JVM.

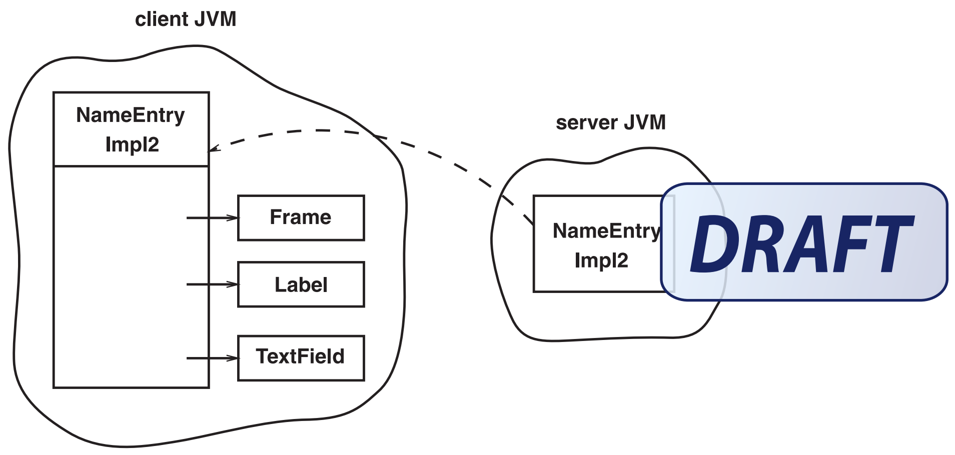

Figure 15-3. *JVM objects for the factory implementation of the user interface*

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

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 used just 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.

For example, it is simple to modify the examples to use a server side-specific class for the action listener, instead of the class itself, as follows:

```
package complex;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/**
 * 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
```

This version of the user interface uses a `NameHandler` class that exists only on the server machine. When the client attempts to deserialize the `NameEntryImpl3` instance, the class loader will fail to find this class and will be unable to complete deserialization. How is this resolved? Well, in the same way as before: by making the class available through the HTTP server.

Figure 15-4 shows which instances are running in which JVM.

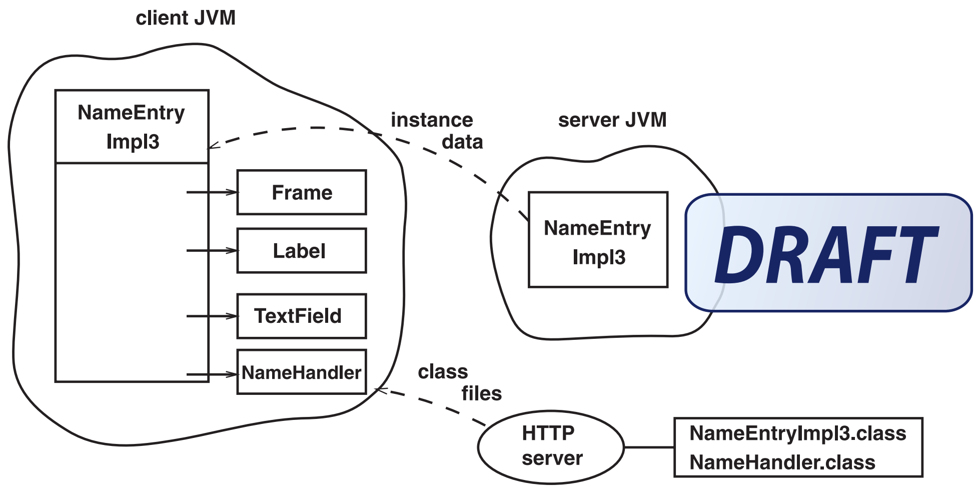

Figure 15-4. *JVM objects for the multiple class files implementation*

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`

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 be either for an exact match on an attribute or for 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 printer interface that will allow us to not only print documents but also access printer speed (plus other capabilities) as follows:

```
package common;
import java.io.Serializable;
/**
 * Printer.java
 */
public interface Printer extends Serializable {

    public void print(String str);
    public int getSpeed();
} // Printer
```

Given such an interface, 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 to 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 users connect their laptops 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, a 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 him- or herself and the printer. However, a local service could be running that 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 arbitrary order, and the code must be structured to deal with this.

The following 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 should be checked at this point.

In this problem, we need to find only one distance service, but possibly many printers. The client code given shortly will save printers in a `Vector`, and it will 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 users 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;
        }
    }
} // TestPrinterDistance
```

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 I provided the code for `PrinterImpl`. A simple program to start up a distance service and two printers is as follows:

```
package complex;
import printer.Printer30;
import printer.Printer20;
import complex.DistanceImpl;
import net.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
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;
import java.rmi.RMISecurityManager;
/**
 * 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() {
        System.setSecurityManager(new RMISecurityManager());
        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 20
            joinMgr = new JoinManager(new Printer20(),
                                      new Entry[] {new Location("1", "20",
                                                                "Building 1")},
                                      this,
                                      mgr,
                                      new LeaseRenewalManager());
            // fast printer in room 30
            joinMgr = new JoinManager(new Printer30(),
                                      new Entry[] {new Location("1", "30",
                                                                "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
```

Leased Changes to a Service

Sometimes a service may allow changes to its state to be made 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 of this section follows the "Granting and Handling Leases" section in Chapter 8. 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 classification will have methods to add and remove MIME mappings:

```
package common;
/**
 * LeaseFileClassifier.java
 */
import net.jini.core.lease.Lease;
public interface LeaseFileClassifier {
    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
```

Here is the remote form:

```
/**
 * RemoteLeaseFileClassifier.java
 */
package lease;
import common.LeaseFileClassifier;
import java.rmi.Remote;
public interface RemoteLeaseFileClassifier extends LeaseFileClassifier, Remote {

} // RemoteLeaseFileClassifier
```

LeaseFileClassifier Implementation

The implementation changes in several ways from the forms we saw earlier. Since it now needs to handle a changing set of MIME types, the types are stored in a map, and lookups are done on this map. Adding and removing types is also done through this map. In addition, adding types now needs to return a lease so that the additions will only last as long as the lease is valid; for this, the implementation will use a landlord to grant and manage leases.

The landlord implements the Landlord interface. In addition, it has a newFileClassifierLease() method, which is called by addType(). The implementation looks like this:

```
package lease;
import java.rmi.Remote;
import java.rmi.RemoteException;
import net.jini.core.lease.Lease;
import net.jini.core.lease.LeaseDeniedException;
import com.sun.jini.landlord.Landlord;
import common.MIMEType;
import common.LeaseFileClassifier;
import java.util.Map;
import java.util.HashMap;
/**
 * FileClassifierImpl.java
 */
public class FileClassifierImpl implements RemoteLeaseFileClassifier {
    public final long DURATION = 2*60*1000L; // 2 minutes
    /**
     * Map of String extensions to MIME types
     */
    protected Map map = new HashMap();
    protected transient FileClassifierLandlord landlord;
    public MIMEType getMIMEType(String fileName) {
        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 Lease addType(String suffix, MIMEType type)
        throws LeaseDeniedException {
        if (map.containsKey(suffix)) {
            throw new LeaseDeniedException("Extension already has a MIME type");
        }
        map.put(suffix, type);
        System.out.println("type added");
        Lease lease = landlord.newFileClassifierLease(this, suffix, DURATION);
        System.out.println("Lease is " + lease);
        return lease;
        //return null;
    }
    public void removeType(String suffix) {
        map.remove(suffix);
    }

    public FileClassifierImpl() throws RemoteException {
        // 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"));
        landlord  = new FileClassifierLandlord();
    }
} // FileClassifierImpl
```

Server

The server for this implementation is the same as the previous servers. It simply creates the service and registers a proxy with lookup services:

```
package lease;
import java.rmi.*;
import net.jini.lease.LeaseRenewalManager;
import java.rmi.RMISecurityManager;
import net.jini.core.lookup.ServiceID;
import net.jini.lookup.ServiceIDListener;
import common.LeaseFileClassifier;
import net.jini.lookup.JoinManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.export.Exporter;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.tcp.TcpServerEndpoint;
/**
 * FileClassifierServer.java
 */
public class FileClassifierServer implements ServiceIDListener  {
    protected FileClassifierImpl impl;

    public static void main(String argv[]) throws Exception {
        FileClassifierServer server = new FileClassifierServer();
        // keep server running forever to
        // - allow time for locator discovery and
        // - keep reregistering the lease
        Object keepAlive = new Object();
        synchronized(keepAlive) {
            try {
                keepAlive.wait();
            } catch(java.lang.InterruptedException e) {
                // do nothing
            }
        }
    }
    public FileClassifierServer() throws Exception {
        System.setSecurityManager(new RMISecurityManager());
        impl = new FileClassifierImpl();
        Exporter exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                                  new BasicILFactory());
        // export an object of this class
        Remote proxy = exporter.export(impl);
        JoinManager joinMgr = null;
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null /* unicast locators */,
                                           null /* DiscoveryListener */);
            joinMgr = new JoinManager(proxy,
                                      null,
                                      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());
    }
} // FileClassifierServer
```

FileClassifierLeasedResource Class

The `FileClassifierLeasedResource` implements the `LeasedResource` interface. It acts as a wrapper around the actual resource (a `LeaseFileClassifier`). It adds cookie and time expiration fields around the resource, and it creates a unique cookie for the resource, in addition to making the wrapped resource visible.

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

```
import common.LeaseFileClassifier;
import com.sun.jini.landlord.LeasedResource;
import net.jini.id.Uuid;
import net.jini.id.UuidFactory;
public class FileClassifierLeasedResource implements LeasedResource  {

    protected Uuid cookie;
    protected LeaseFileClassifier fileClassifier;
    protected long expiration = 0;
    protected String suffix = null;
    public FileClassifierLeasedResource(LeaseFileClassifier fileClassifier,
                                        String suffix) {
        this.fileClassifier = fileClassifier;
        this.suffix = suffix;
        cookie = UuidFactory.generate();
    }
    public void setExpiration(long newExpiration) {
        this.expiration = newExpiration;
    }
    public long getExpiration() {
        return expiration;
    }
    public Uuid getCookie() {
        return cookie;
    }
    public LeaseFileClassifier getFileClassifier() {
        return fileClassifier;
    }
    public String getSuffix() {
        return suffix;
    }
} // FileClassifierLeasedResource
```

Reaper

When leases expire, something should clean them up. For this, we will use a simple *reaper* thread that scans the map of leased resources regularly, looking for expired leases. When the reaper finds an expired lease, it removes the lease from the map of resources and also calls `removeType()` on the file classifier implementation.

```
package lease;
/**
 * Reaper.java
 */
import java.util.Map;
import java.util.Set;
import java.util.Iterator;
import java.rmi.RemoteException;
```

```
/**
 * Every minute, scan list of resources, remove those that
 */
public class Reaper extends Thread {
    private Map leasedResources;
    public Reaper(Map leasedResources) {
        this.leasedResources = leasedResources;
    } // Reaper constructor

    public void run() {
        while (true) {
            try {
                Thread.sleep(10*1000L);
            } catch (InterruptedException e) {
                // ignore
            }
            Set keys = leasedResources.keySet();
            Iterator iter = keys.iterator();
            System.out.println("Reaper running");
            while (iter.hasNext()) {
                Object key = iter.next();
                FileClassifierLeasedResource res =
                        (FileClassifierLeasedResource) leasedResources.get(key);
                long expires = res.getExpiration() - System.currentTimeMillis();
                if (expires < 0) {
                    leasedResources.remove(key);
                    try {
                        res.getFileClassifier().removeType(res.getSuffix());
                    } catch (RemoteException e) {
                        // ignore
                    }

                }
            }
        }
    }

} // Reaper
```

FileClassifierLandlord Class

The `FileClassifierLandlord` class is similar to `FooLandlord`, which was covered in Chapter 8. However, it also includes a reaper to clean up expired leases.

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

```
import common.LeaseFileClassifier;
import net.jini.core.lease.LeaseDeniedException;
import net.jini.core.lease.Lease;
import net.jini.core.lease.UnknownLeaseException;
import net.jini.id.Uuid;
import net.jini.id.UuidFactory;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Map;
import java.util.HashMap;
import com.sun.jini.landlord.Landlord;
import com.sun.jini.landlord.LeaseFactory;
import com.sun.jini.landlord.LeasedResource;
import com.sun.jini.landlord.FixedLeasePeriodPolicy;
import com.sun.jini.landlord.LeasePeriodPolicy;
import com.sun.jini.landlord.LeasePeriodPolicy.Result;
import com.sun.jini.landlord.Landlord.RenewResults;
import com.sun.jini.landlord.LandlordUtil;
import com.sun.jini.landlord.LocalLandlord;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.tcp.TcpServerEndpoint;
import net.jini.export.*;
import java.rmi.Remote;
public class FileClassifierLandlord implements Landlord, LocalLandlord {
    private static final long MAX_LEASE = Lease.FOREVER;
    private static final long DEFAULT_LEASE = 1000*60*5; // 5 minutes
    private Map leasedResourceMap = new HashMap();
    private LeasePeriodPolicy policy = new
        FixedLeasePeriodPolicy(MAX_LEASE, DEFAULT_LEASE);
    private Uuid myUuid = UuidFactory.generate();
    private LeaseFactory factory;
    public FileClassifierLandlord() throws java.rmi.RemoteException {
        Exporter exporter = new
            BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                              new BasicILFactory());
        Landlord proxy = (Landlord) exporter.export(this);
        factory = new LeaseFactory(proxy, myUuid);
        // start a reaper to clean up expired leases
        new Reaper(leasedResourceMap).start();
    }

    public void cancel(Uuid cookie) throws UnknownLeaseException {
        Object value;
        if ((value = leasedResourceMap.remove(cookie)) == null) {
            throw new UnknownLeaseException();
        }
        FileClassifierLeasedResource resource =
                      (FileClassifierLeasedResource) value;

        try {
            resource.getFileClassifier().removeType(resource.getSuffix());
        } catch (RemoteException e) {
            // ignore??
        }
    }
    public Map cancelAll(Uuid[] cookies) {
        return LandlordUtil.cancelAll(this, cookies);
    }
    public long renew(Uuid cookie,
                      long extension)
        throws net.jini.core.lease.LeaseDeniedException,
               net.jini.core.lease.UnknownLeaseException {
        LeasedResource resource = (LeasedResource)
            leasedResourceMap.get(cookie);
        LeasePeriodPolicy.Result result = null;
        if (resource != null) {
            result = policy.renew(resource, extension);
        } else {
            throw new UnknownLeaseException();
        }
        return result.duration;
    }
    public LeasePeriodPolicy.Result grant(LeasedResource resource,
                                          long requestedDuration)
        throws LeaseDeniedException {
        Uuid cookie = resource.getCookie();
        try {
            leasedResourceMap.put(cookie, resource);
        } catch(Exception e) {
            throw new LeaseDeniedException(e.toString());
        }
        return policy.grant(resource, requestedDuration);
    }
    public Lease newFileClassifierLease(LeaseFileClassifier fileClassifier,
                                        String suffixKey, long duration)
        throws LeaseDeniedException {
        FileClassifierLeasedResource resource =
                            new FileClassifierLeasedResource(fileClassifier,
                                                             suffixKey);
        Uuid cookie = resource.getCookie();
        // find out how long we should grant the lease for
        LeasePeriodPolicy.Result result = grant(resource, duration);
        long expiration = result.expiration;
        resource.setExpiration(expiration);
        Lease lease = factory.newLease(cookie, expiration);
        return lease;
    }
    public Landlord.RenewResults renewAll(Uuid[] cookies,
                                          long[] durations) {
        return LandlordUtil.renewAll(this, cookies, durations);
    }
} // FileClassifierLandlord
```

Lease Client

A sample client finds the service and adds a new type to it, getting a lease in return. It renews the lease and finally lets it expire.

```
package client;
import common.LeaseFileClassifier;
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.lease.Lease;
/**
 * TestFileClassifierLease.java
 */
public class TestFileClassifierLease implements DiscoveryListener {
    public static void main(String argv[]) {
        new TestFileClassifierLease();
        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(20*60*1000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
        System.out.println("Exiting normally");
    }
    public TestFileClassifierLease() {
        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[] {LeaseFileClassifier.class};
        LeaseFileClassifier 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];
            try {
                classifier = (LeaseFileClassifier) registrar.lookup(template);
            } catch(java.rmi.RemoteException e) {
                e.printStackTrace();
                System.exit(2);
            }
            if (classifier == null) {
                System.out.println("Classifier null");
                continue;
            }
            MIMEType type;
            try {
                type = classifier.getMIMEType("file1.txt");
                System.out.println("Type of known type file1.txt is " +
type.toString());
                type = classifier.getMIMEType("file1.ps");
                System.out.println("Type of unknown type file1.ps is " + type);
                // Add a type
                Lease lease = classifier.addType("ps",
                                     new MIMEType("text", "postscript"));
                if (lease != null) {
                    System.out.println("Added type for ps");
                    System.out.println("lease for " +
                                       (lease.getExpiration() -
                                        System.currentTimeMillis())/1000 +
                                      " seconds");
                    type = classifier.getMIMEType("file1.ps");
                    System.out.println("Type for now known type file1.ps is " +
                                      type.toString());
                    // sleep for 1 min and try again
                    System.out.println("Sleeping for 1 min");
                    Thread.sleep(1*60*1000L);
                    type = classifier.getMIMEType("file1.ps");
                    System.out.println("Type for still known type file1.ps is " +
                                       type.toString());
                    // renew lease
                    lease.renew(3*60*1000L);
                    System.out.println("renewed lease for " +
                                        (lease.getExpiration() -
                                         System.currentTimeMillis())/1000 +
                                       " seconds");
                    // let lease lapse
                    System.out.println("Sleeping for 4 min to let lease lapse");
                    Thread.sleep(4*60*1000L);
                    type = classifier.getMIMEType("file1.ps");
                    System.out.println("Type for now unknown type file1.ps is " +
                                       type);
                } else {
                    System.err.println("was null");
                }
            } catch(Exception e) {
                e.printStackTrace();
            }
            // System.exit(0);
        }
    }
    public void discarded(DiscoveryEvent evt) {
        // empty
    }
} // TestFileClassifierLease
```

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.