CHAPTER 9

Choices for Service Architecture

A CLIENT WILL ONLY BE LOOKING for an implementation of an interface, and the implementation can be done in many different ways, as discussed in this chapter. In the previous chapter we discussed the roles of service proxy and service backend and briefly talked about how different implementations could place different amounts of processing in the proxy or backend. This can lead to situations such as a thin proxy communicating to a fat backend using RMI, or at the other end of the scale, to a fat proxy and a thin backend. The last chapter showed one implementation—a fat proxy with a backend so thin that it did not exist. This chapter fills in some of the other possibilities.

Proxy Choices

A Jini service will be implemented using a proxy on the client side and a service backend on the service provider side. In RPC-like systems there is little choice: the proxy must be thin and the backend must be fat. Message-based client/server systems allow choices in the distribution of processing, so that one or other side can be fat or thin, or they can equally share. Jini allows a similar range of choices, but does so using the object-oriented paradigm supported by Java. The following sections discuss the choices in detail, giving alternative implementations of a file-classifier service.

Proxy Is the Service

One extreme proxy situation is where the proxy is so fat that there is nothing left to do on the server side. The role of the server is to register the proxy with service locators and just to stay alive (renewing leases on the service locators). The service itself runs entirely within the client. A class diagram for the file classifier problem using this method is given in Figure 9-1. This was the implementation discussed in the previous chapter.

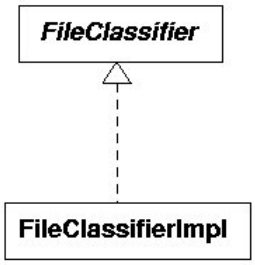

Figure 9-1. Class diagram for file classifier

We have already seen the full object diagram for the JVMs in Chapter 8, but these classes looks like Figure 9-2.

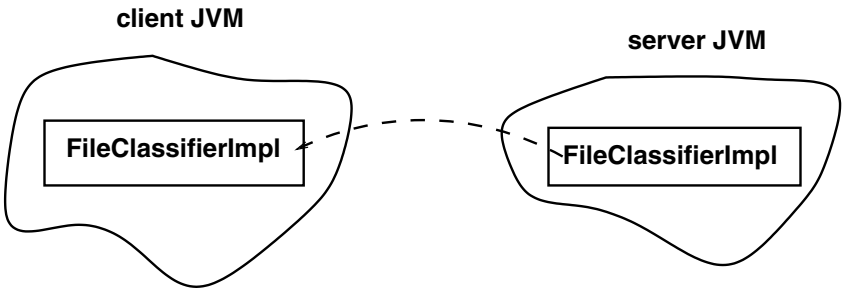

Figure 9-2. Objects in the JVMs

*Figure 9-2. Objects in the JVMs*The client asks for a `FileClassifier`. What is uploaded to the service locators, and thus what the client gets, is a `FileClassifier-Impl`. The `FileClassifierImpl` runs entirely within the client and does not communicate back to its server at all. This can also be done for any service if the service is purely a software one that does not need any link back to the server. It could be something like a calendar that is independent of location, or a diary that uses files on the client side rather than the server side.

RMI Proxy

The opposite proxy extreme is where *all* of the processing is done on the server side. The proxy just exists on the client to take calls from the client, invoke the method in the service on the server, and return the result to the client. Java's RMI

does this in a fairly transparent way (once all the correct files and additional servers are set up!).

A class diagram for an implementation of the file classifier using this mechanism is shown in Figure 9.3.

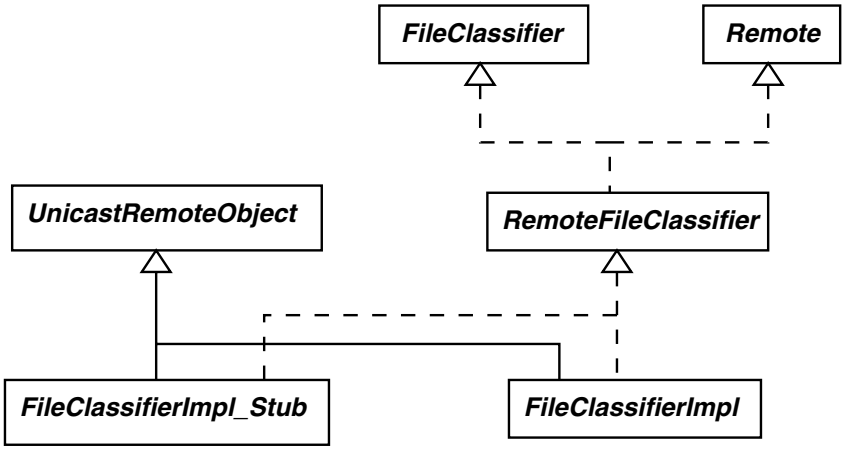

Figure 9-3. Class diagram for RMI proxy

The objects in the JVMs are shown in Figure 9-4.

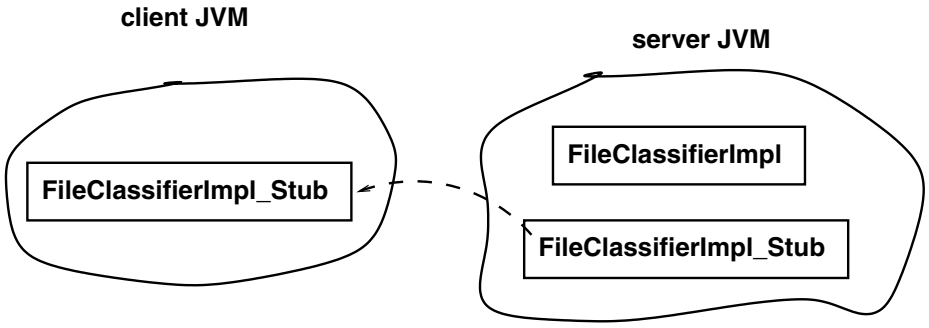

Figure 9-4. JVM objects for RMI proxy

The full code for this mechanism is given later in the chapter in the "RMI Proxy for FileClassifier" section.

The class structure for this mechanism is much more complex than the fat proxy because of RMI requirements. The `RemoteFileClassifier` interface has to be defined, and the implementation class has to implement (or call suitable methods

from) the `UnicastRemoteObject` class. The `FileClassifierImpl_Stub` is generated from `FileClassifierImpl` by using the `rmic` compiler. Implementing the `Remote` interface allows the methods of the `FileClassifierImpl` to be called remotely. Inheriting from `UnicastRemoteObject` allows RMI to export the stub rather than the service, which remains on the server.

Apart from creating the stub class by using `rmic`, the stub is essentially invisible to the programmer; the server code is written to export the implementation, but the RMI runtime component of Java recognizes this and actually exports the stub instead. This can cause a little confusion—the programmer writes code to export an object of one class, but an object of a different class appears in the service locator and in the client.

This structure is useful when the service needs to do no processing on the client side but does need to do a lot on the server side—for example, a diary that stores all information communally on the server rather than individually on each client. Services that are tightly linked to a piece of hardware on the server give further examples.

Non-RMI Proxy

If RMI is not used, and the proxy and backend service want to share processing, then both the backend service and the proxy must be created explicitly on the service provider side. The proxy is explicitly exported by the service provider and must implement the interface, but on the server side this requirement does not hold, since the proxy and backend service are not tightly linked by a class structure any more. The class diagram for the file classifier with this organization is displayed in Figure 9-5.

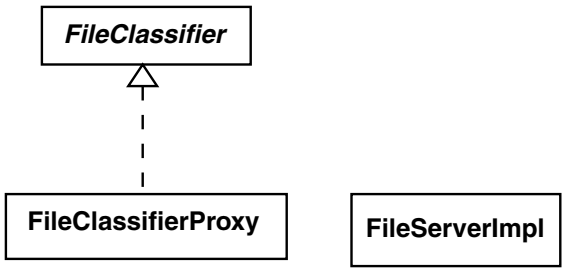

Figure 9-5. Class diagram for non-RMI proxy

The JVMs at runtime for this scenario are shown in Figure 9-6.

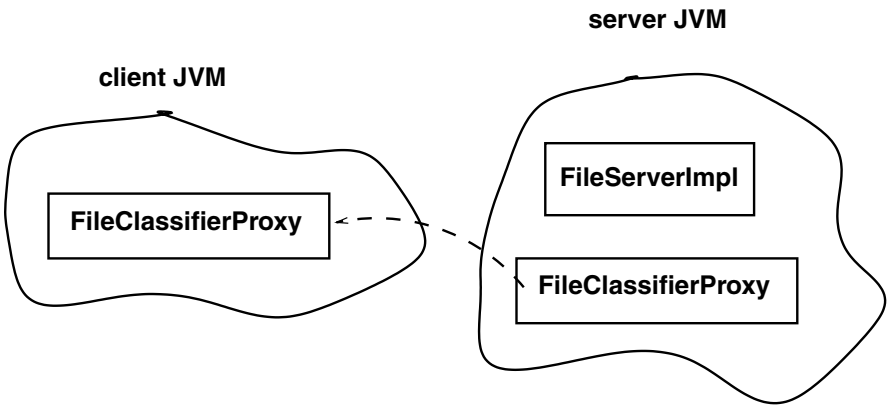

Figure 9-6. JVM objects for a non-RMI proxy

This doesn't specify how the proxy and the server communicate. They could open up a socket connection, for example, and exchange messages using a message structure that only they understand. Or they could communicate using a well-known protocol, such as HTTP. For example, the proxy could make HTTP requests, and the service could act as an HTTP server handling these requests and returning documents. A version of the file classifier using sockets to communicate is given later in this chapter in the "Non-RMI Proxy for FileClassifier" section.

This model is good for bringing "legacy" client/server applications into the Jini world. Client/server applications often communicate using a specialized protocol between the client and server. Copies of the client have to be distributed to all machines, and if there is a bug in the client, they all have to be updated, which is often impossible. Worse, if there is a change to the protocol, the server must be rebuilt to handle old and new versions while attempts are made to update all the clients. This is a tremendous problem with Web browsers, for example, that have varying degrees of support for HTML 3.2 and HTML 4.0 features, let alone new protocol extensions such as style sheets and XML. CGI scripts that attempt to deliver the "right" version of documents to various browsers are clumsy, but necessary, hacks.

What can be done instead is to distribute a "shell" client that just contacts the server and uploads a proxy. The Jini proxy is the real "heart" of the client, whereas the Jini backend service is the server part of the original client/server system. When changes occur, the backend service and its proxy can be updated together, and there is no need to make changes to the shell out on all the various machines.

RMI and Non-RMI Proxies

The last variation is to have a backend service, an explicit proxy, and an RMI proxy. Both of the proxies are exported: the explicit proxy has to be exported by registering it with lookup services, while the RMI proxy is exported by the RMI runtime mechanisms. The RMI proxy can be used as an intermediary for RPC-like communication between the explicit proxy and the backend service. This is just like the last case, but instead of requiring the proxy and service to implement their own communication protocol, it uses RMI instead. The proxy and service can be of any relative size, just like in the last case. What this does is simplify the task of the programmer.

Later in the chapter, in the "RMI and Non-RMI Proxies for FileClassifier" section, there is a non-RMI proxy, `FileClassifierProxy`, implementing the `FileClassifier` interface. This communicates with an object that implements the `ExtendedFileClassifier` interface. There is an object on the server of type `ExtendedFileClassifierImpl` and an RMI proxy for this on the client side of type `ExtendedFileClassifierImpl_Stub` The class diagram is shown in Figure 9-7.

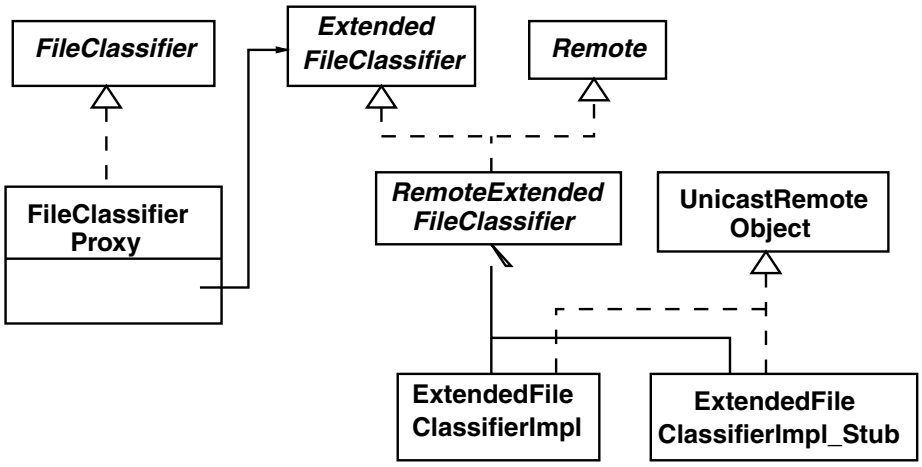

Figure 9-7. Class diagram for RMI and non-RMI proxies

While this looks complex, it is really just a combination of the last two cases. The proxy makes local calls on the RMI stub, which makes remote calls on the service. The JVMs are displayed in Figure 9-8.

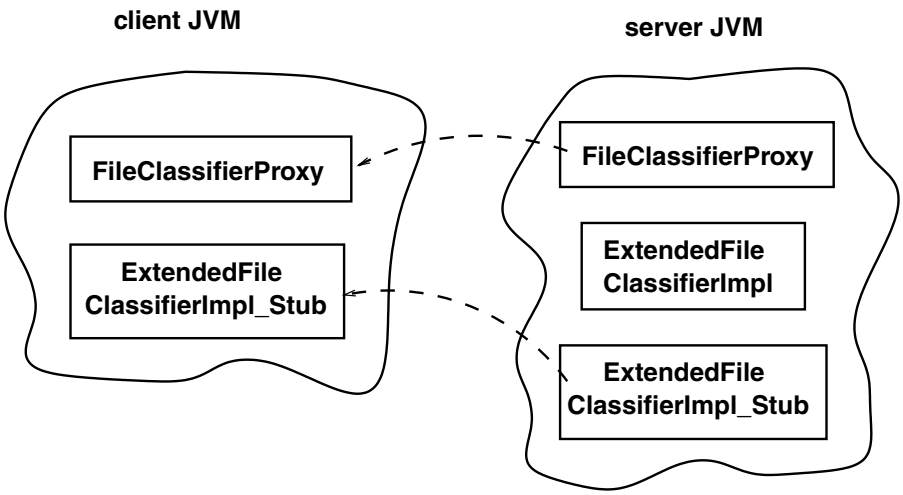

Figure 9-8. JVM objects for RMI and non-RMI proxies

RMI Proxy for FileClassifier

An RMI proxy can be used when all of the work done by the service is done on the server side. In that case, the server exports a thin proxy that simply channels method calls from the client across the network to the "real" service in the server, and returns the result back to the client. The programming for this is relatively simple. The service has to do two major things in its class structure:

1. Implement `Remote`. This is because methods will be called on the service from the proxy, and these will be remote calls on the service.

2. Inherit from `UnicastRemoteObject` (or `Activatable`). This means that it's the backend service's constructor that will create and export a proxy or stub object without the programmer having to do anything more. (An alternative to inheritance is for the object to call the `UnicastRemoteObject.exportObject()` method.)

What Doesn't Change

In Chapter 8, we discussed a file-classifier application built from a client and a service, and in this chapter we have shown a different implementation of the service. A new file-classifier application can be built using this new implementation of the service. Clearly, some things must change in this new version, but because of the Jini architecture, the changes are basically localized to the service implementation. That

is, most of the file-classifier application doesn't change at all, even if the service implementation changes.

The client is not concerned about the implementation of the service at all, and so the client doesn't change. The `FileClassifier` interface doesn't change either, since this is fixed and used by any client and any service implementation. We have already declared its methods to throw `RemoteException`, so a proxy is able to call its methods remotely. The `MIMEType` doesn't change either, since we have already declared it to implement `Serializable`–it is passed back across the network from the service to its proxy.

RemoteFileClassifier

An implementation of the service using an RMI proxy will need to implement both the `FileClassifier` and the `Remote` interfaces. It is convenient to define another interface, called `RemoteFileClassifier`, just to do this. This interface will be used fairly frequently in the rest of this book.

```
package rmi;

import common.FileClassifier;
import java.rmi.Remote;

/**
 * RemoteFileClassifier.java
 */

public interface RemoteFileClassifier extends FileClassifier, Remote {

} // RemoteFileClasssifier
```

FileClassifierImpl

The service provider will run the backend service. When the backend service exports an RMI proxy, it will look like this:

```
package rmi;

import java.rmi.server.UnicastRemoteObject;
import common.MIMEType;
import common.FileClassifier;
```

```
/**
 * FileClassifierImpl.java
 */

public class FileClassifierImpl extends UnicastRemoteObject
                                implements RemoteFileClassifier {

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        System.out.println("Called with " + fileName);
        if (fileName.endsWith(".gif")) {
            return new MIMEType("image", "gif");
        } else if (fileName.endsWith(".jpeg")) {
            return new MIMEType("image", "jpeg");
        } else if (fileName.endsWith(".mpg")) {
            return new MIMEType("video", "mpeg");
        } else if (fileName.endsWith(".txt")) {
            return new MIMEType("text", "plain");
        } else if (fileName.endsWith(".html")) {
            return new MIMEType("text", "html");
        } else
            // fill in lots of other types,
            // but eventually give up and
            return new MIMEType(null, null);
    }


    public FileClassifierImpl()  throws java.rmi.RemoteException {
        // empty constructor required by RMI
    }

} // FileClassifierImpl
```

FileClassifierServer

The service provider changes very little from the version in Chapter 8, which exported a complete service. Both this server and the earlier one export a service object with `register()`, but at this point the RMI runtime intervenes and substitutes an RMI stub object. The other major change is that the server no longer needs to explicitly stay alive. While the RMI system keeps a reference to the RMI stub object, it keeps alive the JVM that contains the stub object. This means that the daemon threads that are looking after the discovery process will continue to

run, and in turn, since they have a reference to the service provider as listener, the service provider will continue to exist.

The following server creates and manages the RMI service:

```
package rmi;

import rmi.FileClassifierImpl;
import rmi.RemoteFileClassifier;

import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
// import 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;
import java.rmi.RMISecurityManager;

/**
 * FileClassifierServer.java
 */

public class FileClassifierServerRMI implements DiscoveryListener, LeaseListener {

    protected FileClassifierImpl impl;
    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

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

        // no need to keep server alive, RMI will do that
    }

    public FileClassifierServerRMI() {
        try {
            impl = new FileClassifierImpl();
        } catch(Exception e) {
            System.err.println("New impl: " + e.toString());
```

```
            System.exit(1);
        }

        // install suitable security manager
        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();
        RemoteFileClassifier service;

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

            // export the proxy service
            ServiceItem item = new ServiceItem(null,
                                               impl,
                                               null);
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.print("Register exception: ");
                e.printStackTrace();
                // System.exit(2);
                continue;
            }
            try {
                System.out.println("service registered at " +
                                   registrar.getLocator().getHost());
            } catch(Exception e) {
            }
            leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);
```

```
        }
    }

    public void discarded(DiscoveryEvent evt) {

    }

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

} // FileClassifierServerRMI
```

What Classes Need to Be Where?

This chapter deals with a number of different implementations of the file-classifier service. Each implementation introduces some new classes, but also depends on some of the classes we have developed in earlier chapters. In deploying the service, we need to pay attention to this set of classes and determine which classes need to be known to the different parts of the Jini system. This "What Classes Need to Be Where?" section is repeated for each of the different service implementations, and it describes the configuration issues for each of these different implementation choices.

For the RMI proxy implementation, we need to consider these classes:

- `common.MIMEType`
- `common.FileClassifier`
- `rmi.RemoteFileClassifier`
- `rmi.FileClassifierImpl`
- `rmi.FileClassifierImpl_Stub`
- `rmi.FileClassifierServer`
- `client.TestFileClassifier`

(The `FileClassifierImpl_Stub` class is added to our classes by `rmic` as discussed in the next section.)

These classes could be running on up to four different machines:

- The server machine for `FileClassifierServer`
- The HTTP server, which may be on a different machine
- The machine for the lookup service
- The machine running the `TestFileClassifier` client

So, which classes need to be known to which machines?

The server running `FileClassifierServer` needs to know the following classes and interfaces:

- The `common.FileClassifier` interface
- The `rmi.RemoteFileClassifier` interface
- The `common.MIMEType` class
- The `rmi.FileClassifierServer` class
- The `rmi.FileClassifierImpl` class

The lookup service does not need to know any of these classes. It just deals with them in the form of a `java.rmi.MarshalledObject`.

The client needs to know the following:

- The `common.FileClassifier` interface
- The `common.MIMEType` class

In addition, the HTTP server needs to be able to load and store classes. It needs to be able to access the following:

- The `rmi.FileClassifierImpl_Stub` interface
- The `rmi.RemoteFileClassifier` interface
- The `common.FileClassifier` interface
- The `common.MIMEType` class

The reason for all of these is slightly complex. In the `FileClassifierProxy` constructor, the `FileClassifierImpl` class is passed in. The RMI runtime converts this to `FileClassifierImpl_Stub`. This class implements the same interfaces as `FileClassifierImpl`, that is, `RemoteFileClassifier` and hence `FileClassifier`, so these also need to be available. In the implementation, `FileClassifierImpl` references the `MIMEType` class, so this must also be available.

So, what does the phrase "available" mean in the last sentence? The HTTP server will look for files based on the `java.rmi.server.codebase` property of the application server. The value of this property is a URL. Often, URLs can be file references such as `file://home/jan/index.html` or HTTP references such as `http://host/index.html`. But for this case, clients running anywhere will use the URL, so it cannot be a file reference specific to a particular machine. For the same reason, it cannot be just `localhost`, unless you are running every part of a Jini federation on a single computer!

If `java.rmi.server.codebase` is an HTTP reference, then the preceding class files must be accessible from that reference. For example, suppose the property is set to

```
java.rmi.server.codebase=http://myWebHost/classes
```

(where `myWebHost` is the name of the HTTP server's host) and this Web server has its `DocumentRoot` set to `/home/webdocs`. In that case, these files must exist:

```
/home/webdocs/classes/rmi/FileClassifierImpl_Stub.class
/home/webdocs/classes/rmi/RemoteFileClassifier.class
/home/webdocs/classes/common/FileClassifier.class
/home/webdocs/classes/common/MIMEType.class
```

Running the RMI Proxy FileClassifier

As with the file classifier developed in Chapter 8, we again have a server and a client to run. The client does not depend on how the service is implemented, and it does not even find out about the service until it has been started and has performed a search for the service. That means the client is started in exactly the same way as it was started in Chapter 8:

```
java -Djava.security.policy=policy.all client.TestFileClassifier
```

The server in this situation is more complex than the one in Chapter 8, because the RMI runtime is manipulating RMI stubs, and these have additional requirements. Firstly, RMI stubs must be generated during compilation. Secondly, security rights must be set, because an `RMISecurityManager` is used.

Although the `FileClassifierImpl` is created explicitly by the server, it is not this class file that is moved around. The `FileClassifierImpl` object continues to exist on the server machine. Rather, a stub object is moved around and will run on the client machine. This stub is responsible for sending the method requests back to the implementation class on the server. The client machine must be able to access the class file for the stub. This class file has to be generated from the implementation class by the stub compiler `rmic` with the following command:

```
rmic -v1.2 -d /home/webdocs/classes rmi.FileClassifierImpl
```

Here, the `-v1.2` option says to generate JDK 1.2 stubs only, and the `-d` option says where to place the resultant stub class files so that they can be located by the HTTP server (in this case, in the local file system). If the `-v1.2` option is omitted, `rmic` will also generate Java 1.1 skeleton files, which are not needed. In Java 1.3, it may not be necessary to even run `rmic`. Note that the pathnames for directories here and later do not include the package name of the class files. The class files (here `FileClassifierImpl_Stub.class`) will be placed in and looked for in the appropriate subdirectories.

The value of `java.rmi.server.codebase` must specify the protocol used by the HTTP server to find the class files. This could be the `file` protocol or the `http` protocol. For example, if the class files are stored on my Web server's pages under `classes/rmi/FileClassifierImpl_Stub.class`. the codebase would be specified as

```
java.rmi.server.codebase=http://myWebHost/classes/
```

(where `myWebHost` is the name of the HTTP server).

The server also sets a security manager. This is a restrictive one, so it needs to be told to allow access. This can be done by setting the `java.security.policy` property to point to a security policy file, such as `policy.all`.

Combining all these points leads to startups such as this:

```
java -Djava.rmi.server.codebase=http://myWebHost/classes/ \
     -Djava.security.policy=policy.all \
     rmi.FileClassifierServer
```

Non-RMI Proxy for FileClassifier

Many client-server programs communicate by message passing, often using a TCP socket. The two sides need to have an agreed-upon protocol; that is, they must have a standard set of message formats and know what messages to receive and what replies to send at any time. Jini can be used in this sort of case by providing a wrapper around the client and server, and making them available as a Jini service.

The original client then becomes a proxy agent for the server and is distributed to Jini clients for execution. The original server runs within the Jini server and performs the real work of the service, just as in the thin proxy model. What differs is the class structure and how the components communicate.

The proxy and the service do not need to belong to the same class, or even share common superclasses. Unlike the RMI case, the proxy is not derived from the service, so they do not have a shared class structure. The proxy and the service are written independently, using their own appropriate class hierarchies. However, the proxy still has to implement the `FileClassifier` interface, since that is what the client is asking for and the proxy is delivering.

If RMI is not used, then any other distributed communication mechanism can be employed. Typically client-server systems will use something like reliable TCP ports—this is not the only choice, but it is the one used in this example. Thus, the service listens on an agreed-upon port, the client connects to this port, and they exchange messages.

The message format adopted for this solution is really simple:

- The proxy sends a message giving the file extension that it wants classified. This can be sent as a newline-terminated string (terminated by the `'\n'` character).

- The service will either succeed or fail in the classification. If it fails, it sends a single line of the empty string `""` followed by a newline. If it succeeds, it sends two lines, the first being the content type, the second the subtype.

The proxy will then use this reply to either return `null` or a new `MIMEType` object.

FileClassifierProxy

The proxy object will be exported completely to a Jini client, such as `TestFileClassifier`. When this client calls the `getMIMEType()` method, the proxy opens up a connection to the service on an agreed-upon TCP port and exchanges messages on this port. It then returns a suitable result. The code looks like this:

```
package socket;

import common.FileClassifier;
import common.MIMEType;
import java.net.Socket;

import java.io.Serializable;
import java.io.IOException;
```

```
import java.rmi.Naming;

import java.io.*;

/**
 * FileClassifierProxy
 */

public class FileClassifierProxy implements FileClassifier, Serializable {

    static public final int PORT = 2981;
    protected String host;

    public FileClassifierProxy(String host) {
        this.host = host;
    }

    public MIMEType getMIMEType(String fileName)
        throws java.rmi.RemoteException {
        // open a connection to the service on port XXX
        int dotIndex = fileName.lastIndexOf('.');
        if (dotIndex == -1 || dotIndex + 1 == fileName.length()) {
            // can't find suitable index
            return null;
        }
        String fileExtension = fileName.substring(dotIndex + 1);

        // open a client socket connection
        Socket socket = null;
        try {
             socket = new Socket(host, PORT);
        } catch(Exception e) {
            return null;
        }

        String type = null;
        String subType = null;

        /*
         * protocol:
         * Write: file extension
         * Read: "null" + '\n'
         *       type + '\n' + subtype + '\n'
         */
```

```
        try {
            InputStreamReader inputReader =
                new InputStreamReader(socket.getInputStream());
            BufferedReader reader = new BufferedReader(inputReader);
            OutputStreamWriter outputWriter =
                new OutputStreamWriter(socket.getOutputStream());
            BufferedWriter writer = new BufferedWriter(outputWriter);

            writer.write(fileExtension);
            writer.newLine();
            writer.flush();

            type = reader.readLine();
            if (type.equals("null")) {
                return null;
            }
            subType = reader.readLine();
        } catch(IOException e) {
            return null;
        }
        // and finally
        return new MIMEType(type, subType);
    }
} // FileClassifierProxy
```

FileServerImpl

The `FileServerImpl` service will be running on the server side. It will run in its own thread (inheriting from `Thread`) and will listen for connections. When one is received, it will create a new `Connection` object in its own thread, to handle the message exchange. (This creation of another thread is probably overkill here where the entire message exchange is very short, but it is good practice for more complex situations.)

```
/**
 * FileServerImpl.java
 */

package socket;

import java.net.*;
import java.io.*;
```

```
public class FileServerImpl extends Thread {

    protected ServerSocket listenSocket;

    public FileServerImpl() {
        try {
            listenSocket = new ServerSocket(FileClassifierProxy.PORT);
        } catch(IOException e) {
            e.printStackTrace();
        }
    }

    public void run() {
        try {
            while(true) {
                Socket clientSocket = listenSocket.accept();
                new Connection(clientSocket).start();
            }
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
} // FileServerImpl

class Connection extends Thread {

    protected Socket client;

    public Connection(Socket clientSocket) {
        client = clientSocket;
    }

    public void run() {
        String contentType = null;
        String subType = null;

        try {
            InputStreamReader inputReader =
                new InputStreamReader(client.getInputStream());
            BufferedReader reader = new BufferedReader(inputReader);
            OutputStreamWriter outputWriter =
                new OutputStreamWriter(client.getOutputStream());
            BufferedWriter writer = new BufferedWriter(outputWriter);
```

```
            String fileExtension = reader.readLine();

            if (fileExtension.equals("gif")) {
                contentType = "image";
                subType = "gif";
            } else if (fileExtension.equals("txt")) {
                contentType = "text";
                subType = "plain";
            } // etc

            if (contentType == null) {
                writer.write("null");
            } else {
                writer.write(contentType);
                writer.newLine();
                writer.write(subType);
            }
            writer.newLine();
            writer.close();
        } catch(IOException e) {
            e.printStackTrace();
        }
    }
}
```

Service Provider

The Jini service provider must start a `FileServerImpl` to listen for later connections. Then it can register a `FileClassifierProxy` proxy object with each lookup service, which will send them on to interested clients. The proxy object must know where the service backend object (the `FileServerImpl`) is listening in order to attempt a connection to it, and this information is given by first making a query for the local host and then passing the hostname to the proxy in its constructor.

```
package socket;

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;  // Jini 1.0
// import com.sun.jini.lease.LeaseListener;        // Jini 1.0
// import com.sun.jini.lease.LeaseRenewalEvent;    // Jini 1.0
import net.jini.lease.LeaseRenewalManager;
import net.jini.lease.LeaseListener;
import net.jini.lease.LeaseRenewalEvent;
import java.rmi.RMISecurityManager;
import java.net.*;

/**
 * FileClassifierServer.java
 */

public class FileClassifierServer implements DiscoveryListener, LeaseListener {

    protected FileClassifierProxy proxy;
    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

    public static void main(String argv[]) {
        new FileClassifierServer();
        try {
            Thread.sleep(1000000L);
        } catch(Exception e) {
        }
    }

    public FileClassifierServer() {
        try {
            new FileServerImpl().start();
        } catch(Exception e) {
            System.err.println("New impl: " + e.toString());
            System.exit(1);
        }

        // set RMI scurity manager
        System.setSecurityManager(new RMISecurityManager());

        // proxy primed with address
        String host = null;
        try {
            host = InetAddress.getLocalHost().getHostName();
        } catch(UnknownHostException e) {
            e.printStackTrace();
            System.exit(1);
```

```
            }

            proxy = new FileClassifierProxy(host);
            // now continue as before
            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("found registrars");
                ServiceRegistrar registrar = registrars[n];

                // export the proxy service
                ServiceItem item = new ServiceItem(null,
                                                   proxy,
                                                   null);
                ServiceRegistration reg = null;
                try {
                    reg = registrar.register(item, Lease.FOREVER);
                } catch(java.rmi.RemoteException e) {
                    System.err.print("Register exception: ");
                    e.printStackTrace();
                    // System.exit(2);
                    continue;
                }
                try {
                    System.out.println("service registered at " +
                                       registrar.getLocator().getHost());
                } catch(Exception e) {
                }
                leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);
            }
        }
```

```
    public void discarded(DiscoveryEvent evt) {

    }

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

} // FileClassifierServer
```

What Classes Need to Be Where?

This section has considered a non-RMI proxy implementation. An application that uses this service implementation will need to deal with these classes:

- `common.MIMEType`
- `common.FileClassifier`
- `socket.FileClassifierProxy`
- `socket.FileServerImpl`
- `socket.FileClassifierServer`
- `client.TestFileClassifier`

Objects in these classes could be running on up to four different machines:

- The server machine for `FileClassifierServer`
- The HTTP server, which may be on a different machine
- The machine for the lookup service
- The machine running the `TestFileClassifier` client

So, what classes need to be known to which machines?

The server running `FileClassifierServer` needs to know the following classes and interfaces:

- The `common.FileClassifier` interface

- The `common.MIMEType` class
- The `socket.FileClassifierServer` class
- The `socket.FileClassifierProxy` class
- The `socket.FileServerImpl` class

The lookup service does not need to know any of these classes. It just deals with them in the form of a `java.rmi.MarshalledObject`.

The client needs to know the following:

- The `common.FileClassifier` interface
- The `common.MIMEType` class

In addition, the HTTP server needs to be able to load and store classes. It needs to be able to access the following:

- The `socket.FileClassifierProxy` interface
- The `common.FileClassifier` interface
- The `common.MIMEType` class

Running the RMI Proxy FileClassifier

A file classification application will have to run a server and a client, as in the earlier standalone implementation in Chapter 8 and the RMI implementation just a few pages earlier. The client is unchanged, as it does not care which server implementation is used:

```
java -Djava.security.policy=policy.all client.TestFileClassifier
```

The value of `java.rmi.server.codebase` must specify the protocol used by the HTTP server to find the class files. This could be the `file` protocol or the `http` protocol. For example, if the class files are stored on my Web server's pages under `classes/socket/FileClassifierProxy.class`, the codebase would be specified as

```
java.rmi.server.codebase=http://myWebHost/classes/
```

(where `myWebHost` is the name of the HTTP server host).

The server also sets a security manager. This is a restrictive one, so it needs to be told to allow access. This can be done by setting the `java.security.policy` property to point to a security policy file, such as `policy.all`.

Combining all these points leads to startups such as this:

```
java -Djava.rmi.server.codebase=http://myWebHost/classes/ \
     -Djava.security.policy=policy.all \
     FileClassifierServer
```

RMI and non-RMI Proxies for FileClassifier

An alternative that is often used for client/server systems instead of message passing is remote procedure calls (RPC). This involves a client that does some local processing and makes some RPC calls to the server. We can also bring this into the Jini world by using a proxy that does some processing on the client side, and that makes use of an RMI proxy/stub when it needs to make calls back to the service. The RPC mechanism would most naturally be done using RMI in Java.

Some file types are more common than others: GIF, DOC, and HTML files, abound, but there are many more types, ranging from less common ones, such as FrameMaker MIF files, to downright obscure ones, such as PDP11 overlay files. An implementation of a file classifier might place the common types in a proxy object that makes them quickly available to clients, and the less common ones back on the server, accessible through a (slower) RMI call.

FileClassifierProxy

The proxy object will implement `FileClassifier` so that clients can find it. The implementation will handle some file types locally, but others it will pass on to another object that implements the `ExtendedFileClassifier` interface. The `ExtendedFileClassifier` has one method: `getExtraMIMEType()`. The proxy is told about this other object at constructor time. The `FileClassifierProxy` class is as follows:

```
/**
 * FileClassifierProxy.java
 */

package extended;

import common.FileClassifier;
import common.ExtendedFileClassifier;
import common.MIMEType;
```

```
import java.rmi.RemoteException;

public class FileClassifierProxy implements FileClassifier {

    /**
     * The service object that knows lots more MIME types
     */
    protected ExtendedFileClassifier extension;

    public FileClassifierProxy(ExtendedFileClassifier ext) {
        this.extension = ext;
    }

    public MIMEType getMIMEType(String fileName)
        throws RemoteException {
            if (fileName.endsWith(".gif")) {
            return new MIMEType("image", "gif");
        } else if (fileName.endsWith(".jpeg")) {
            return new MIMEType("image", "jpeg");
        } else if (fileName.endsWith(".mpg")) {
            return new MIMEType("video", "mpeg");
        } else if (fileName.endsWith(".txt")) {
            return new MIMEType("text", "plain");
        } else if (fileName.endsWith(".html")) {
            return new MIMEType("text", "html");
        } else {
            // we don't know it, pass it on to the service
            return extension.getExtraMIMEType(fileName);
        }
    }
} // FileClassifierProxy
```

ExtendedFileClassifier

The `ExtendedFileClassifier` interface will be the top-level interface for the service and an RMI proxy for the service. It will be publicly available for all clients to use. An immediate subinterface, `RemoteExtendedFileClassifier`, will add the `Remote` interface:

```
/**
 * ExtendedFileClassifier.java
 */
```

```
package common;

import java.io.Serializable;
import java.rmi.RemoteException;

public interface ExtendedFileClassifier extends Serializable {

    public MIMEType getExtraMIMEType(String fileName)
        throws RemoteException;

} // ExtendedFileClassifier
```

and

```
/**
 * RemoteExtendedFileClassifier.java
 */

package extended;

import java.rmi.Remote;

interface RemoteExtendedFileClassifier extends common.ExtendedFileClassifier,
Remote {

} // RemoteExtendedFileClassifier
```

ExtendedFileClassifierImpl

The implementation of the `ExtendedFileClassifier` interface is done by an `ExtendedFileClassifierImpl` object. This will also need to extend `UnicastRemoteObject` so that the RMI runtime can create an RMI proxy for it. Since this object may handle requests from many proxies, an alternative implementation of searching for MIME types using a hash table is given. This is more efficient for repeated searches:

```
/**
 * ExtendedFileClassifierImpl.java
 */

package extended;

import java.rmi.server.UnicastRemoteObject;
import common.MIMEType;
```

```
import java.util.HashMap;
import java.util.Map;

public class ExtendedFileClassifierImpl extends UnicastRemoteObject
    implements RemoteExtendedFileClassifier {

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

    public ExtendedFileClassifierImpl() throws java.rmi.RemoteException {
        /* This object will handle all classification attempts
         * that fail in client-side classifiers. It will be around
         * a long time, and may be called frequently, so it is worth
         * optimizing the implementation by using a hash map
         */
        map.put("rtf", new MIMEType("application", "rtf"));
        map.put("dvi", new MIMEType("application", "x-dvi"));
        map.put("png", new MIMEType("image", "png"));
        // etc
    }

    public MIMEType getExtraMIMEType(String fileName)
        throws java.rmi.RemoteException {
        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;
    }
} // ExtendedFileClassifierImpl
```

FileClassifierServer

The final piece in this jigsaw puzzle is the server that creates the service (and implicitly the RMI proxy for the service) and also the proxy primed with knowledge of the service:

```
package extended;

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;
import java.rmi.RMISecurityManager;

/**
 * FileClassifierServer.java
 */

public class FileClassifierServer implements DiscoveryListener, LeaseListener {

    protected FileClassifierProxy proxy;
    protected ExtendedFileClassifierImpl impl;
    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

    public static void main(String argv[]) {
        new FileClassifierServer();
        // RMI keeps this alive
    }

    public FileClassifierServer() {
        try {
            impl = new ExtendedFileClassifierImpl();
        } catch(Exception e) {
            System.err.println("New impl: " + e.toString());
            System.exit(1);
```

```
            }

            // set RMI scurity manager
            System.setSecurityManager(new RMISecurityManager());

            // proxy primed with impl
            proxy = new FileClassifierProxy(impl);

            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("found registrars");
                ServiceRegistrar registrar = registrars[n];

                // export the proxy service
                ServiceItem item = new ServiceItem(null,
                                                   proxy,
                                                   null);
                ServiceRegistration reg = null;
                try {
                    reg = registrar.register(item, Lease.FOREVER);
                } catch(java.rmi.RemoteException e) {
                    System.err.print("Register exception: ");
                    e.printStackTrace();
                    continue;
                }
                try {
                    System.out.println("service registered at " +
                                       registrar.getLocator().getHost());
                } catch(Exception e) {
                }
```

```
            leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);
        }
    }

    public void discarded(DiscoveryEvent evt) {

    }

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

} // FileClassifierServer
```

What Classes Need to Be Where?

The implementation of the file classifier in this section uses both RMI and non-RMI proxies. As in other implementations, there is a set of classes involved that need to be known to different parts of an application. We have these classes:

- `common.MIMEType`
- `common.FileClassifier`
- `common.ExtendedFileClassifier`
- `extended.FileClassifierProxy`
- `extended.RemoteExtendedFileClassifier`
- `extended.ExtendedFileServerImpl`
- `extended.FileClassifierServer`
- `client.TestFileClassifier`

The server running `FileClassifierServer` needs to know the following classes and interfaces:

- The `common.FileClassifier` interface
- The `common.MIMEType` class
- The `common.ExtendedFileClassifier` class

- The `extended.FileClassifierServer` class
- The `extended.FileClassifierProxy` class
- The `extended.RemoteExtendedFileClassifier` class
- The `extended.ExtendedFileServerImpl` class

The lookup service does not need to know any of these classes. It just deals with them in the form of a `java.rmi.MarshalledObject`.

The client needs to know the following:

- The `common.FileClassifier` interface
- The `common.MIMEType` class

In addition, the HTTP server needs to be able to load and store classes. It needs to be able to access the following:

- The `extended.FileClassifierProxy` interface
- The `extended.RemoteExtendedFileClassifier` class
- The `extended.ExtendedFileServerImpl_Stub` class
- The `common.FileClassifier` interface
- The `common.MIMEType` class

Using Other Services

In all the examples so far, a proxy has been created in a server and registered with a lookup service. Meanwhile, a service backend has usually been left behind in the server to handle calls from the proxy. However, there may be no need for the service to exist on the server, and the proxy could make use of other services elsewhere. This may be subject to security restrictions imposed by the client, which may disallow connections to some hosts.

In this section, we shall give an example of using a non-Jini service on another host. Recently an Australian, Pat Farmer, attempted to set a world record for jogging the longest distance. While he was running around, I became involved in a small project to broadcast his heartbeat live to the Web; a heart monitor was attached to him, which talked via an RS232 link to a mobile phone he was carrying.

This did a data transfer to a program running at `http://www.micromed.com.au` located at the Gold Coast, which forwarded the data to a machine at the Distributed Systems Technology Centre (DSTC) in Brisbane. This ran a Web server delivering an applet, and the applet talked back to a server on the DSTC machine, which sent out the data to each applet as it was received from the heart monitor.

Now that the experiment is over, the broadcast data is sitting as a file at `http://www.micromed.com.au/patfarmer/v2/patfhr.ecg`, and it can be viewed on the applet from `http://www.micromed.com.au/patfarmer/v2/heart.html`. We can make it into a Jini service as follows:

1. Create a service that we can locate using the service type ("display a heart monitor trace") and information about it, such as whose heart trace it is showing.

2. Have the service connect to an HTTP address encoded into the service by its constructor (or other means), and read from this and display the contents, assuming it is heart cardiograph data.

3. The information about whose trace it is can be given by a `Name` entry.

The client shows what you see in Figure 9-9. The break towards the right-hand side shows where the current trace is being written (it scans from left to right, overwriting as it goes). Cardiologists do not seem to be concerned about the lack of horizontal or vertical scales, as long as the trace is physically the right size!

Figure 9-9. Heart monitor trace service

The heart monitor service can be regarded in a couple of ways:

- It is a full-blown service uploaded to the client that just happens to use an external data source supplied from an HTTP server.

- It is a "fat" proxy to the HTTP service, and it acts as a client to this service by displaying the data.

Many other non-RMI services can be built that act in this "fat proxy" style.

Heart Interface

The `Heart` interface only has one method, and that is to `show()` the heart trace in some manner:

```
/**
 * Heart.java
 */

package heart;

public interface Heart extends java.io.Serializable {

    public void show();
} // Heart
```

Heart Server

The `HeartServer` is similar to the method discussed in Chapter 8, of uploading a complete implementation of the service. This service, of type `HeartImpl`, is primed with a URL identifying where the heart data is stored. An HTTP server will later deliver this data.

This implementation is enough to locate the service. However, rather than just getting anyone's heart data, a client may wish to search for a particular person's data. This can be done by adding a `Name` entry as additional information about the service. A server that exports the complete service, plus the entry information, is as follows:

```
package heart;

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;
import net.jini.core.entry.Entry;
import net.jini.lookup.entry.Name;

/**
 * HeartServer.java
 */

public class HeartServer implements DiscoveryListener,
                                            LeaseListener {

    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

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

        // 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 HeartServer() {

        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];

                ServiceItem item = new ServiceItem(null,
                                                   // new HeartImpl("file:/home/jan/
projects/jini/doc/heart/TECG3.ecg"),
                                                   new HeartImpl("http://
www.micromed.com.au/patfarmer/v2/patfhr.ecg"),
                                                new Entry[] {new Name("Pat Farmer")});
                ServiceRegistration reg = null;
                try {
                    reg = registrar.register(item, Lease.FOREVER);
                } catch(java.rmi.RemoteException e) {
                    System.err.println("Register exception: " + e.toString());
                    continue;
                }
                System.out.println("service registered");

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

        public void discarded(DiscoveryEvent evt) {

        }

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

    } // HeartServer
```

Heart Client

The client searches for a service implementing the `Heart` interface, with the additional requirement that it be for a particular person. Once it has this, the client just calls the `show()` method on the service to display this in some manner:

```
package heart;

import heart.Heart;

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.entry.Entry;
import net.jini.lookup.entry.Name;

/**
 * HeartClient.java
 */

public class HeartClient implements DiscoveryListener {

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

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(1000000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public HeartClient() {
        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[] {Heart.class};
            Entry [] entries = new Entry[] {new Name("Pat Farmer")};
            Heart heart = null;
            ServiceTemplate template = new ServiceTemplate(null, classes,
                                                           entries);

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

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

Heart Implementation

The HeartImpl class opens a connection to an HTTP server and requests delivery of a file. Heart data needs to be displayed at a reasonable rate, so it reads, draws, and sleeps, in a loop. It acts as a fat client to the HTTP server, displaying the data in a suitable format (in this case, it uses HTTP as a transport mechanism for data delivery). As a "client-aware" service, it customizes this delivery to the characteristics of the client platform, just occupying a "reasonable" amount of screen space and using local colors and fonts.

```
/**
 * HeartImpl.java
 */

package heart;

import java.io.*;
import java.net.*;
import java.awt.*;

public class HeartImpl implements Heart {

    protected String url;

    /*
     * If we want to run it standalone we can use this
     */
    public static void main(String argv[]) {

        HeartImpl impl =
            new HeartImpl("file:/home/jan/projects/jini/doc/heart/TECG3.ecg");
        impl.show();
    }

    public HeartImpl(String u) {
        url = u;
    }

    double[] points = null;
    Painter painter = null;

    String heartRate = "--";
```

```
public void setHeartRate(int rate) {
    if (rate > 20 && rate <= 250) {
        heartRate = "Heart Rate: " + rate;
    } else {
        heartRate = "Heart Rate: --";
    }
    // ? ask for repaint?
}

public void quit(Exception e, String s) {
    System.err.println(s);
    e.printStackTrace();
    System.exit(1);
}

public void show() {
    int SAMPLE_SIZE = 300 / Toolkit.getDefaultToolkit().
                                                getScreenResolution();
    Dimension size = Toolkit.getDefaultToolkit().
                             getScreenSize();
    int width = (int) size.getWidth();
    // capture points in an array, for redrawing in app if needed
    points = new double[width * SAMPLE_SIZE];
    for (int n = 0; n < width; n++) {
        points[n] = -1;
    }


    URL dataUrl = null;
    InputStream in = null;

    try {
        dataUrl = new URL(url);
        in = dataUrl.openStream();
    } catch (Exception ex) {
        quit(ex, "connecting to ECG server");
        return;
    }

    Frame frame = new Frame("Heart monitor");
    frame.setSize((int) size.getWidth()/2, (int) size.getHeight()/2);
    try {
        painter = new Painter(this, frame, in);
        painter.start();
```

```
        } catch (Exception ex) {
            quit(ex, "fetching data from ECG server");
            return;
        }
        frame.setVisible(true);
    }
} // HeartImpl

class Painter extends Thread {

    static final int DEFAULT_SLEEP_TIME = 25; // milliseconds
    static final int CLEAR_AHEAD = 15;
    static final int MAX = 255;
    static final int MIN = 0;
    final int READ_SIZE = 10;

    protected HeartImpl app;
    protected Frame frame;

    protected InputStream in;
    protected final int RESOLUTION = Toolkit.getDefaultToolkit().
                                                    getScreenResolution();
    protected final int UNITS_PER_INCH = 125;
    protected final int SAMPLE_SIZE = 300 / RESOLUTION;
    protected int sleepTime = DEFAULT_SLEEP_TIME;

    public Painter(HeartImpl app, Frame frame, InputStream in) throws Exception {
        this.app = app;
        this.frame = frame;
        this.in = in;
    }

    public void run() {

        while (!frame.isVisible()) {
           try {
               Thread.sleep(1000);
           } catch(Exception e) {
               // ignore
           }
        }

        int height = frame.getSize().height;
        int width = frame.getSize().width;
```

```
int x = 1; // start at 1 rather than 0 to avoid drawing initial line
           // from -128 .. 127
int magnitude;
int nread;
int max = MIN; // top bound of magnitude
int min = MAX;  // bottom bound of magnitude
int oldMax = MAX + 1;
byte[] data = new byte[READ_SIZE];
Graphics g = frame.getGraphics();
g.setColor(Color.red);
try {
    Font f = new Font("Serif", Font.BOLD, 20);
    g.setFont(f);
} catch (Exception ex) {
    // ....
}



try {
    boolean expectHR = false;   // true ==> next byte is heartrate

    while ((nread = in.read(data)) != -1) {
        for (int n = 0; n < nread; n++) {
            int thisByte = data[n] & 0xFF;
            if (expectHR) {
                expectHR = false;
                app.setHeartRate(thisByte);
                continue;
            } else if (thisByte == 255) {
                expectHR = true;
                continue;
            }

            // we are reading bytes, from -127..128
            // convert to unsigned
            magnitude = thisByte;

            // then convert to correct scale
            magnitude -= 128;
            // scale and convert to window coord from the top downwards
            int y = ((128 - magnitude) * RESOLUTION) / UNITS_PER_INCH;
            app.points[x] = y;
```

```
                    // draw only on multiples of sample size
                    if (x % SAMPLE_SIZE == 0) {
                        // delay to draw at a reasonable rate
                        Thread.sleep(sleepTime);

                        int x0 = x / SAMPLE_SIZE;
                        g.clearRect(x0, 0, CLEAR_AHEAD, height);
                        if (oldMax != MAX + 1) {
                            g.drawLine(x0-1, oldMax, x0, min);
                        }
                        g.drawLine(x0, min, x0, max);
                        oldMax = max;
                        min = 1000;
                        max = -1000;
                        if (app.heartRate != null) {
                            g.setColor(Color.black);
                            g.clearRect(0, 180, 200, 100);
                            g.drawString(app.heartRate, 0, 220);
                            g.setColor(Color.red);
                        }
                    } else {
                        if (y > max) max = y;
                        if (y < min) min = y;
                    }
                    if (++x >= width * SAMPLE_SIZE) {
                        x = 0;
                    }
                }
            }
        } catch(Exception ex) {
            if (! (ex instanceof SocketException)) {
                System.out.println("Applet quit -- got " + ex);
            }
        } finally {
            try {
                if (in != null) {
                    in.close();
                    in = null;
                }
            } catch (Exception ex) {
                // hide it
            }
        }
    }
}
```

Summary

Clients are built to make use of the services they find, but they do not need to be concerned with how the services are implemented. On the other hand, service implementers need to be aware of the choices they have in building services, and they need to choose the architecture that best suits the needs of the service. This chapter has looked at a number of possibilities and has used a simple running example to illustrate some of the possible design patterns.