CHAPTER 11

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 Chapter 9, we discussed the roles of the service proxy and service back-end, and briefly talked about how different implementations could place different amounts of processing in the proxy or back-end. This can lead to situations such as a thin proxy communicating to a fat back-end using RMI, or at the other end of the scale, to a fat proxy and a thin back-end. The last chapter showed one implementation: a fat proxy with a back-end 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 back-end on the service provider side. In RPC-like systems there is little choice: the proxy must be thin and the back-end 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 11-1.

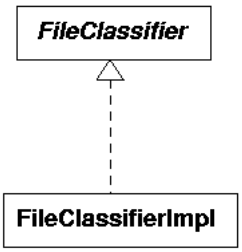


Figure 11-1. *Class diagram for file classifier*

You have already seen the full object diagram for the JVMs in Chapter 9, but just concentrating on the service and proxy classes looks like Figure 11-2.

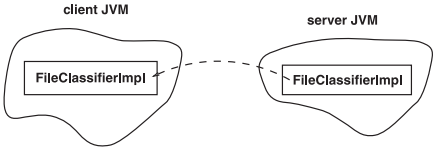


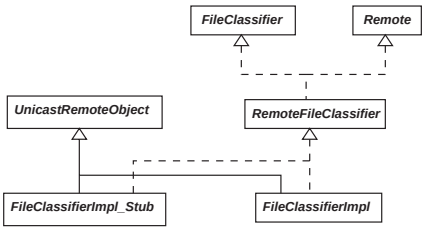
Figure 11-2. *Objects in the JVMs*

The client asks for a `FileClassifier`. What is uploaded to the service locators, and thus what the client gets, is a `FileClassifierImpl`. 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!).

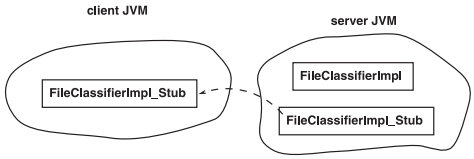
A class diagram for an implementation of the file classifier using this mechanism is shown in Figure 11-3.



FPO

Figure 11-3. *Class diagram for RMI proxy*

The objects in the JVMs are shown in Figure 11-4.



FPO

Figure 11-4. *JVM objects for RMI proxy*

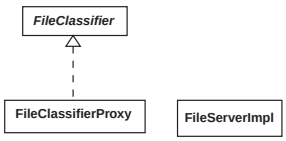
The full code for this mechanism is given later in the chapter in the "RMI Proxy for FileClassifier" section.

The class structure for this mechanism is more complex than the fat proxy because of RMI requirements. The `RemoteFileClassifier` interface is defined for convenience (the `FileClassifierImpl` could have implemented `FileClassifier` and `Remote` directly). Before Jeri, it was customary for the implementation to subclass from the `UnicastRemoteObject` class, but now it is recommended to use methods of an `Exporter` object (not shown in the figure). Implementing the `Remote` interface allows the proxy to be generated, which can call the methods of a `FileClassifierImpl` objects remotely.

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

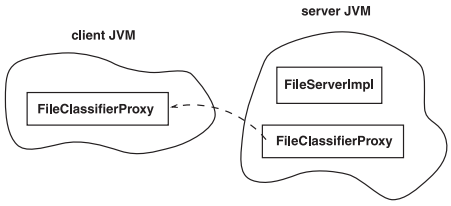
If RMI is not used, and the proxy and service want to share processing, then both the service and the proxy must be created explicitly on the server side. The proxy is explicitly registered with a lookup service, just as with an RMI proxy. The major differences are that the server creates the proxy and does not use an `exporter` for this, and that the proxy must implement the interface, but the service need not do so since the proxy and 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 11-5.



FPO

Figure 11-5. *Class diagram for a non-RMI proxy*

The JVMs at runtime for this scenario are shown in Figure 11-6.



FPO

Figure 11-6. *JVM objects for a non-RMI proxy*

Jini 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, then 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" versions 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 back-end service is the server part of the original client/server system. When changes occur, the back-end 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 back-end service, an explicit (smart) proxy, and an RMI proxy. The RMI proxy is created from the service using an `Exporter`. The smart proxy is created by the server and will typically be told about the RMI proxy in its constructor. The smart proxy is registered on a lookup service.

The RMI proxy can be used as an RPC-like communication mechanism between the smart proxy and the service. This is just like the last case, but instead of requiring the smart proxy and service to implement their own communication protocol, the smart proxy calls methods on the local RMI proxy, which uses RMI protocols to talk across the network to the service. The smart proxy and service can be of any relative size, just like in the last case. This simplifies the task of the programmer, as no distributed protocol needs to be devised and implemented.

Later in the chapter, in the "RMI and Non-RMI Proxies for FileClassifier" section, is a non-RMI proxy, `FileClassifierProxy`, that implements the `FileClassifier` interface. The proxy 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 11-7.

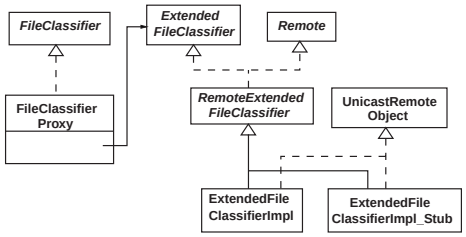


Figure 11-7. *Class diagram for RMI and non-RMI proxies*

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 11-8.

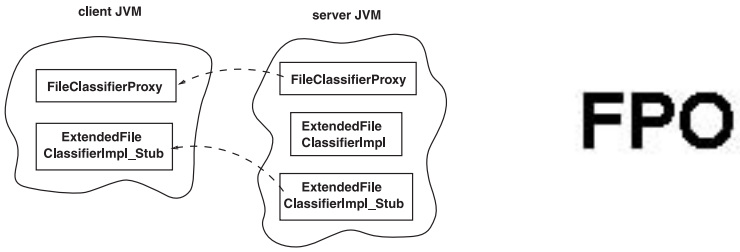


Figure 11-8. *JVM objects for RMI and non-RMI proxies*

RMI Proxy for FileClassifier

An RMI proxy can be used when all of the work performed by the service is done on the server side. In this case, the server makes available 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:

- Implement `Remote`. This is because methods will be called on the service from the proxy, and these will be remote calls on the service.
- Locate and use an `Exporter` object to create an RMI proxy.

What Doesn't Change

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 it 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. And the `MIMEType` doesn't change, because 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 back-end service. When the back-end service exports an RMI proxy, the service will look like this:

```
package rmi;
import common.MIMEType;
import common.FileClassifier;
/**
 * FileClassifierImpl.java
 */
public class FileClassifierImpl 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 server changes by first getting an `Exporter` object and using this to create a proxy. This proxy implements `RemoteFileClassifier` as shown by the class cast, but it is only necessary for it to be a `Remote` object.

```
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 net.jini.lease.LeaseRenewalManager;
import net.jini.lease.LeaseListener;
import net.jini.lease.LeaseRenewalEvent;
import java.rmi.RMISecurityManager;
import net.jini.config.*;
import net.jini.export.*;
/**
 * FileClassifierServerRMI.java
 */
public class FileClassifierServerRMI implements DiscoveryListener, LeaseListener {
    protected FileClassifierImpl impl;
    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();
    // explicit proxy for Jini 2.0
    protected RemoteFileClassifier proxy;
```

```
    private static String CONFIG_FILE = "jeri/file_classifier_server.config";

    public static void main(String argv[]) {
        new FileClassifierServerRMI();
        Object keepAlive = new Object();
        synchronized(keepAlive) {
            try {
                keepAlive.wait();
            } catch(java.lang.InterruptedException e) {
                // do nothing
            }
        }
    }
    public FileClassifierServerRMI() {
        try {
            impl = new FileClassifierImpl();
        } catch(Exception e) {
            System.err.println("New impl: " + e.toString());
            System.exit(1);
        }
        String[] configArgs = new String[] {CONFIG_FILE};
        try {
            // get the configuration (by default a FileConfiguration)
            Configuration config = ConfigurationProvider.getInstance(configArgs);

            // and use this to construct an exporter
            Exporter exporter = (Exporter) config.getEntry( "FileClassifierServer",
                                                            "exporter",
                                                            Exporter.class);
            // export an object of this class
            proxy = (RemoteFileClassifier) exporter.export(impl);
        } catch(Exception e) {
            System.err.println(e.toString());
            e.printStackTrace();
            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 - use the actual proxy in 2.0
            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());
    }

} // FileClassifierServerRMI
```

The server makes use of a configuration provider to locate a `Configuration` object and hence an `Exporter`. As before, the default `Configuration` object is a `FileConfiguration` that uses a configuration file (here given as `jeri/file_classifier_server.config`). For Jeri, the contents of this file are as follows:

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
FileClassifierServer {
    exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                     new BasicILFactory());
}
```

If instead the older Java Remote Method Protocol (JRMP) version of RMI is used, the configuration file would look like this:

```
import net.jini.jrmp.*;
FileClassifierServer {
    exporter = new JrmpExporter();
}
```

Jeri: What Classes Need to Be Where?

Using the new Jini extensible remote invocation (ERI), we have the following classes:

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

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 client `TestFileClassifier`

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 the older JRMP-style RMI, one of the main functions of the HTTP server was to download the `rmic`-generated RMI stub. Using Jeri, this is no longer needed. Does this mean the HTTP server isn't necessary anymore? Regrettably, no. While the client knows the "commonly known" interfaces and gets the proxy, there is the `RemoteFileClassifier` interface, which so far is only known on the server side. In order for the client to be able to unmarshall the proxy, it needs to get this interface from the HTTP server. Thus, the HTTP server needs to be able to access the following:

- The `rmi.RemoteFileClassifier` interface

The proxy contains the value of the `java.rmi.server.codebase` set by the server. Jini code running in the client examines this and uses it to determine the URL of class files that it needs to download. In general, URLs can be *file* or *http* references. But for this case, the URL will be used by clients running anywhere, 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/RemoteFileClassifier.class
```

An Ant file to build and deploy server files where the service uses Jeri (or any protocol that generates its own proxies at runtime) is as follows:

```
<!--
     Project name must be the same as the file name, which must
     be the same as the main.class. Builds jar files with the
     same name.
  -->

<project name="rmi.FileClassifierServerRMI">
    <!-- Inherits properties from ../build.xml:
         jini.home
         jini.jars
         src
         dist
         build
         httpd.classes
      -->
    <!-- Files for this project -->
    <!-- Source files for the server -->
    <property name="src.files"
              value="
                     common/MIMEType.java,
                     common/FileClassifier.java,
```

```
                    rmi/RemoteFileClassifier.java,
                    rmi/FileClassifierImpl.java,
                    rmi/FileClassifierServerRMI.java
                   "/>
    <!-- Class files to run the server -->
    <property name="class.files"
              value="
                    common/MIMEType.class,
                    common/FileClassifier.class,
                    rmi/RemoteFileClassifier.class,
                    rmi/FileClassifierImpl.class,
                    rmi/FileClassifierServerRMI.class
                   "/>
    <!-- Class files for the client to download -->
    <property name="class.files.dl"
              value="
                    rmi/RemoteFileClassifier.class,
                   "/>
    <!-- Uncomment if no class files downloaded to the client -->
    <!-- <property name="no-dl" value="true"/> -->
    <!-- derived names - may be changed -->
    <property name="jar.file"
              value="${ant.project.name}.jar"/>
    <property name="jar.file.dl"
              value="${ant.project.name}-dl.jar"/>
    <property name="main.class"
              value="${ant.project.name}"/>
   <property name="codebase"
              value="http://${localhost}/classes/${jar.file.dl}"/>
    <!-- targets -->
    <target name="all" depends="compile"/>
    <target name="compile">
        <javac destdir="${build}" srcdir="${src}"
               classpath="${jini.jars}"
               includes="${src.files}">
        </javac>
    </target>
    <target name="dist" depends="compile"
            description="generate the distribution">
        <jar jarfile="${dist}/${jar.file}"
             basedir="${build}"
             includes="${class.files}"/>
        <antcall target="dist-jar-dl"/>
    </target>
    <target name="dist-jar-dl" unless="no-dl">
        <jar jarfile="${dist}/${jar.file.dl}"
             basedir="${build}"
```

```
            includes="${class.files.dl}"/>
    </target>
    <target name="build" depends="dist,compile"/>
    <target name="run" depends="deploy">
        <java classname="${main.class}"
              fork="true"
              classpath="${jini.jars}:${dist}/${jar.file}">
             <jvmarg value="-Djava.rmi.server.codebase=${codebase}"/>
             <jvmarg value="-Djava.security.policy=${res}/policy.all"/>
        </java>
    </target>
    <target name="deploy" depends="dist" unless="no-dl">
        <copy file="${dist}/${jar.file.dl}"
              todir="${httpd.classes}"/>
    </target>
</project>
```

JRMP: What Classes Need to Be Where?

This section discusses Jini's use of RMI as it used to be before Jeri and shows what had to be done in Jini 1.2 and earlier. It is recommended that you not use this procedure any more.

We have the following 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 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 client `TestFileClassifier`

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 reason for all of these classes and interfaces is slightly complex. In the `FileClassifierProxy` constructor, the class `FileClassifierImpl` is passed in. The RMI runtime converts this to `FileClassifierImpl_Stub`. This class implements the same interfaces as `FileClassifierImpl`—that is, `RemoteFileClassifier` also needs to be available.

So, what does the term "available" mean in the last paragraph? The client will look for files based on the `java.rmi.server.codebase` property of the application server. The value of this property is a URL. In general, URLs can be file or http references. But in this case, the URL will be used by clients running anywhere, 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

Again, we have a server and a client to run. The client is called in the same way as in the previous chapter, using the server in the `complete` package, since the client is independent of any server implementation.

```
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/rmi/FileClassifierImpl_Stub.class`, the codebase would be specified as follows:

```
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 start-ups such as this:

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

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

The proxy will then use this reply to return either `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 on an agreed-upon TCP port to the service 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 listen for connections. When one is received, `FileServerImpl` will create a new `Connection` object also 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();
        }
    }
}
```

Server

The Jini server must start a `FileServerImpl` to listen for later connections. Then it can register a `FileClassifierProxy` proxy object with each lookup service, which will send it on to interested clients. The proxy object must know where the service is listening in order to attempt a connection to it, and this information is given by first making a query for the localhost 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 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++) {
                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`

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 client `TestFileClassifier`

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 `common.MIMEType` class
- The `socket.FileClassifierServer` class
- The `socket.FileClassifierProxy` class
- The `socket.FileServerImpl` class
- The `socket.Connection` class (a class in `FileServerImpl.java`)

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 Non-RMI Proxy FileClassifier

Again, we have a server and a client to run. 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. If the class files are stored on my web server's pages under `classes/socket/FileClassifierProxy.class`, the codebase would be specified as follows:

```
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 start-ups such as this:

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

An Ant file to build and deploy this project is `socket.FileClassifierServer.xml`:

```
<!--
     Project name must be the same as the file name, which must
     be the same as the main.class. Builds jar files with the
     same name.
  -->

<project name="socket.FileClassifierServer">
    <!-- Inherits properties from ../build.xml:
         jini.home
         jini.jars
         src
         dist
         build
         httpd.classes
      -->
    <!-- Files for this project -->
    <!-- Source files for the server -->
    <property name="src.files"
              value="
                     common/MIMEType.java,
                     common/FileClassifier.java,
                     socket/FileClassifierProxy.java,
                     socket/FileServerImpl.java,
                     socket/FileClassifierServer.java,
```

```
                "/>
<!-- Class files to run the server -->
<property name="class.files"
          value="
                 common/MIMEType.class,
                 common/FileClassifier.class,
                 socket/FileClassifierProxy.class,
                 socket/FileServerImpl.class,
                 socket/FileClassifierServer.class,
                 socket/Connection.class
                "/>
<!-- Class files for the client to download -->
<property name="class.files.dl"
          value="
                 socket/FileClassifierProxy.class,
                 "/>
<!-- Uncomment if no class files downloaded to the client -->
<!-- <property name="no-dl" value="true"/> -->
<!-- derived names - may be changed -->
<property name="jar.file"
          value="${ant.project.name}.jar"/>
<property name="jar.file.dl"
          value="${ant.project.name}-dl.jar"/>
<property name="main.class"
          value="${ant.project.name}"/>
<property name="codebase"
          value="http://${localhost}/classes/${jar.file.dl}"/>
<!-- targets -->
<target name="all" depends="compile"/>
<target name="compile">
    <javac destdir="${build}" srcdir="${src}"
           classpath="${jini.jars}"
           includes="${src.files}">
    </javac>
</target>
<target name="dist" depends="compile"
        description="generate the distribution">
    <jar jarfile="${dist}/${jar.file}"
         basedir="${build}"
         includes="${class.files}"/>
    <antcall target="dist-jar-dl"/>
</target>
<target name="dist-jar-dl" unless="no-dl">
    <jar jarfile="${dist}/${jar.file.dl}"
         basedir="${build}"
         includes="${class.files.dl}"/>
</target>
```

```
    <target name="build" depends="dist,compile"/>
    <target name="run" depends="deploy">
        <java classname="${main.class}"
              fork="true"
              classpath="${jini.jars}:${dist}/${jar.file}">
             <jvmarg value="-Djava.rmi.server.codebase=${codebase}"/>
             <jvmarg value="-Djava.security.policy=${res}/policy.all"/>
        </java>
    </target>
    <target name="deploy" depends="dist" unless="no-dl">
        <copy file="${dist}/${jar.file.dl}"
              todir="${httpd.classes}"/>
    </target>
</project>
```

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.

Some file types are more common than others—GIF, DOC, and HTML files abound, but there are many more file 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. This `FileClassifierProxy` class is as follows:

```
/**
 * FileClassifierProxy.java
 */
package extended;
import common.FileClassifier;
import common.ExtendedFileClassifier;
import common.MIMEType;
import java.rmi.RemoteException;
import java.rmi.Remote;
public class FileClassifierProxy implements FileClassifier, java.io.Serializable {
```

```
    /**
     * The service object that knows lots more MIME types
     */
    protected RemoteExtendedFileClassifier extension;
    public FileClassifierProxy(Remote ext) {
        this.extension = (RemoteExtendedFileClassifier) 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, Re-
mote {
} // RemoteExtendedFileClassifier
```

ExtendedFileClassifierImpl

The implementation of the `ExtendedFileClassifier` interface is done by an `ExtendedFileClassifierImpl` object. Since this object may handle requests from many proxies, an alternative implementation of searching for MIME types that is more efficient for repeated searches is used:

```
/**
 * ExtendedFileClassifierImpl.java
 */
package extended;
import java.rmi.server.UnicastRemoteObject;
import common.MIMEType;
import java.util.HashMap;
import java.util.Map;
public class ExtendedFileClassifierImpl
    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 net.jini.lease.LeaseRenewalManager;
import net.jini.lease.LeaseListener;
import net.jini.lease.LeaseRenewalEvent;
import java.rmi.RMISecurityManager;
import java.rmi.Remote;
import net.jini.config.*;
import net.jini.export.*;
import rmi.RemoteFileClassifier;
/**
 * FileClassifierServer.java
 */
public class FileClassifierServer implements DiscoveryListener, LeaseListener {
    protected FileClassifierProxy smartProxy;
    protected Remote rmiProxy;
    protected ExtendedFileClassifierImpl impl;
    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();
    private static String CONFIG_FILE = "jeri/file_classifier_server.config";

    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);
        }
        String[] configArgs = new String[] {CONFIG_FILE};
        try {
            // get the configuration (by default a FileConfiguration)
            Configuration config = ConfigurationProvider.getInstance(configArgs);

            // and use this to construct an exporter
            Exporter exporter = (Exporter) config.getEntry( "FileClassifierServer",
                                                            "exporter",
                                                            Exporter.class);
            // export an object of this class
            rmiProxy = (RemoteFileClassifier) exporter.export(impl);
        } catch(Exception e) {
            System.err.println(e.toString());
            e.printStackTrace();
            System.exit(1);
        }
        // set RMI security manager
        System.setSecurityManager(new RMISecurityManager());
        // proxy primed with impl
        smartProxy = new FileClassifierProxy(rmiProxy);
        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }
        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {
        ServiceRegistrar[] registrars = evt.getRegistrars();
        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];
            // export the proxy service
            ServiceItem item = new ServiceItem(null,
                                               smartProxy,
                                               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`
- `rmi.RemoteFileClassifier`
- `extended.FileClassifierProxy`
- `extended.RemoteExtendedFileClassifier`
- The class `extended.ExtendedFileClassifierImpl`
- `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 `rmi.RemoteFileClassifier` class
- The `extended.FileClassifierServer` class
- The `extended.FileClassifierProxy` class
- The `extended.RemoteExtendedFileClassifier` class
- The `extended.ExtendedFileClassifierImpl` 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. This HTTP codebase must have all the files related to an exported object except for those classes the client already has (they would be redundant). So it needs to be able to access the following:

- The `rmi.RemoteFileClassifier` class
- The `common.ExtendedFileClassifier` interface
- The `extended.FileClassifierProxy` interface
- The `extended.RemoteExtendedFileClassifier` class

An Ant file to build and deploy these is `extended.FileClassifierServer.xml`:

```
<!--
     Project name must be the same as the file name, which must
     be the same as the main.class. Builds jar files with the
     same name.
  -->

<project name="extended.FileClassifierServer">
    <!-- Inherits properties from ../build.xml:
         jini.home
         jini.jars
         src
         dist
         build
         httpd.classes
      -->
    <!-- Files for this project -->
    <!-- Source files for the server -->
    <property name="src.files"
              value="
```

```
                common/MIMEType.java,
                common/FileClassifier.java,
                common/ExtendedFileClassifier.java,
                rmi/RemoteFileClasifier.java,
                extended/RemoteExtendedFileClassifier.java,
                extended/FileClassifierProxy.java,
                extended/ExtendedFileClassifierImpl.java,
                extended/FileClassifierServer.java
               "/>
<!-- Class files to run the server -->
<property name="class.files"
          value="
                common/MIMEType.class,
                common/FileClassifier.class,
                common/ExtendedFileClassifier.class,
                rmi/RemoteFileClassifier.class,
                extended/RemoteExtendedFileClassifier.class,
                extended/FileClassifierProxy.class,
                extended/FileClassifierServer.class,
                extended/ExtendedFileClassifierImpl.class,
               "/>
<!-- Class files for the client to download -->
<property name="class.files.dl"
          value="
                rmi/RemoteFileClassifier.class,
                common/ExtendedFileClassifier.class,
                extended/FileClassifierProxy.class,
                extended/RemoteExtendedFileClassifier.class,
               "/>
<!-- Uncomment if no class files downloaded to the client -->
<!-- <property name="no-dl" value="true"/> -->
<!-- derived names - may be changed -->
<property name="jar.file"
          value="${ant.project.name}.jar"/>
<property name="jar.file.dl"
          value="${ant.project.name}-dl.jar"/>
<property name="main.class"
          value="${ant.project.name}"/>
<property name="codebase"
          value="http://${localhost}/classes/${jar.file.dl}"/>
<!-- targets -->
<target name="all" depends="compile"/>
<target name="compile">
    <javac destdir="${build}" srcdir="${src}"
           classpath="${jini.jars}"
           includes="${src.files}">
    </javac>
```

```
    </target>
    <target name="dist" depends="compile"
            description="generate the distribution">
        <jar jarfile="${dist}/${jar.file}"
             basedir="${build}"
             includes="${class.files}"/>
        <antcall target="dist-jar-dl"/>
    </target>
    <target name="dist-jar-dl" unless="no-dl">
        <jar jarfile="${dist}/${jar.file.dl}"
             basedir="${build}"
             includes="${class.files.dl}"/>
    </target>
    <target name="build" depends="dist,compile"/>
    <target name="run" depends="deploy">
        <java classname="${main.class}"
              fork="true"
              classpath="${jini.jars}:${dist}/${jar.file}">
             <jvmarg value="-Djava.rmi.server.codebase=${codebase}"/>
             <jvmarg value="-Djava.security.policy=${res}/policy.all"/>
        </java>
    </target>
    <target name="deploy" depends="dist" unless="no-dl">
        <copy file="${dist}/${jar.file.dl}"
              todir="${httpd.classes}"/>
    </target>
</project>
```

Summary

Clients don't care how services are implemented; they just want a service that implements the service specification. But service authors have a large variety of choices about where the service runs, and how a proxy communicates with a back-end service. This chapter has considered some of the alternatives and discussed details of which classes will be needed where.