CHAPTER 8

A Simple Example

THIS CHAPTER LOOKS AT A SIMPLE PROBLEM to give a complete example of a Jini service and client.

Before a Jini service can be built, common knowledge must be defined about the type of service that will be offered. This involves designing a set of "well-known" classes and interfaces. Based on a well-known interface, a client can be written to search for and use services implementing the interface.

The client can use either a unicast or multicast search to find services, but it will be uninterested in how any particular service is implemented. This chapter looks at building clients using both methods, and these clients will be heavily reused throughout the rest of the book.

The service, on the other hand, is implemented by each vendor in a different way. This chapter discusses a simple choice, with alternatives being dealt with in the next chapter. It is difficult to get a Jini service and client functioning correctly, as there are many configuration issues to be dealt with. These are discussed in some detail.

By the end of this chapter you should be able to build a client and a service, and configure your system so that they are able to run and communicate with each other.

Problem Description

Applications often need to work out the type of a file to see if it is a text file, an HTML document, an executable, etc. This can be done in two ways:

- By examining the file's name
- By examining the file's contents

Utilities such as the Unix `file` command use the second method and have a complex description file (such as `/etc/magic` or `/usr/share/magic`) to aid in this. Many other applications, such as Web browsers, mail readers, and even some operating systems, use the first method and work out a file's type based on its name.

A common way of classifying files is into MIME types, such as `text/plain` and `image/gif`. There are tables of "official" MIME types (unofficial ones can be added on an ad hoc basis), and there are also tables of mappings from filename endings to corresponding MIME types. These tables have entries such as these:

```
application/postscript         ai eps ps
application/rtf                rtf
application/zip                zip
image/gif                      gif
image/jpeg                     jpeg jpg jpe
text/html                      html htm
text/plain                     txt
```

These tables are stored in files for applications to access.

Storing these tables separately from the applications that would use them is considered bad from the object-oriented point of view, since each application would need to have code to interpret the tables. Also, the multiplicity of these tables and the ability of users to modify them makes this a maintenance problem. It would be better to encapsulate at least the filename to MIME type mapping table in an object.

We could define a MIME class as follows:

```
package standalone;

/**
 * MIMEType.java
 */

public class MIMEType  {

    /**
     * A MIME type is made up of 2 parts
     * contentType/subtype
     */
    protected String contentType;
    protected String subtype;

    public MIMEType(String type) {
        int slash = type.indexOf('/');
        contentType = type.substring(0, slash-1);
        subtype = type.substring(slash+1, type.length());
    }

    public MIMEType(String contentType, String subtype) {
        this.contentType = contentType;
```

```
        this.subtype = subtype;
    }

    public String toString() {
        return contentType + "/" + subtype;
    }
} // MIMEType
```

We could then define a mapping class like this:

```
package standalone;

/**
 * FileClassifier.java
 */

public class FileClassifier  {

    static MIMEType getMIMEType(String 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 null;
    }
} // FileClassifier
```

This mapping class has no constructors, because it just acts as a lookup table via its static method `getMIMEType()`.

Applications can make use of these classes as they stand, by simply compiling them and having the class files available at run time. This would still result in duplication throughout JVMs, possible multiple copies of the class files, and potentially severe maintenance problems if applications need to be recompiled, so it might be better to have the `FileClassifier` as a network service. Let's consider what would be involved in this.

Service Specification

If we wish to make a version of `FileClassifier` available across the network, there are a number of possibilities. The client will be asking for an instance of a class, and generally will not care too much about the details of this instance. It will want an object that belongs to the `FileClassifierClass` or one of its subclasses and will not usually care which of these it gets, as long as it contains the method `getMIMEType()`.

Services will have particular implementations, and will upload these to the service locators. The uploaded service will be of a specific class and may have associated entries.

There are several options that the client could use in trying to locate a suitable service:

1. This is the silly option: push the entire implementation up to the lookup service and make the client ask for it by its class. Then the client might just as well create the classifier as a local object, because it has all the information needed! This doesn't lend itself to flexibility with new unknown services coming along, because the client already has to know the details. So this option is not feasible.

2. Let the client ask for a superclass of the service. This is better, as it allows new implementations of a service to just be implemented as new subclasses. It is not ideal, as classes have implementation code, and if this changes over time, there is a maintenance issue with the possibility of version "skew." This can be used for Jini; it just isn't the best way.

3. Separate the interface completely from the implementation. Make the interface available to the client, and upload the implementation to the lookup service. Then, when the client asks for an instance object that implements the interface, it will get an object for this interface, which implements the interface in some way or other. The client generally will not care how the object does this. This will reduce maintenance: if the client is coded just in terms of the interface, then it will not need recompilation even if the implementation changes. Note that these words will translate straight into Java terms; the client knows about a Java `interface`, whereas the service provider deals in terms of a Java class that implements the `interface`.

The ideal mechanism in the Jini world is to specify services by interfaces and have all clients know this interface. Then each service can be an implementation of this interface. This is simple in Java terms, simple in specification terms, and simple for maintenance. This is not the complete set of choices for the service, but it is enough to allow a service to be specified and to get on with building the client.

One possibility for service implementation is looked at later in this chapter, and the next chapter is devoted to the full range of possibilities.

Common Classes

The client and any implementations of a service must share some common classes. For a file-classification service, the common classes are the classifier itself (which can be implemented as many different services) and the return value, the MIME type. These have to change very slightly from their standalone form.

MIMEType

The `MIMEType` class is known to the client and to any file-classifier service. The `MIMEType` class file can be expected to be known to the JVMs of all clients and services. That is, this class file needs to be in the `CLASSPATH` of every file-classifier service and of every client that wants to use a file-classifier service.

The `getMIMEType()` method will return an object from the file-classifier service. There are implementation possibilities that can affect this object:

- If the service runs in the client's JVM, then nothing special needs to be done.
- If the service is implemented remotely and runs in a separate JVM, then the `MIMEType` object must be serialized for transport to the client's JVM. For this to be possible, it must implement the `Serializable` interface. Note that while the class files are accessible to both client and service, the instance data of the `MIMEType` object needs to be serializable to move the object from one machine to the other.

There can be differences in the object depending on the implementation. If it implements `Serializable`, it can be used in both the remote and local cases, but if it doesn't, it can only be used in the local case.

Making decisions about interfaces based on future implementation concerns is traditionally rated as poor design. In particular, the philosophy behind remote procedure calls is that they should hide the network as much as possible and make the calls behave as though they were local calls. With this philosophy, there is no need to make a distinction between local and remote calls at design time. However, a document from Sun, "A Note on Distributed Computing" by Jim Waldo and others argues that this is wrong, particularly in the case of distributed objects. The basis of their argument is that the network brings in a host of other factors, in particular that of partial failure. That is, part of the network, itself, may fail, or a component on the network may fail without all of the network or all of the components failing. If other components do not make allowance for this possible (or maybe even likely) behavior, then the system as a whole will not be robust and could be brought down by the failure of a single component.

According to this document, it is important to determine whether the objects could be running remotely and to adjust interfaces and classes accordingly at the design stage. This lets you to take into account possible extra failure modes of methods, and in this case, an extra requirement on the object. This important paper is reprinted in the Jini specification book from Sun (*The Jini Specification* by Ken Arnold and others) and is also at `http://www.sun.com/research/techrep/1994/abstract_29.html`.

These considerations lead to an interface that adds the `Serializable` interface to the original version of the `MIMEType` class, as objects of this class could be sent across the network.

```
package common;

import java.io.Serializable;

/**
 * MIMEType.java
 */

public class MIMEType implements Serializable {

    /**
     * A MIME type is made up of 2 parts
     * contentType/subtype
     */
    protected String contentType;
    protected String subtype;

    public MIMEType(String type) {
        int slash = type.indexOf('/');
        contentType = type.substring(0, slash-1);
        subtype = type.substring(slash+1, type.length());
    }

    public MIMEType(String contentType, String subtype) {
        this.contentType = contentType;
        this.subtype = subtype;
    }

    public String toString() {
        return contentType + "/" + subtype;
    }
} // MIMEType
```

FileClassifier Interface

Changes have to be made to the file-classifier interface, as well. First, interfaces cannot have static methods, so we will have to turn the `getMIMEType()` method into a public instance method.

In addition, all methods are defined to throw a `java.rmi.RemoteException`. This type of exception is used throughout Java (not just by the RMI component) to mean "a network error has occurred." This could be a lost connection, a missing server, a class not downloadable, etc. There is a little subtlety here, related to the `java.rmi.Remote` class: the methods of `Remote` must all throw a `RemoteException`, but a class is not required to be `Remote` if its methods throw `RemoteExceptions`. If all the methods of a class throw `RemoteException`, it does not mean the class implements or extends `Remote`. It only means that an implementation may be implemented as a remote (distributed) object, and that an implementation might also use the RMI `Remote` interface.

There are some very fine points to this, which you can skip if you want. Basically, though, you can't go wrong if every method of a Jini interface throws `RemoteException` and the interface does not extend `Remote`. In fact, prior to JDK 1.2.2, making the interface extend `Remote` would force each implementation of the interface to actually be a remote object. At JDK 1.2.2, however, the semantics of `Remote` were changed a little, and this requirement was relaxed. From JDK 1.2.2 onwards, an interface can extend `Remote` without implementation consequences. At least, that is almost the case: "unusual" ways of implementing RMI, such as over IIOP (IIOP is the transport protocol for CORBA, and RMI can use this), have not yet caught up to this. So for maximum flexibility, just throw `RemoteException` from each method and don't extend `Remote`.

Doing so gives the following interface:

```
package common;

/**
 * FileClassifier.java
 */

public interface FileClassifier {

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

} // FileClasssifier
```

Why does this interface throw a `java.rmi.RemoteException` in the `getMIMEType()` method? Well, an interface is supposed to be above all possible implementations and should never change. The implementation discussed later in this chapter does not throw such an exception. However, other implementations in other sections use a `Remote` implementation, and this will require that the method throws a `java.rmi.RemoteException`. Since it is not possible to just add a new exception in a subclass or interface implementation, the possibility must be added in the interface specification.

The Client

The client is the same for all of the possible server implementations discussed throughout this book. The client does not care how the service implementation is done, just as long as it gets a service that it wants, and it specifies this by asking for a `FileClassifier` interface.

Unicast Client

If there is a known service locator that will know about the service, then there is no need to search for the service locator. This doesn't mean that the location of the service is known, only the location of the locator. For example, there might be a (fictitious) organization "All About Files" at `www.all_about_files.com` that would know about various file services, keeping track of them as they come online, move, disappear, etc. A client would ask the service locator running on this site for the service, wherever it is. This uses the unicast lookup techniques:

```
package client;

import common.FileClassifier;
import common.MIMEType;

import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import java.rmi.RMISecurityManager;
import net.jini.core.lookup.ServiceTemplate;

/**
 * TestUnicastFileClassifier.java
 */

public class TestUnicastFileClassifier {
```

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

public TestUnicastFileClassifier() {
    LookupLocator lookup = null;
    ServiceRegistrar registrar = null;
    FileClassifier classifier = null;

    try {
        lookup = new LookupLocator("jini://www.all_about_files.com");
    } catch(java.net.MalformedURLException e) {
        System.err.println("Lookup failed: " + e.toString());
        System.exit(1);
    }

    System.setSecurityManager(new RMISecurityManager());

    try {
        registrar = lookup.getRegistrar();
    } catch (java.io.IOException e) {
        System.err.println("Registrar search failed: " + e.toString());
        System.exit(1);
    } catch (java.lang.ClassNotFoundException e) {
        System.err.println("Registrar search failed: " + e.toString());
        System.exit(1);
    }

    Class[] classes = new Class[] {FileClassifier.class};
    ServiceTemplate template = new ServiceTemplate(null, classes, null);
    try {
        classifier = (FileClassifier) registrar.lookup(template);
    } catch(java.rmi.RemoteException e) {
        e.printStackTrace();
        System.exit(1);
    }

    if (classifier == null) {
        System.out.println("Classifier null");
        System.exit(2);
    }
    MIMEType type;
    try {
```

```
            type = classifier.getMIMEType("file1.txt");
            System.out.println("Type is " + type.toString());
        } catch(java.rmi.RemoteException e) {
            System.err.println(e.toString());
        }
        System.exit(0);
    }
} // TestUnicastFileClassifier
```

The client's JVM is illustrated in Figure 8-1. This shows a UML class diagram, surrounded by the JVM in which the objects exist.

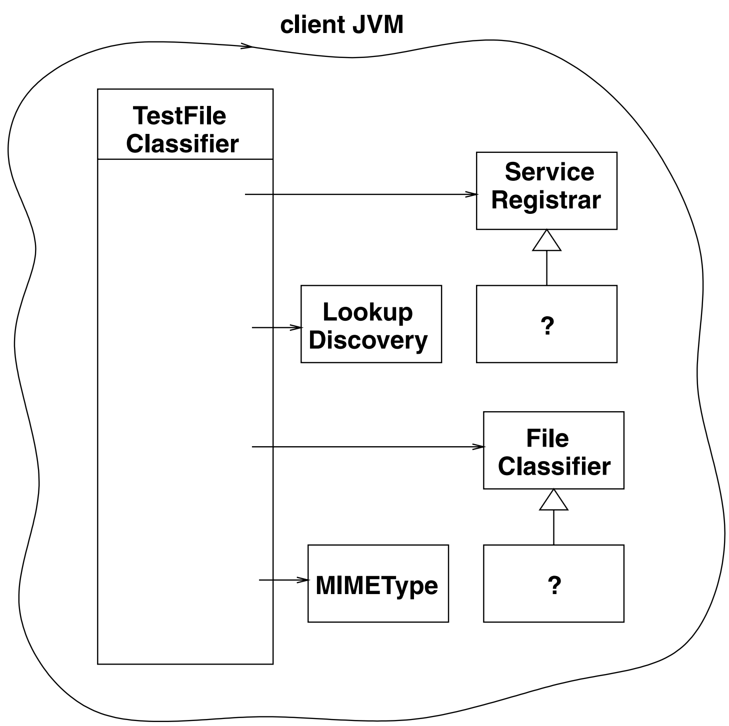

Figure 8-1. Objects in client JVM

The client has a main `TestFileClassifier` class, which has two objects of types `LookupDiscovery` and `MIMEType`. It also has objects that implement the interfaces `ServiceRegistrar` and `FileClassifier`, but it doesn't know, or need to know, what classes they are. These objects have come across the network as implementation objects of the two interfaces.

Figure 8-2 shows the situation when the service locator's JVM is added in. The lookup service has an object implementing `ServiceRegistrar`, and this is the object exported to the client.

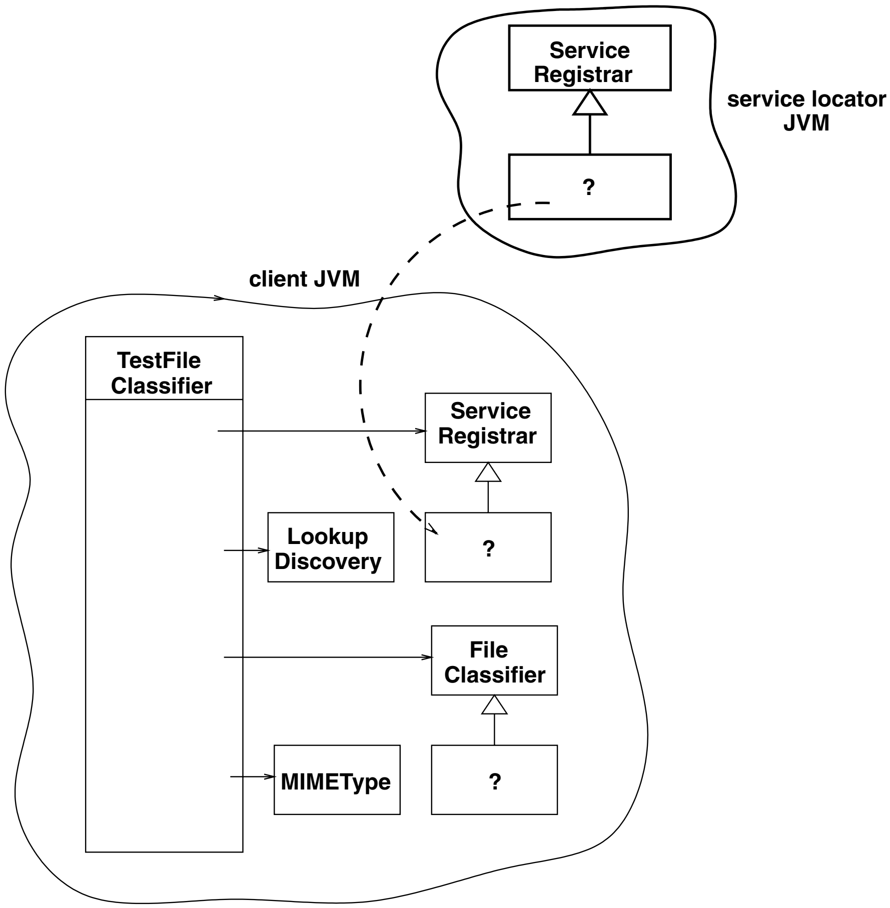

Figure 8-2. Objects in the client and service locator JVMs

This figure shows that the client gets its `registrar` from the JVM of the service locator. This `registrar` object is not specified in detail. Sun supplies a service locator known as `reggie`, which implements the `ServiceRegistrar` using an implementation that neither clients nor services are expected to know. The classes that implement the `ServiceRegistrar` object are contained in the `reggie-dl.jar` file and are downloaded to the clients and services using (typically) an HTTP server.

The figure also shows a question mark for the object in the client implementing `FileClassifier`. The source of this object is not yet shown; it will get the object from a service, but we haven't yet discussed any of the possible implementations of a FileClassifier service. One implementation will be discussed in the "Uploading a Complete Service" section later in this chapter, and others will be discussed in Chapter 9.

Multicast Client

We have looked at the unicast client, where the location of the service locator is already known. However, it is more likely that a client will need to search for service locators until it finds one holding a service it is looking for. It would need to use a multicast search for this. If it only needs one occurrence of the service, then it can exit after using the service. More complex behavior will be illustrated in later examples.

In this situation, the client does not need to have long-term persistence, but it does need a user thread to remain in existence for long enough to find service locators and find a suitable service. Therefore, in main() a user thread sleeps for a short period (ten seconds).

```
package client;

import common.FileClassifier;
import common.MIMEType;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;

/**
 * TestFileClassifier.java
 */

public class TestFileClassifier implements DiscoveryListener {

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

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(100000L);
        } catch(java.lang.InterruptedException e) {
```

```
            // do nothing
        }
    }

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

        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            System.exit(1);
        }

        discover.addDiscoveryListener(this);

    }

    public void discovered(DiscoveryEvent evt) {

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

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

            // Use the service to classify a few file types
            MIMEType type;
            try {
```

```
                String fileName;

                fileName = "file1.txt";
                type = classifier.getMIMEType(fileName);
                printType(fileName, type);

                fileName = "file2.rtf";
                type = classifier.getMIMEType(fileName);
                printType(fileName, type);

                fileName = "file3.abc";
                type = classifier.getMIMEType(fileName);
                printType(fileName, type);
            } catch(java.rmi.RemoteException e) {
                System.err.println(e.toString());
                continue;
            }
            // success
            System.exit(0);
        }
    }

    private void printType(String fileName, MIMEType type) {
        System.out.print("Type of " + fileName + " is ");
        if (type == null) {
            System.out.println("null");
        } else {
            System.out.println(type.toString());
        }
    }

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

Exception Handling

A Jini program can generate a huge number of exceptions, often related to the network nature of Jini. This is not accidental, but lies at the heart of the Jini approach to network programming. Services can disappear because the link to them has vanished, the server machine has crashed, or the service provider has died. Class files can disappear for similar problems with the HTTP server that delivers them.

Timeouts can occur due to unpredictable network delays. Many of these exceptions have their own exception types, such as `LookupUnmarshalException`, which can occur when unmarshalling objects. Many others are simply wrapped in a `RemoteException`, which has a `detail` field for the wrapped exception.

Since many Jini calls can generate exceptions, these must be handled somehow. Many Java programs (or rather, their programmers!) adopt a somewhat cavalier attitude to exceptions: catch them, maybe put out an error message, and continue—Java makes it easy to handle errors! More seriously, whenever an exception occurs, the question has to be asked: Can the program continue, or has its state been corrupted but not so badly that it cannot recover, or has the program's state been damaged so much that the program must exit.

The multicast `TestFileClassifier` of the last section can throw exceptions at a number of places:

- The `LookupDiscovery` constructor can fail. This is indicative of some serious network error. The created `discover` object is needed to add a listener, and if this cannot be done, then the program really can't do anything. So it is appropriate to exit with an error value.

- The `ServiceRegistrar.lookup()` method can fail. This is indicative of some network error in the connection with a particular service locator. While this may have failed, it is possible that other network connections may succeed. The application can restore a consistent state by skipping the rest of the code in this iteration of the `for()` loop by using a `continue` statement.

- The `FileClassifier.getMIMEType()` method can fail. This can be caused by a network error, or perhaps the service has simply gone away. Regardless, consistent state can again be restored by skipping the rest of this loop iteration.

Finally, if one part of a program can exit with an abnormal (non-zero) error value, then a successful exit should signal its success with an exit value of 0. If this is not done, then the exit value becomes indeterminate and is of no value to other processes that may wish to know whether the program exited successfully or not.

The Service Proxy

A service will be delivered from out of a service provider. That is, a server will be started to act as a service provider. It will create one or more objects, which between them will implement the service. Amongst these will be a distinguished object—the service object. The service provider will register the service object with service locators and then will wait for network requests to come in for the service.

What the service provider will actually export as a service object is usually a proxy for the service. The proxy is an object that will eventually run in a client and will usually make calls back across the network to service backend objects. These backend objects running within the server actually complete the implementation of the service.

The proxy and the service backend objects are tightly integrated; they must communicate using a protocol known to them both, and they must exchange information in an agreed upon manner. However, the relative *size* of each is up to the designer of a service and its proxy. For example, the proxy may be "fat" (or "smart"), which means it does a lot of processing on the client side. Backend object(s) within the service provider are then typically "thin," not doing much at all. Alternatively, the proxy may be "thin," doing little more (or nothing more) than passing requests between the client and "fat" backend objects, and most processing will be done by the backend objects running in the service provider.

As well as this choice of size, there is also a choice of communication mechanisms between the client and service provider objects. Client/server systems often have the choice of message-based or remote procedure call (RPC) communications. These choices are also available between a Jini proxy and its service. Since they are both in Java, there is a standard RPC-like mechanism called RMI (Remote Method Invocation), and this can be used if wanted. There is no need to use this, but many implementations of Jini proxies will do so because it is easy. RMI does force a particular choice of thin proxy to fat service backend, though, and this may not be ideal for all situations.

This chapter will look at one possibility only, where the proxy is fat and is the whole of the service implementation (the service backend is an empty set of objects). This is the simplest way of implementing the file-classifier service, but not always the most desirable. Chapter 9 will look in more detail at the other possibilities.

Uploading a Complete Service

The file-classifier service does not rely on any particular properties of its host—it is not hardware or operating-system dependent, and does not make use of any files on the host side. In this case, it is possible to upload the entire service to the client and let it run there. The proxy is the service, and no processing elements need to be left on the server.

FileClassifier Implementation

The implementation of the FileClassifier is straightforward:

```
package complete;

import common.MIMEType;
import common.FileClassifier;

/**
 * FileClassifierImpl.java
 */

public class FileClassifierImpl implements FileClassifier, java.io.Serializable {

    public MIMEType getMIMEType(String 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 null;
    }


    public FileClassifierImpl() {
        // empty
    }

} // FileClassifierImpl
```

FileClassifierServer Implementation

The service provider for the file-classifier service needs to create an instance of the exportable service object, register this, and keep the lease alive. In the `discovered()`

method, it not only registers the service but also adds it to a `LeaseRenewalManager`, to keep the lease alive "forever." This manager runs its own threads to keep re-registering the leases, but these are daemon threads. So in the `main()` method, the user thread goes to sleep for as long as we want the server to stay around.

The following code uses an "unsatisfied wait" condition that will sleep forever until interrupted. Note that if the server does terminate, then the lease will fail to be renewed and the exported service object will be discarded from lookup locators even though the server is not required for delivery of the service.

The `serviceID` is initially set to `null`. This may be the first time this service is ever run, or at least the first time it is ever run with this particular implementation. Since service IDs are issued by lookup services, it must remain `null` until at least the first registration. Then the service ID can be extracted from the registration and reused for all further lookup services. In addition, the service ID can be saved in some permanent form so that if the server crashes and restarts, the service ID can be retrieved from permanent storage. The following server code saves and retrieves this value in a `FileClassifier.id` file. Note that we get the service ID from the registration, not from the registrar.

```
package complete;

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 net.jini.core.lookup.ServiceID ;
// 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.LeaseListener;             // Jini 1.1
import net.jini.lease.LeaseRenewalEvent;         // Jini 1.1
import net.jini.lease.LeaseRenewalManager;       // Jini 1.1

import java.io.*;

/**
 * FileClassifierServer.java
 */

public class FileClassifierServer implements DiscoveryListener,
```

```
                                        LeaseListener {

protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();
protected ServiceID serviceID = null;

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

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

public FileClassifierServer() {
    // Try to load the service ID from file.
    // It isn't an error if we can't load it, because
    // maybe this is the first time this service has run
    DataInput din = null;
    try {
        din = new DataInputStream(new FileInputStream("FileClassifier.id"));
        serviceID = new ServiceID(din);
    } catch(Exception e) {
        // ignore
    }

    System.setSecurityManager(new RMISecurityManager());

    LookupDiscovery discover = null;
    try {
        discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
    } catch(Exception e) {
        System.err.println("Discovery failed " + 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(serviceID,
                                               new FileClassifierImpl(),
                                               null);
            ServiceRegistration reg = null;
            try {
                reg = registrar.register(item, Lease.FOREVER);
            } catch(java.rmi.RemoteException e) {
                System.err.println("Register exception: " + e.toString());
                continue;
            }
           System.out.println("Service registered with id " + reg.getServiceID());

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

            // set the serviceID if necessary
            if (serviceID == null) {
                serviceID = reg.getServiceID();

                // try to save the service ID in a file
                DataOutputStream dout = null;
                try {
                    dout = new DataOutputStream(new
FileOutputStream("FileClassifier.id"));
                    serviceID.writeBytes(dout);
                    dout.flush();
                } catch(Exception e) {
                    // ignore
                }

            }
        }
    }

    public void discarded(DiscoveryEvent evt) {
```

```
    }

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

} // FileClassifierServer
```

Figure 8-3 shows the server, by itself, running in its JVM.

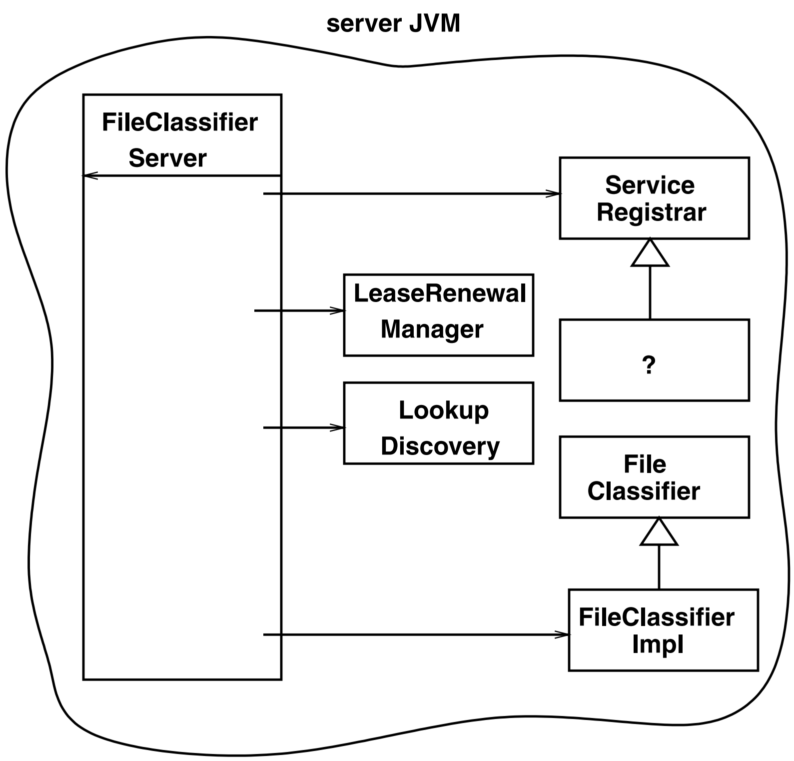

Figure 8-3. Objects in the server JVM

Figure 8-2 showed that the client receives an object implementing `ServiceRegistrar` from the service locator (such as `reggie`). When we add in the service locator and the client in their JVMs, the picture looks like what is shown in Figure 8-4.

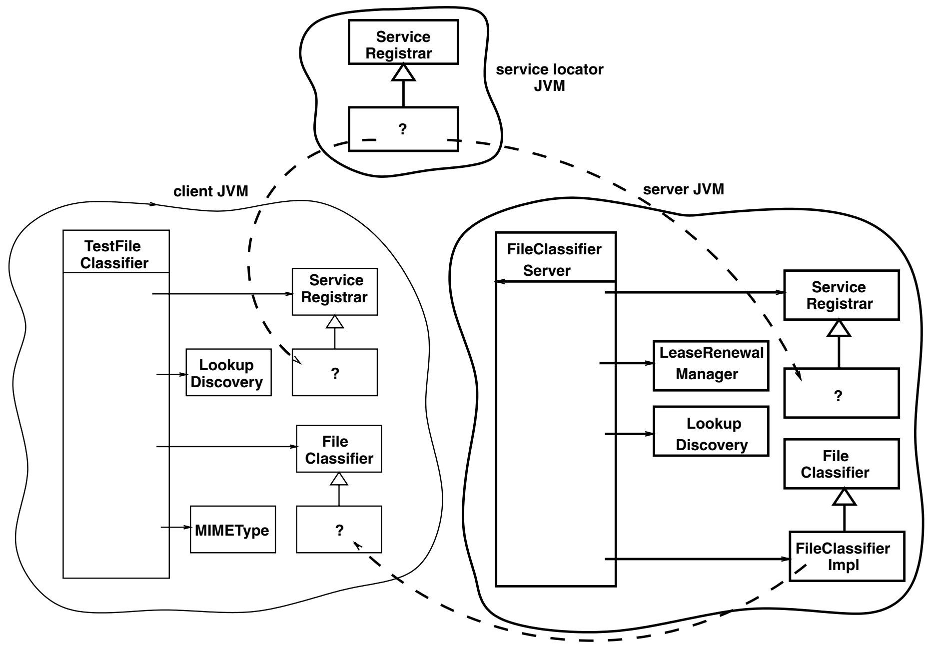

Figure 8-4. Objects in all the JVMs

The unknown `FileClassifier` object in the client is here supplied by the service object `FileClassifierImpl` (via the lookup service, where it is stored in passive form).

Client Implementation

The client for this service was discussed earlier in the section "The Client." The client does not need any special information about this implementation of the service and so can remain quite generic.

What Classes Need to Be Where?

In this chapter we have defined the following classes:

- `common.MIMEType` (in the section "Common Classes")
- `common.FileClassifier` (in the section "Common Classes")
- `complete.FileClassifierImpl` (in the section "Uploading a Complete Service")

- `complete.FileClassifierServer` (in the section "Uploading a Complete Service")
- `client.TestFileClassifier` (in the section "The Client")

These classes are all required to run a file-classifier application that consists of a file-classifier client and a file-classifier service.

Instance objects of these classes could be running on up to four different machines:

- The server machine for `FileClassifier`
- The machine for the lookup service
- The machine running the `TestFileClassifier` client
- An HTTP server will need to run somewhere to deliver the class file definition of `FileClassifierImpl` to clients

What classes need to be "known" to which machines? The term "known" can refer to different things:

- The class may be in the `CLASSPATH` of a JVM
- The class may be loadable across the network
- The class may be accessible by an HTTP server

Service Provider

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

- The `common.FileClassifier` interface
- The `common.MIMEType` class
- The `complete.FileClassifierServer` class
- The `complete.FileClassifierImpl` class

These classes all need to be in the `CLASSPATH` of the server.

HTTP Server

The class `complete.FileClassifierImpl` will need to be accessible to an HTTP server, as discussed in the next section.

Lookup Service

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`.

Client

The client needs to know the following:

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

These all need to be in the `CLASSPATH` of the client. In addition, the client will need to know the class files for `complete.FileClassifierImpl`. However, these will come across the network as part of the discovery process, and this will be invisible to the client's programmer.

Running the FileClassifier

We now have a `FileClassifierServer` service and a `TestFileClassifier` client to run. There should also be at least one lookup locator already running. The `CLASSPATH` should be set for each to include the classes discussed in the last section, in addition to the standard ones.

A serialized instance of `complete.FileClassifierImpl` will be passed from the server to the locator and then to the client. Once on the client, the Jini classes will need to be able to restore the `FileClassifierImpl` object from the instance data and from the class file, and so will need to load the `FileClassifierImpl` class file from an HTTP server. The location of this class file relative to the server's `DocumentRoot` will need to be specified by the service invocation. For example, if it is stored in `/DocumentRoot/classes/complete/FileClassifierImpl.class`, then the service will be started by this command:

```
java -Djava.rmi.codebase=http://hostname/classes \
     complete.FileClassifierServer
```

In this command, `hostname` is the name of the host the server is running on. Note that this host name cannot be `localhost`, because the local host for the server will not be the local host for the client!

The client will be loading a class definition across the network. It will need to allow this in a security policy file with the following statement:

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

Summary

The material of the previous chapters is put together in this chapter in a simple example. The requirements of class structures for a Jini system are discussed, along with details of what classes need to be available to each component of a Jini system.