CHAPTER 4

Discovering a Lookup Service

Jini uses a lookup service in much the same way as other distributed systems use naming services and traders. Services register with lookup services, and clients use them to find services they are interested in. Jini lookup services are designed to be an integral part of the Jini system, and they have their own set of classes and methods. This chapter looks at what is involved in discovering a lookup service/service locator; this is common to both services and clients. The chapter also discusses issues particular to the Sun lookup service `reggie`.

Running a Lookup Service

A client locates a service by querying a lookup service (service locator). In order to do this, it must first locate a lookup service. Similarly, a service must register itself with the lookup service, and in order to do so it must also first locate a lookup service.

The initial task for both a client and a service is thus discovering a lookup service. Such a service (or set of services) will usually have been started by some independent mechanism. The search for a lookup service can be done either by unicast or by multicast. *Unicast* means that you know the address of the lookup service and can contact it directly. *Multicast* is used when you do not know where a lookup service is and have to broadcast a message across the network so that any lookup service can respond. In fact, the lookup service is just another Jini service, but it is one that is specialized to store services and pass them on to clients looking for them.

reggie

Sun supplies a lookup service called `reggie` as part of the standard Jini distribution. The specification of a lookup service is public, and in the future we can expect to see other implementations of lookup services.

There may be any number of these lookup services running in a network. A local area network (LAN) may run many lookup services to provide redundancy in case one of them crashes. Similarly, across the Internet, people may run lookup services for a variety of reasons; for example, a public lookup service is sometimes running on `http://jan.netcomp.monash.edu.au` to aid people trying Jini clients and services so they don't need to also set up a lookup service. Other lookup services may act as coordination centers, such as a repository of locations for all of the atomic clock servers in the world.

Anybody can start a lookup service (depending on access permissions), but it will usually be started by an administrator, or started at boot time. Starting a lookup service used to be the hardest part of getting Jini working for the beginner. It could take hours or even days of playing with configuration files and network settings. It has now been made substantially easier: just run a DOS batch file or Unix shell script. At the top level of the Jini distribution is the directory `installverify`. Change to this directory and run the program `LaunchAll`, which will start an HTTP server, the lookup service `reggie`, and several other useful services.

For the curious, `LaunchAll` uses the `ServiceStarter` described in a later chapter, which in turn uses the configuration file `startAll.config`. Configuration files are described in Chapter 19.

Unicast Discovery

Unicast discovery can be used when you know the machine on which the lookup service resides and can ask for it directly. This approach is expected to be used for a lookup service that is outside of your local network, but that you know the address of anyway (such as your home network while you are at work, or a network identified in a newsgroup or e-mail message, or maybe even one advertised on TV).

Unicast discovery relies on a single class, `LookupLocator`, which is described in the next section. Basic use of this class is illustrated in the sections on the `InvalidLookupLocator` program. The `InvalidLookupLocator` should be treated as an introductory Jini program that you can build and run without having to worry about network issues. Connecting to a lookup service using the network is done with the `getRegister` method of `LookupLocator`, and an example program using this is shown in the `UnicastRegistrar` program in the "getRegistrar" section.

LookupLocator

The `LookupLocator` class in the `net.jini.core.discovery` package is used for unicast discovery of a lookup service. There are two constructors:

```
package net.jini.core.discovery;
public class LookupLocator {
     LookupLocator(java.lang.String url) throws
          java.net.MalformedURLException;
     LookupLocator(java.lang.String host,int port);
}
```

For the first constructor, the URL must be of the form `jini://host/` or `jini://host:port/`. If no port is given, it defaults to 4160. The host should be a valid Domain Name System (DNS) name (such as `www.jini.monash.edu.au`) or an IP address (such as `137.92.11.13`). No unicast discovery is performed at this stage, though, so any rubbish could be entered. Only a check for syntactic validity of the URL is performed. This syntactic check is not even done for the second constructor.

InvalidLookupLocator

The following program creates some objects with valid and invalid host/URLs. They are only checked for syntactic validity rather than existence as URLs; that is, no network lookups are performed. This should be treated as a basic example to get you started building and running a simple Jini program.

```
package basic;
import net.jini.core.discovery.LookupLocator;
/**
 * InvalidLookupLocator.java
 */
public class InvalidLookupLocator  {
    static public void main(String argv[]) {
        new InvalidLookupLocator();
    }

public InvalidLookupLocator() {
   LookupLocator lookup;
   // this is valid
   try {
       lookup = new LookupLocator("jini://localhost");
       System.out.println("First lookup creation succeeded");
   } catch(java.net.MalformedURLException e) {
       System.err.println("First lookup failed: " + e.toString());
   }
   // this is probably an invalid URL,
   // but the URL is syntactically okay
   try {
       lookup = new LookupLocator("jini://ABCDEFG.org");
       System.out.println("Second lookup creation succeeded");
   } catch(java.net.MalformedURLException e) {
       System.err.println("Second lookup failed: " + e.toString());
   }
   // this IS a malformed URL, and should throw an exception
   try {
       lookup = new LookupLocator("A:B:C://ABCDEFG.org");
       System.out.println("Third lookup creation succeeded");
   } catch(java.net.MalformedURLException e) {
       System.err.println("Third lookup failed: " + e.toString());
   }
   // this is valid, but no check is made anyway
   lookup = new LookupLocator("localhost", 80);
   System.out.println("Fourth lookup creation succeeded");
    }

} // InvalidLookupLocator
```

Running the InvalidLookupLocator

All programs in this book can be compiled using the JDK 1.4 compiler. The Java 1.5 compiler can be used, although the Jini class libraries do not use any of the 1.5 features.

The following program defines the `InvalidLookupLocator` class in the `basic` package. The source code will in the `InvalidLookupLocator.java` file in the `basic` subdirectory. From the parent directory, this can be compiled by a command such as this:

```
javac basic/InvalidLookupLocator.java
```

to leave the class file also in the `basic` subdirectory.

When you compile the code, the `CLASSPATH` will need to include some Jini `.jar` files. In versions 2.0 and earlier, the `jini-core.jar` file was required. This has changed for Jini 2.1; the preferred files are `jsk-platform.jar` and `jsk-lib.jar` for compilation of the source code. These files are in the `lib` subdirectory of the Jini distribution. When a service is run, these Jini files will need to be in its `CLASSPATH`. Similarly, when a client runs, it will also need these files in its `CLASSPATH`. The reason for this repetition is that the service and the client are two separate applications, running in two separate JVMs, and quite likely they will be on two separate computers.

The `InvalidLookupLocator` has no additional requirements. It does not perform any network calls and does not require any additional service to be running. It can be run simply by entering this command:

```
 java -Djava.security.policy=policy.all -classpath ... basic.InvalidLookupLocator
```

where the policy file could be the permissive security policy file

```
 grant { permission java.security.AllPermission; };
```

An Ant file to build, deploy, and run this class is basic.InvalidLookupLocator.xml:

```
<project name="basic.InvalidLookupLocator" default="usage">
    <!-- files for this project -->
    <property name="src.files"   value="basic/InvalidLookupLocator.java"/>
    <property name="class.files" value="basic/InvalidLookupLocator.class"/>

    <!-- 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="${ant.project.name}"/>
    <property name="no-dl" value="true"/>
    <!-- targets -->
    <target name="all" depends="compile"/>
    <target name="compile">
        <javac destdir="${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="${class.files.dl}"/>
    </target>
    <target name="build" depends="dist,compile"/>
    <target name="deploy" depends="dist" unless="no-dl">
        <copy file="${dist}/${jar.file.dl}"
              todir="${httpd.classes}"/>
    </target>
    <target name="run">
        <java classname="${main.class}"
              classpath="${jini.jars}:${dist}/${jar.file}"/>
    </target>
</project>
```

Information from the LookupLocator

Two of the methods of `LookupLocator` are as follows:

```
String getHost();
int getPort();
```

These methods will return information about the hostname that the locator will use, and the port it will connect on or is already connected on. This is just the information fed into the constructor or left to default values, though; it doesn't offer anything new for unicasting. However, this information will be useful in the multicast situation if you need to find out where the lookup service is.

getRegistrar

Search and lookup is performed by the `getRegistrar()` method of the `LookupLocator`, which returns an object of class `ServiceRegistrar`.

```
public ServiceRegistrar getRegistrar()
     throws java.io.IOException,
     java.lang.ClassNotFoundException
```

The `ServiceRegistrar` class is discussed in detail later. This class performs network lookup on the URL given in the `LookupLocator` constructor.

UML sequence diagrams are useful for showing the timelines of object existence and the method calls that are made from one object to another. The timeline reads down, and method calls and their returns read across. A UML sequence diagram augmented with a jagged arrow showing the network connection is shown in Figure 4-1. The `UnicastRegister` object makes a new() call to create a LookupLocator, and this call returns a lookup object. The getRegistrar() method call is then made on the lookup object, and this causes network activity. As a result, a ServiceRegistrar object is created in some manner by the lookup object, and this object is returned from the method as the registrar.

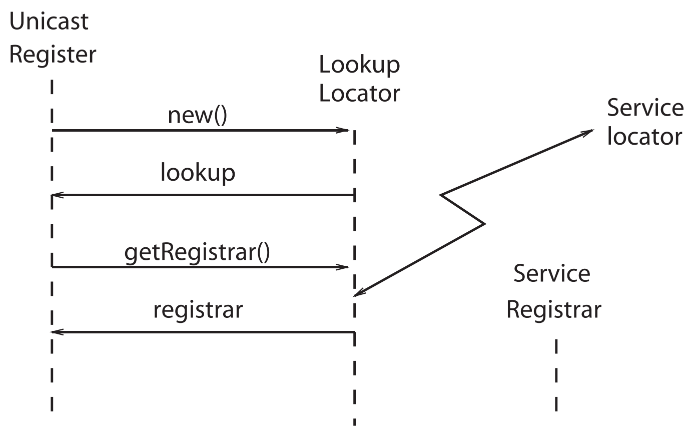

Figure 4-1. *UML sequence diagram for lookup*

By this stage, the UnicastRegister program that implements Figure 4-1 and performs the connection to get a ServiceRegistrar object looks like this:

```
package basic;
import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.ServiceRegistrar;
import java.rmi.RMISecurityManager;
/**
 * UnicastRegistrar.java
 */
public class UnicastRegister  {

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

    public UnicastRegister() {
        LookupLocator lookup = null;
        ServiceRegistrar registrar = null;
        System.setSecurityManager(new RMISecurityManager());
        try {
            lookup = new LookupLocator("jini://localhost");
        } catch(java.net.MalformedURLException e) {
            System.err.println("Lookup failed: " + e.toString());
            System.exit(1);
        }
        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);
        }
        System.out.println("Registrar found");
        // the code takes separate routes from here for client or service
    }

} // UnicastRegister
```

The `registrar` object will be used in different ways for clients and services: the services will use it to register themselves, and the clients will use it to locate services.

Running the UnicastRegister

When the `UnicastRegistrar` program in the previous section needs to be compiled and run, it has to have `jsk-platform.jar` and `jsk-lib.jar` in its `CLASSPATH`.

```
 javac -classpath ... basic/UnicastRegister.java
```

When run, it will attempt to connect to the service locator, so obviously the service locator needs to be running on the machine specified in order for this to happen. Otherwise, the program will throw an exception and terminate. In this case, the host specified is `localhost`. It could, however, be any machine accessible on the local or remote network (as long as it is running a service locator). For example, to connect to the service locator running on my current workstation, `jan.netcomp.monash.edu.au`, the parameter to `LookupLocator` would be `jini://jan.netcomp.monash.edu.au`.

The `UnicastRegister` program will receive a `ServiceRegistrar` from the service locator. However, it does so by a simple `readObject()` on a socket connected to the service locator, so it does not need any additional support services such as `rmiregistry` or `rmid`. The program can be run by this command:

```
 java -Djava.security.policy=policy.all -classpath ... basic.UnicastRegister
```

An Ant file to build, deploy, and run this class is basic.UnicastRegister.xml:

```
<project name="basic.UnicastRegister" default="usage">
    <!-- Inherits properties
         jini.home
         jini.jars
         src
         dist
         build
```

```
     httpd.classes
  -->
<!-- files for this project -->
<property name="src.files"
          value="
                 basic/UnicastRegister.java
                "/>
<property name="class.files"
          value="
                 basic/UnicastRegister.class
                "/>
<property name="class.files.dl"
          value="
                "/>
<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="${ant.project.name}"/>
<property name="jini.jars.start"
          value="${jini.jars}:${jini.home}/lib/start.jar"/>
<!-- targets -->
<target name="all" depends="compile"/>
<target name="compile">
    <javac destdir="${build}" srcdir="${src}"
           classpath="${jini.jars.start}"
           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="${class.files.dl}"/>
</target>
<target name="build" depends="dist,compile"/>
<target name="run" depends="build">
    <java classname="${main.class}"
```

```
              fork="true"
              classpath="${jini.jars.start}:${dist}/${jar.file}">
             <jvmarg value="-Djava.security.policy=${res}/policy.all"/>
        </java>
    </target>
    <target name="deploy" depends="dist" unless="no-dl">
        <copy file="${dist}/${jar.file.dl}"
              todir="${httpd.classes}"/>
    </target>
</project>
```

Broadcast Discovery

If the location of a lookup service is unknown, it is necessary to make a broadcast search for one. The User Datagram Protocol (UDP) supports a multicast mechanism that the current implementations of Jini use. Because multicast is expensive in terms of network requirements, most routers block multicast packets. This usually restricts broadcast to a LAN, although this depends on the network configuration and the time to live (TTL) of the multicast packets.

Any number of lookup services can be running on the network accessible to the broadcast search. On a small network, such as a home network, there may be just a single lookup service, but in a large network there may be many—perhaps one or two per department. Each one of these may choose to reply to a broadcast request.

Groups

Some services may be meant for anyone to use, but some may be more restricted in applicability. For example, the engineering department may wish to keep lists of services specific to that department, including a departmental diary service, a departmental inventory, and so forth. The services themselves may be running anywhere in the organization, but the department would like to be able to store information about them and to locate them from their own lookup service. Of course, this lookup service may be running anywhere, too!

So there could be lookup services specifically for a particular group of services, such as the engineering department services, and others for the publicity department services. Some lookup services may cater to more than one group—for example, a company lookup service may want to hold information about all services running for all groups on the network.

When a lookup service is started, it can be given a list of groups to act for as a command-line parameter. A service may include such group information by giving a list of groups that it belongs to. This is an array of strings, such as the following:

```
String [] groups = {"Engineering dept"};
```

LookupDiscovery

The `LookupDiscovery` class in the `net.jini.discovery` package is used for broadcast discovery. There are two constructors:

```
LookupDiscovery(java.lang.String[] groups)
LookupDiscovery(java.lang.String[] groups, Configuration config)
```

We will look at only the first one for now. The second one is new to Jini 2.0.

The parameter to the first `LookupDiscovery` constructor can take three cases:

- `null`, or `LookupDiscovery.ALL_GROUPS`, means that the object should attempt to discover all reachable lookup services, no matter which group they belong to. This will be the normal case.
- An empty list of strings, or `LookupDiscovery.NO_GROUPS`, means that the object is created but no search is performed. In this case, the `setGroups()` method will need to be called in order to perform a search.
- A nonempty array of strings can be given. This will attempt to discover all lookup services in that set of groups.

DiscoveryListener

A broadcast is a multicast call across the network, and lookup services are expected to reply as they receive the call. Doing so may take time, and there will generally be an unknown number of lookup services that can reply. To be notified of lookup services as they are discovered, the application must register a listener with the `LookupDiscovery` object, as follows.

```
public void addDiscoveryListener(DiscoveryListener l)
```

The listener must implement the `DiscoveryListener` interface:

```
package net.jini.discovery;

public abstract interface DiscoveryListener {
     public void discovered(DiscoveryEvent e);
     public void discarded(DiscoveryEvent e);
}
```

The `discovered()` method is invoked whenever a lookup service has been discovered. The API recommends that this method should return quickly and not make any remote calls. However, the `discovered()` method is the natural place to register the service, and for a client it is the natural place to ask if there is a service available and to invoke the service. It may be better to perform these lengthy operations in a separate thread.

Other timing issues are involved: when the `DiscoveryListener` is created, the broadcast is made, and after this, a listener is added to this discovery object. What happens if replies come in very quickly, before the listener is added? The Jini Discovery Utilities Specification guarantees that these replies will be buffered and delivered when a listener is added. Conversely, no replies may come in for a long time—what is the application supposed to do in the meantime? It cannot simply exit, because then there would be no object to reply to! It has to be made persistent enough to last till replies come in. One way of handling this is for the application to have a GUI interface, in which case the application will stay until the user dismisses it. Another possibility is that the application may be prepared to wait for a while before giving up. In that case, the `main()` method could sleep for, say, ten seconds and then exit. This will depend on what the application should do if no lookup service is discovered.

The `discarded()` method is invoked whenever the application discards a lookup service by calling `discard()` on the `registrar` object.

DiscoveryEvent

The parameter of the `discovered()`method of the `DiscoveryListener` interface is a `DiscoveryEvent` object.

```
package net.jini.discovery;

public Class DiscoveryEvent {
     public net.jini.core.lookup.ServiceRegistrar[] getRegistrars();
}
```

This has one public method, `getRegistrars()`, which returns an array of `ServiceRegistrar` objects. Each one of these implements the `ServiceRegistrar` interface, just like the object returned from a unicast search for a lookup service. More than one `ServiceRegistrar` object can be returned if a set of replies has come in before the listener was registered—they are collected in an array and returned in a single call to the listener. Figure 4-2 shows a UML sequence diagram augmented with jagged arrows showing the network broadcast and replies.

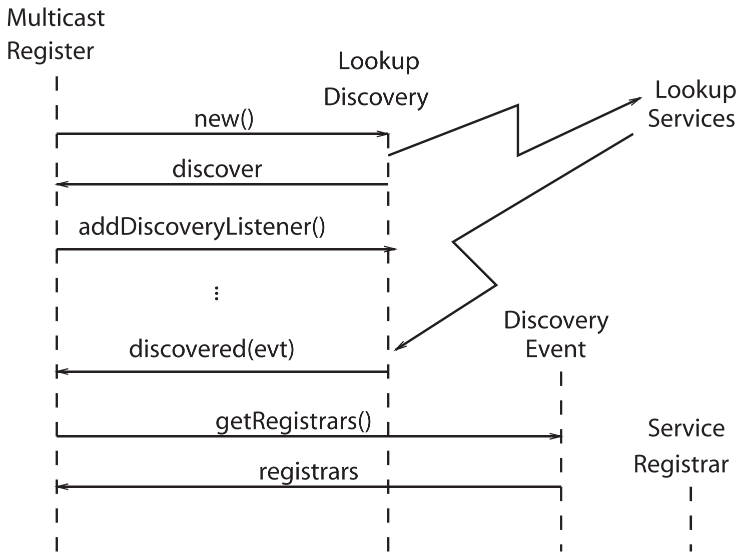

Figure 4-2. *UML sequence diagram for discovery*

In Figure 4-2, the creation of a `LookupDiscovery` object starts the broadcast search, and it returns the `discover` object. The `MulticastRegister` adds itself as a listener to the `discover` object. The search continues in a separate thread, and when a new lookup service replies, the `discover` object invokes the `discovered()` method in the `MulticastRegister`, passing it a newly created `DiscoveryEvent`. The `MulticastRegister` object can then make calls on the `DiscoveryEvent`, such as `getRegistrars()`, which will return suitable `ServiceRegistrar` objects.

By this stage, the program looks like this:

```
package basic;
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.discovery.LookupLocator;
import java.rmi.RemoteException;
/**
 * MulticastRegister.java
 */
public class MulticastRegister implements DiscoveryListener {

    static public void main(String argv[]) {
        new MulticastRegister();
        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(10000L);
        } catch(java.lang.InterruptedException e) {
            // do nothing
        }
    }

    public MulticastRegister() {
        System.setSecurityManager(new java.rmi.RMISecurityManager());
        LookupDiscovery discover = null;
        try {
            discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println(e.toString());
            e.printStackTrace();
            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];
            // the code takes separate routes from here for client or service
            try {
                System.out.println("found a service locator at " +
                                   registrar.getLocator().getHost() +
                                   " at port " +
                                registrar.getLocator().getPort());
                } catch(RemoteException e) {
                    e.printStackTrace();
                }
            }
        }
        public void discarded(DiscoveryEvent evt) {
        }
} // MulticastRegister
```

Staying Alive

In the preceding constructor for the `MulticastRegister` program, we create a `LookupDiscovery` object, add a `DiscoveryListener`, and then the constructor terminates. The `main()` method, having called this constructor, promptly goes to sleep. What is going on here? The constructor for `LookupDiscovery` actually starts up a number of threads to broadcast the service and to listen for replies. When replies come in, the listener thread will call the `discovered()` method of the `MulticastRegister`. However, these threads are daemon threads. Java has two types of threads, daemon threads and user threads, and at least one user thread must be running or the application will terminate. All these other threads are not enough to keep the application alive, so it keeps a user thread running in order to continue to exist.

The `sleep()` method ensures that a user thread continues to run, even though it apparently does nothing. This will keep the application alive, so that the daemon threads (running in the background) can discover some lookup locators. Ten seconds (10,000 milliseconds) is long enough for that. To stay alive after this ten seconds expires requires either increasing the sleep time or creating another user thread in the `discovered()` method (for example, by creating an AWT frame) or by some other method.

I have placed the `sleep()` call in the `main()` method. It is perfectly reasonable to place it in the application constructor, and some examples do this. However, it looks a bit strange in the constructor, because it looks like the constructor does not terminate (so is the object created or not?), so I prefer this placement. Note that although the constructor for `MulticastRegister` will have terminated without us assigning its object reference, a live reference has been passed into the `discover` object as a `DiscoveryListener`, and it will keep the reference alive in its own daemon threads. This means that the application object will still exist for its `discovered()` method to be called.

Any other method that results in a user thread continuing to exist will do just as well. For example, a client that has an AWT or Swing user interface will stay alive because there are many user threads created by any of these GUI objects.

For services, which typically will not have a GUI interface running, another simple way to keep them alive is to create an object and then wait for another thread to `notify()` it. Since nothing will, the thread (and hence the application) stays alive. Essentially, this is an unsatisfied wait that will never terminate—usually an erroneous thing to do, but here it is deliberate.

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

This will keep the service alive indefinitely, and it will not terminate unless interrupted. This is unlike `sleep()`, which will terminate eventually.

Running the MulticastRegister

The `MulticastRegister` program needs to be compiled and run with `jsk-platform.jar` and `jsk-lib.jar` in its `CLASSPATH`.

```
javac -classpath ... basic/MulticastRegister.java
```

When run, the program will attempt to find all service locators that it can. If there are none, it will find none—pretty boring. So one or more service locators should be set running in the near network or on the local machine.

```
java -Djava.security.policy=policy.all -classpath ... basic.MulticastRegister
```

This program will receive `ServiceRegistrars` from the service locators. However, it does so with a simple `readObject()` on a socket connected to a service locator, and so does not need any additional support services such as `rmiregistry`.

An Ant file to build, deploy, and run this class is basic.MulticastRegister.xml:

```
<project name="basic.MulticastRegister" default="usage">
    <!-- Inherits properties
         jini.home
         jini.jars
         src
         dist
         build
         httpd.classes
      -->
    <!-- files for this project -->
    <property name="src.files"
              value="
                     basic/MulticastRegister.java
                    "/>
    <property name="class.files"
              value="
                     basic/MulticastRegister.class
                    "/>
    <property name="class.files.dl"
              value="
                    "/>
    <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="${ant.project.name}"/>
    <!-- targets -->
    <target name="all" depends="compile"/>
    <target name="compile">
        <javac destdir="${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="${class.files.dl}"/>
    </target>
    <target name="build" depends="dist,compile"/>
    <target name="run" depends="build">
        <java classname="${main.class}"
              fork="true"
              classpath="${jini.jars}:${dist}/${jar.file}">
             <jvmarg value="-Djava.security.policy=${res}/policy.all"/>
        </java>
    </target>
    <target name="deploy" depends="dist" unless="no-dl">
        <copy file="${dist}/${jar.file.dl}"
              todir="${httpd.classes}"/>
    </target>
</project>
```

Broadcast Range

Services and clients search for lookup locators using the multicast protocol by sending out packets as UDP datagrams. A `LookupDiscovery` object makes announcements on UDP 224.0.1.84 on port 4160. How far do these announcements reach? This is controlled by two things:

- The time to live (TTL) field on the packets
- The network administrator settings on routers and gateways

By default, the current implementation of `LookupDiscovery` sets the TTL to 15. Common network administrative settings restrict such packets to the local network. However, the TTL may be changed by giving the system property `net.jini.discovery.ttl` a different value. But be careful about setting this, as many people will get irate if you flood the networks with multicast packets.

ServiceRegistrar

The `ServiceRegistrar` is an abstract class implemented by each lookup service. The actual details of this implementation are not relevant here. The role of a `ServiceRegistrar` is to act as a proxy for the lookup service. This proxy runs in the application, which may be a service or a client.

This is the first object that is moved from one Java process to another in Jini. It is shipped from the lookup service to the application looking for the lookup service, using a socket connection. From then on, it runs as an object in the application's address space, and the application makes normal method calls to it. When needed, it communicates back to its lookup service. The implementation used by Sun's `reggie` uses RMI to communicate, but the application does not need to know this, and anyway, it could be done in different ways. This proxy object should not cache any information on the application side, but instead should get "live" information from the lookup service as needed. The implementation of the lookup service supplied by Sun does exactly this.

The `ServiceRegistrar` object has two major methods. One is used by a service attempting to register:

```
public ServiceRegistration register(ServiceItem item, long leaseDuration)
     throws java.rmi.RemoteException
```

The other method is used by a client trying to locate a particular service:

```
public java.lang.Object lookup(ServiceTemplate tmpl)
     throws java.rmi.RemoteException;
public ServiceMatches lookup(ServiceTemplate tmpl, int maxMatches)
     throws java.rmi.RemoteException;
```

The details of these methods are given in Chapters 6 and 7. For now, an overview will suffice.

A service provider will register a service object (i.e., an instance of a class) and a set of attributes for that object. For example, a printer may specify that it can handle PostScript documents, or a toaster might specify that it can deal with frozen slices of bread. The service provider may register a singleton object that completely implements the service, but more likely it will register a service proxy that will communicate back to other objects in the service provider. Note carefully that *the registered object will be shipped around the network, and when it finally gets to run, it may be a long way away from where it was originally created.* It will have been created in the service's JVM, transferred to the lookup locator by `register()`, and then to the client's JVM by `lookup()`.

A client is trying to find a service using some properties of the service that it knows about. Whereas the service can export a live object, the client cannot use a service object as a property, because then it would already have the thing, and wouldn't need to try to find one! What it can do is use a class object, and try to find instances of this class lying around in service locators. As discussed later in Chapter 7, it is best if the client asks for an `interface` class object. In addition to this class specification, the client may specify a set of attribute values that it requires from the service.

The next step is to look at the possible forms of attribute values, and how matching will be performed. This is done using Jini `Entry` objects. The simplest services, and the least demanding clients, will not require any attributes: the `Entry[]` array will be `null`. You may wish to skip ahead to Chapter 6 or Chapter 7 and come back to the discussion of entries in Chapter 5 later.

Information from the ServiceRegistrar

The `ServiceRegistrar` is returned after a successful discovery has been made. This object has a number of methods that will return useful information about the lookup service. So, in addition to using this object to register a service or to look up a service, you can use it to find out about the lookup locator. The major methods are as follows:

```
String[] getGroups();
LookupLocator getLocator();
ServiceID getServiceID();
```

The first method, `getGroups()`, will return a list of the groups that the locator is a member of. The second method, `getLocator()`, is more interesting. This returns exactly the same type of object as is used in the unicast lookup, but now its fields are filled in by the discovery process. You can find out which host the locator is running on and its hostname by using the following statement:

```
registrar.getLocator().getHost();
```

Applications usually do not care where the lookup services are running. However, if you are curious you can use the `getLocator()` method to find this:

```
public void discovered(DiscoveryEvent evt) {
     ServiceRegistrar[] registrars = evt.getRegistrars();
     for (int n = 0; n < registrars.length; n++) {
          ServiceRegistrar registrar = registrars[n];
          System.out.println("Service locator at " +
               registrar.getLocator().getHost());
     }
}
```

The third method, `getServiceID()`, is unlikely to be of much use to you. In general, service IDs are used to give a globally unique identifier for the service (different services should not have the same ID), and a service should have the same ID with all service locators. However, this is the service ID of the lookup service, not of any services registered with it.

Summary

Both services and clients need to find lookup services. Discovering a lookup service may be done using unicast or multicast protocols. Unicast discovery is a synchronous mechanism. Multicast discovery is an asynchronous mechanism that requires the use of a listener to respond when a new service locator is discovered.

When a service locator is discovered, it sends a `ServiceRegistrar` object to run in the client or service. This object acts as a proxy for the locator and may be queried for information, such as the host the service locator is on. The major uses of the `ServiceRegistrar` object are to register services (covered in Chapter 6) and by clients searching for services (covered in Chapter 7).