CHAPTER 3

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 or 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 a 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 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 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: a public lookup service is running on `http://www.jini.canberra.edu.au` to aid people trying Jini clients and services so that 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 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.

`Reggie` requires support services: an HTTP server and an RMI daemon, `rmid`. These need to be already running by the time `reggie` is started. If there is already an HTTP server running, it can be used, or a new one can be started.

If you don't have access to an HTTP server (such as Apache), then there is a simple one supplied by Jini. This server is incomplete, and it is only good for downloading Java class files—it cannot be used as a general-purpose Web server. The Jini HTTP server is in the `tools.jar` file, and it can be started with this command:

```
java -jar tools.jar
```

This Jini HTTP server runs on a default port (8080), which means that any user can start it as long as local network policies do not forbid it. It uses the current directory as the document root for locating class files. These can be controlled by parameters:

```
java -jar tools-jarfile [-port port-number] [-dir document-root-dir] [-trees] [-
verbose]
```

The HTTP server is needed to deliver the stub class files (of the `registrar`) to clients. These class files are stored in `reggie-dl.jar`, so this file must be reachable from the document root. For example, on my machine the jar file has the full path `/home/jan/tmpdir/jini1_0/lib/reggie-dl.jar`. I set the document root to `/home/jan/tmpdir/jini1_0/lib`, so the relative URL from this server is just `/reggie-dl.jar`.

The other support service needed for `reggie` is an RMI daemon. This is `rmid`, and it is a part of the standard Java distribution. Vendors could implement other RMI daemons, but this is unlikely to happen. `rmid` must be run on the same machine as `reggie`. The following command is a Unix command that runs `rmid` as a background process:

```
rmid &
```

This command also has major options:

```
rmid [-port num] [-log dir]
```

These options can specify the TCP port used (which defaults to 4160). You can also specify the location for the log files that `rmid` uses to store its state—they default to being in the `log` subdirectory.

There is a security issue with `rmid` on multiuser systems such as Unix. The activation system that it supports allows anyone on the same machine to run programs using the user ID that `rmid` is running under. That means you should never run `rmid` using a sensitive user ID such as `root`, but instead should run it as the least privileged user, `nobody`.

Once the HTTP server and `rmid` are running, `reggie` can be started with a number of compulsory parameters:

```
java -jar lookup-server-jarfile lookup-client-codebase lookup-policy-file \
        output-log-dir lookup-service-group
```

The parameters are as follows:

- The `lookup-server-jarfile` will be `reggie.jar` or some path to it.

- The `lookup-client-codebase` will be the URL for the reggie stub class files, using the HTTP server started earlier. In my case, this is `http://jannote.dstc.edu.au:8080/reggie-dl.jar`. Note that an absolute IP hostname must be used—you cannot use `localhost` because to the `reggie` *service* that means `jannote.dstc.edu.au`. To the *client* it would be a different machine altogether, because to the client `localhost` is their own machine, not `jannote.dstc.edu.au`! The client would then fail to find `reggie-dl.jar` on its own machine. Even an abbreviated address, such as `jannote`, would fail to be resolved if the client is external to the local network.

- The `lookup-policy-file` controls security accesses. Initially you can set this to the `policy.all` path in the Jini distribution, but for deployment, use a less dangerous policy file. The topic of security is discussed in Chapter 12, but in brief, Jini code mobility allows code from other sources to run within the client machine. If you trust the other code, then that may be fine, but can you *really* trust it? If not, you don't want to run it, and Jini security can control this. However, in the debugging and testing phases, this security can cause extra complications, so you should turn off security while testing other aspects of your code by using a weak security policy. Then make sure you turn it back on later!

- The `output-log-dir` can be set to any (writable) path to store the log files.

- The `lookup-service-group` can be set to the public group `public`.

As an example, on my own machine, I start `reggie` like this:

```
java -jar /home/jan/tmpdir/jini1_0/lib/reggie.jar \
     http://jannote.dstc.edu.au:8080/reggie-dl.jar \
     /home/jan/tmpdir/jini1_0/example/lookup/policy.all \
     /tmp/reggie_log public
```

After starting, `reggie` will promptly exit! Don't worry about this—it is actually kept in a passive state by `rmid` and will be brought back into existence whenever necessary (this is done by the new `Activation` mechanism of RMI in JDK 1.2).

You only need to start `reggie` once, even if your machine is switched off or rebooted. The activation daemon `rmid` restarts it on an as-needed basis, since it keeps information about `reggie` in its log files.

Rmid and JDK 1.3

`rmid` is responsible for starting (or restarting) services such as `reggie`. It will create a new JVM on demand to run the service. `rmid` may look after a number of services, not just `reggie`, and they will all be run in their own JVMs. In JDK 1.2 there was no difference in handling these different JVMs. However, in JDK 1.3, the ability to set different security policies was introduced. This topic is dealt with in detail in Chapter 12.

In JDK 1.3, starting `rmid` requires an extra parameter to set the `sun.rmi.activation.execPolicy` policy. It is simplest to set it so that `rmid` behaves the same way as it did in JDK 1.2. This can be done with the following command:

```
rmid -J-Dsun.rmi.activation.execPolicy=none
```

This setting ignores the new security mechanism, and it is not recommended as a long-term or production solution.

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 email 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 `getRegistrar()`method of `LookupLocator`, and an example program using this is shown in the `UnicastRegistrar` program in the "Get Registrar" 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` parameter follows the standard URL syntax of "protocol://host" or "protocol://host:port". The protocol is `jini`. If no port is given, it defaults to 4160. The host should be a valid DNS name (such as `pandonia.canberra.edu.au` or an IP address (such as `137.92.11.13`). So for example, `"jini://pandonia.canberra.edu.au:4160"` may be given as the URL for the first constructor. No unicast discovery is performed at this stage, though, so any rubbish could be entered. Only a check for the syntactic validity of the URL is performed. The first constructor will throw an exception if it discovers a syntax error. This syntactic check is not even done for the second constructor, which takes a host name and port separately.

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 Jini programs will need to be compiled using the JDK 1.2 compiler. Jini programs will not compile or run under JDK 1.1 (any versions).

The `InvalidLookupLocator` program defines the `InvalidLookupLocator` class in the `basic` package. The source code will be 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
```

This will leave the class file also in the `basic` subdirectory.

When you compile the source code, the `CLASSPATH` will need to include the `jini-core.jar` Jini file. Similarly, when a service is run, this Jini file will need to be in its `CLASSPATH`, and when a client runs, it will also need this file 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 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 -classpath ... basic.InvalidLookupLocator
```

Information from the LookupLocator

Two of the methods of `LookupLocator` are these:

```
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. This information will be useful in the multicast situation, though, if you need to find out where the lookup service is.

Get Registrar

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 3-1. 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 of this, a `ServiceRegistrar` object is created in some manner by the `lookup` object, and this is returned from the method as the `registrar`.

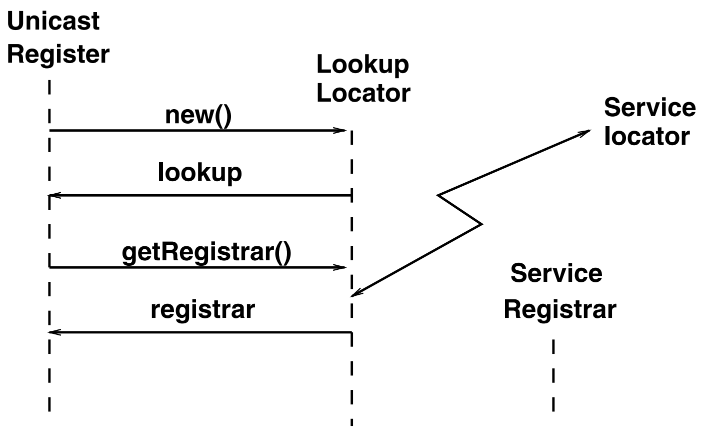

Figure 3-1. UML sequence diagram for lookup

The `UnicastRegistrar` program that implements Figure 3-1 and performs the network connection to get a `ServiceRegistrar` object is as follows:

```
package basic;

import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.ServiceRegistrar;

/**
 * UnicastRegistrar.java
 */

public class UnicastRegister  {

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

    public UnicastRegister() {
    LookupLocator lookup = null;
    ServiceRegistrar registrar = null;

        try {
            lookup = new LookupLocator("jini://www.jini.canberra.edu.au");
        } 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.

NOTE Tip title goes here

This program might not run as is, due to security issues. If that is the case, see the first section of Chapter 12.

Running the UnicastRegister

When the `UnicastRegistrar` program in the previous section program needs to be compiled and run, it has to have the file `jini-code.jar` in its `CLASSPATH`. 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 `www.jini.canberra.edu.au`. 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, the parameter for `LookupLocator` would be `jini://pandonia.canberra.edu.au`.

The `UnicastRegister` program will receive a `ServiceRegistrar` from the service locator. However, it does so with 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  basic.UnicastRegister
```

The `CLASSPATH` for the `UnicastRegister` program should contain the Jini jar files as well as the path to `basic/UnicastRegister.class`.

Broadcast Discovery

If the location of a lookup service is unknown, it is necessary to make a broadcast search for one. 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 broadcasts to a local area network, although this depends on the network configuration and the time-to-live (TTL) of the multicast packets.

There can be any number of lookup services 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. This may include a departmental diary service, a departmental inventory, etc. 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 may have a lookup service 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, like this:

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

LookupDiscovery

The `LookupDiscovery` class in package `net.jini.discovery` is used for broadcast discovery. There is a single constructor:

```
LookupDiscovery(java.lang.String[] groups)
```

The parameter in the `LookupDiscovery` constructor can take three possible values:

- `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 method `setGroups()` will need to be called in order to perform a search.

- A non-empty 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 for a service to register, and it is also the natural place for a client 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.

There are other timing issues 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 until 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 have come in before the listener was registered—they are collected in an array and returned in a single call to the listener. A UML sequence diagram augmented with jagged arrows showing the network broadcast and replies is shown in Figure 3-2.

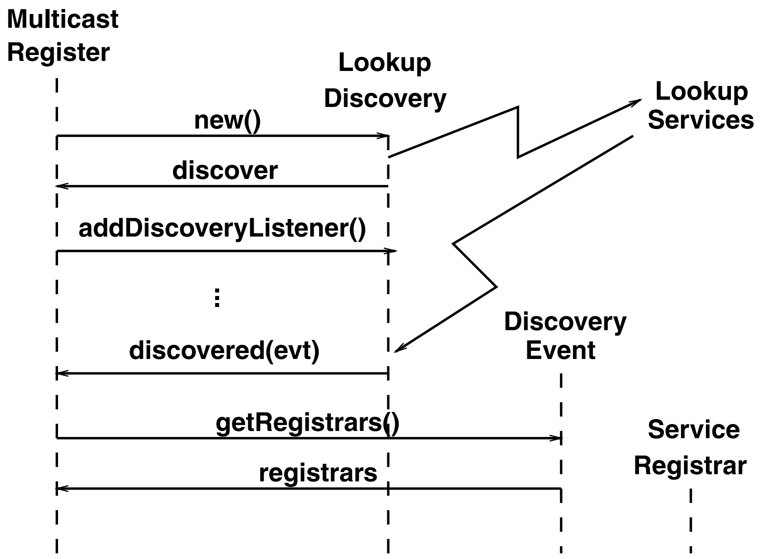

Figure 3-2. UML sequence diagram for discovery

In Figure 3-2, 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 `Multicast-Register` object can then make calls on the `DiscoveryEvent`, such as `getRegistrars()`, which will return suitable `ServiceRegistrar` objects. There is no line connecting to the `ServiceRegistrar` because the `DiscoveryEvent` creates the `ServiceRegistrar` somehow, but the actual mechanism that is used is hidden in the implementation of the `DiscoveryEvent`.

A `MulticastRegister` program that implements multicast searches for lookup services would look 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;

/**
 * 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
        System.out.println("found a service locator");
      }
    }

    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 (see Chapter 21).

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 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. In Chapter 7, use is made of a useful constant, `Lease.FOREVER`. It is tempting to use the `FOREVER` constant if you want a thread to sleep forever. While the "leasing" system understands this `FOREVER` constant, the standard Java `sleep()` method does not treat it any special way and merely uses its

Long.MAX_VALUE value and treats it as the maximum value of a long, so that it just sleeps for a very lengthy period.

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 jini-core.jar and jini-ext.jar in its CLASSPATH. The extra jar file is needed because it contains the class files from the net.jini.discovery package. When run, the program will attempt to find all the 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 network or on the local machine. Service locators running in the network must be accessible by multicast calls or they will not be found. This usually means that they will have to be on the same LAN as the MulticastRegister program.

This program will receive `ServiceRegistrar`s 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 RMI support services, such as `rmiregistry`.

Broadcast Range

Services and clients search for lookup locators using the multicast protocol by sending out packets as UDP datagrams. It 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 be 15. Common network administrative settings restrict such packets to the local network. However, the TTL can be changed by giving the system property `net.jini.discovery.ttl` a different value. However, be careful about setting this; many people will get irate if you flood the networks with multicast packets.

`ServiceRegistrar`

The `ServiceRegistrar` is an abstract class that is 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 JVM to another by 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 (with two forms) 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 Chapter 5 and Chapter 6. For now, an overview will suffice.

A service provider will register a service object (that is, 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 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: 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 6, 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 at how matching will be performed. This is done using Jini `Entry` objects, which are discussed in Chapter 4. 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 5 or to Chapter 6 and come back to the discussion of entries in Chapter 4 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 these:

```
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();
```

The following code shows how this can be used in the `discovered()` method to print information about each lookup service that replies to the multicast request:

```
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());
    }
}
```

You could use the `discovered()` method to find out where a service locator is so that the next time this program runs, it could connect directly by unicast.

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 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 acts as a proxy for the locator. This object may be queried for information, such as the host the service locator is on.