CHAPTER 26

Introspection

Questions often asked in the Jini mailing lists are as follows: "How do I find all services?" and "How do I deal with a service if I don't know what it is?" The first question is answered by searching for `Object`. Introspection is the answer to the second question, but if you require your services to be introspected, then you have to pay extra attention to the deployment environment.

In this chapter, we look at how a client can deal with a service that it knows nothing about, and how services can cooperate up front by making enough information available to clients.

Basic Service Lister

The client of Chapter 9 looked for a particular class by specifying the class in the `ServiceItem`. How do we find all services? Well, all classes inherit from `Object`, so one way is to just look for all services that implement `Object` (this is one of the few cases where we might specify a class instead of an interface). We can adapt the client quite simply by looking for all services, and then doing something simple like printing its class:

```
package client;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceMatches;
import net.jini.core.lookup.ServiceItem;
/**
 * BasicServiceLister
 */
public class BasicServiceLister implements DiscoveryListener {
    public static void main(String argv[]) {
        new BasicServiceLister();
        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(1000000L);
        } catch(java.lang.InterruptedException e) {
```

```
            // do nothing
        }
    }
    public BasicServiceLister() {
        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[] {Object.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);

        for (int n = 0; n < registrars.length; n++) {
            ServiceRegistrar registrar = registrars[n];
            System.out.print("Lookup service found at ");
            try {
                System.out.println(registrar.getLocator().getHost());
            } catch(RemoteException e) {
                continue;
            }
            ServiceMatches matches = null;
            try {
                matches = registrar.lookup(template, Integer.MAX_VALUE);
            } catch(RemoteException e) {
                System.err.println("Can't describe service: " + e.toString());
                continue;
            }
            ServiceItem[] items = matches.items;
            for (int m = 0; m < items.length; m++) {
                Object service = items[m].service;
                if (service != null) {
                    printObject(service);
                } else {
                    System.out.println("Got null service");
                }
            }
        }
    }
```

```
    public void discarded(DiscoveryEvent evt) {
        // empty
    }
    /** Print the object's class information within its hierarchy
     */
    private void printObject(Object obj) {
        System.out.println("Discovered service belongs to class \n" +
                           obj.getClass().getName());
        printInterfaces(obj.getClass());
        /*
        Class[] interfaces = obj.getClass().getInterfaces();
        if (interfaces.length != 0) {
            System.out.println("  Implements interfaces");
            for (int n = 0; n < interfaces.length; n++) {
                System.out.println("    " + interfaces[n].getName());
            }
        }
        */
        printSuperClasses(obj.getClass());
    }
    /** Print information about superclasses
     */
    private void printSuperClasses(Class cls) {
        System.out.println("  With superclasses");
        while ((cls = cls.getSuperclass()) != null) {
            System.out.println("    " + cls.getName());
            printInterfaces(cls);
        }
    }
    private void printInterfaces(Class cls) {
        Class[] interfaces = cls.getInterfaces();
        if (interfaces.length != 0) {
            System.out.println("      which implements interfaces");
            for (int n = 0; n < interfaces.length; n++) {
                System.out.println("        " + interfaces[n]);
                printInterfaces(interfaces[n]);
            }
        }
    }
} // BasicServiceLister
```

Unknown Services

A common question from Jini programmers is, "How do I deal with services that I know nothing about?" There are two answers:

- Are you sure you want to? If you don't know about the service beforehand, then what are you going to sensibly infer about its behavior just from discovering the interface? In most cases, if you don't already know the service interface and have some idea of what it is supposed to do, then getting this service isn't going to be of much use.
- On the other hand, service browsers may no have prior knowledge of the services they discover, but may still wish to use information about these services. For example, a service browser could present this information to a user and ask if the user wants to invoke the service. Or a client using artificial intelligence techniques may be able to guess at behavior from the interface and invoke the service based on this.

This (short) chapter is concerned with the second case.

Introspection

Java has a well-developed introspection library, which allows a Java program to take a class and find all of the methods (including constructors), and to find the parameters and return types of these methods. For noninterface classes, the fields can also be found. The classes that a class implements or extends can also be determined. The access methods (private, public, and protected) and the thrown exceptions can be found as well. In other words, all of the important information (except Javadoc comments) can be retrieved from the object's class.

The starting point for introspection is various `Class` methods, including the following:

```
Constructor[] getConstructors();
Class[]       getClasses();
Field[]       getFields();
Class[]       getInterfaces();
Method[]      getMethods();
int           getModifiers();
Package       getPackage();
```

Methods in the classes `Field`, `Method`, and so on allow you to gain extra details.

For example, you can use the following code to find information about interfaces of services on a lookup service:

```
ServiceRegistrar registrar = ...
ServiceTemplate templ = new ServiceTemplate(null, null, null);
ServiceMatches matches = registrar.lookup(templ, Integer.MAX_VALUE);
ServiceItem[] items = matches.items;
for (int n = 0; n &lt; items.length; n++) {
    Object service = items[n].service;
    if (service != null) {
        Class cls = service.getClass();
        System.out.println("Class is " + cls.getName());
        Class[] ifaces = cls.getInterfaces();
        for (int m = 0; m &lt; ifaces.length; m++) {
            System.out.println("  implements " + ifaces[m].getName());
        }
    }
}
```

Unknown Classes

In earlier chapters, we assumed that a client will know at least the interfaces of the services it is attempting to use. For a browser or a "smart" client, this may not be the case: the client will often come across services that it does not know much about. When a client discovers a service, it must be able to reconstitute it into an object that it can deal with, and for this it needs to be able to find the class files. If any one of the needed class files are missing, then the service comes back as `null`. That is why there is a check for `null` service in the previous example code: a service has been found, but cannot be rebuilt into an object due to missing class files.

Clients get the class files from two sources:

- Already known, and in their class path
- Accessed from a web server by the `java.rmi.server.codebase` property of the service

If you are a service provider, you may wish to make your service available to clients who have never heard of it before. In this case, you cannot rely on the client knowing anything except for the core Java classes. This may be in doubt if the client is using one of the "limited device" Java memory models—this is not a problem currently, since these models do not yet support Jini. You can make a pretty solid bet that the core Jini classes will have to be there, too, but nonessential classes in the package `jini-ext.jar` may not be present.

The example that we have been using so far is an implementation of `FileClassifier`. A typical implementation uses these noncore classes/interfaces:

- `FileClassifier`
- `RemoteFileClassifier`
- `MIMEType`
- `FileClassifierImpl`

The assumption in earlier chapters is that `FileClassifier` and `MIMEType` are well known and the others need to be accessible from a web server. For robust introspection, this assumption must be dropped: `FileClassifier` and `MIMEType` must also be available from the service's web server. This a *server* responsibility, not a *client* responsibility; the client can do nothing if the service does not make its class files available.

In summary, if a service wishes to be discovered by clients that have no prior knowledge of the service, then it must make *all* of its interface and specification classes publicly available from a web server. Any other classes that are nonstandard or potentially missing from the client should be on this public web server, too. There is a restriction: you cannot make some classes available to nonsignatories to various copyright agreements, meaning that licensing restrictions may not allow you to make some classes publicly available. For example, you cannot make any of the Jini files publicly available "just in case the client doesn't have them."

Summary

This short chapter considered some of the issues for a client to deal with services about which it has no previous knowledge. Services that would like to visible to all clients also have to take steps to make sure all required classes are available for download.