CHAPTER 6

Service Registration

This chapter looks at how services register themselves with lookup services so that they can later be found by clients. From a lookup service, the service will get a `ServiceRegistrar` object. The server will prepare a description of the service in a `ServiceItem` and will then call the `ServiceRegistrar`'s `register()` method with the `ServiceItem` as a parameter. The `ServiceItem` can contain additional information about a service as well as its type, and this information is stored in `Entry` objects.

ServiceRegistrar

A server for a service finds a service locator using either a unicast lookup with a `LookupLocator` or a multicast search using `LookupDiscovery`. In both cases, a `ServiceRegistrar` object is returned to act as a proxy for the lookup service. The server then registers the service with the service locator using the `ServiceRegistrar`'s `register()` method:

```
package net.jini.core.lookup;
public Class ServiceRegistrar {
    public ServiceRegistration register(ServiceItem item,
                                        long leaseDuration)
                                throws java.rmi.RemoteException;
}
```

The second parameter here, `leaseDuration`, is a request for the length of time (in milliseconds) the lookup service will keep the service registered. A request for a time period need not be honored—the lookup service may reject it completely, or only grant a lesser time interval. Leasing is discussed in more detail in Chapter 8.

The first parameter is of the following type:

```
package net.jini.core.lookup;
public Class ServiceItem {
    public ServiceID serviceID;
    public java.lang.Object service;
    public Entry[] attributeSets;
    public ServiceItem(ServiceID serviceID,
```

```
                    java.lang.Object service,
                    Entry[] attrSets);
}
```

ServiceItem

The service provider will create a `ServiceItem` object by using the constructor and pass it into `register()`. The `serviceID` is set to `null` when the service is registered for the first time. The lookup service will set a non-null value as it registers the service. On subsequent registrations or reregistrations, this non-null value should be used. The `serviceID` is used as a globally unique identifier (GUID) for the service.

The second parameter, `service`, is the service object that is being registered. This object will be serialized and sent to the service locator for storage. When a client later requests a service, this is the object it will be given. There are several things to note about the service object:

- The object must be serializable. Some objects, such as Swing's `JTextArea`, are not serializable at present and so cannot be used.
- The object is created in the service's JVM. However, when it runs, it will do so in the client's JVM, so it may need to be a proxy for the actual service. For example, the object may be able to show a set of toaster controls, but it might have to send messages across the network to the real toaster service, which is connected to the physical toaster.
- If the service object is an RMI proxy, then the object in the `ServiceItem` is given by the programmer as the `UnicastRemoteObject` for the proxy stub, not the proxy itself. The Java runtime substitutes the proxy. This subtlety is explored in Chapter 10.

The third parameter is a set of entries giving information about the service in addition to the service object/service proxy itself. If there is no additional information, this can be `null`.

Registration

The service attempts to register itself by calling `register()`. This may throw a `java.rmi.RemoteException`, which must be caught. The second parameter is a request to the service locator for the length of time to store the service. The time requested may or may not be honored. The return value is of type `ServiceRegistration`.

ServiceRegistration

The `ServiceRegistration` object is created by the lookup service and is returned to run in the service provider. This object acts as a proxy object that will maintain the state information for the service object exported to the lookup service.

Actually, the `ServiceRegistration` object can be used to make changes to the entire `ServiceItem` stored on the lookup service. The `ServiceRegistration` object maintains a `serviceID` field, which is used to identify the `ServiceItem` on the lookup service. The `ServiceItem` value can be retrieved by `getServiceID()` for reuse by the server if it needs to do so

(which it should, so that it can use the same identifier for the service across all lookup services). These objects are shown in Figure 6-1.

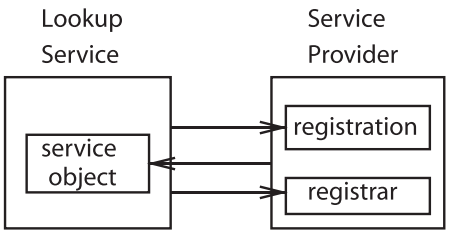

Figure 6-1. *Objects in service registration*

Other methods such as the following can be used to change the entry attributes stored on the lookup service:

```
void addAttributes(Entry[] attrSets);
void modifyAttributes(Entry[] attrSetTemplates, Entry[] attrSets);
void setAttributes(Entry[] attrSets);
```

The final public method for the `ServiceRegistration` class is `getLease()`, which returns a `Lease` object that allows renewal or cancellation of the lease. This is discussed in more detail in Chapter 8.

The major task of the server is then over. It will have successfully exported the service to a number of lookup services. What the server then does depends on how long it needs to keep the service alive or registered. If the exported service can do everything that the service needs to do, and does not need to maintain long-term registration, then the server can simply exit. More commonly, if the exported service object acts as a proxy and needs to communicate back to the service, then the server can sleep so that it maintains the existence of the service. If the service needs to be reregistered before timeout occurs, then the server can also sleep in this situation.

The SimpleService Program

A unicast server that exports its service and does nothing else is shown in the following program:

```
package basic;
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.io.Serializable;
import java.rmi.RMISecurityManager;
/**
 * SimpleService.java
```

```
 */
public class SimpleService implements Serializable {

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

    public SimpleService() {
        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("Found a registrar");
        // register ourselves as service, with no serviceID
        // or set of attributes
        ServiceItem item = new ServiceItem(null, this, null);
        ServiceRegistration reg = null;
        try {
            // ask to register for 10,000,000 milliseconds
            reg = registrar.register(item, 10000000L);
        } catch(java.rmi.RemoteException e) {
            System.err.println("Register exception: " + e.toString());
        }
        System.out.println("Service registered with registration id: " +
                           reg.getServiceID());
        // we can exit here if the exported service object can do
        // everything, or we can sleep if it needs to communicate
        // to us or we need to renew a lease later
        //
        // Typically, we will need to renew a lease later
    }

} // SimpleService
```

Running the SimpleService Program

The `SimpleService` program needs to be compiled and run with `jsk-platform.jar` and `jsk-lib.jar` in its `CLASSPATH`. In order to run the program, a security policy must be specified, as it uses an `RMIClassLoader`.

When run, the `SimpleService` program will attempt to connect to the service locator, so obviously one needs to be running on the machine specified in order for this to happen. Otherwise, the program will throw an exception and terminate.

The instance data for the service object is transferred in serialized form across socket connections. The instance data is kept in this serialized form by the lookup services. Later, when a client asks for the service to be reconstituted, it will use this instance data and also will need the class files. At this point, the class files will also need to be transferred, probably by an HTTP server. There is no need for additional RMI support services, such as `rmiregistry` or `rmid`, since all registration is done by the `register()` method.

An Ant file to build and run the `SimpleService` program is `basic.SimpleService.xml`:

```
<project name="basic.SimpleService" default="usage">
    <!-- Inherits properties
         jini.home
         jini.jars
         src
         dist
         build
         httpd.classes
      -->
    <!-- files for this project -->
    <property name="src.files"
              value="
                     basic/SimpleService.java
                    "/>
    <property name="class.files"
              value="
                     basic/SimpleService.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>
```

Information from the ServiceRegistration

The `ServiceRegistrar` object's `register()` method is used to register the service, and in doing so returns a `ServiceRegistration` object. This object can be used to give information about the registration itself. The relevant methods are as follows:

```
ServiceID getServiceID();
Lease getLease();
```

The service ID can be stored by the application if it is going to reregister later. The lease object can be used to control the lease granted by the lookup locator, and will be discussed in more detail in Chapter 8. For now, we can just use the lease object to find out how long the lease has been granted for by using its `getExpiration()` method:

```
long duration = reg.getLease().getExpiration() -
                System.currentTimeMillis();
```

```
System.out.println("Lease expires at: " +
                   duration +
                   " milliseconds from now");
```

Service ID

A service is unique in the world. It runs on a particular machine and performs certain tasks. However, it will probably register itself with many lookup services; it should have the same "identity" on all of these. In addition, if either the service or one of these locators crashes or restarts, then this identity should be the same as before.

The `ServiceID` plays the role of unique identifier for a service. It is a 128-bit number generated in a pseudo-random manner, and it should be effectively unique—the chance that a generator might duplicate this number is vanishingly small. There are two ways to get a service ID: ask a lookup service for one or generate it yourself.

If the first argument to `ServiceItem` is `null`, this is a request to a lookup service to generate and return a service ID. The returned service ID can then be used as the first parameter to other lookup services. This used to be the preferred method, but if you register with multiple lookup services, it can lead to slightly messy logic.

These days, the preferred method seems to be for a service to generate the service ID itself and give this to all the lookup services it finds, as follows:

```
import net.jini.id.Uuid;
import net.jini.id.Uuidfactory;
Uuid uuid = UuidFactory.generate();
ServiceID serviceID = new ServiceID(uuid.getMostSignificantBits(),
                                    uuid.getLeastSignificantBits());
```

Entries

A service can announce a number of entry attributes when it registers itself with a lookup service. It does so by preparing an array of `Entry` objects and passing them into the `ServiceItem` used in the `register()` method of the registrar. There is no limitation to the amount of information the service can include; in later searches by clients, each entry is treated as though it was OR'ed with the other entries. In other words, the more entries that are given by the service, the greater the chance of matching a client's requirements.

For example, suppose we have a coffee machine on the seventh floor of our building, which is known as both "GP South Building" and "General Purpose South Building." Information such as this, and general information about the coffee machine, can be encapsulated in the convenience classes `Location` and `Comment` from the `net.jini.lookup.entry` package. If this were on our network as a service, it would advertise itself as follows:

```
import net.jini.lookup.entry.Location;
import net.jini.lookup.entry.Comment;
Location loc1 = new Location("7", "728",
                             "GP South Building");
Location loc2 = new Location("7", "728",
                             "General Purpose South Building");
Comment comment = new Comment("DSTC coffee machine");
```

```
Entry[] entries = new Entry[] {loc1, loc2, comment};
ServiceItem item = new ServiceItem(..., ..., entries);
registrar.register(item, ...);
```

Summary

As you learned in this chapter, a service uses the `ServiceRegistrar` object, which is returned as a proxy from a locator, to register itself with that locator. The service prepares a `ServiceItem` that contains a service object and a set of entries, and the service object may be a proxy for the real service. It registers this service object and entry information using the `register()` method of the `ServiceRegistrar` object. Information about a registration is returned as a `ServiceRegistration` object, which may be queried for information such as the lease and its duration.