CHAPTER 10

Jini Extensible Remote Invocation

Jini Extensible Remote Invocation, or Jeri, is an alternative to the Java Remote Method Protocol (JRMP) used by "traditional" RMI. It incorporates lessons learned from RMI over IIOP and other transports.

Jini is middleware for distributed processing. As such, it relies on a number of mechanisms for distributed processing. One of these is the ability of proxies and services to communicate, so that client calls on the proxy can result in remote invocations of service methods.

Initial versions of Jini often used the Java remote method invocation (RMI). In RMI, an object stays on a server and a dumb proxy is sent to a client. The client makes method calls on this proxy. The proxy just transmits calls across the network to the server, which calls the original object. The result of this method call is then sent back to the proxy, which returns the result to the client. This is an object-oriented version of remote procedure calls. It is somewhat similar to CORBA remote references, except that the proxy is a Java object that runs in the client and does not require the "backplane" support of CORBA.

The original implementation of RMI used JRMP, a particular protocol built directly on TCP. Since then, a number of other ways of doing RMI have emerged, such as RMI over HTTP (the web transport protocol), RMI over IIOP (the CORBA transport protocol), and RMI over Secure Socket Layer (SSL). There is even an implementation of RMI on FireWire (IEEE 1394), the high-speed transport layer designed for audio/visual data such as high-definition TV.

The different ways of doing RMI each have their own programmatic interface, with specialized classes. This should be abstracted to one mechanism, with configurations used to select the actual protocol implementation used. For example, a configuration file could specify whether to use RMI over TCP or RMI over IIOP. Then the application could be written in an implementation-independent way, with the protocol chosen at runtime based on runtime configuration information. This is what Jini has now done, and Sun has developed a new protocol, Jeri, which also solves some other issues.

In this chapter, we'll cover this new protocol and look at the changes that are needed compared to traditional RMI. While some parts seem more complex than RMI, Jeri provides a more coherent programming model with greater flexibility. Jeri supports the standard RMI semantics but is designed to be more flexible than existing RMI implementations such as JRMP and RMI-over-IIOP. It can support the following:

- The new Jini trust model
- Elimination of the compile-time generation of stubs

- Non-TCP transports
- More flexible distributed garbage collection
- Much greater customization

Traditional RMI

Most books on Java include a section on RMI, and there are complete books devoted to this topic. In traditional RMI, a class typically subclasses `UnicastRemoteObject`. After compilation, the RMI compiler `rmic` is run on the class file to produce a proxy object. When the service is run, this proxy object must be made network-visible in some way so that external clients can locate it. This visibility may be to an RMI `Naming` service to a Jini registry or to other directory services. Then a client can use the directory to find the proxy object, and use the proxy object to make remote calls on the original service object.

A little bit of chicanery takes place before an object is made network-visible: a class registers itself with the Java runtime by an operation called *exporting*, and then methods that should use the proxy object instead use the original service object. The Java runtime looks out for a `UnicastRemoteObject` instance and substitutes the proxy object in its place. This means that the programmer does not deal explicitly with the proxy at all, and seemingly writes code that does not use proxies; the Java runtime takes care of substituting the proxy when necessary. However, this process may be confusing, and it certainly does not make clear exactly what is going on: this kind of trick is not one that will be in most programmers' experience.

The most common way to use RMI is simply to declare an object that extends `UnicastRemoteObject` and implements the `Remote` interface:

```
package jeri;
import java.rmi.*;
import java.rmi.server.*;
public class RmiImplicitExportDemo extends UnicastRemoteObject implements Remote {
    public static void main(String[] args) throws Exception {
        // this exports the RMI stub to the Java runtime
        // a thread is started to keep the stub alive
        new RmiImplicitExportDemo();
        System.out.println("Proxy is now exported");
        // this application will then stay alive until killed by the user
    }
    // An empty constructor is needed for the runtime to construct
    // the proxy stub
    public RmiImplicitExportDemo() throws java.rmi.RemoteException {
    }
}
```

A `UnicastRemoteObject` does lots of things in the constructor behind the scenes: it uses the class name to construct a proxy object using the zero args constructor, it starts an extra thread to keep things alive, and it registers the object as a remote object requiring special attention. While the intention is to keep things simple, these activities can prove somewhat unsettling.

The code in RmiImplicitExportDemo does not mention a proxy object. The runtime will create the proxy when it constructs the RmiImplicitExportDemo object. But just like most other Java objects, it needs to have a class definition for the proxy. The proxy class is created using the rmic compiler. This needs to be run on the implementation class file, for example:

```
javac jeri/RmiImplicitExportDemo.java
rmic -v1.2  jeri.RmiImplicitExportDemo
```

This will create an RmiImplicitDemo_Stub.class proxy.

An alternative approach makes some of the actions dealing with proxies more explicit:

```
package jeri;
import java.rmi.*;
import java.rmi.server.*;
public class RmiExplicitExportDemo implements Remote {
    public static void main(String[] args) throws Exception {
        Remote demo = new RmiExplicitExportDemo();
        // this exports the RMI stub to the Java runtime
        RemoteStub stub = UnicastRemoteObject.exportObject(demo);
        System.out.println("Proxy is " + stub.toString());

        // This application will stay alive until killed by the user,
        // or it does a System.exit()
        // or it unexports the proxy
        // Note that the demo is "apparently" unexported, not the proxy
        UnicastRemoteObject.unexportObject(demo, true);
    }
}
```

Traditionally, this mechanism has only been used when the class has to inherit from some other class and cannot also inherit from UnicastRemoteObject. Again, the proxy class has to be created by running the rmic compiler.

```
javac jeri/RmiExplicitExportDemo.java
rmic -v1.2  jeri.RmiExplicitExportDemo
```

Exporter Class

From Jini 2.0 onward, exporting and unexporting are made into explicit operations, using static methods of an Exporter class. So, for example, to export an object using JRMP, an exporter of type JRMPExporter is created and used:

```
package jeri;
import java.rmi.*;
import net.jini.export.*;
import net.jini.jrmp.JrmpExporter;
public class ExportJrmpDemo implements Remote {
    public static void main(String[] args) throws Exception {
        Exporter exporter = new JrmpExporter();
```

```
        // export an object of this class
        Remote proxy = exporter.export(new ExportJrmpDemo());
        System.out.println("Proxy is " + proxy.toString());
        // now unexport it once finished
        exporter.unexport(true);
    }
}
```

An exporter can export only one object. To export two objects, create two exporters and use each one to export an object. The proxy classes have to be generated using `rmic` again.

To export an object using IIOP, an exporter of type `IIOPExporter` is used:

```
package jeri;
import java.rmi.*;
import net.jini.export.*;
import net.jini.iiop.IiopExporter;
public class ExportIiopDemo implements Remote {
    public static void main(String[] args) throws Exception {

        Exporter exporter = new IiopExporter();
        // export an object of this class
        Remote proxy = exporter.export(new ExportIiopDemo());
        System.out.println("Proxy is " + proxy.toString());

        // now unexport it once finished
        exporter.unexport(true);
    }
}
```

Jeri Exporter

The standard exporter for Jeri is `BasicJeriExporter`. Its constructor takes parameters that specify the transport protocol (e.g., TCP on an arbitrary port) and an invocation object that handles the details of RMI, such as marshalling and unmarshalling parameters and return values, and specifying methods and exceptions. Other Jeri exporters can wrap around this class. The most common use is to create a TCP-based exporter:

```
package jeri;
import java.rmi.*;
import net.jini.export.*;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.tcp.TcpServerEndpoint;
public class ExportJeriDemo implements Remote {
    public static void main(String[] args) throws Exception {
        Exporter exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                                  new BasicILFactory());
        // export an object of this class
        Remote proxy = exporter.export(new ExportJeriDemo());
```

```
        System.out.println("Proxy is " + proxy.toString());
        // now unexport it once finished
        exporter.unexport(true);
    }
}
```

Note that there is no need to generate the proxy class as part of the compile-time build. The proxy is generated at runtime by the Jeri system.

Exported Interfaces

The exported object, the proxy, is declared to be of the `Remote` interface. In fact, the specification says that the proxy will implement all of the remote interfaces of the original `Remote` object. In the preceding examples there are none, but in general a remote object will implement one or more `Remote` interfaces, and so will the proxy.

This last point is worth expanding on. Suppose we have an interface, `Iface`, that does *not* extend `Remote`. The interface `RemoteIface` extends both `Iface` and `Remote`. From there, we could have the implementations `RemoteIfaceImpl` and `IfaceImpl`, and we could generate proxies from each of these using `Exporter`. The class names of the proxies are automatically generated (and are obscure), so let's call these classes `RemoteIfaceImplProxy` and `IfaceImplProxy`, respectively. The resulting class diagram is (with the nonstandard dotted arrow showing the generation of the proxies) is shown in Figure 10-1.

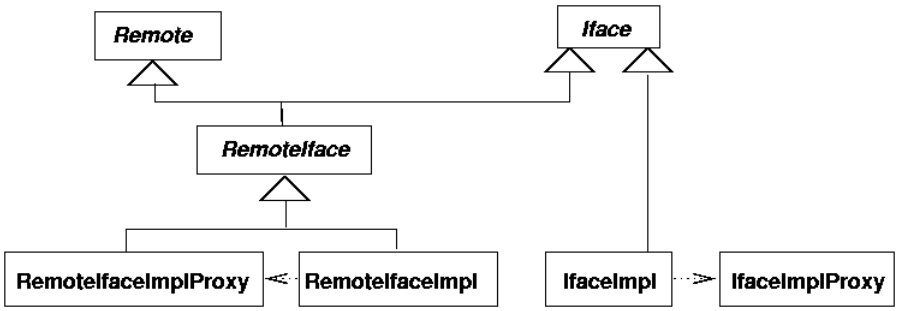

Figure 10-1. *Proxies generated by Exporter*

Since `IfaceImpl` does not implement any remote interfaces, then neither does `IfaceImplProxy` (strictly, it *may* implement some additional ones, but probably not ones we are interested in here). There are no inheritance lines leading to `IfaceImplProxy`. But since `RemoteIfaceImpl` does implement a remote interface, then so does its proxy.

The most notable consequence of this is that the `IfaceImplProxy` cannot be cast to an `Iface`, whereas `IfaceRemoteImplProxy` can be:

```
Iface iface = (Iface) ifaceImplProxy        // class cast error
Iface iface = (Iface) ifaceRemoteImplProxy  // okay
```

Thus, it will be important to include a remote interface somewhere in the interface hierarchy for any implementation.

Failure to include the `Remote` *interface will mean that you can't look up the object.* You won't be able to look up an Iface proxy object if it is just derived from an Iface object; it needs to be a proxy object derived from an RemoteIface object.

Configuration

The choices of transport protocol (TCP, FireWire, etc.) and invocation protocol (JRMP, IIOP, Jeri) are usually hard-coded into an application. That is, they are made as *compile-time* choices. More flexibility is gained if they are left as *runtime* choices. Jini 2.0 onward supports a configuration mechanism that allows a single compile-time object to be customized at runtime. The general mechanism is explored in more detail in Chapter 19. It is recommended that you use it for exporting objects.

The configuration mechanism uses a number of levels of indirection to gain a single compile-time object customized in different ways at runtime. First, it gains its `Exporter` from a `Configuration` object, which can contain a variety of runtime configuration information. The `Configuration` object is obtained from a `ConfigurationProvider`. The `ConfigurationProvider` can be set to return a custom `Configuration`, but it defaults to a `ConfigurationFile`, which takes a configuration file as parameter.

The `ConfigurationFile` object has a constructor, `ConfigurationFile(String[] args)`. The first element of this list is a file name for a configuration file. The list is passed to the `ConfigurationFile` constructor from the `ConfigurationProvider.getInstance(args)` method. Once a configuration object is found, configurable objects can be extracted from it by the `Configuration.getEntry(String component, String name, Class type)` method. The parameters for this method are as follows:

- `component`: The component being configured
- `name`: The name of the entry for the component
- `type`: The type of the object to be returned

To make this framework more concrete, the contents of the `jeri/jrmp.config` configuration file for a `ConfigurationFile` may be as follows:

```
import net.jini.jrmp.*;
JeriExportDemo {
    exporter = new JrmpExporter();
}
```

This file specifies an `Exporter` object constructed from a component name `JeriExportDemo` with the name `exporter`, of type `JrmpExporter` (which is a subclass of `Exporter`). This configuration file specifies that traditional RMI using JRMP is being used.

The configuration used can be changed by modifying the contents of the file or by using different files. To specify the IIOP protocol, change the configuration file to `jeri/iiop.config`:

```
import net.jini.iiop.*;
JeriExportDemo {
    exporter = new IiopExporter();
}
```

or use a file with multiple entries, such as `jeri/many.config`, and different component names to select which one is used:

```
import net.jini.jrmp.*;
import net.jini.iiop.*;
JrmpExportDemo {
    exporter = new JrmpExporter();
}
IiopExportDemo {
     exporter = new IiopExporter();
}
```

To use the new Jeri protocol, use the configuration file `jeri/jeri.config`:

```
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.tcp.TcpServerEndpoint;
JeriExportDemo {
    exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                     new BasicILFactory());
}
```

Once an `Exporter` has been found, an object can be exported to the Java runtime by the `Exporter.export()` method. This method takes the implementation object and returns a proxy that can be registered with a lookup service or any other remote directory.

The program to export an object using this configuration mechanism is as follows:

```
package jeri;
import java.rmi.*;
import net.jini.config.*;
import net.jini.export.*;
public class ConfigExportDemo implements Remote {
    // We are using an explicit config file here so you can see
    // where the Configuration is coming from. Really, another
    // level of indirection (such as a command-line argument)
    // should be used
    private static String CONFIG_FILE = "jeri/jeri.config";
    public static void main(String[] args) throws Exception {
        String[] configArgs = new String[] {CONFIG_FILE};
        // get the configuration (by default a FileConfiguration)
        Configuration config = ConfigurationProvider.getInstance(configArgs);
        System.out.println("Configuration: " + config.toString());
        // and use this to construct an exporter
        Exporter exporter = (Exporter) config.getEntry( "JeriExportDemo",
                                                        "exporter",
                                                        Exporter.class);
        // export an object of this class
        Remote proxy = exporter.export(new ConfigExportDemo());
        System.out.println("Proxy is " + proxy.toString());
```

```
        // now unexport it once finished
        exporter.unexport(true);
    }
}
```

Garbage Collection

Jini is a distributed system. Any JVM will have references to local objects and to remote objects. In addition, remote JVMs will have references to objects running in any particular JVM. In earlier versions of Jini that used "traditional" RMI, remote references to local objects were enough to keep the local objects from being garbage collected—a thread created by RMI did this. This often led to "sloppy" code, where objects would be created and then apparently all (local) references to those objects would go out of scope. Nevertheless, RMI would often keep these objects alive, and they would not be garbage collected.

Java distinguishes between "strong" and "weak" references. Objects that have only weak references to them may be garbage collected, while objects with strong references to them cannot be garbage collected. If there is an explicit reference to an object from another one that in turn has strong references, then the first object cannot be garbage collected. Objects with only weak references may be garbage collected. Earlier versions of this book (up to Jini 1.2) had service objects that did not have strong references. While those examples ran OK under Jini 1.2, that is no longer the case with Jini 2: running examples unchanged may result in this error when the client tries to invoke the service:

```
java.rmi.NoSuchObjectException: no such object in table
```

The solution is to make strong references to service objects. For example, the server's `main()` method will have a (static) reference to the server object, which in turn will have a reference to the service.

To make this a little more concrete, the `FileClassifierServer` of Chapter 9 used to have code that looked like this:

```
public class FileClassifierServer {
    public static void main(String argv[]) {
        new FileClassifierServer();
        ...
    }
    public FileClassifierServer() {
        // Create the service
        new FileClassifierImpl();
        ...
    }
}
```

No reference was kept to the server or to the implementation. This didn't matter in Jini 1.2, because the RMI runtime system would keep a reference to the implementation alive. Since Jini 2.0, explicit references should be kept:

```
public class FileClassifierServer {

    protected FileClassifierImpl impl;
    public static void main(String argv[]) {
        FileClassifierServer s = new FileClassifierServer();
        ... // sleep forever
        }
    }
    public FileClassifierServer() {
        // Create the service
        impl = new FileClassifierImpl
        ...
    }
}
```

Then as long as the `main()` method remains alive, there will be strong references kept to the implementation.

Proxy Accessor

In most of the examples in this book, a server will create a service and then create a proxy for it, export the proxy, and use the proxy later for such tasks as registration with a lookup service. But there are some occasions (such as activation) where it will not be feasible for the server to create the proxy; instead, the service itself will need to create the proxy.

If the service creates its own proxy, then the server may still need to get access to it, such as in registration with a lookup service. The server will then need to ask the service for its proxy. The service can signal that it is able to do this—and supply an access method—by implementing the `ProxyAccessor` interface:

```
interface ProxyAccessor {
    Object getProxy();
}
```

The service will return a suitable proxy when this method is called. You'll see examples of this in use occasionally.

Summary

This chapter covered the new model for remote invocation, called Jeri. The chapter showed how Jeri differs from the traditional RMI, and also how to deal with the explicit exporter model it uses. Jeri is the preferred mechanism in Jini for remote objects.

■ **Note** For more information on the topics covered in this chapter, see Frank Sommers' article titled "Call on extensible RMI: An introduction to JERI" at `http://www.javaworld.com/javaworld/jw-12-2003/jw-1219-jiniology.html`.
