CHAPTER 2

Troubleshooting Jini

Jini is advertised as being "network plug and play," which carries with it the idea of *zero administration*, where you buy a device or install a software service, switch it on, and *voila!*—it is there and available. Well, this may happen in the future, but right now there are a number of backroom games that you have to succeed at. Once you have won these games, "network plug and play" *does* work, but if you lose at any stage, you have an uphill battle to fight.

The difficult parts are getting the right files in the right places with the right permissions. About 50 percent of the messages in the Jini mailing list relate to these configuration problems, which shouldn't occur.

This chapter looks at some of the problems that can arise in a Jini system, most of which are configuration issues of some kind. Each of the early sections contains step-by-step instructions on what to do to get the example programs working. Because this is only the second chapter in this book, and right now you shouldn't have managed to fail at anything, feel free to skip to the next chapters, but do come back here when things go wrong.

Java Packages

The following is a typical Java package-related error:

```
Exception in thread "main" java.lang.NoClassDefFoundError: basic/
InvalidLookupLocator
```

Most of the code in this tutorial is organized into packages. To run the book's examples, the classes must be accessible from your classpath. For example, one of the programs in the `basic` directory is `InvalidLookupLocator.java`. This defines the class `InvalidLookupLocator` in the package `basic`. The program must be run using the fully qualified path name, as follows:

```
java basic.InvalidLookupLocator
```

(Note the use of ".", not "/".)

To find this class, the classpath must be set correctly for the Java runtime. If you have copied the file `classes.zip`, then you can find the class files for this tutorial there. You only need to reference this:

```
CLASSPATH=classes.zip:...
```

If you have downloaded the source files, then you can find the class files in subdirectories such as `basic`, `complex`, and so on. After compilation, the class files should also be in these subdirectories, for example, `basic/InvalidLookupLocator.class`. An alternative to using `classes.zip` is to set the classpath to include the directory containing those subdirectories. For example, if the full path is `/home/jan/classes/basic/InvalidLookupLocator.class`, then set classpath to

```
CLASSPATH=/home/jan/classes:...
```

An alternative to setting the `CLASSPATH` environment variable is to use the `-classpath` option to the Java runtime engine:

```
java -classpath /home/jan/classes basic.InvalidLookupLocator
```

Jini and Java Versions

There are five versions of Jini: 1.0, 1.1, 1.2, 2.0, and now 2.1. The core classes are the same in each version. In this book, we'll deal only with the new version, 2.1. Jini 2.1 requires Java Development Kit (JDK) 1.4 or later, not earlier versions of Java. It will work with JDK 1.5 but does not require it.

The changes for 2.1 are listed in the document `jini2_1/doc/release-notes/new.html`.

The main classes that have changed for 2.0 are as follows:

- `LookupDiscovery` (now has an additional constructor)
- `LeaseRenewalManager`
- `ServiceIDListener`

The main new classes are as follows:

- `LookupLocatorDiscovery`
- `LookupDiscoveryManager`
- `ClientLookupManager`

If you get syntax errors or runtime errors relating to these classes, it is possible that you are using Jini 1 instead of Jini 2. If you get "deprecated" warnings, it is likely that you are using the Jini 1 classes in a Jini 2 environment. The old classes are supported for now, but are not approved.

Jini Packages

The following is a typical Jini package-related error:

```
Exception in thread "main" java.lang.NoClassDefFoundError: net/jini/discovery/
DiscoveryListener
```

The Jini class files are all in `.jar` files. The Jini distribution has them in a subdirectory, `lib`. The files were repackaged in Jini 2.0: formerly you would use `jini-core.jar`, `jini-ext.jar` and sometimes `sun-util.jar`. Now you should use `jsk-platform.jar` and `jsk-lib.jar`.

A compile or run of a Jini application will typically have an environment set something like this:

```
JINI_HOME=wherever_Jini_home_is CLASSPATH=.:$JINI_HOME/lib/jsk-plat-
form.jar:$JINI_HOME/lib/jsk-lib.jar
```

HTTP Server

Jini requires a server to deliver class files to a client. Usually this is done using an HTTP server. One of the common errors related to this is as follows:

```
java.rmi.ServerException: RemoteException in server thread; nested exception is:
java.rmi.UnmarshalException: unmarshalling method/arguments; nested exception is:
java.lang.ClassNotFoundException: could not obtain preferred value for: ...
```

The most likely cause of this exception is that you aren't running an HTTP server on the machine that `java.rmi.server.codebase` is pointing to. Note that using `localhost` is a common error, since it may refer to a different machine from the one intended.

Network Configuration

A long-term aim in pervasive computing is to have *zero configuration*, whereby you can plug devices into a network and things "just work." Jini goes a long way toward making this possible at the *service* level, but the current implementation relies heavily on a functioning network layer: misconfiguration of the network can cause a great deal of problems in Jini.

The following is a typical network configuration error:

```
java.rmi.ConnectException: connection refused or timed out to BasicObjectEnd-
point[88133900-39f9-466a-880b-de8ce6653a63, TcpEndpoint[0.0.0.0:1831]]; nested
exception is: java.net.ConnectException: Connection refused
```

This error can occur by using the new configuration mechanism, where a service is exported by Jeri as follows:

```
 exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0), new
BasicILFactory());
```

"Exporting a service" means finding the localhost, getting its hostname and IP address, and listening on any available port. I lost several days' work over this, as the hostname on one machine was incorrectly set, and the Java network layer (by `InetAddress.getLocalHost()`) was unable to determine the IP address of `localhost` and returned "0.0.0.0"—and nothing could connect to that address!

The solution was to correctly set the hostname on that machine; then services could be found and run on that machine. Alternatively, `TcpServerEndpoint.getInstance(0)` could be replaced by `TcpServerEndpoint.getInstance("my_ip_address", 0)` (for a suitable "my_ip_address", of course!) in the configuration files for the services in that machine.

Could Not Obtain Preferred Value

When you receive a "Could not obtain preferred value for . . ." message, it means that Jini can't find a class file—something is wrong with the classpath or the codebase. This can occur if the codebase points to a directory, and the value is not terminated with a forward slash (`/`).

Lookup Service

The following is a typical lookup service-related error:

```
java.rmi.activation.ActivationException: ActivationSystem not running; nested excep-
tion is: java.rmi.NotBoundException: java.rmi.activation.ActivationSystem
java.rmi.NotBoundException: java.rmi.activation.ActivationSystem
```

The command `rmid` starts the activation system running. If the activation system cannot start properly or dies just after starting, you will get this message. Usually it is caused by incorrect file permissions.

RMI Stubs

This is a typical RMI stubs-related error:

```
java.rmi.StubNotFoundException: Stub class not found: rmi.FileClassifierImpl_Stub;
nested exception is: java.lang.ClassNotFoundException: rmi.FileClassifierImpl_Stub
```

This error does not occur as frequently as it used to. From Jini 2.0 onward, proxies should be generated using Jeri instead of RMI, and this error will only occur when using RMI. If it does occur, then the best thing to do is change the application to use Jeri. See Chapter 10 for more details.

Garbage Collection

The following is a typical garbage collection-related error:

```
java.rmi.ConnectException: connection refused or timed out to BasicObjectEnd-
point[afeb7958-8cff-41cb-8042-ec884a52e9a6,TcpEndpoint[192.168.2.1:3558]]; nested
exception is: java.net.ConnectException: Connection refused
```

If the service has been garbage collected, then there will be no server listening for connections to it, so any connection request will be refused. This error is more likely to happen with Jini 2.0, where objects may be garbage collected if there are no active references.

The solution is to ensure that an active reference is kept to the service. The `main()` method should contain a reference to the server (not just create it, but also keep a variable pointing to it). The server should also keep a reference to the service implementation. An alternative is to keep a static reference to the service implementation. Similarly, if you are using a `JoinManager` to keep services leased, then there should be an active reference to it or it may be garbage collected and cause any leases to expire.

Debugging

Debugging a Jini application is difficult because there are so many bits to it, and these bits are all running separately: the server for a service, the client, lookup services, possibly remote activation daemons, and HTTP servers. There are a few (not many) errors within the Jini objects themselves, but more important, many of these objects are implemented using multiple threads, and the flow of execution is not always clear. There are no magic debug flags that can be turned on to show what is happening.

On either the client or service side, a debugger such as `jdb` can be used to step through or trace execution of the client or the server. Having lots of print statements helps, too, and you can also turn on the following three flags:

```
java -Djava.security.debug=access \ -Dnet.jini.discovery.debug=1 \ -Djava.rmi.server.logCalls=true ...
```

These flags don't give complete information, but they do give some, and can at least tell you if the application parts are still living.

The logging API introduced in Jini 1.4 has been adopted by Jini 2.0. It can also be used for debugging and is discussed in Chapter 20.

Summary

As discussed in this chapter, getting a Jini application to run should be easy, but sometimes it isn't. Issues specific to Jini that you may encounter include the following:

- Using the correct Jini packages
- Running an HTTP server
- Having a properly configured network
- Codebase settings
- Weak references causing services to be garbage collected