CHAPTER 2

Troubleshooting Jini Configuration Problems

JINI IS ADVERTISED AS "network plug and work," which carries the idea of zero administration, where you buy a device, 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 back-room games that you have to succeed at. Once you have won at these, network plug and work *does* indeed work, but if you lose at any stage, then it can be all uphill!

The difficulty is getting the right files in the right places with the right permissions. About 50 percent of the messages in the Jini mailing list are about these configuration problems. They shouldn't occur, and that is why this is "The Chapter That Shouldn't Exist." This chapter looks at some of the problems that can arise in a Jini system. Most of them are configuration problems of some kind.

This is the second chapter in the book, so right now you shouldn't have managed to fail at anything! In the following chapters, the sections contains instructions on what to do to get the example programs working, and include step-by-step instructions, so skip on to the next chapters, but come back here when things go wrong. Your luck may vary: I got a reasonable way into my first attempts without problems, and some people are even luckier. Some aren't. . . .

Java Packages

A typical Java packages error looks like this:

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

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

```
java basic.InvalidLookupLocator
```

Note the use of the period (.), not a slash (/).

In order to find this class, the CLASSPATH must be set correctly for the Java runtime. If you have copied the classes.zip file, the class files for this tutorial are in there. You only need to reference this:

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

If you have downloaded the source files, then they are all in subdirectories, such as basic, complex, etc. After compilation, the class files should also be in the subdirectories, such as 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 the 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, like this:

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

Jini Versions

At the time of writing, there are two versions of Jini: 1.0 and a version of 1.1. The core classes are all the same for versions 1.0 and 1.1. The only changes in version 1.1 for the programmer are that some classes from Jini 1.0 have been better specified and are in different packages, and some classes are new.

These are the main classes that have changed:

- JoinManager
- LeaseRenewalManager
- ServiceIDListener

These are the main new classes:

- LookupLocatorDiscovery
- LookupDiscoveryManager
- ClientLookupManager

If you get syntax or runtime errors relating to these classes, then it is possible that you are using Jini 1.0 instead of Jini 1.1. If you get "deprecated" warnings, then it is likely that you are using the Jini 1.0 classes in a Jini 1.1 environment. The old classes are supported for now, but are not approved.

Jini Packages

A typical Jini package error looks like this:

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

The Jini class files are all in jar files. The Jini distribution puts them in a `lib` subdirectory when they are unpacked. There are a whole bunch of these jar files:

```
jini-core.jar            mahalo-dl.jar     sun-util.jar
jini-examples-dl.jar     mahalo.jar        tools.jar
jini-examples.jar        reggie-dl.jar     reggie.jar
jini-ext.jar
```

The `jini-core.jar` jar file contains the major packages of Jini:

```
net.jini.core                net.jini.core.discovery
net.jini.core.entry          net.jini.core.event
net.jini.core.lease          net.jini.core.lookup
net.jini.core.transaction
```

If the Java compiler or runtime can't find a class in one of these packages, then you need to make sure that the `jini-core.jar` file is in your `CLASSPATH`.

The jar file `jini-ext.jar` contains a set of packages that are not in the core, but are still heavily used:

```
net.jini.admin      net.jini.discovery
net.jini.entry      net.jini.lease
net.jini.lookup     net.jini.lookup.entry
net.jini.space
```

If the Java compiler or runtime can't find a class in one of these packages, then you need to make sure that the `jini-ext.jar` file is in your `CLASSPATH`.

The `sun-util.jar` jar file contains the packages from the `com.sun.jini` hierarchy. These contain a number of "convenience" classes that are not essential but can be useful. These are less frequently used.

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

```
JINI_HOME=wherever_Jini_home_is
CLASSPATH=.:$JINI_HOME/lib/jini-core.jar:$JINI_HOME/lib/jini-ext.jar
```

Lookup Service

A typical lookup service error looks like this:

```
java.rmi.activation.ActivationException: ActivationSystem not running;
nested exception is:
    java.rmi.NotBoundException: java.rmi.activation.ActivationSystem
java.rmi.NotBoundException: java.rmi.activation.ActivationSystem
```

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

RMI Stubs

A typical RMI stubs error looks like this:

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

Many of the examples in this book export services as remote RMI objects. These objects are subclasses of `UnicastRemoteObject`. What gets exported is not the object itself, but a stub that will act as a proxy for the object (which continues to run back in the server). The stub has to be created using the `rmic` compiler, like this:

```
rmic -v1.2 -d . rmi.FileClassifierImpl
```

This will create a `FileClassifierImpl_Stub.class` in the `rmi` subdirectory. The stub class file needs to be accessible to the Java runtime in the same way as the original class file.

Another typical error is this:

```
java.rmi.ServerException: RemoteException occurred in server thread; nested excep-
tion is:
  java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is:
  java.lang.ClassNotFoundException: rmi.FileClassifierImpl_Stub
```

This error arises when an object is trying to get a remote reference to `FileClassifierImpl`, and it is trying to load the class file for the stub from an HTTP server. What makes this one particularly annoying is that it may not be referring to the `FileClassifierImpl_Stub` at all! The class will often implement a remote interface, such as `RemoteFileClassifier`. This, in turn, implements the common class `FileClassifier`, as shown in Figure 2-1.

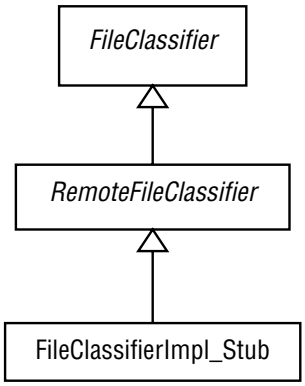

Figure 2-1. Interfaces and superclasses for an exported stub

Class files for all of these classes and interfaces have to be available! The `FileClassifier` interface may be "well known," with a class file on each client and server. However, an interface such as `RemoteFileClassifier`, as well as the

implementation files for `FileClassifierImpl`, may only be known to a particular server. The HTTP server must carry not only the class files for the stubs, but the class files for all superclasses and interfaces that are not available to all—in this case, for `RemoteFileClassifier` as well as `FileClassfier`.

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, the lookup services, the remote activation daemons, and the HTTP servers. There are a few (not many) errors within the Jini objects themselves, but more importantly, 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 a client or server. Lots of print statements help too. There are also three flags that can be turned on to help:

```
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 they can at least tell you if the application's parts are still living! If the `java.security.debug` property is set to `access`, then every time the application needs to check a security access (such as making a network connection, opening a file, etc.) it will print a message. If `net.jini.discovery.debug` is set to any non-null value, then any exceptions thrown during the discovery process will be printed. The final property will set on logging of RMI calls.

Summary

Setting up and running a Jini system is complex at present, with many things that can go wrong. This chapter looked at some of the problems that can occur and some of the solutions. The list is not complete, but it may help in the most common situations.