7168ch01.fm Page 1 Tuesday, August 22, 2006 2:44 PM

CHAPTER 1

Overview of Jini

Jini grew from early work in Java to make distributed computing easier. It intends to make network devices and network services into standard components of everyone's computing environment. The computing world is currently abuzz about service-oriented systems, and Jini has been a major platform for service-oriented computing since its inception, long before the term became popular.

Jini supports both software and hardware services. When you buy a new piece of office computing equipment such as a desk lamp, or a new home computer appliance such as an alarm clock, it will not only carry out its "traditional" functions, but will also join a network of other computer devices and services. The desk lamp will turn itself off when you leave your desk, informed of your departure by sensors in your chair; the alarm clock will tell your coffee-maker to switch on a few minutes before it wakes you up. These hardware services will interact with software services such as calendar and diary services, and possibly with external services such as weather and stock exchange services (to wake you up early if the weather is cold or if there is sudden movement on the market). Jini doesn't care what is behind services; it just makes the services available to applications.

Homes, offices, and factories are becoming increasingly networked. Current twisted-pair wiring will remain, but it will be augmented by wireless networks and networks built on phone lines and power cables. On top of this will be an infrastructure to allow services to communicate. TCP/IP will be a part of this, but it will not be enough. There will need to be mechanisms for service discovery, for negotiation of properties, and for event signaling (e.g., "My alarm has gone off—does anyone want to know?").

Jini supplies this higher level of interaction. This chapter provides a brief overview of Jini, and the components of a Jini system and the relationships between them.

Note The licensing model for Jini has now changed to an open source Apache license from a proprietary license. This change, plus the continued quality of the middleware, has sparked renewed interest in Jini.

Jini

Jini is the name for a distributed computing environment that can offer *network plug and play*, meaning that a device or a software service can be connected to a network and announce its presence, and clients that wish to use such a service can then locate it and call it to perform tasks. Jini can be used for mobile computing tasks where a service may be connected to a network for only a short time, but it can more generally be used in any network where there is some degree of change. Jini is useful in a large number of scenarios, including the following:

- A new printer can be connected to the network and announce its presence and capabilities. A client can then use this printer without having to be specially configured to do so.
- A digital camera can be connected to the network and present a user interface that will not only allow pictures to be taken, but also be aware of any printers so that the pictures can be printed.
- A configuration file that is copied and modified on individual machines can be made into a network service from a single machine, reducing maintenance costs.
- New capabilities extending existing ones can be added to a running system without disrupting existing services, or without any need to reconfigure clients.
- Services can announce changes of state, such as when a printer runs out of paper. Listeners, typically of an administrative nature, can watch for these changes and flag them for attention.

Jini is not an acronym for anything, and it does not have a particular meaning (although it gained the post-hoc interpretation of “Jini Is Not Initials”). A Jini system or *federation* is a collection of clients and services all communicating by the Jini protocols. Often this federation will consist of applications written in Java, communicating using the Java Remote Method Invocation (RMI) mechanism. Although Jini is written in pure Java, neither clients nor services are constrained to be in pure Java. They may include native code methods, act as wrappers around non-Java objects, or even be written in some other language altogether. Jini supplies a *middleware* layer to link services and clients from a variety of sources.

When you download a copy of Jini, you actually get a mixture of things. First, Jini is a specification of a set of middleware components, including an application programming interface (API) so that you as a programmer can write services and components that make use of this middleware. Second, it includes an implementation (in pure Java) of the middleware, as a set of Java packages. By including these packages in the classpath of your client or service, you can invoke the Jini middleware protocols to join in whatever Jini services and clients are currently running. (This collection of clients of services is sometimes called a *djinn*.) You also get source code to these packages as a bonus. Finally, Jini requires a number of “standard” services, and Sun gives basic implementations of each. These implementations are not an official part of Jini, but are included to get you going, and in practice, most users find these implementations sufficient to do substantial work with Jini.

Jini was not born in a vacuum. It was based on long experience within Sun Microsystems of building networking applications and frameworks. Many of the most important lessons from this were summarised in the Eight Fallacies of Distributed Computing. For the last few years Jini has not been highly visible. There are various non-technical reasons for this, but also technology that works and works well is often not reported. Nevertheless, to show that Jini came from somewhere and has been used in substantial projects, in the subsections that follow, I discuss the Eight Fallacies of Distributed Computing and how they relate to Jini, and I also describe some Jini success stories.

Eight Fallacies of Distributed Computing

Since the early days, computers have been linked in networks. For over 20 years, the mantra from Sun Microsystems has been "The Network Is the Computer," and this idea has been a cornerstone of much work in distributed computing. Based on the experience of Sun engineers over many years, Peter Deutsch took a critical look at the state of distributed computing in 1999 and concluded that early optimism was in many ways misplaced. Networks and applications that run on them are prone to all sorts of problems that do not occur with stand-alone applications, and ignoring these problems can lead to unreliable and unstable applications. Deutsch identified the following fallacies of networking (extended by James Gosling):

- The network is reliable.
- Bandwidth is infinite.
- The network is secure.
- Topology doesn't change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.
- Latency is zero.

Typical ways of "hiding" the network such as remote procedure calls (RPCs) assume these fallacies are all true. For example, Sun's RPC was one of the earliest widespread RPC frameworks. This *does* assume that the network is reliable, and it *does* assume that bandwidth is infinite, so applications assume that remote calls will always succeeed and that there is no overhead in making remote calls. Network calls are several orders of magnitude slower than local function calls and do sometimes fail, but there is no recognition of this in the RPC programming model—that was its purpose: to hide the network! These assumptions have been continued into many later middleware systems such as CORBA, and even into quite recent (and popular) frameworks.

Jini recognizes these fallacies and attempts to deal with them. For example, specifications of services have to be marked as "potentially remote" and all method calls have to handle possible network failures.

Jini Success Stories

Jini has been around since 1999, and while it has achieved some notable successes, it does not have the visibility of many other middleware systems. In part, this is because Jini simply *works* and has been a stable but evolving platform over these years. The Sun web site lists many successful projects using Jini. This section covers a couple of them plus some other systems. For more stories, visit `http://www.sun.com/software/jini/news/success.xml`.

■ **Note** You should be aware that Jini is only one competitor in a growing market. What conditions the success or failure of Jini is partly the politics of the market, but also (hopefully) the technical capabilities of Jini. This book deals with some of these technical issues involved in using Jini.

Note that the systems described in the sections that follow often have to deal directly with the issues raised by the Eight Fallacies described in the previous section. For example, they need to work in changing topologies, on unreliable networks, and with limited bandwidth. Jini is middleware designed to manage such issues, but it is not perfect. Nevertheless, it recognises the Fallacies and provides mechanisms adequate for many systems, with the possibility of sophisticated configuration for more demanding situations.

Rubean, A.G.

Frank Sommers reports on a banking system developed by Rubean, A.G., in an article at `http://www.artima.com/lejava/articles/banking_on_jiniP.html`. A group of German and Swiss banks with 35,000 customers run a centralized Java 2 Platform, Enterprise Edition (J2EE) system. However, this system needs to interact with ATM machines, cash-dispensing machines, and other devices linked to PCs typically through an RS232 cable. These devices have differing capabilities, their PCs are not always on, and they have IP addresses assigned by the Dynamic Host Configuration Protocol (DHCP), so they are continually changing. This dynamic environment could be a maintenance nightmare to a centralized system. The solution by Rubean uses Jini services running on each PC, talking to the devices using the Java Comms API. Each service advertises itself to Jini lookup services, which can handle the dynamic nature of the services without needing configuration. The J2EE system just sees collections of services and is unaware of configuration details.

Magnetti Marelli

Magneti Marelli Motorsport builds monitoring equipment for Formula One racing cars, and its hardware and software is used by most racing teams. However, its software used to run on proprietary platforms and lacked the flexibility and robustness required. In the cars, sensors collect a great deal of information, which is relayed by radio links to teams in the pits. The environment in the pits is often hot and noisy, and real-time responses are needed to deal both with the data itself and with a changing environment (computers die, network cables are tripped over, etc.). New software was developed using not only the discovery mechanisms of Jini, but also its ability to self-heal the service environment. Being able to run in multiple operating system environments was also a help. The project is described at `http://wiki.javapolis.com/confluence/display/JP05/Formula+One+Telemetry+with+Java`.

Nedap N.V.

Nedap N.V. is a security management company used by over 6 million people every day. Some years ago, the company saw the need for a next generation of security systems and began to design one from the ground up. The heart of the system is a 64MB controller with an Ethernet connection to the network and CAN connection to devices. Each controller exposes itself to the network as a Jini service. Examples of use include the security system controlling elevators within the Eiffel Tower in Paris. Every time an elevator comes within wireless range of a Jini lookup service, it is discovered. This allows security access of an elevator to any floor to be controlled, with Jini dynamic service management handling access to services. This system is described at `http://www.jini.org/meetings/eighth/J8abstracts.html#Wegman`.

Orbitz

The previous examples seem to suggest that Jini is just for linking hardware systems into software systems. Jini is certainly good at doing this, since it makes the devices appear as services, making them first-class citizens in a service-oriented system. But that is not all that Jini is good for: it *is* a framework for service-oriented middleware, and it excels in purely software-based systems too. An example of such a system is Orbitz, a multibillion dollar online travel company based in the United States. Orbitz uses over 1,000 Linux servers, and it has both a changing set of machines and evolving applications running on them. For example, if a supplier is having a sale, Orbitz will allocate more services to that company. Obviously, reconfiguring applications to use new services could be an ongoing horrific task, but Jini dynamic discovery allows it to be done transparently. In addition, services can be upgraded in place and the new versions become automatically available. The Orbitz system is described here: `http://www.sun.com/software/jini/news/Jini_Orbitz_Profile_Final.pdf`.

Orange

Orange is a major mobile communications company, with over 44 million customers worldwide. It offers a range of services, many of them supplied by external organizations. Again, this is a dynamic service environment, with new services begin deployed and existing services being revised, all in a high-volume environment, and Jini manages many aspects of this dynamism automatically. For more information, see `http://www.sun.com/products-n-solutions/telecom/docs/orangesp_1.pdf.`

Jini Licensing and Apache

Jini for may years has been licensed under a Sun Community License. While generous in many respects (for example, Jini source code has always been available) it is, nevertheless, a proprietary license. Recently the Jini group has decided that this has possibly limited the uptake of Jini, and so early in 2006 the license was changed to the open source Apache license.

However, grander changes are underway. At the time of writing, negotiations are under way to turn Jini into an Apache "incubator project", and later perhaps into a top-level Apache project. This will turn Jini into a true open source project and hopefully will bring in a new set of users and developers, in addition to being build on, and contribute to, other Apache projects. Among the issues still to be resolved is the name of the project—since Jini is all of: a concept for middleware, a specification and API and an implementation, the name might change in moving to Apache. One possible name that has been suggested is Babylon. Associative thinking leads to "Rivers of Babylon", then to "disco", and from there to "discovery"—which is one of the principal features of Jini :-)

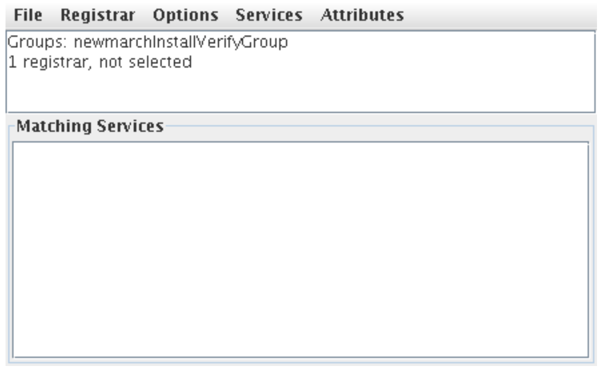
Until recently, the primary site for Jini was `http://www.jini.org`. A new site has been added `https://jini.dev.java.net/` and most of the projects have been moved to there. In the future, it is hoped that Jini will be on the Incubator's page on the Apache site at `http://incubator.apache.org/` and later on perhaps will have its own top-level page on the main Apache site.

Jini in One Hour

Our homes are becoming full of more and more complex pieces of electronic equipment—TVs, microwaves, stereo systems, and so forth—and many of these items have clocks. It's pretty common to find one or more of these clocks flashing; particularly the one on the VCR. Whenever there is a power failure, all the clocks on these pieces of equipment start up again with an incorrect time and signal this by flashing until they are manually corrected. In addition, many offices synchronize the clocks in a building to a central time server, but this is a luxury most homes do not have. But wouldn't it be nice if every clock in the house could find a correct, central clock and set itself? In this section, we'll walk through the steps of setting up a Jini system that will do this for some "software" clocks. No real clocks currently support Jini, but it would be nice if they did!

The first step is to get the Jini classes, which you can download from `http://www.jini.org` by following the "Getting Started" link to the Jini Downloads page. This page has several options; you only need the Jini Technology Starter Kit for now, but you may like to download other projects as well.

The Starter Kit installation process for platforms such as Windows and Linux will check your network settings and Java installation, and start up a key component of Jini called a *registrar.* Once you see a window called Service Browser with the message "1 registrar, not selected," you'll know this key component is running and you can start up some services and clients. You should see something similar to Figure 1-1.



```
Groups: newmarchInstallVerifyGroup
1 registrar, not selected
```

Figure 1-1. *Service Browser window showing the registrar running*

At later times, you can get to this page by just running the command `LaunchAll` from the `installverify` directory.

At this stage, you will use three files in the Jini directories:

- `jsk-lib.jar`, in the `lib` directory, contains many of the Jini classes.
- `jsk-platform.jar`, in the `lib` directory, contains more Jini classes.

- `jsk-all.policy`, in the `installverify/support` directory, controls Jini security, and here just turns it off. (This is OK for the purposes of this demonstration, but not for real systems!)

As you already know, this book is all about how programmers can build Jini services and clients. The flashing clocks problem has all the source code explained in Chapter 18. For now, you can download `.jar` files containing all the compiled classes from `http://jan.netcomp.monash.edu.au/java/jini/tutorial/programs.zip`. Unzip the files into any directory you want. Three `.jar` files are of interest, under the `dist` directory:

- `clock.clock.ticker.jar`: This file contains the class files for a standard "dumb" clock that starts off with a random time and just ticks away. However, it is smart enough to look around the network to see if there are any other clocks it can synchronize with.
- `clock.clock.computer.jar`: This file is a "smarter" clock that gets its time from the built-in computer clock, which we assume has the correct time. This clock also looks around the network to see of there are clocks that should synchronize with it.
- `clock.clock-dl.jar`: This file contains special classes that can be downloaded across the network. Systems like CORBA and web services rely on getting references to remote services and making calls using specific protocols by these references. Jini, on the other hand, relies on downloading Java classes representing a service: once a client has these classes, then it just makes local calls and doesn't care how the downloaded classes talk to the service.

When clients find services, they download a proxy for the service. Support code for this proxy is usually in a `.jar` file on an HTTP server. So the file `clock.clock-dl.jar` has to be on an HTTP server somewhere. You can copy this file to an HTTP server you have access to, or you can just use the file on the HTTP server that I run at `jan.netcomp.monash.edu.au`. (If you have a firewall between my server and your computers, then it may be easier to put the file on a local server than to get Java to talk through the firewall. You *can* get away with not using these classes in this example; the clocks will work fine, but the service browser won't see the services properly.)

That's all you need to get this demonstration working. You can start up a flashing clock by running Java from, say, a command box under Windows or a terminal window under Unix. You will need to set your classpath so that it contains the Jini files `jsk-platform.jar` and `jsk-lib.jar`, and also the clock file `clock.clock.ticker.jar`. For example, under Unix you could run

```
JINI_HOME=...
CLOCK_DIR=...
CLASSPATH=$JINI_HOME/lib/jsk-lib.jar:$JINI_HOME/lib/jsk-platform.jar:$CLOCK_DIR/
clock.clock.ticker.jar
export CLASSPATH
```

and under Windows, you could run

```
set JINI_HOME = ...
set CLASSPATH = %JINI_HOME%/lib/jsk-lib.jar;%JINI_HOME%/lib/jsk-plat-
form.jar;%CLOCK_DIR%/clock.clock.ticker.jar
```

After setting the classpath, run a dumb ticking clock:

```
java \
     -Djava.rmi.server.codebase=http://jan.netcomp.monash.edu.au/classes/
clock.clock-dl.jar \
     -Djava.security.policy=JINI_HOME/installverify/support/jsk-all.policy \
     clock.clock.TickerClock \
     "Ticking Clock"
```

where `JINI_HOME` is replaced by the directory name where you installed Jini. The first parameter (`codebase`) lets the service tell clients where the downloadable files are; the second parameter (`security`) sets the policy for what remote code is allowed to do to this service. The third parameter is the `main` class file, and the last parameter is just a string to be displayed as window title.

You should see a clock like the one shown in Figure 1-2, flashing every second.

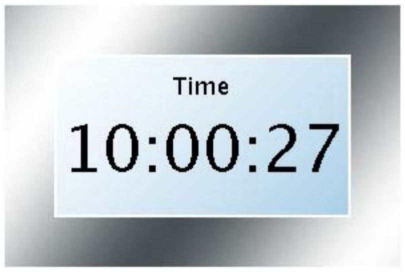


Figure 1-2. *Jini service browser*

You can run this command as often as you want, on the same or different machines. Each one should start up a new flashing clock. These clocks will all discover one another, but since none of them shows a valid time, there is nothing they can do to each other.

Now start up a "smart" clock that is showing the right time. The classpath needs to be set to the Jini files `jsk-platform.jar` and `jsk-lib.jar` again, but this time it should include `clock.clock.computer.jar` instead of `clock.clock.ticker.jar`. The you run the good clock as follows:

```
java \
     -Djava.rmi.server.codebase=http://jan.netcomp.monash.edu.au/classes/
clock.clock-dl.jar \
     -Djava.security.policy=JINI_HOME/installverify/support/jsk-all.policy \
     clock.clock.ComputerClock \
     "Computer Clock"
```

As this one starts up, it will discover the other clocks and they will discover it. The wrong clocks will ask the right clocks for the correct time; the right clocks will tell the wrong clocks to reset their time. This is a peer-to-peer system, and I don't know whether right tells wrong the correct time or wrong gets the correct time from right—it doesn't matter. All that matters is that the correct time will soon show on all clocks. Later, the clocks will "drift," but after reading Chapter 18 you will easily be able to add code to resynchronize on a regular basis.

If you now start up another possible flashing clock, it will quickly discover the other correct clocks and may not even flash at all.

So, what is going on with these clocks that is valuable to a distributed application's programmer?

- The clocks demonstrate *discovery*. New clocks start and both discover and are discovered by existing clocks. This is a general property of Jini: clients discover services they are interested in.
- A clock can make a call on another clock to get or set the time. The clocks are making remote method calls, but as you will discover later, the *protocol* isn't specified by Jini: all each clock knows is that it has a local proxy representing the remote service and is making local calls on that proxy. How the proxy talks to its service is of no interest to the client. Of course, it *is* of interest to the service programmer, and Jini allows the service programmer full control of how this is done, while giving default mechanisms good enough for many cases.
- Some clocks can crash and the others will carry on. Well, OK, there isn't much interaction going on. But a clock can crash *after* but *before* being called. Jini will throw exceptions to signal failed calls so that the client programmer can handle failure.
- While method calls are synchronous, Jini also allows events to be generated and delivered asynchronously to listeners. When a clock changes state, it can inform any interested listener. So Jini can handle both synchronous and asynchronous method calls.

Finally in this section, let's look at pseudocode for the clocks:

```
main:
    allow remote code to be downloaded and run within this VM
    start a thread to asynchronously discover proxies for clock services,
        calling us as listener
service discovered:
    if we are invalid and the remote clock is valid
        set our time from the remote clock
        set state to valid
    else if we are valid and the remote clock is invalid
        set the time on the remote clock
```

That's it! The rest of the clocks' code (less than 700 lines total) is the service specification, user interface classes, and code to keeping the clocks ticking.

Components

When running Jini system, you are dealing with three main players: a service, a client, and a lookup service. The *service* could be something such as a printer, a toaster, a marriage agency, and so forth. The *client* would like to make use of this service, and the *lookup service* acts as a broker/trader/locator between the service and client. (The generic term for the lookup service seems to be settling on *service cache manager*.) An additional component is a *network* connecting all three main players, and this network will generally be running TCP/IP. (The Jini *specification* is fairly independent of network protocol, but the only current *implementation* is on TCP/IP.)

Code is moved around between these three pieces by *marshaling* the objects. Marshaling involves serializing the objects in such a way that they can be moved around the network, stored in a “freeze-dried” form and later reconstituted by using included information about the class files as well as instance data. This process is performed using Java’s socket support to send and receive objects.

In addition, objects in one Java Virtual Machine (JVM) may need to invoke methods on an object in another JVM. Often this will be done using RMI, although the Jini specification does not require this, and there are many other possibilities.

Figure 1-3 shows the components of a Jini system discussed in this section.

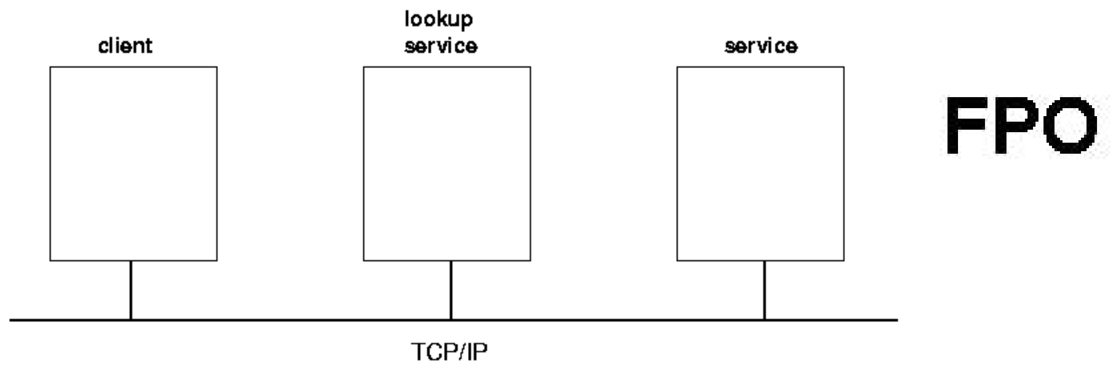


Figure 1-3. *Components of a Jini system*

Service Registration

As mentioned previously, a *service* is a logical concept such as a blender, a chat service, or a disk. It will usually turn out to be defined by a Java interface, and often the service itself will be identified by this interface. Each service can be implemented in many ways, by many different vendors. For example, there may be Joe’s dating service, Mary’s dating service, and any number of others. What makes them the “same” service is that they implement the same interface; what distinguishes one from another is that each different implementation uses a different set of objects (or maybe just one object) belonging to different classes.

A service is created by a *service provider*. A service provider plays a number of roles:

- It creates the objects that implement the service.
- It registers one of these objects, the *service object*, with lookup services. The service object is the publicly visible part of the service, and it will be downloaded to clients.
- It stays alive in a server role, performing various tasks such as keeping the service “alive.”

In order for the service provider to register the service object with a lookup service, the server must first find the lookup service. This can be done in two ways. If the location of the lookup service is known, then the service provider can use unicast TCP to connect directly to it. If the location is not known, the service provider will make UDP multicast requests, and lookup services may respond to these requests. Lookup services will be listening on port 4160 for both the unicast and multicast requests. (Port 4160 is the decimal representation of hexadecimal (CAFEBABE). Oh well, these numbers have to come from somewhere.) When the lookup service gets a request on this port, it sends an object back to the server. This object, known as a *registrar*, acts as a proxy to the lookup service and runs in the service's JVM. Any requests that the service provider needs to make of the lookup service are made through this proxy registrar. Any suitable protocol may be used to do this, but in practice the implementations that you get of the lookup service (such as those from Sun) will probably use RMI.

What the service provider does with the registrar is *register* the service with the lookup service. This involves taking a copy of the service object and storing it on the lookup service as shown in Figures 1-4, 1-5, and 1-6.

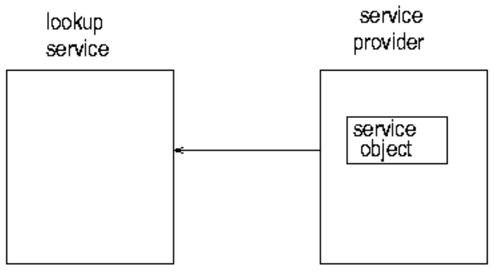


Figure 1-4. *Querying for a service locator*

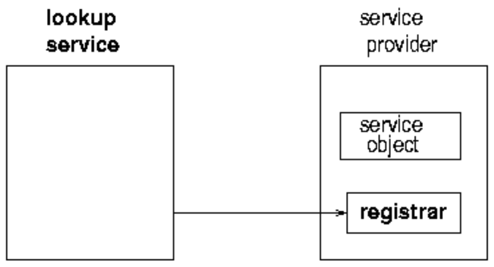


Figure 1-5. *Registrar returned*

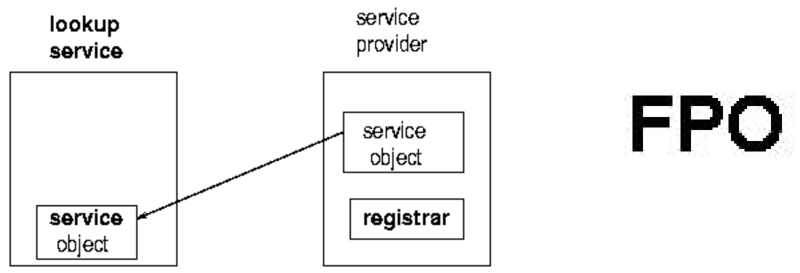


Figure 1-6. *Service uploaded*

Client Lookup

The client, on the other hand, is trying to get a copy of the service into its own JVM. It goes through the same mechanism to get a registrar from the lookup service. But this time it does something different, which is to request the service object to be copied across to it. This process is shown in Figures 1-7, 1-8, 1-9, and 1-10.

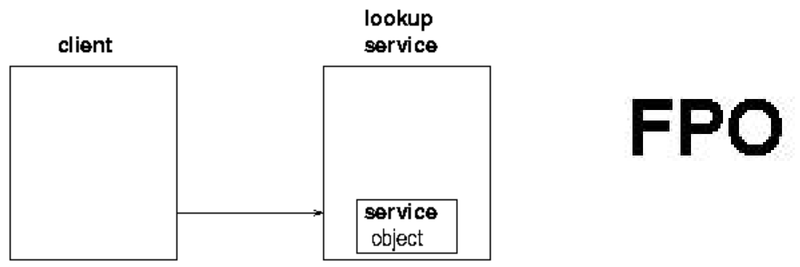


Figure 1-7. *Querying for a service locator*

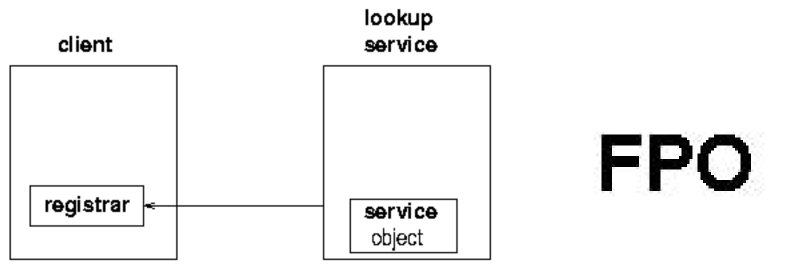


Figure 1-8. *Registrar returned*

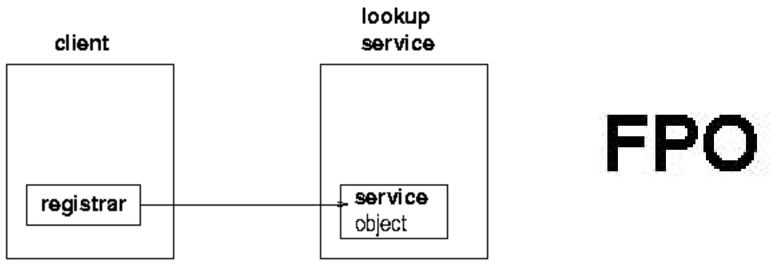


Figure 1-9. *Asking for a service*

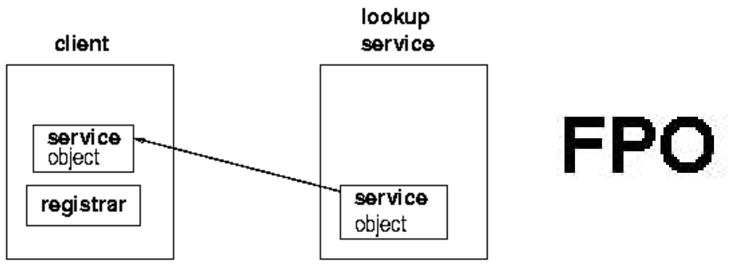


Figure 1-10. *Service returned*

At this point, the original service object is running on its host, there is a copy of the service object stored in the lookup service, and there is a copy of the service object running in the client's JVM. The client can make requests of the service object running in its own JVM.

Proxies

Some services can be implemented by a single object, the service object. How does this work if the service is actually a toaster, a printer, or is controlling some piece of hardware? By the time the service object runs in the client's JVM, it may be a long way away from its hardware. It cannot control this remote piece of hardware all by itself. In this situation, the implementation of the service must be made up of at least two objects: one running in the client and another distinct one running in the service provider.

The service object is really a *proxy*, which will communicate back to other objects in the service provider, probably using RMI. The proxy is the part of the service that is visible to clients, but its function will be to pass method calls back to the rest of the objects that form the total implementation of the service. There isn't a standard nomenclature for these server-side implementation objects. I will refer to them in this book as the *service back-end* objects.

The motivation for discussing proxies is when a service object needs to control a remote piece of hardware that is not directly accessible to the service object. However, it need not be hardware: there could be files accessible to the service provider that are not available to objects running in clients. There could be applications local to the service provider that are useful in implementing the service. Or it could simply be easier to program the service in ways that involve objects on the service provider, with the service object being just a proxy. The majority of service implementations end up with the service object being just a proxy to service back-end objects, and it is quite common to see the service object being referred to as a *service proxy*. It is sometimes referred to a the service proxy even if the implementation doesn't use a proxy at all!

The proxy needs to communicate with other objects in the service provider, but this begins to look like a chicken-and-egg situation: how does the proxy find the service back-end objects in its service provider? Use a Jini lookup? No, when the proxy is created it is "primed" with its own service provider's location so that when run it can find its own "home," as shown in Figure 1-11.

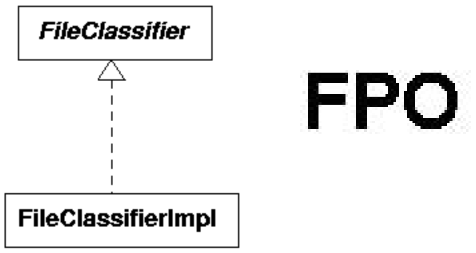


Figure 1-11. *A proxy service*

How is the proxy primed? This isn't specified by Jini, and it can be done in many ways. For example, an RMI naming service can be used, such as `rmiregistry`, where the proxy is given the name of the service. This isn't very common, as RMI proxies can be passed more directly as returned objects from method calls, and these can refer to ordinary RMI server objects or to RMI activateable objects. Another option is that the proxy can be implemented without any direct use of RMI and can then use an RMI-exported service or some other protocol altogether, such as FTP, HTTP, or a home-grown protocol. These various possibilities are all illustrated in later chapters.

Client Structure

Internally a client will look as shown in Table 1-1.

Table 1-1. *Client Pseudocommand*

Pseudocommand	Where Discussed
`prepare for discovery`	Chapter 4, "Discovering a Lookup Service"
`discover a lookup service`	Chapter 4, "Discovering a Lookup Service"
`prepare a template for lookup search`	Chapter 5, "Entry Objects" and "Client Search"
`look up a service`	Chapter 7, "Client Search"
`call the service`	

The following code is a simplified version of a real case, with various checks on exceptions and other conditions omitted. It attempts to find a `FileClassifier` service, and then calls the method `getMIMEType()` on this service. The full version of the code is given in a later chapter. I don't provide detailed code explanations right now, as this example is just intended to show how the preceding schema translates into actual code.

```
package nonworking;
public class TestUnicastFileClassifier {
    public static void main(String argv[]) {
   new TestUnicastFileClassifier();
    }
    public TestUnicastFileClassifier() {
   LookupLocator lookup = null;
   ServiceRegistrar registrar = null;
   FileClassifier classifier = null;
        // Prepare for discovery
        lookup = new LookupLocator("jini://www.all_about_files.com");
        // Discover a lookup service
        // This uses the synchronous unicast protocol
   registrar = lookup.getRegistrar();
        // Prepare a template for lookup search
   Class[] classes = new Class[] {FileClassifier.class};
   ServiceTemplate template = new ServiceTemplate(null, classes, null);
        // Lookup a service
   classifier = (FileClassifier) registrar.lookup(template);
        // Call the service
   MIMEType type;
   type = classifier.getMIMEType("file1.txt");
        System.out.println("Type is " + type.toString());
    }
} // TestUnicastFileClassifier
```

Server Structure

A server application will internally look as shown in Table 1-2.

Table 1-2. *Server Pseudocommand*

Pseudocode	Where Discussed
`prepare for discovery`	Chapter 4, "Discovering a Lookup Service"
`discover a lookup service`	Chapter 4, "Discovering a Lookup Service"
`create information about a service`	Chapter 5, "Entry Objects" and "Client Search"
`export a service`	Chapter 6, "Service Registration"
`renew leasing periodically`	Chapter 8, "Leasing"

Again, the following code is simplified, with various checks on exceptions and other conditions omitted. It exports an implementation of a file classifier service as a `FileClassifierImpl` object. The full version of the code is given in a later chapter. I don't provide detailed code explanations right now, as this example is just intended to show how the preceding schema translates into actual code.

```
package nonworking;
public class FileClassifierServer implements DiscoveryListener {

    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();
    public static void main(String argv[]) {
   new FileClassifierServer();
        // keep server running (almost) forever to
   // - allow time for locator discovery and
   // - keep reregistering the lease
        Thread.currentThread().sleep(Lease.FOREVER);
    }
    public FileClassifierServer() {
   LookupDiscovery discover = null;
        // Prepare for discovery - empty here
        // Discover a lookup service
        // This uses the asynchronous multicast protocol,
        // which calls back into the discovered() method
        discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
        discover.addDiscoveryListener(this);
    }

    public void discovered(DiscoveryEvent evt) {
        ServiceRegistrar registrar = evt.getRegistrars()[0];
        // At this point we have discovered a lookup service
        // Create information about a service
   ServiceItem item = new ServiceItem(null,
              new FileClassifierImpl(),
                 null);
        // Export a service
   ServiceRegistration reg = registrar.register(item, Lease.FOREVER);
   // Renew leasing
   leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);
    }
} // FileClassifierServer
```

Partitioning an Application

Jini uses a *service* view of applications, in contrast to the simple object-oriented view of an application. Of course, a Jini "application" is made up of objects, but these will be distributed out into individual services, which will communicate via their proxy objects. The Jini specification claims that in many monolithic applications, there are one or more services waiting to be released, and making them into services increases their possible uses.

To see this, let's look at a smart file viewer. This application will be given a file name, and the structure of the name will determine what type of file it is (`.rtf` is Rich Text Format file, `.gif` is a Graphics Interchange Format file, etc.). Using this classification, the application will then call up an appropriate viewer for a given type of file, such as an image viewer or document viewer. A UML class diagram for this application might look like Figure 1-12.

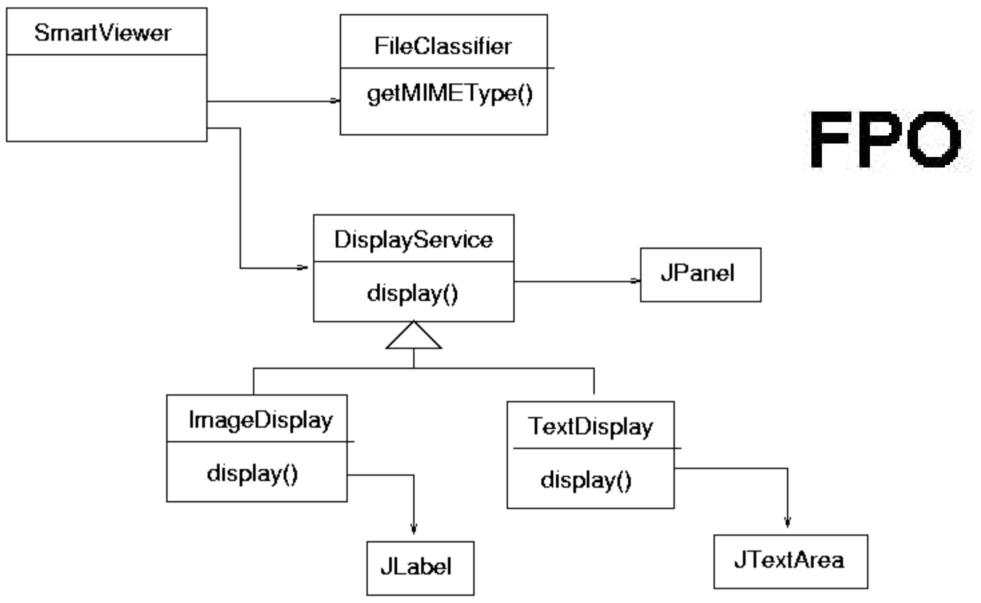


Figure 1-12. *A UML diagram for a smart file viewer application*

If we take a service-oriented view of the smart file viewer, then we can see a number of possible services in this application. Classifying a file into types is one possible service (which will be used heavily in the sequel, because it is simple). A file classification service can be used in many different situations, in addition to determining the file type for viewing contents of files. Each of the different viewer classes is another possible candidate for a service: an image display service, a text display service and so on. This is not to say that every class should become a service; that would be overkill. What makes these qualify as services is that they

- Have a simple interface
- Are useful in more than one situation
- Can be replaced or varied

They are *reusable*, and this is makes them good candidates for services. They do not require high-bandwidth communication and are not completely trivial.

If the application is reorganized as a collection of services, it might look like Figure 1-13.

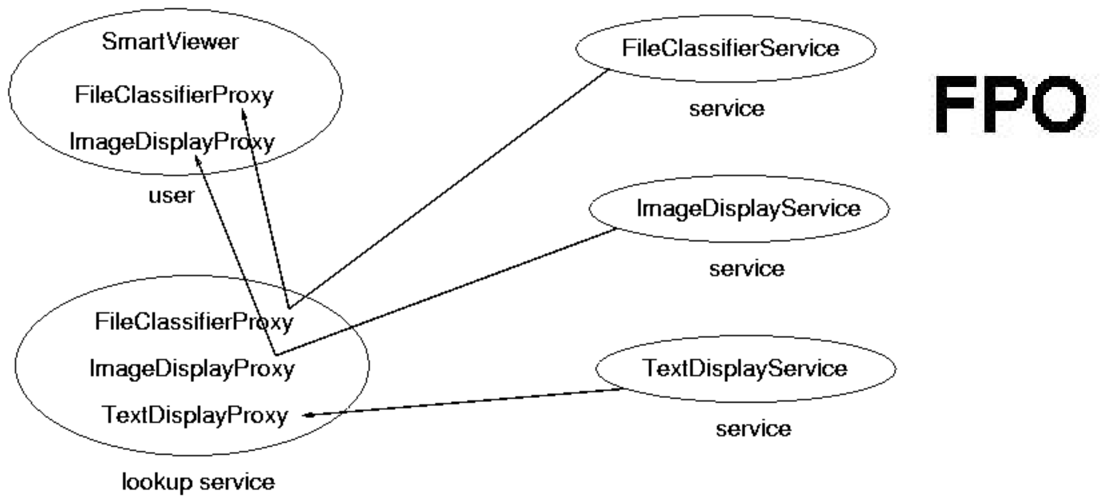


Figure 1-13. *An application as a collection of services*

Each service may be running on a different machine on the network (or on the same machine; it doesn't matter). Each service exports a proxy to whatever service locators are running. The `SmartViewer` application finds and downloads whatever services it needs, as it needs them.

Support Services

As previously discussed, the three components of a Jini system are clients, services, and service locators, each of which can run anywhere on the network. These will be implemented using Java code running in JVMs. The implementation may be in pure Java, but it could make use of native code by Java Native Interface (JNI) or make external calls to other applications. Often, each application will run in its own JVM on its own computer, although they could run on the same machine or even share the same JVM. When they run, they will need access to Java class files, just like any other Java application. Each component will use the `CLASSPATH` environment variable or use the `classpath` option to the runtime to locate the classes it needs to run.

Jini also relies heavily on the ability to move objects across the network, from one JVM to another. In order to do this, particular implementations must make use of support services such as an HTTP server. The particular support services required depend on implementation details, and so may vary from one Jini component to another.

The End of Protocols

Client/server systems built from scratch typically require the design of a communications protocol. For example, before the Web could became as important as it is today, the Hypertext Transfer Protocol, or HTTP, had to be designed so that clients and servers could communicate.

This protocol has been through several public versions: 0.9, 1.0, and 1.1. Clients and servers have had to be rebuilt on each version change of even this simple protocol.

RPC systems such as Sun's ONC, CORBA, COM+, and more recently SOAP also rely on a fixed protocol. In these cases, though, there are usually tools available that will generate code to manage the protocol messaging. However, these tools need to generate *two* sets of code: one for the *client* side and one for the *server* side. Any change to the protocol means that both client and server need to have this code regenerated.

This dependence on protocol is tightly bound to how clients address services, which depends on how they *find* these services. For example, to address a web service, you often need a Web Services Description Language (WSDL) document that contains the URL of the service's server and method names of the service. Coupled with knowledge that the service is addressed using SOAP, a client can then talk to it. With CORBA, you could obtain an object reference in a variety of ways (through a name server, using "stringified" references, or through a trader). Given this reference and the knowledge that CORBA uses the IIOP protocol, a client could then talk to a server.

Changes to a protocol are a nightmare once the protocol has become popular. It took years for all clients and servers to upgrade to HTTP 1.1, and the situation was similar for CORBA protocol changes. The protocol used by `sendmail` hasn't been touched for years because of the chaos to e-mail that would almost certainly result from any changes, even though many people think it is well past its use-by date.

When you discover a Jini service, you don't get an address to be dealt with by a particular protocol. Instead, you get a Java proxy object with known methods. This alters the playing field in a significant way: the client doesn't need to know the communications protocol *at all*! It just makes local method calls on the proxy.

The proxy comes from the server, and this is key to *client ignorance*. The client doesn't know the protocol used between proxy and service since it never has to know. The communications protocol is private to the proxy and the service. That means that the proxy and the service can use any protocol they wish, and it has no effect on the client at all. The proxy and service can change the protocol, and the client never knows.

There is often an assumption that Jini systems must be "all Java." This isn't quite true. Certainly, the client has to be a Java client, although even here there are caveats: strictly speaking, the client must be able to invoke a Java object through a JVM, and there are many languages that can now do this. The proxy needs to be JVM bytecode. Most likely, this bytecode is generated from Java source code, but not necessarily. But on the service side, who knows? The proxy can talk to any service it wants to, using any protocol it has chosen. For example, many people (including myself) have built proxies that will talk SOAP to UPnP devices or web services. The client is ignorant of the service language and has no need to care.

It isn't quite the end of protocols, unfortunately. Jini leads to the end of client knowledge of *invocation* protocols, but the client still has to *discover* the Jini proxy; a discovery protocol is still involved. However, the Jini discovery protocol is fairly lightweight and is customized to just this task rather than being a general-purpose protocol. Jini discovery involves a simple protocol to discover a lookup service and then a mechanism for downloading bytecode. In the last section, I pointed out the use of an HTTP server to deliver the proxy bytecode, but this is not a prescribed mechanism for Jini. It would be possible to use other mechanisms such as e-mail or FTP, although I don't think anyone has yet seriously considered doing this, since HTTP seems to be good enough so far.

Summary

In this chapter, you learned that a Jini system is made up of three parts:

- Service
- Client
- Service locator

Code is moved between these applications. A registrar acts as a proxy to the lookup locator and runs on both the client and service.

A service and a client both possess a certain structure, which is detailed in the following chapters. Services may require support from other non-Jini servers, such as an HTTP server.