

The Jini™ Architecture: Dynamic Services in a Flexible Network

Ken Arnold
Sun Microsystems, Inc.
1 Network Drive
Burlington, MA 01804
+01-781-442-0720
ken.arnold@sun.com

1. ABSTRACT

This paper gives an overview of the Jini™ architecture, which provides a federated infrastructure for dynamic services in a network. Services may be large or small.

1.1 Keywords

Jini, Java, networks, distribution, distributed computing

2. INTRODUCTION

The Jini™ architecture provides an infrastructure for defining, advertising, and finding services in a network. Services are defined by one or more Java™ language interfaces or classes. These types define a contract with the clients of the service.

For example, if a service supports a `Printer` interface it supports a `Printer` service. If that interface has a `printText` method, client processes that invoke the method will be able to print a text message, regardless of the implementation of the printing service.

Because the Jini architecture is defined in terms of the Java programming language, the type system used for service descriptions is universal—the Java virtual machine provides a single execution environment, no matter which platform hosts the virtual machine.

The Jini Lookup service provides a place for services to advertise their presence in a network. Services place a serialized proxy object into one or more lookup services. The types implemented by the proxy are the service types. A print service would register a serialized object that implemented the `Printer` interface. Each service's proxy object implements that interface in an appropriate way for the particular service. If a `Printer` service uses a PostScript

capable printer, the proxy object that implements the `Printer` interface will convert the text to PostScript and send it to the printer. A different printing service whose printer uses PCL as a printing language will have a proxy object that converts the text to PCL commands.

The universal platform provided by the virtual machine also means that code can be downloaded to the client if necessary. This means that the client can use services whose implementations were previously unknown. If a client using the PostScript capable print service has never used it before, the conversion code can be automatically downloaded on demand by the virtual machine. Clients can talk to services that they have never seen before (and may never see again) without any human intervention, such as installing drivers on all the systems that will use a new printing service.

A discovery protocol lets clients and services find available lookup services for advertising in the local network, and lets clients find lookup services in which to search for services of desired types. Clients and services can also use specific lookup services (such as your home system's lookup service) from anywhere where the lookup service is reachable.

If you want to write a service you must have a Java virtual machine, but that machine need not be part of the device or software that is providing the service. A virtual machine anywhere in the network can perform the required functions on behalf of a small device or a legacy server. I will discuss some designs for such configurations.

3. GOALS

The Jini architecture [1] is designed to allow a service on a network be available to anyone who can reach it, and to do so in a type-safe and robust way. The goals of the architecture are:

- **Network plug-and-work:** You should be able to plug a service into the network and have it be visible and available to those who want to use it. Plugging something into a network should be all or almost all you need to do to deploy the service.
- **Erase the hardware/software distinction:** You want a service. You don't particularly care what part of it is software and what part is hardware as long as it does

Permission to make digital/hardcopy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 99, New Orleans, Louisiana
(c) 1999 ACM 1-58113-109-7/99/06..\$5.00

what you need. A service on the network should be available in the same way under the same rules whether it is implemented in hardware, software, or a combination of the two.

- **Enable spontaneous networking:** When services plug into the network and are available, they can be discovered and used by clients and by other services. When clients and services work in a flexible network of services, they can organize themselves the most appropriate way for the set of services actually available in the environment
- **Promote service-based architecture:** With a simple mechanism for deploying services in a network, more products can be designed as services instead of stand-alone applications. Inside almost every application is a service or two struggling to get out. An application lets people who are in particular places (such as in front of a keyboard and monitor) use its underlying service. The easier it is to make the service itself available on the network, the more services you will find on the network.
- **Simplicity:** We are aesthetically driven to make things simple because simple systems please us. Much of our design time is spent trying to throw things out of a design. We try to throw out everything we can, and where we can't throw something out, we try to invent reusable pieces so that one idea can do duty in many places. You benefit because the resulting system is easier to learn to use, and easier to provide systems in. Being a well-behaved Jini service is relatively simple, and much of what you need to do can be automated by other tools, leaving you with a few necessary pieces of work to do. Equally important, a large system built on simple principles is going to be more robust than a large complicated system.

4. THE JAVA PLATFORM

The Java platform [2,3,4] gives the Jini architecture a universal type system. The types are fully object-oriented, and understood the same way on all platforms. The Jini architecture relies upon several properties of the Java virtual machine:

- **Homogeneity:** The Java virtual machine provides a homogeneous platform: a single execution environment that allows downloaded code to behave the same everywhere.
- **A Single Type System:** This homogeneity results in types that mean the same thing on all platforms. The same typing system can be used for local and remote objects and the objects passed between them.
- **Serialization:** Java objects typically can be serialized into a transportable form that can later be deserialized.
- **Code Downloading:** Serialization can mark an object with a codebase: the place or places from which the object's code can be downloaded. Deserialization can then download the code for an object when needed.
- **Safety and Security:** The Java virtual machine protects

the client machine from viruses that could otherwise come with downloaded code. Downloaded code is restricted to operations allowed by the virtual machine's security policy.

Together these properties provide a single platform that can be trusted by users to download and execute code. When code can be downloaded, the network is more flexible—you can interact with implementations you have never seen before.

4.1 Object Oriented Types

Object oriented programming is very concerned with types. For the purposes of this paper I will describe the Java platform's notion of types. I will not distinguish between general and Java platform specific notions of types since the purpose is to introduce ideas are used in the Jini architecture with which the reader may not be familiar that, not to give an overview of object oriented programming in general.

The Java programming language requires each object to have a type. This type is named, and the type may contain methods (functions), which are the requests you can make of the object. In our `Printer` example, `Printer` is a type name, and `printText` is the name of one of its methods.

Types are organized into trees. The `Printer` interface can be implemented by, for example, a `PCLPrinter` class, which means that the `PCLPrinter` class implements all of the methods of the `Printer` interface (and possibly more besides). In this case the `PCLPrinter` object has more than one type—it is both a `PCLPrinter` object, and a `Printer` object, since it can respond to both `Printer` and `PCLPrinter` method calls. So any code that requires a `Printer` object can use a `PCLPrinter` object as well. The most common term for this feature is *polymorphism*, because a single object (the `PCLPrinter` object) can be viewed as having many (*poly-*) forms (*-morph*), in this case both as `PCLPrinter` and as `Printer`. In such a hierarchy `Printer` is called a *supertype* of `PCLPrinter`; conversely, `PCLPrinter` is a *subtype* of `Printer`.

The Jini architecture uses the polymorphism of object oriented types—a client can ask for a `Printer` object and get any kind of `Printer` object, including a `PCLPrinter` (which is, after all, a kind of `Printer`).

A single object can implement more than one interface. A combination printer and fax machine could create a single service object that implemented both the `Printer` and `Fax` interfaces. This object could be returned from searches for either type of object.

4.2 Serialization

Serialization is the process of taking the state of an object and turning it into a form that is transmissible to another system where it will be deserialized, thereby creating a new object that is equivalent to the original. Serialization on the Java platform encodes the actual type of the object, and deserialization can therefore recreate a new object of the same type. If necessary, the code can be downloaded to the

deserializing virtual machine if the type is one previously unknown to that virtual machine.

For example, a client that is written to use `Printer` objects already knows about the `Printer` type, but may not have used a `PCLPrinter` before. When it deserializes a `Printer` object and finds a `PCLPrinter`, the Java virtual machine will try to download the code for `PCLPrinter`.

5. THE LOOKUP SERVICE

The lookup service is the Jini architecture's corollary to a traditional distributed system's naming or directory service: it is the place where clients go to find services. Services are stored in a lookup service by a serialized proxy object.

When a service boots up or initially connects to a network, it typically will find a lookup service using a Jini Discovery protocol that sends messages to the local network(s) asking for available lookup services. Each lookup service found is sent a serialized object whose types are the advertised services. A print service will use a serialized `Printer` object of an appropriate type.

When a service registers its serialized proxy object, it is returned a lease. The lease is for an amount of time allowed by the lookup service. The service must renew the lease to sustain its presence in the lookup service. This mechanism maintains the freshness of the list of services—services that are down or no longer reachable due to network failures will not renew their leases and be dropped from the lookup service. So except for the “slop” time given by the lease, the list of services in the lookup service is a list of services actually available in the network.

When a client needs a service, it first contacts a lookup service. It either discovers the lookup service using a discovery protocol (just like a service), or talks to one directly using a URL-style identifier. Once the client has a proxy for the lookup service, it asks the lookup service to find one or more services that match a template. Templates define the client's requirement on the service including the types the client wants to use.

The lookup service uses object oriented type rules to match a search request (such as for the type `Printer`) against all the services currently registered (such as `PCLPrinter`, `PostScriptPrinter`, or `FaxAndPrinter`). The client may ask for a single matching proxy object, an array of matching proxy objects, or an array of service description information for interactive browsing of the lookup service's contents. For the purposes of this paper, let us suppose the simplest “any match” form of search is performed and a match of `PCLPrinter` is found.

The lookup service returns the serialized `PCLPrinter` proxy object. When the client deserializes the proxy, any necessary code will be downloaded to the client. (The lookup service does not store this code itself—the location of the code is stored in the serialized object in the lookup service. The service publishes its own code for the client to download.)

Then the client invokes methods, such as `printText`, on the proxy to send requests to the print service. The client is typically unaware of the details of the implementation of the particular proxy. It will invoke the `Printer` methods on whatever object it gets back. The specific proxy's code (`PCLPrinter`, `PostScriptPrinter`, or whatever) will implement the `Printer` methods as appropriate for the given service.

5.1 Lookup Attributes

Clients and people will sometimes need to distinguish between services by something other than their type. For example, if you have two printers, you may care which of them you print something on. This notion of “location” can be as simple as a printer name (fine for a small office or workgroup) or very complex (in a large company it may include the country, city, building, floor, department, and office number). Or a user may be looking for a printer that has A4 or legal-sized paper loaded. These are not directly expressed in the service type.

A Jini service can be stamped with one or more *attributes*. These attributes may be stamped on it by the local administrator (such as a location) or maintained by the service itself (the sizes of paper currently available). These attributes can be used when a client searches the lookup service by specifying them in the service template. This constrains the search to services that have particular attributes, possibly with specified values.

The matching protocol for attributes is quite simple: you can say of an attribute that it must be present or that you don't care about it. If you say it must be present you can say of each field of the attribute a similar thing: it must be a particular exact value or you don't care about the value. Attributes are objects, and their type is also hierarchical—matching for an attribute may match a subtype of the attribute.

This is the same matching strategy used in JavaSpaces [1] and is an extension of the tuple matching model used in Linda [5]. This simple matching model, while not as powerful as a full database query language, is powerful enough for most uses that are required of it. And since it is simple and sufficient, it fits the simplicity goal of the Jini architecture.

5.2 Lookup Groups

Lookup services can be part of one or more groups. Services can be configured to register with lookups that are part of particular groups. Combined, these features allow you to group sets of services into logical sets governed by particular lookup services in the network.

For example, if you have a conference room next to some offices you probably want the people in the conference room to use the printer in that room, not the one for the adjacent offices, and you almost certainly don't want the people in the offices printing on the conference room's printer. You can accomplish this by running two lookup services, configuring one to be in the “conference room”

group and the other to be in the default (public) group. When a service is installed in the conference room, you configure it to register only with “conference room” lookup services. Other services will join the other lookup service.

When people come to the conference room they can use the “conference room” lookup service to find only those services in the conference room. The people in the surrounding offices will use the default (public) lookup service to find services. This kind of separation allows services to be administratively grouped into practical sets.

6. THE VALUE OF A PROXY

Implementation hiding, also known as *encapsulation*, is standard object oriented philosophy. It frees the designer of the print service to design a good programming API for network printing, rather than a good network protocol.

In traditional distributed computing systems, an abstract interface definition (commonly expressed in an interface definition language such as IDL [6,7]) describes the methods that a remote server understands. This description defines a wire protocol—a method in an IDL interface defines bits that will be transmitted across a network to a remote server. Once this interface is defined all servers must be able to receive and execute the method calls. If the `Printer` interface in IDL contains a `printPostScript` method, then every printer that supports the interface must be able to receive and understand PostScript page descriptions.

Network protocols are thus very rigid in nature—they define exactly and only what they were originally designed to define, and they place strong requirements at the receiving end of the messages. The receiver must either understand PostScript or at least be able to find a way to forward incoming PostScript to a translator that *can* understand it, translating it into (say) PCL. Because an IDL description defines wire protocols, it defines either capabilities (the printer must understand PostScript) or bottlenecks (the printer must find some extra, remote translator process).

Defining network services at the API level is much more flexible when combined with downloaded code. The proxy that implements a `Printer` interface in a Jini system can be small or large, simple or complex. A `printPostScript` method can be implemented as a simple pass-through for PostScript printers or as a PostScript-to-PCL translator for PCL printers, or the proxy might search for a PostScript-to-PCL translating service elsewhere in the network. Or it might choose a completely different strategy.

The ability to add a layer of client-side code allows the designers of remote services to concentrate on what makes a good programming API for clients of a network printer, rather than what makes a good wire protocol. In a Jini system the wire protocol designs are left to the implementors of each service, and need not be agreed upon among vendors. Only the API must be standardized, and only to the point of common functionality. A company can extend the functionality of its printer services by adding

methods to its own printer proxy objects, as long as it also supports the standard `Printer` methods.

7. DISCOVERY PROTOCOLS

A discovery protocol is used for finding “nearby” lookup services in the network. Specific lookup services can be located by a URL-style lookup location identifier of the form `jini://host[:port]`.

Because IP is a common network, we have initially defined the IP discovery protocol. In this protocol, a newly-installed device sends out an initial “looking are lookups” multicast message with a (configurable) time-to-live in network hops. These messages contain the groups that the service is configured to join. These messages are sent at most ten times by the service at boot time. After that the service is passive.

Lookup services listen for these multicast messages. When received, they are examined to see if the lookup service is managing any of the groups the service wants to join. If it does, the lookup service sends its direct URL-style name to the service, and the service engages in the unicast join protocol using that URL.

Lookup services also intermittently multicast “here I am” messages of their existence, including the groups they manage. This way if a service cannot reach a lookup service when it is booted, the service will receive a “here I am” message after the service becomes reachable and then register with the lookup at that point.

The unicast join protocol uses the URL-style location definition to get the host and port of the lookup service. It then connects to it, asking for a `ServiceRegistrar` proxy. The `ServiceRegistrar` interface is the primary interface of the lookup service. It is used just like any other service proxy in the Jini architecture: the implementor of the lookup service defines how the `ServiceRegistrar` methods (such as the `register` method) are implemented in the proxy. The service downloads the proxy code when it deserializes the lookup services’ proxy object. This gives the same flexibility to the designers, as well as the implementors, of the lookup service that the proxy model gives to all other service designers and implementors.

We have defined the discovery protocol for IP networks. This is the first, but by no means the last, Jini Discovery protocol. Other networks will require different discovery protocols based on different addressing and messaging mechanisms. We invite people who require discovery protocols for other networks to initiate and/or participate in the design of those protocols.

8. LOOKUP ROBUSTNESS

The combination of the discovery protocols, defined service behaviors, and leasing makes the lookup model particularly robust in the presence of network failures. Let us take a few interesting cases to illustrate:

- Suppose that the network is up, and has a running lookup service. All the services in the network will

register. Now suppose a network partition isolates some part of the network. The services in that part of the network will be unable to renew their leases and so will soon drop out of the lookup service, preserving the general freshness of the service list.

- Now suppose the network is fixed. When the lookup service sends its next “here I am” message, the services that were isolated will see a lookup service in which they are not currently registered. They will register with that lookup service, adding them back in to the list of available services. This heals that list with no human intervention. Services drop back in as automatically as they drop out.
- People often ask about replication of the lookup service. Our initial example implementation of the lookup service is not replicated, but there is a simple workaround: Start a second lookup service in the network. When the new lookup service starts up, it will send out a “here I am” multicast of its own. When it does so, the services will see a lookup service in which they are not registered and then register. This will give you two separate lookup services that both have all the available services registered. This requires no explicit replication strategy in the lookup service itself: the two lookup service implementations can be completely different and unaware of each other. The replication is as automatic as the self healing; it comes from the same service behavior.
- If you want replication for “fail over” recovery from a crash of the lookup service itself, you can do this even after the crash. The second lookup service’s “here I am” message will cause the same registration behavior. This is, effectively, *post facto* fail over, with no pre-planning required.

9. SMALL DEVICES AND LEGACY CODE

Services are composed of proxy objects and the network entities they must use to fulfill the contracts defined by the service types they implement. We have been talking consistently about printers in this paper because printers are a common piece of equipment with which we are all familiar. But printers are relatively powerful devices compared to many devices which one might want to make available in a Jini system. What about smaller devices, such as pagers, phones, and washing machines? At current cost levels one would probably not want to add a Java virtual machine to each of these devices.

The Jini architecture requires a Java virtual machine to be present for each service. It places no requirements at all on where that virtual machine should be placed, nor on how many services might share a single virtual machine. Let us consider, for example, a cellular phone as a service that does not include a virtual machine, but which you want to make available on Jini systems.

One design strategy would be this: you can create a docking station into which the phone will be plugged in order to be attached to a Jini system. The dock will contain a Java

virtual machine that will, when the phone is plugged in, engage in the discovery protocol and otherwise handle all the Jini service related requirements. In effect the phone plus its dock are a Phone service in a Jini system.

This requires a dock which can be sold separately. In fact, it can be created by a third party not connected to phone’s manufacturer (excepting, of course, any relevant legal requirements, which the Jini architecture does not, itself, resolve). The third party manufacturer can create this device to address a perceived niche market, or to compete with the phone’s own dock.

A software solution will work as well. A simpler plug, rather than a dock, can raise a signal when a phone is connected. A software server running on a connected computer can detect this signal and fire up its Jini service when the phone is plugged in. In this case a single Java virtual machine might perform the Jini service duties for any number of such phones. This might be more cost effective in many environments. And again, the software solution could be provided by a third party.

Legacy servers and equipment can be incorporated in the same way. Anybody can write a proxy for a device or server, as long as there is some way for the proxy to communicate with the device or server to provide the proxy’s advertised service.

The fact that being a Jini service is relatively simple comes back to help us here. If you have a piece of equipment or legacy server that you want to integrate into a Jini system, you can ask the manufacturer for help, or find a third party that sells a proxy solution, or hire a consultant to write one, or write one yourself. The Jini technology part of the proxy will be fairly simple to write (we believe, from experience, that it takes about two weeks to the Jini service part of a proxy after you are familiar with the system). Simplicity gives you several alternatives when you need to integrate a non-Jini device or server into a Jini system.

Combined with the flexibility of the downloadable proxy, this makes the Jini architecture an attractive infrastructure for standardization of device administration and data collection. Rather than defining what network protocols various devices must all share, which often defines hard requirements on the network types and speed capabilities of the hardware involved, companies can agree on a reasonable client-level programming API for the device type and let the separate virtual machine proxies handle the work for the smaller versions of these devices. As hardware prices come down and it becomes easier to install more capabilities in the devices, the programming API—the Jini service definition—does not have to change. Only the proxies for the newer, more capable devices will change, and the clients that use them will automatically adapt. A programming API standard is more likely to be robust over time than a network-level protocol standard.

10. SOME EXAMPLES

To give this all some concrete instantiation, here are some

possible uses of the Jini system:

- You could design a kiosk that allowed the user to download information. For example, I might plug my PDA into the kiosk and ask the kiosk for directions to someplace. The kiosk can publish the information as a simple `TextPublisher` service, which I would use to download the directions onto a text device such as a pager, as well as an `HTMLPublisher` service which I would use to download them onto a more capable device, such as a PalmPilot.
- You could have expense sources (such as a taxi meter or credit card scanner) provide an `ExpenseSource` service that my PDA could use to download expense details. When I return to my office my PDA could be its own `ExpenseSource` service that my spreadsheet or company expense report software could use as a source for expense report information.
- You could make sensors in a gas supply system be Jini services, and have several monitoring and report generating applications adapt automatically to new sensors that are added to the network. Adding a new sensor would then be as simple as plugging it into the network: the monitoring applications would find the new service and incorporate it into the data flow. New “sensors” could be software services that aggregate and analyze information from sensors into higher-level data. The clients will be blissfully unaware of this hardware-software distinction.

11. CONCLUSION

The Jini architecture provides a platform that is robust in many dimensions:

- It is robust in the face of network failures: the set of services automatically adapts the actual state of the network and service topology.
- It is robust in the face of changes in the composition of services: as long as the service interface is implemented, the details of the service implementation can change as you buy new equipment and as equipment generally becomes more capable.
- It is robust in the face of old services: it is relatively easy to incorporate old devices and servers seamlessly instead

of leaving them as an impediment to progress.

- It is robust in the face of competition: the minimum standards necessary for cooperation are defined in the architecture—the definition of what constitutes a service (a Java type) and how you find them (in a lookup service)—and lets variation exist where it needs to. An industry can standardize on common ground (such as the basic `Printer` interface) and individual companies can add specific features in company-specific interfaces (such as `MyCompanysPrinter`) for clients that want to use them, without breaking generic clients that only want the common `Printer` functionality.
- It is robust in the face of scale: Jini services can be very large or very small, and can work with small devices via a supporting virtual machine.

We feel that the Jini architecture is a solid base for designing networked device and service systems. As the world becomes more networked—as it does by the minute—the Jini architecture will be a robust platform on which to build networked devices and services.

12. REFERENCES

- [1] The Jini Architecture Team, <http://sun.com/jini/specs/>. See also Arnold, K., O’Sullivan, B., Scheiffler, R.W., Waldo, J., and Wollrath, A. *The Jini Specification*, Addison-Wesley, in press.
- [2] Arnold, K. and Gosling, J., *The Java Programming Language, Second Edition*, Addison-Wesley, ISBN 0-201-31006-6.
- [3] Gosling, J., Joy, W., and Steele, G., *The Java Language Specification*, Addison-Wesley, ISBN 0-201-63451-1.
- [4] Lindholm, T. and Yellin, F., *The Java Virtual Machine Specification*, Addison-Wesley, ISBN 0-201-63452-X.
- [5] Carriero, N. and Gelernter, D., *How to Write Parallel Programs: A Guide to the Perplexed*, *ACM Computing Surveys*, Sept., 1989
- [6] The Object Management Group, *Common Object Request Broker: Architecture and Specification*, OMG Document Number 91.12.1 (1991)
- [7] Rogerson, D., y Microsoft Press (1997)