# THE **JINI ARCH**
# NETWORK-CENT

***A federation of spontaneously networked electronic components of all types can communicate, interact, and share their services and functions, as explained by Jini's lead architect.***

JIM WALDO

THE JINI™ ARCHITECTURE EXEMPLIFIES A NEW APPROACH TO COMPUTING systems—making the network the central connecting tissue. By replacing the notion of peripherals and applications with that of network-available services and clients that use those services, the Jini system breaks down the conventional view of what a computer is, while including new classes of devices in a unified architecture.

Jini technology assumes a changing network, in terms of both the components that make up the network and the way these components interact. Networks are generally long-lived entities that, as they grow to include ever-larger populations of users and machines, become increasingly difficult to upgrade as a single entity. That's why the Jini architecture is designed around support for incrementally upgrading network components (hardware and software) [9]. So, for example, installing a

ITECTURE FOR
RIC COMPUTING

network printer in a Jini environment involves simply plugging it into the network and turning on the power; removing it from the network involves no more than unplugging it.

Because it is designed for the network, Jini challenges the presuppositions that have shaped conventional thinking about computers and how software is written for them. The result is a system that offers considerable new power but is simpler to use and adapt than current systems.

Jini allows anything with a processor, some memory, and a network connection to offer services to other entities on the network or to use the services that are so offered. This class of devices includes all the things we traditionally think of as computers but also most of the things we think of as peripherals, such as printers, storage devices, and specialized hardware. In the near future, the definition will also encompass a host of other devices, such as cell phones, personal digital assistants, and microprocessor-controlled devices, such as televisions, stereo components, and even modern thermostats.

Making the network central requires a design that allows updates and changes to individual components without the wholesale shutdown of the network. Unlike a single machine, a large network cannot be shut down without great difficulty; updating the entire network is more difficult still. So the Jini system allows upgrades and updates to be installed and used by the components being networked without requiring that the network be shut down or all individual components be updated.

An additional result of building around the network is that the data and code running on any device in the network cannot be assumed by users or developers to have been built especially for that device. Indeed, given the longevity of networks and the rapid rate of change in small devices, the code and the information used on a particular processor is often constructed or gathered long before the processor is designed or built.

The combination of rapid change and long-running networks imposed another goal on Jini's designers. Users of a Jini-

TERRY MIURA

**Figure 1. Jini system structure, including its relationship with the Java language.**

| | Infrastructure | Programming Model | Services |
|---|---|---|---|
| Java | Java virtual machine<br>Java Remote Method Invocation | Beans<br>Swing graphics toolkit | Enterprise Java Beans<br>Java naming and directory services<br>Java transaction service |
| Jini | Discovery<br>Lookup service | Leasing<br>Transactions<br>Distributed events | JavaSpaces<br>Transaction manager |
| | **Lets objects find and communicate with one another** | **Adds simple APIs for remote objects and basic distributed computing** | **Everything else is a service** |

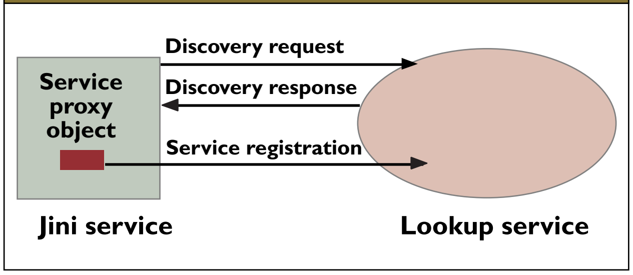


**Figure 2. Jini stores the proxy for a service in the Lookup service as part of registration**

based network should be able to add or remove member components without having to update other member components in the network community. Further, the way these components communicate with one another had to be able to change over time.

A final goal for Jini's designers was imposed by the size of today's networks and how rapidly they are growing. If we have all the embedded systems that could possibly be given access to a network as part of our system, Jini technology has to be able to scale to levels previously unthought of. (The specifications for the Jini system, along with the source code for the reference implementation, are at www.sun.com/jini.)

## A Simple Set of Conventions

Jini technology is not a distributed operating system (in the traditional sense) or an application. It is, in a classic sense, a system defining a small, simple set of conventions that allows services and clients to form a flexible distributed system that can change easily over time.

We separated the system's various components into the infrastructure, the programming model, and the clients and services themselves. While each of these components is logically independent, together they can use one another to make an overall system that is more flexible and reliable than the sum of its parts.

Each component of the Jini system can be viewed as a logical extension of the Java language system to the fully distributed case [1, 3] (see Figure 1). The Jini infrastructure is built on the Java Remote Method Invocation system, which has been part of the Java platform since the release of Java 1.1 in January 1997 [12]. On top of this base, Jini adds to the infrastructure two components: the discovery protocol, which allows an entity wishing to join a Jini network to find a lookup service, and the lookup service, which acts as a place where services advertise themselves and clients go to find a service.

The Jini programming model consists of three sets of interfaces meant to extend the usual single virtual machine programming model at the core Java programming libraries to allow the connection of distributed objects in robust ways. One set of interfaces defines a distributed event model that is an extension of the standard Java event model in Java Beans [7]. A second set of interfaces enables a two-phase-commit protocol—a simplified distributed version of the transaction model in the Java transaction service [8]. Finally, there is a set of interfaces and classes that define the notion of leasing, developed specifically for problems in resource allocation and reclamation in distributed systems.

The Jini system's services and the clients that use them are open-ended; the services offered depend on the Jini federation—the informal group of clients and services that use the Jini-defined interaction patterns—in question and the time one happens to be looking at the federation. The other parts of the system aid in offering and finding these services. The vast range of services that can live in the system includes hardware implementations of Jini interfaces, software services that act as distributed components, and hardware/software combinations.

## Jini and Java

The Jini system is Java-centric—because it builds on the existing Java environment and because it requires features that are widely available only within the Java platform. The Jini enabler is the ability of a service to move code into a client that wants to make use of that service. Such mobility is not unique to the Java environment; indeed, other systems, such as Inferno, a network operating system from Bell Laboratories [2], Telescript, an object-oriented programming language from General Magic, Inc. [11], and Tcl, a scripting language from Scriptics Corp. [6], have all been used for mobile code. However, the Java environment combines mobile code with other important properties that make exploitation of that mobility easy and safe.

Java's most basic property is that it turns an otherwise heterogeneous network of computing entities into a homogeneous collection of Java virtual machines. By ensuring a basic and consistent environment in which the Jini system can exist, services written in the Java language can provide implementations that run in the environment of the clients that want to use these services.

While the Java environment provides homogeneity with respect to the virtual machine and its basic class libraries, the resources on a particular machine can vary widely. But in a Jini environment, such resource variations are far less important than they would be in a more traditional mobile-code environment, since programs written for the Jini environment look for all such resources on the network, not only those on an individual machine.

A second property enabled by the Java environment is mobile object code. Most software engineers are accustomed to the notion of code portability, but it has always been source code that is portable. Java allows the byte codes the Java source is compiled into to be moved from machine to machine. This portability also yielded Java's "write-once-run-everywhere" slogan.

The write-once-run-everywhere property of Java byte codes, when coupled with the dynamic nature of Java, allows object code to be moved and dynamically loaded into a process even while the process is running. The Jini system uses the dynamic moving and loading of code to allow new functions to be introduced into a running program. This dynamic nature of the Java environment is a third enabler of the Jini system.

**EACH COMPONENT OF THE JINI SYSTEM CAN BE VIEWED AS A LOGICAL EXTENSION OF THE JAVA LANGUAGE SYSTEM TO THE FULL CASE.**

Being able to move code to a new system and dynamically load that code into a running process allows great flexibility, but it also introduces serious risk to the program accepting the code. Allowing code to be imported into a running process from possibly unknown sources requires a level of trust few users are willing to grant. However, Java's inherent security can be relied on to allow such trust.

Another property of the Java environment enabling Jini is its inherent safety—in the sense of referential integrity, array-bounds checking, and type safety. The Java security model also allows fine-grain control of the operations that can be performed by any code.

While the Jini system design assumes the Java environment will run on all components, Jini requires only that there be a Java virtual machine somewhere on the network. Components unable to run the Java environment can delegate functions requiring a Java virtual machine to this other machine [10].

## Spontaneous Networking

The Jini infrastructure, combined with the Java environment's ability to move code safely, allows the system to represent a spontaneous form of networking. Services and clients can join or leave a network federation anytime. More important, new and enhanced services can be introduced to extend the functionality of the networked federation.

The notion of a proxy is central to the Jini system—as well as to many other distributed systems. A proxy is a local object that stands in for the remote object. While presenting the same programmatic interface to the local code, the proxy deals with any network-related functions, transmitting any parameters to the remote service and receiv-

**Figure 3. Upon receipt of a proxy from a lookup service associated with Jini, a client can make requests of a service.**

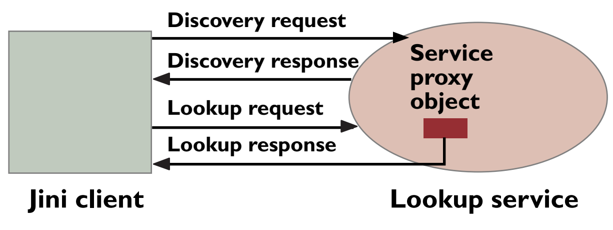
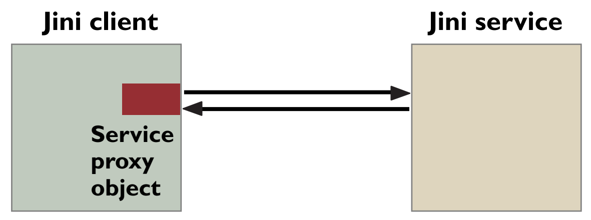


**Figure 4. The proxy and the service can communicate via whatever protocol they need.**



Network communication is between the proxy provided by the service and the service itself, without intervention from the lookup service.

ing any return values from that service.

A service (hardware or software) that wants to join a Jini federation sends out a packet, multicast over the LAN to a well-known port, asking for any lookup service to respond. The packet might specify that only lookup services within a particular (named) group respond, but in the simplest case, any and all lookup services on the local network would respond. The packet also contains the information necessary for any lookup service to respond to the requester.

Upon receipt of such a request, a lookup service responds by sending the requester a local proxy to the lookup service. This proxy, when loaded into the Java virtual machine running on the requester, contains enough information that, if the code needed for the proxy is not present at the site of the requester, it can be downloaded over the network. Therefore, the proxy returned to the requester is always matched to the lookup service that sent it (see Figure 2).

Upon receipt of a proxy for a lookup service, a service can register to offer itself for use to other members of the Jini federation by placing a proxy object of its own in the lookup service. If the service has received a response to more than one lookup service, it can register itself in any or all of these services.

Clients looking for a service find the lookup service in the same way—by multicasting a discovery request. Upon receipt of a proxy from any available lookup service, a client can request a service. Such a request takes the form of asking for an object implementing a particular Java language type. So, for example, a client could ask for something—either hardware or software—that implements a Java printing interface, rather than asking for something called a printer (see Figure 3).

This distinction is important, as it allows the lookup service to return a subtype of the requested type. If a client requests something of type `printer`, the lookup service is permitted to return a proxy for a subtype of the printer type, possibly a color printer. As in any strongly typed object-oriented system, this subtype includes all the characteristics of the requested super type, but also has additional behaviors. The client can then use these additional behaviors, which can be found through the Java language's reflection and type-discovery operations. If the client needs the code that implements the new behaviors, this code is downloaded when the proxy is recreated in the client's Java virtual machine.

Since the proxy used by the client to talk to a service is placed in the lookup service by the entity to which it communicates, the proxy can know details about the service to which it talks. All the client needs to know about the proxy is the Java interface it supports. This interface-based communication means that the proxy and the service can talk by way of whatever protocol they need. Further, the way the proxy talks to the service can change over time without the client's needing to be altered or even to be aware of the change (see Figure 4).

The ability of the service to move code into the client means that all the client has to worry about is the interface to the service. The actual implementation of the interface is a private matter between the service and the proxy supplied by the service. Since the client finds the service through its Java language type, the client knows the programmatic interface needed to talk to the service. This interface-based approach moves object-oriented programming techniques out of the address space and onto the network; all the client need worry about is what it needs done (expressed in the interface), not how it is done.

The ability to move code from the service to its client is the core difference between the Jini environment and other distributed systems, such as the Common Object Request Broker Architecture (CORBA) [4] and the Distributed Common Object Model (DCOM) [5]. In such systems, the code used to communicate with a service is associated with the client and knows how to transfer information to the service using a static protocol defined in terms of an interface definition language, which defines the information exchanged. This built-in knowledge means that any change in the communications protocol requires a coordinated change in both the client and the service.

Using mobile code, the Jini environment downloads the code used to communicate with the service into the client at the time the client wants to use the service. Changes in the protocol are private to the service and to the code sent to the client by that service. This privacy allows changes to be propagated as needed without the client's being aware of the changes.

The discovery protocol and the lookup service allow clients and service providers to join a Jini federation spontaneously. However, a genuinely spontaneous federation also requires that clients and services be able to leave the federation easily in a way that is not disruptive to its other members. The ability to leave is accomplished through the Jini programming model, which includes the notion of "leasing" resources. Services and clients can leave a Jini federation easily by way of the Jini leasing model.

Jini's leasing model introduces time into the allocation of resources. A Jini member component offering a resource does so through a lease, in a way that does not allow the resource to be used until it is explicitly released. The lease represents a period of time during which the resource is available. The client wanting the resource requests a lease period, but the actual period of the lease is determined by the grantor of the resource.

Even a lease that is handed out can be cancelled by the client holding the lease—if the client is finished with the leased resource before the lease expires. The lease can also be renewed—if the client requests it and the grantor agrees. However, when the lease expires, the lease grantor may free up the leased resource, and the lease holder knows the resource's availability is no longer guaranteed.

To see how Jini service leasing works, consider the Jini lookup service, which leases service registrations. Any service registering with a lookup service is granted a lease on that registration. Cancelling the lease is equivalent to unregistering with the lookup service. The service is expected to renew the lease for as long as it wants to be available to clients looking for that kind of service.

However, if the lease expires because it was not renewed (due to, say, a network failure, a service crash, or the service's being abruptly removed from the network), the service registration is dropped. Such lease expiration allows services to be removed from the Jini federation without warning or administrative intervention. Moreover, the federation can reflect the loss of that particular service in a timely fashion. This quick reflection of the loss of a service represents the other half of spontaneous networking, complementing the discovery and join functions.

**THE JINI SYSTEM AIMS TO PROVIDE THE MINIMAL SET OF RULES TO ALLOW CLIENTS AND SERVICES TO FIND EACH OTHER AND INTERACT.**

## Federation and Centralized Control

In the broadest sense, a Jini federation is defined by the set of services registered with a particular (set of) lookup service(s) and the clients using these lookup services to find these registered needed services.

But the term "federation" conveys more than just this limited definition. In a federated system of government, most of the power belongs to the local authorities, with federal authorities having only the authority to ensure local entities work together. Similarly, the Jini system aims to provide the minimal set of rules to allow clients and services to find each other and interact. By supplying the lookup service and the discovery protocol, a Jini federation dictates how its members join, leave, and find one another. And by dictating the use of the Java language's type system, the Jini federation dictates how services are identified. The Jini programming model codifies certain common styles of object interaction. Finally, by requiring that the proxy code for a service be downloadable, the Jini federation dictates how the services and their clients manage change in their implementations and the way they are extended.

Using a federated model instead of the usual model of centralized control (as in distributed operating systems) was a conscious decision by the Jini designers. While centralized systems can be optimized for some cases, changing them over time is difficult. More important, centralized systems do not scale well—and as Jini federations (and federations of federations) begin to emerge, they have to scale to very large numbers. The federated approach allows such scaling in ways not available to centralized systems.

## Disk-centric Computing

The concentration on the network, the ability to move code, and the federated nature of the Jini system result in an architecture that is fundamentally different from the one used for the past 50 years. Traditional computing architectures are built around three central components: the central processing unit, which does the computing; dynamic memory, where temporary results are stored; and a disk, which acts as stable storage. These elements have supported the computing industry since the earliest days of the stored-program computer in the late 1940s and early 1950s.

Given the changes in the technology, it is surprising that this architecture has remained so stable for so long. Each of the components has decreased in size and increased in capacity and speed during the 50 years of computing history, but their relationship has remained remarkably constant. The transitions from mainframes to minicomputers to workstations to personal computers changed many things about the way we build hardware and design software. But in all these transitions, the base of the computer itself has remained the CPU, dynamic memory, and a disk for stable storage.

We are so familiar with this kind of architecture, we tend to forget that there are implications to this approach. An obvious example is the nearly universal use of virtual memory, which allows part of the disk to be used to augment a computer's physical memory. The fact that virtual memory is part of nearly every commercial operating system is an outgrowth of the assumption that memory and disk are always present together.

There are more interesting, and subtler, implications of this architectural assumption. For example, because we assume that the processor is tightly associated with a disk, we store programs on disks that are compiled specifically for a particular kind of processor. Associating storage and processor leads to binary programs that are finely tuned to a particular kind of processor—and cannot be used on any other kind.

Moving to a network-centric design means that these assumptions cannot be made. The code that is to be run on a particular processor may come from any part of the network and therefore cannot be specialized for any particular kind of processor. Indeed, given the long-lived nature of networks, the code run on a particular processor may have been placed on the network long before the processor was even designed.

While processor-independence makes it impossible to specialize the code to a processor, the coupling between code and machine is far looser than has been the case historically. This looser coupling allows new services to be introduced into the network and inject their code into the processors that want to use the service for instant connection. This architecture makes the network the computer and allows connectivity to achieve a simplicity never before possible. C

**References**

1. Arnold, K., and Gosling, J. *The Java Programming Language.* Addison-Wesley Longman, Reading, Mass., 1996.
2. Dorward, S., Pike, R., Presotto, D., Ritchie, D., Trickey, H., and Winterbottom, P. *The Inferno Operating System*. Tech. Rep., Bell Laboratories; see www.lucent.com/ideas2/perspectives/bltj/winter_97/pdf/paper01.pdf.
3. Gosling, J., Joy, B., and Steel, G. *The Java Language Specification.* Addison-Wesley Longman, Reading, Mass., 1996.
4. Object Management Group. *Common Object Request Broker: Architecture and Specification.* OMG Doc. No. 91.12.1, 1991.
5. Rogerson, D. *Inside COM*. Microsoft Press, Redmond, Wash., 1997
6. Scriptics Corp. Tcl home page; see www.scriptics.com/.
7. Sun Microsystems, Inc., Java Beans documentation; see www.java.sun.com/beans/glasgow/.
8. Sun Microsystems, Inc. Java transaction service specification; see www.java.sun.com/products/jts/jts_090.pdf.
9. Sun Microsystems, Inc. Jini architectural overview; see www.sun.com/jini/whitepapers/architecture.html.
10. Sun Microsystems, Inc., Jini device architecture; see www.sun.com/jini/specs/deviceArch.ps.
11. Telescript Language Reference Manual; see www.science.gmu.edu/~mchacko/Telescript/docs/telescript.html.
12. Wollrath, A., Riggs, R., and Waldo, J. A distributed object model for the Java system. *Comput. Syst. 9,* 4 (Dec. 1996), 265–290.

**Jim Waldo** (jim.waldo@sun.com) is a Distinguished Engineer at Sun Microsystems in Burlington, Mass., where he is the lead architect of Jini, and an adjunct faculty member in the department of computer science at Harvard University, Cambridge, Mass.