Alive and Well: Jini Technology Today

Jim Waldo, Sun Microsystems

It's been nearly 18 months since Sun Microsystems launched the Jini connection technology. At the time of its launch, much enthusiasm surrounded Jini's promise that you could plug various "gizmos" into the network infrastructure and have them just work. While the excitement about the technology has diminished somewhat since Jini's debut, the original cause for all that excitement still remains. In fact, with all we've learned over the course of the past year and a half, there's more reason than ever to celebrate.

We wouldn't be celebrating so much, however, without all the Jini community's contributions. Sun originally released the Jini technology with a unique license, a "community source license" in which Sun makes both the source and binary code for the sample implementation available for use, inspection, and experimentation by developers who wish to join the Jini community.

The rules of that community are few, and mostly entail keeping the type-space clean and sharing profits from commercial applications of the technology with its originators. Most notable about the license, however, is that it lets developers use the source and ideas behind the Jini technology for research without obligation or limit.

The idea behind the license was to expand the number of developers who would try the technology, thereby giving Sun rapid feedback on the adequacy and usefulness of the approach. Jini's licensing plan proved effective. We are now a full Moore's-law generation beyond the initial release of the technology. In that time, we've learned several useful (and sometimes surprising) things about Jini.

A BRIEF OVERVIEW

As shown in Figure 1, the Jini technology is an approach to distributed systems that uses several Java properties. In particular, the Jini environment makes heavy use of Java's ability to move objects—including each object's data and code—from one Java environment to another. The Java environment, with its portable bytecodes, dynamic loading, code verification, and security allows such an approach straightforwardly.

The mechanisms for using this ability in Jini include a set of conventions for advertising and finding services on the network, a service that acts as a place to do such service advertising and finding, and a means for bootstrapping the system without human intervention.

Core services

In a Jini network, services advertise themselves by saying what Java language interfaces they implement. Clients look for services by specifying what Java language interfaces they need. This matching of client and service differs from more usual naming or directory lookups in that it allows the services to evolve over time without losing the ability to be used by clients that only expect the functionality that the service previously provided.

Advertisement and matching of client and service occurs in a *Jini lookup service*, a place where providers of a service can register what they provide and clients of services can look for what they need. The service actually stores the information needed to reconstruct a Java object—the proxy for the service in the client's virtual machine.

This approach means that all the client need know about the service is the interface the service provides; everything else (such as how the proxy and the service communicate, or what wire protocols are used, or even whether or not there is a networking component to the service) is hidden behind the implementation of the proxy object.

The Jini community is an ongoing experiment in trying to mix open source development techniques with industrial engineering development.

You find lookup services—and there may be many on any particular Jini-enabled network—by using the Jini discovery protocol. This is the one place in the Jini specification where the actual wire representation of anything is specified and fixed for a particular type of network.

Discovery protocol

When a client or service wishes to find the Jini lookup services on a network, it multicasts a packet with a particular form to that network. The packet includes the name of the lookup group, which simply identifies different sets of Jini lookup services. A group name of the empty string is the default, publicly available group. All the lookup services on the network in

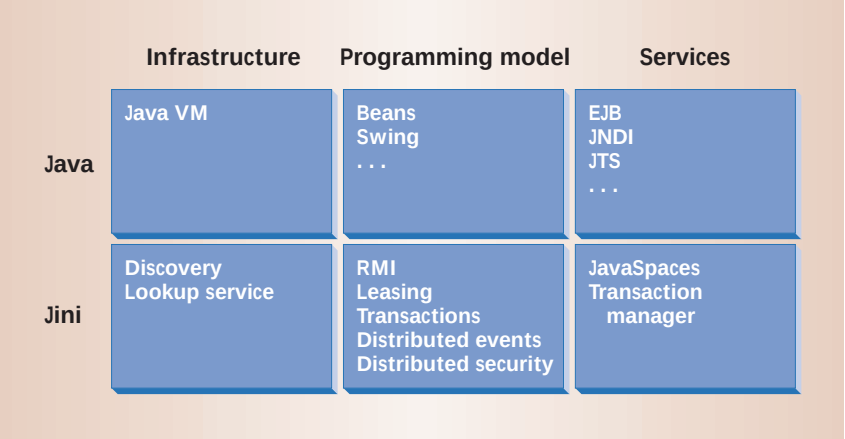

Figure 1. Jini extends Java's capabilities. When you let objects communicate and you add simple APIs for remote objects and basic distributed computing, everything else is a service.

that group respond to the object doing the discovery.

This response comes in the form of an object that implements the Jini lookup service interface, which is reconstructed in the requester's virtual machine and then used to communicate with the lookup service. Like all other Jini parts, the lookup service does not depend on any particular communication mechanism or wire protocol; these implementation details are hidden in the proxy object for the lookup service.

The first release of Jini included three sets of interfaces that define what we believed would be common interaction patterns between the client of a service and a service. These sets offered a way of receiving event notifications, a way of leasing resources for some period, and a two-phase commit protocol that would allow different services to be coordinated in a transactional fashion.

But the real core of the Jini technology is the dynamic combination of moving objects, including their implementation code, from the service to the client of the service on demand; the identification of services by their Java language type; the use of a lookup service or group of services for client-service rendezvous; and the mechanisms for discovering those lookup services for bootstrapping into the network without explicit administration.

NOT JUST FOR DEVICES

Perhaps the least surprising thing we learned in the year and a half since Jini's debut is that the Jini architecture has applications far broader than the device-connection technology toward which Sun originally targeted it. This discovery came as no surprise to Jini's developers, who had always envisioned the Jini approach as a general architecture for networks of computing devices based on services that could be implemented either as hardware or software.

Most demonstrations of the technology centered on connecting devices to the network, since this approach offered the most effective way of illustrating Jini's functionality. From these demos came the impression that Sun designed the Jini technology specifically for attaching devices to the network. But developers who began using the code soon found they could use the techniques for registering and finding software services on the network as well. Indeed, given the difference in manufacturing times between hardware and software, the first Jini services to be developed were these software services.

A central design feature of the Jini architecture is that there is no difference, from the client's point of view, between a hardware and a software service. Both are just ways of implementing an interface, which is all the client sees. The code moved into the client as the proxy for the service might be quite different, but the client's view is constant.

Devices

The connection of small devices into a Jini network actually requires extending the basic architecture. Since we based the interoperation of Jini services and clients on the ability to move Java bytecodes from the service to the client, many developers incorrectly assume that a Java virtual machine must be running on all members of the Jini federation. While much progress has been made in the design and implementation of small Java virtual machines, many interesting network devices are still incapable of hosting even the smallest JVM.

The original Jini specifications include what we call a *device architecture* to deal with the limited devices that can only supply the needed Java code as proxy for the device. This proxy code needs to know how to talk directly to the device, but having that knowledge means it isn't necessary for the device to host a Java virtual machine. Such devices would need to find a service that could register the proxy with any lookup services and maintain those registrations, but such a service would be fairly simple to provide.

Surrogates

While the device architecture approach has sufficed for a large variety of devices, it requires that the device allow some generic code to do all of the work for it in the Jini federation, such as registering with the lookup service, renewing leases, and the like. When developers wanted more flexibility, we developed a *surrogate architecture*, which extends the device architecture.

With the surrogate approach, a device can supply a Java language object with a known interface to a surrogate host service, found using a variant discovery protocol. On receipt of this object, the surrogate host will run the object, which uses the device as part of its own implementation and can then register as a Jini service and use other Jini services. Since the surrogate host offers an environment without memory constraints, runs a full Java virtual machine, and has access to all Java 2 classes, the object offered by the device can be considerably richer than objects in a vanilla device architecture.

SYSTEM ENHANCEMENTS

The surrogate design is one of several additions to the basic Jini architecture that we will include in Jini's next release, early versions of which have already shipped. The new version does not change the basic infrastructure but does provide a set of programming tools that make it easier to build Jini services and clients.

For example, the next release contains utilities (Java objects that can be incorporated into a Jini service or application) and services (standalone Jini entities). The utilities provide the functionality to find lookup services, register with them, or find their contents. The services will let you delegate the renewal of leases and the reception of event notifications, thus allowing unused Jini services or clients to deactivate themselves without fear of prematurely releasing needed resources or missing events of interest.

Perhaps the most significant functionality change is occurring not in the Jini technology but in the Java Remote Method Invocation (RMI) system that provides the communication model for Jini, and is used as the communication infrastructure for the sample implementation. The work going on with RMI adds full distributed security mechanisms to the RMI model and implementation, allowing mutual authentication, privacy of communication, anonymity, and delegation of rights within the RMI framework. This work, under review by both the Java and Jini communities, will form the basis of the security services that will be introduced into the Jini system in the future.

THE JINI COMMUNITY

While the Jini technology was itself an experiment in how distributed systems could be built, the Jini community is an ongoing experiment in trying to mix the benefits of open source software development techniques with the discipline of industrial engineering development. We had no idea what the response of the development community would be to such an approach, but our initial experience has been extremely promising.

The Jini users' mailing list and the archive it has generated have become one of the community's major assets. With more than 1,000 active members, the list has become the first level of support for both new users getting started and experienced hands running into the technology's limits. The community has provided tutorials, examples, discussion, and handholding that has gone far beyond what the original development group could have provided.

While the internal development team has been occupied with the next release and the addition of security, the Jini community has been busy working on other additions to the technology. Some industry groups have formed to specify interfaces for the services that will be provided by those parts of the industry.

> The Jini architecture has applications far broader than the device-connection technology toward which Sun originally targeted it.

Perhaps the most active of these is the group defining a set of interfaces for printing services in a Jini network. This specification has gone through a number of drafts within the group, and has been reviewed by the original Jini architecture group. At this point, the printing specification is going through what may be its final public review, after which it will become part of the Jini specifications.

A less formal group has also been engaged in defining the interfaces that can be used by a service to provide a user interface to that service. This group, known as the ServiceUI project, did most of their design work via e-mail, with a few face-to-face meetings at Jini community gatherings over the year. This group has also produced a specification that has been reviewed by the original Jini architects, and a final revision has been publicly reviewed. The specification and a sample implementation should be available by June.

These two projects show two very different organizing paradigms for the Jini community. The printing project has been run much more like common efforts within standards organizations to put together a set of interfaces that can be generally supported within an industry segment. The ServiceUI project, on the other hand, has had a much less formal organization, running much more like an open source community project. Both have made progress within the Jini community, which is also actively discussing how to pursue such projects in the future.

Jini Resources

Sun's Jini link: http://www.sun.com/jini/overview/index.html
The Jini community: http://jini.org/
Directory of Jini resources: http://www.litefaden.com/sv/jd/
Community source license: http://www.sun.com/981208/scsl/principles.html
Jini books: http://java.sun.com/docs/books/jini/

The Jini technology appears to be alive and well, both in terms of the number of new projects being undertaken with Jini and the number of application areas to which Jini is being applied. The Jini community, an experiment in allowing access to commercial source code, has shown itself capable of helping new users, mutually supporting its membership, and defining new parts of the technology. ✸

Jim Waldo is a distinguished engineer at Sun Microsystems, where he is the lead architect for Jini. Contact him at jim.waldo@sun.com.

Editor Info: Michael J. Lutz, Rochester Institute of Technology, Department of Computer Science, 102 Lomb Memorial Drive, Rochester, NY 14623; (phone) +1 716 465 2909; (fax) +1 716 475 7100; mjl@cs.rit.edu