June 1999
JavaWorld
FUELING INNOVATION

JINIOLOGY

Jini: New technology for a networked world

How Jini raises the level of abstraction for distributed systems programming

Summary
This article, the first installment of *JavaWorld*'s newest column, **Jiniology**, introduces Sun Microsystems's Jini technology; demonstrates how Jini enables services on a network to find each other through the processes of discovery, join, and lookup; and explains how Jini's architecture brings the benefits of object-oriented programming to the network. (*2,000 words*)

By Bill Venners

Welcome to the first installment of **Jiniology**, *JavaWorld*'s new column dedicated to distributed systems programming with Java, RMI (Remote Method Invocation), and Jini. In this first article, I'll present an introduction to Jini technology and show how Jini's architecture helps to raise the level of abstraction in distributed systems programming, effectively turning network programming into object-oriented programming.

Jini in context

Traditionally, operating systems have been designed with the assumption that a computer will have a processor, some memory, and a disk. When you boot a computer, the first thing it does is look for a disk. If it doesn't find a disk, it can't function as a computer. Increasingly, however, computers are appearing in a different guise: as embedded devices that have a processor, some memory, and a network connection -- but no disk. The first thing a cellphone does when you boot it up, for example, is look for the telephone network. If it doesn't find the network, it can't function as a cellphone. This trend in the hardware environment, from disk-centric to network-centric, will affect how we organize our software -- and that's where Jini comes in.

Jini is an attempt to rethink computer architecture, given the rising importance of the network and the proliferation of processors in devices that have no disk drive. These devices, which will come from many different vendors, will need to interact over a network. The network itself will be very dynamic -- devices and services will be added and removed regularly. Jini provides mechanisms to enable smooth adding, removal, and finding of devices and services on the network. In addition, Jini provides a programming model that makes it easier for programmers to get their devices talking to each other.

Building on top of Java, object serialization, and RMI, which enable objects to move around the network from virtual machine to virtual machine, Jini attempts to extend the benefits of object-oriented programming to the network. Instead of requiring device vendors to agree on the network protocols through which their devices can interact, Jini enables the devices to talk to each other through interfaces to objects.

What is Jini?

Jini is a set of APIs and network protocols that can help you build and deploy distributed systems that are organized as *federations of services.* A *service* can be anything that sits on the network and is ready to perform a useful function. Hardware devices, software, communications channels -- even human users themselves -- can be services. A Jini-enabled disk drive, for example, could offer a "storage" service. A Jini-enabled printer could offer a "printing" service. A *federation of services,* then, is a set of services, currently available on the network, that a client (meaning a program, service, or user) can bring together to help it accomplish some goal.

To perform a task, a client enlists the help of services. For example, a client program might upload pictures from the image storage service in a digital camera, download the pictures to a persistent storage service offered by a disk drive, and send a page of thumbnail-sized versions of the images to the printing service of a color printer. In this example, the client program builds a distributed system consisting of itself, the image storage service, the persistent storage service, and the color-printing service. The client and services of this distributed system work together to perform the task: to offload and store images from a digital camera and print out a page of thumbnails.

The idea behind the word *federation* is based on the Jini view of the network -- there isn't a central controlling authority. Because no one service is in charge, the set of all services available on the network form a federation -- a group composed of equal peers. Instead of a central authority, Jini's runtime infrastructure merely provides a way for clients and services to find each other (via a lookup service, which stores a directory of currently available services). After services locate each other, they are on their own. The client and its enlisted services perform their task independently of the Jini runtime infrastructure. If the Jini lookup service crashes, any distributed systems brought together via the lookup service before it crashed can continue their work. Jini even includes a network protocol that clients can use to find services in the absence of a lookup service.

How Jini works

Jini defines a *runtime infrastructure* that resides on the network and provides mechanisms that enable you to add, remove, locate, and access services. The runtime infrastructure resides on the network in three places: in lookup services that sit on the network; in the service providers (such as Jini-enabled devices); and in clients. *Lookup services* are the central organizing mechanism for Jini-based systems. When new services become available on the network, they register themselves with a lookup service. When clients wish to locate a service to assist with some task, they consult a lookup service.

The runtime infrastructure uses one network-level protocol, called *discovery*, and two object-level protocols, called *join* and *lookup.* Discovery enables clients and services to locate lookup services. Join enables a service to register itself in a lookup service. Lookup enables a client to query a lookup service for services that can help the client accomplish its goals.

The discovery process

Discovery works like this: Imagine you have a Jini-enabled disk drive that offers a persistent storage service. As soon as you connect the drive to the network, it broadcasts a *presence announcement* by dropping a multicast packet onto a well-known port. Included in the presence announcement is an IP address and port number where the disk drive can be contacted by a lookup service.

Lookup services monitor the well-known port for presence announcement packets. When a lookup service receives a presence announcement, it opens and inspects the packet. The packet contains information that enables the lookup service to determine whether or not it should contact the sender of the packet. If so, it contacts the sender directly by making a TCP connection to the IP address and port number extracted from the packet. Using RMI, the lookup service sends an object, called a *service registrar,* across the network to the originator of the packet. The purpose of the service registrar object is to facilitate further communication with the lookup service. By invoking methods on this object, the sender of the announcement packet can perform join and lookup on the lookup service. In the case of the disk drive, the lookup service would make a TCP connection to the disk drive and would send it a service registrar object, through which the disk drive would then register its persistent storage service via the join process.

The join process

Once a service provider has a service registrar object, the end product of discovery, it is ready to do a join -- to become part of the federation of services that are registered in the lookup service. To do a join, the service provider invokes the `register()` method on the service registrar object, passing as a parameter an object called a *service item,* a bundle of objects that describe the service. The `register()` method sends a copy of the service item up to the lookup service, where the service item is stored. Once this has completed, the service provider has finished the join process: its service has become registered in the lookup service.

The service item is a container for several objects, including an object called a *service object,* which clients can use to interact with the service. The service item can also include any number of *attributes,* which can be any object. Some potential attributes are icons, classes that provide GUIs for the service, and objects that give more information about the service.

Service objects usually implement one or more interfaces through which clients interact with the service. For example, a lookup service is a Jini service, and its service object is the service registrar. The `register()` method invoked by service providers during join is declared in the `ServiceRegistrar` interface, which all service registrar objects implement. Clients and service providers talk to the lookup service through the service registrar object by invoking methods declared in the `ServiceRegistrar` interface. Likewise, a disk drive would provide a service object that implemented some well-known storage service interface. Clients would look up and interact with the disk drive by this storage service interface.

The lookup process

Once a service has registered with a lookup service via the join process, that service is available for use by clients who query that lookup service. To build a distributed system of services that will work together to perform some task, a client must locate and enlist the help of the individual services. To find a service, clients query lookup services via a process called *lookup.*

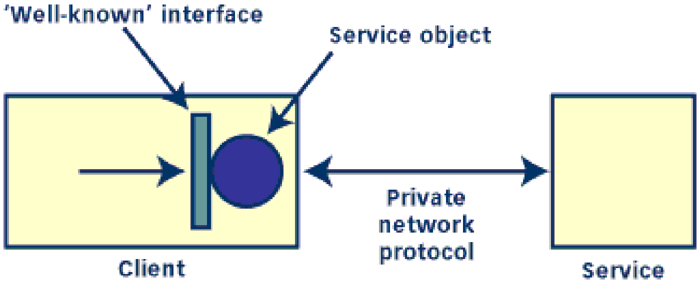
To perform a lookup, a client invokes the `lookup()` method on a service registrar object. (A client, like a service provider, gets a service registrar through the process of discovery, as described earlier in this article.) The client passes as an argument to `lookup()` a *service template,* an object that serves as search criteria for the query. The service template can include a reference to an array of `Class` objects. These `Class` objects indicate to the lookup service the Java type (or types) of the service object desired by the client. The service template can also include a *service ID*, which uniquely identifies a service, and attributes, which must exactly match the attributes uploaded by the service provider in the service item. The service template can also contain wildcards for any of these fields. A wildcard in the service ID field, for example, will match any service ID. The `lookup()` method sends the service template to the lookup service, which performs the query and sends back zero to many matching service objects. The client gets a reference to the matching service objects as the return value of the `lookup()` method.

In the general case, a client looks up a service by Java type, usually an interface. For example, if a client needed to use a printer, it would compose a service template that included a `Class` object for a well-known interface to printer services. All printer services would implement this well-known interface. The lookup service would return a service object (or objects) that implemented this interface. Attributes can be included in the service template to narrow the number of matches for such a type-based search. The client would use the printer service by invoking on the service object methods declared in the well-known printer service interface.

Separation of interface and implementation

Jini's architecture brings object-oriented programming to the network by enabling network services to take advantage of one of the fundamentals of object-oriented programming: the separation of interface and implementation. For example, a service object can grant clients access to the service in many ways. The object can actually represent the entire service, which is downloaded to the client during lookup and then executed locally. Alternatively, the service object can serve merely as a proxy to a remote server. When the client invokes methods on the service object, it sends the requests across the network to the server, which does the real work. The local service object and a remote server can each do part of the work as well.

One important consequence of Jini's architecture is that the network protocol used to communicate between a proxy service object and a remote server does not need to be known to the client. As illustrated in the figure below, the network protocol is part of the service's implementation. This protocol is a private matter decided upon by the developer of the service. The client can communicate with the service via this private protocol because the service injects some of its own code (the service object) into the client's address space. The injected service object could communicate with the service via RMI, CORBA, DCOM, some home-brewed protocol built on top of sockets and streams, or anything else. The client simply doesn't need to care about network protocols, because it can talk to the well-known interface that the service object implements. The service object takes care of any necessary communication on the network.



The client talks to the service through a well-known interface

Different implementations of the same service interface can use completely different implementation approaches and completely different network protocols. A service can use specialized hardware to fulfill client requests, or it can do all its work in software. In fact, the implementation approach taken by a single service can evolve over time. The client can be sure it has a service object that understands the current implementation of the service, because the client receives the service object (by way of the lookup service) from the service provider itself. To the client, a service looks like the well-known interface, regardless of how the service is implemented.

Conclusion

As we have seen in this introductory column, Jini attempts to raise the level of abstraction for distributed systems programming, from the network protocol level to the object interface level. In the emerging proliferation of embedded devices connected to networks, many pieces of a distributed system may come from different vendors. Jini makes it unnecessary for vendors of devices to agree on network level protocols that allow their devices to interact. Instead, vendors must agree on Java interfaces through which their devices can interact. The processes of discovery, join, and lookup, provided by the Jini runtime infrastructure, will enable devices to locate each other on the network. Once they locate each other, devices will be able to communicate with each other through Java interfaces.

Next month

Although this column will focus mainly on how to solve specific programming problems using Jini, such as adding a GUI to a service or making a service administratable, next month I'm going to discuss Jini's real-world problems and prospects.

Discussing Jini

To discuss the material presented in this article, visit:
http://www.artima.com/jini/jf/intro/index.html

About the author

Bill Venners has been writing software professionally for 14 years. Based in Silicon Valley, he provides software consulting and training services and maintains a Web site for Java and Jini developers, artima.com. He is author of the book: *Inside the Java Virtual Machine*, published by McGraw-Hill.

Resources

- Visit the Jini Community for information about the process by which well-known interfaces will be defined:
  http://www.jini.org

- Download page for the current Jini release (at the Java Developer Connection):
  http://developer.java.sun.com/developer/products/jini
- Download page for JDK 1.2 FCS Release, on which the current Jini release runs:
  http://java.sun.com/products/jdk/1.2/
- An online Jini tutorial:
  http://pandonia.canberra.edu.au/java/jini/tutorial/Jini.xml
- Online lecture notes for a course about RMI and Jini:
  http://www.eli.sdsu.edu/courses/spring99/cs696/notes/index.html
- "The Network Revolution," Clyde Higaki and Bill Venners (Sun's Jini Technology Homepage, 1999). Authors Clyde Higaki and Bill Venners offer a number of scenarios to describe how Jini might be used in the real world:
  http://java.sun.com/features/1999/01/jini_scenario.html
- Links to Jini resources:
  http://www.artima.com/jini/resources/index.html
- The main Jini page at Sun:
  http://java.sun.com/products/jini/
- The Jini Community, the central site for interaction among signers of the Jini Sun Community Source License:
  http://www.jini.org
- Download page for all the Jini specifications:
  http://java.sun.com/products/jini/specs/
- JINI-USERS mailing list archives. To subscribe to the JINI-USERS mailing list, send e-mail to `listserv@java.sun.com`. In the body of the message, type `subscribe jini-users`:
  http://archives.java.sun.com/archives/jini-users.html
- A Jini FAQ for the JINI-USERS mailing list:
  http://www.artima.com/jini/faq.html

Feedback: jweditors@javaworld.com
Technical difficulties: webmaster@javaworld.com
URL: http://www.javaworld.com/jw-06-1999/jw-06-jiniology.html
Last modified: Thursday, July 01, 1999