Sun's Jini: Science, Not Magic

The Goal Is Smarter Networks, But Microsoft Has a Plan, Too

by Tom R. Halfhill

As if "write once, run anywhere" weren't an ambitious enough target, Sun is now aiming for "write once, run everywhere." Sun's new Java-based Jini technology tries to make it easier for IT administrators and befuddled users to add hardware devices and software services to networks. "Plug and work, not plug and play" is the new mantra.

Jini leverages Java's ability to move program objects, in the form of bytecode, over networks and run them on any device that has a Java virtual machine (JVM). Today, that ability allows any PC with a JVM-enabled Web browser to run a Java applet, without the need to install native binary code. When the applet's task is done, it goes away, leaving no permanent mark on the system. Jini extends that model to all interactions between devices on networks. Any kind of client can interact with anything else (hardware devices or software services) without the need to permanently install a native program or device driver.

One goal is to solve the problem of replicating native device drivers across an unmanageable variety of clients. With new types of clients emerging all the time, the current driver model will soon reach the breaking point. The larger vision, however, is to make networks more friendly and more fluid. Anybody from an IT administrator to a casual home user should be able to plug any kind of device into a LAN without fussing over control panels, property sheets, registries, IP addresses, and subnet masks. The network should instantly adapt to the new arrival. The device should automatically gain access to hardware and software services on the network—or, if the device has services of its own to offer, other clients should automatically gain access to them. If a device leaves the LAN, the network should adjust to that, too.

Realizing this dream means adding more intelligence to devices and networks—intelligence in the form of processing power and software. That has obvious implications (mostly good ones) for vendors of embedded hardware and software. If Jini makes it easier to manage and expand networks, it could encourage the development and acquisition of more networking products. It could also improve the outlook for home networking, a potentially lucrative market that's the target of several other industry initiatives.

Jini sounds wonderful but isn't a clear winner. In addition to the usual technical challenges, Jini faces opposition from an alternative proposed by Microsoft: Universal Plug and Play (UPnP). Jini and UPnP take different but similar technical approaches to the same problems.

Whenever Microsoft and Sun go head to head, marketing becomes as important as technology. Microsoft has much more influence over the evolution of the PC platform than Sun does. Although neither Jini nor UPnP is PC-specific, the need to fit into a Windows-centric world means that Jini faces an uphill battle for acceptance.

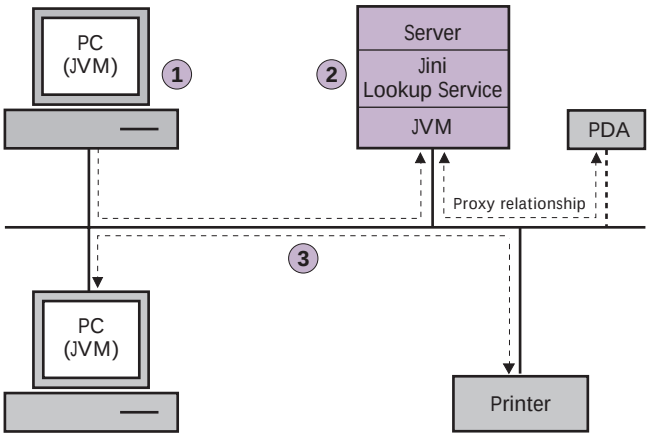

Figure 1. Jini services use RMI to interact over networks. Services without a Java virtual machine can use another service as a proxy. Newly attached devices (1) register with the lookup service (2). In response to queries, the lookup service arranges links (3) between clients and services. Clients and devices talk directly via RMI.

Abstracting Device Drivers

As Figure 1 shows, Jini relies on three components: a lookup service that allows clients to find services on networks, defined interfaces that allow clients and services to interact with each other, and a programming model based on Java (or at least, any programming language that compiles to Java bytecode).

Java's primary role is to solve the device-driver dilemma. Allowing clients as disparate as a Palm handheld computer and a Windows PC to share the same networked laser printer or flatbed scanner might normally require a client-specific driver for every conceivable type of device as well as a repository for the drivers. As the definition of a network device expands to include everything from mobile phones and digital cameras to personal organizers and video-game consoles, the drivers will start multiplying like digital rabbits.

Jini divides a driver's functions into two parts. Ideally, low-level code that drives the hardware runs on the device itself. Developers don't have to write this code in Java, and they probably won't. Instead, they can wrap native code in Jini classes that provide well-defined interfaces to other devices on the network. Devices interact by calling those interfaces, using Java remote method invocations (RMIs), as Figure 2 shows.

Devices don't have to be capable of running a JVM if another device on the network acts as a proxy. The proxy can be a server or any client that has a JVM. Devices that use services and devices that provide services can avoid having local JVMs if they negotiate through proxies. Although using a proxy may be less efficient, Jini offers these options so developers have more latitude and so existing devices can take advantage of Jini.

Xerox has demonstrated this capability by adapting its DocuPrint N17 network laser printer to work with Jini. According to Xerox, the difficulty of adding a Jini wrapper to the existing driver was minimal.

When a new Jini device arrives on a network, it sends out a Jini-specific multicast packet to find a Jini lookup service (or a proxy, if the device needs one). Upon finding the lookup service—which probably resides on a server, perhaps integrated with a standard directory service—the device joins the Jini "federation" by first obtaining an RMI reference to the lookup service. Then it uses RMI to send the lookup service a Java object that describes the device's capabilities. If the device is a printer, for example, it will identify itself as such and enumerate its properties (its output resolution, whether it can print in color or in monochrome, the configuration of its paper trays, and so on).

Clients that need services can query the lookup service to see what's available. Because Jini encapsulates the interfaces within Java objects, queries can pass Java types as parameters. A multifunction peripheral could respond as a 300-dpi printer, a 200-dpi fax machine, or a 600-dpi scanner, depending on what kind of service a client wants. In contrast, an ordinary network directory might opaquely describe the same device as "HP MFP—ACCT DEPT."

After locating a service, the client (or its proxy) receives a Java object that contains the interfaces to that service. From that point, the client communicates directly with the provider of the service. Jini's lookup service steps out of the way; it's a matchmaker, not a chaperone.

Jini doesn't define exactly how a client and a service must converse through their interfaces. The bitstream between them is private. In Xerox's demo, a laptop computer sent a document to the N17 laser printer through a PC acting as a proxy by using the standard Unix line printer (LPR) protocol. The N17's native device driver was not installed on the laptop. The Jini objects that enabled this interaction remained on the laptop only until the print job was finished.

Jini also allows proxies to poll a network in search of new devices and draft them into a Jini federation. This allows a Jini lookup service to assimilate non-Jini devices that don't know how to ask for a proxy.

Essentially then, Jini inserts an abstraction layer of Java code between the consumers and providers of services on networks. Although this extra level of indirection is less efficient than a local, native device driver, it does address the problem of writing drivers for every conceivable client-device combination. Clients and devices deal with each other only through their Jini interfaces. And because those interfaces are Java objects, they're compatible with different platforms and can move freely around a network.

The Incredible Shrinking JVM

Sun says the ideal Jini device would have enough CPU cycles, nonvolatile storage, and memory to support a JVM, the required Java class libraries, the new Jini classes, and whatever additional software (such as a user interface) the maker of the device deems necessary. If a device already meets those requirements—as in the case of a PC—the new Jini classes add only about 48K of additional code to the core Java platform. But printers, scanners, disk drives, handheld computers, and most other embedded devices are rarely so well endowed. They must either learn to speak Java or use a proxy.

At this point, there are barriers to embedding Java in a device. Part of the problem is that Jini is so new (Sun formally introduced it in January) that not everything is in place yet. For instance, the version of RMI that Jini requires is the one included in the latest Java 2 release (formerly known as 1.2). But the EmbeddedJava (eJava) and PersonalJava (pJava) subplatforms that are best suited to small devices don't yet support Java 2. Sun expects them to catch up later this year.

The cost of embedding Java in a small device has been high because JVMs and the Java class libraries normally require megabytes of storage. Recently, lighter-weight JVMs have been appearing. Two examples are Insignia Solutions' Jeode and Oberon's JBed, which require as little as 42K of ROM and RAM, although an implementation capable of supporting Jini would require more like 256K–640K. They also need a 32-bit CPU. That's not a major obstacle; claims Jini chief architect Jim Waldo, "I took apart my microwave oven and discovered it has a Motorola 68000. It could run Unix System V."

Java chips based on Sun's PicoJava core, such as the MicroJava 701, might be suitable for relatively high-end devices like Sun's network computers. But they will probably be too expensive, and are certainly too power hungry, for small or mobile embedded devices such as smart phones and

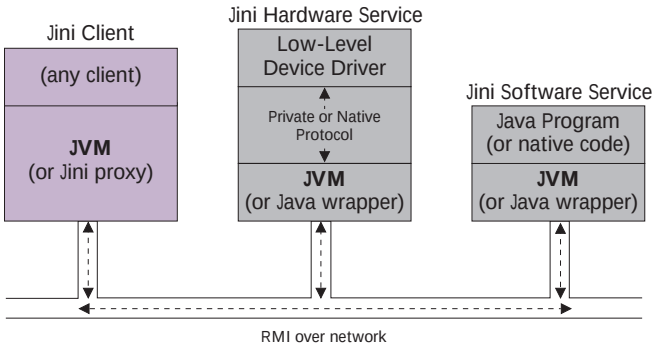

Figure 2. Jini services can be provided by hardware devices (printers, scanners, fax machines, etc.) or by programs written in Java or native code.

handheld computers. Some better bytecode-savvy alternatives might be Patriot Scientific's PSC1000 (see MPR 4/15/96, p. 1) or Imsys's GP1000 (see MPR 12/28/98, p. 14).

Dallas Semiconductor has another solution: a Jini-enabling chip set. It consists of a microcontroller, a ROM containing a JVM and core Java classes, and an Ethernet interface chip. The MCU is a proprietary 8-bit ASIC core that runs at 33 MHz—plenty fast, says Dallas Semi, for the relatively simple bitstream processing expected of a Jini interface. It has two asynchronous serial ports, I^2C, and a single-wire interface (1-Wire MicroLAN) for optional coprocessors. The ROM chip was originally a 256K part, but the company is moving to a 512K ROM to make room for the Java 2 version of RMI. The Ethernet chip includes a TCP/IP stack. Designers can add their own RAM (up to 1M).

The chip set is scheduled to be commercially available in June for $50 in single-unit quantities. That would make Jini a costly addition for small devices, but a more integrated single-chip version is targeted to sell in mid-2000 for $15. Dallas Semiconductor is delivering a $500 beta developer's kit this month.

Until more cost-effective solutions for embedded Java appear, fully enabled Jini implementations will probably appear first in peripherals that already have a fair amount of processing power and firmware: network laser printers, server-sized disk drives, and professional scanners. Even those kinds of devices may rely heavily on proxies until Jini proves itself in the marketplace.

Microsoft's Java-Free Alternative

Why bother with eJava or pJava when nJava (no Java) could work just as well? asks Microsoft. UPnP is similar to Jini but standardizes on service protocols instead of Java interfaces.

When a UPnP device attaches to a network, it seeks an IP address from a dynamic host configuration protocol (DHCP) server or, if necessary, assigns itself one using automatic private IP addressing (APIPA). APIPA is a proposed Internet Engineering Task Force (IETF) standard that allows a client to randomly pick an IP address from a 16-bit table of entries while checking for collisions with existing addresses. It's ideal for small LANs. APIPA is already built into Windows 98 and Mac OS 8.

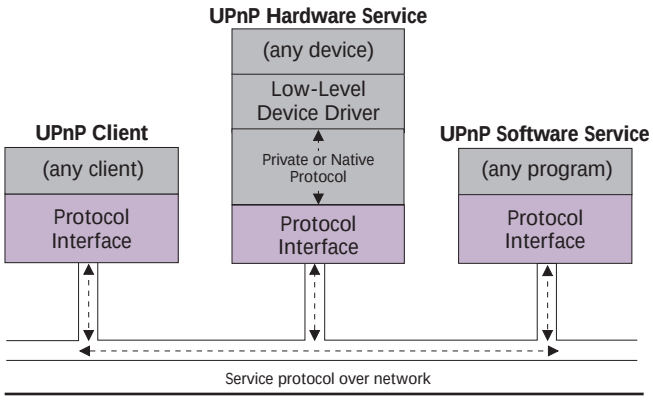

Figure 3. Microsoft's Universal Plug and Play relies on standardized service protocols (such as LPR) instead of Java interfaces.

Newly connected UPnP devices, like Jini devices, send multicast packets over the network in search of a lookup service, which Microsoft calls an announcement listener. The protocol for the multicast packet is another proposed IETF standard: simple server discovery protocol (SSDP).

On a large network, the announcement listener would be integrated with a directory service. The listener stores the attributes of all available services, though not in the form of program objects as Jini does. Jini's approach might allow more flexible queries, because devices can find services by matching Java object types not just strings and values. For example, "printer" could be an object type, with subclasses for various kinds of printers.

On a small network, such as a home LAN, a UPnP device stops looking for an announcement listener if it doesn't receive a response. It then waits for a peer device to query for a service by multicasting another SSDP packet.

Either way, UPnP devices that have services to offer respond by returning a URL. The URL contains the responding device's IP address and the name of the protocol required to access the service. In the case of a printer, scanner, or digital camera, it might specify Internet printing protocol (IPP), another proposed IETF standard.

UPnP Requires Standardized Protocols

This approach dispenses with Sun's Java-centric Jini interfaces and RMI. But it means the industry must define a protocol for every class of service, and it also requires every client that uses a service to natively support the appropriate protocol. A single device could support more than one protocol—for example, a printer might accept IPP and LPR.

UPnP's reliance on standardized protocols is similar to the way Web servers communicate with clients using hypertext transfer protocol (HTTP). A Web server doesn't need to know anything about a client system that requests a Web page. The server merely sends an HTTP-encoded package of HTML text and graphics over the network to the client, which implements whatever code is necessary to render the content on a screen. HTTP insulates the Web server from specific knowledge about the client's CPU architecture, OS, APIs, screen size, and other native details. UPnP will use similar protocols to extend that model to any kind of device or service, as Figure 3 shows.

In this manner, UPnP, like Jini, tries to isolate the most bothersome low-level driver code on the device itself by adding a level of indirection on the network. Microsoft's goal is not only to simplify things for users but also to reduce a burden on its OS developers. Microsoft says that the testing of an OS against thousands of devices and drivers is a major factor delaying the release of new versions of Windows.

Microsoft estimates that adding UPnP to a device might require only about 90K of x86 code, or perhaps

65,000 gates on a chip. If a device already supports a TCP/IP stack, a UPnP implementation might require as little as 5K of additional code. (Minimally, a UPnP device must support a TCP/IP stack and the browserlike capability to navigate URLs and use service protocols.) The precise requirements are uncertain because UPnP is even newer than Jini. Microsoft plans to distribute the first specifications at the Windows Hardware Engineering Conference in April. Some vendors are testing prototypes of UPnP-enabled devices.

Both Microsoft and Sun are enlisting allies to their causes and are trying to seed the market. Microsoft is making the UPnP specifications available for free, but it may charge later for SDKs. Jini is available under the new Sun community source license. This allows anyone to download and experiment with Jini source code for free, but Sun charges fees when developers distribute Jini source code or bytecode commercially.

The Distributed-Computing Debate

Easier networking isn't the whole story. Jini and UPnP also build the foundation of a distributed-computing architecture, which is why Jini is often compared to Microsoft's Millennium, a research project that tries to make distributed computing invisible to programmers. Services brokered by Jini and UPnP can be provided by hardware or software—clients don't care. They're aware of only the network-level interfaces, which can present any kind of service.

Vendors such as Xerox want to take advantage of those capabilities to streamline document processing. For example, a Jini-enabled scanner could automatically send the scanned image of a document to a Jini-aware OCR program running on a server. After converting the scan to text, the OCR program could pass the document to another Jini-aware program that sorts documents and forwards them to the appropriate users or archives. Xerox says it's possible to do all that today on a network of Windows PCs, but Jini offers a better cross-platform solution.

The abstraction of hardware and software services means that a Jini client can distribute just about any task to Jini-aware processes running on multiple computers across a network. The remote processes could provide services ranging from file filtering to parallel computation. The same is true of UPnP. Neither distribution model is transparent, however. Developers must explicitly write distributed applications with this architecture in mind.

In that sense, both Jini and UPnP fall short of Millennium. Microsoft has a prototype JVM called Borg that allows any Java program to distribute its workload across multiple systems. To the program (and, more important, to the programmer), those systems appear as a single process. Another Millennium prototype known as Coign optimizes the performance of distributed Common Object Model (COM) applications. But Millennium is still a research project with no announced product plans.

For More Information

For more information about the technologies and products discussed in this article, check out the following Web sites:

- Jini, *www.sun.com/jini*
- Universal Plug and Play, *www.microsoft.com/homenet*
- Dallas Semiconductor, *www.dallassemiconductor.com/News_Center/Press_Releases/1999/prjini.html*
- Oberon's JBed, *www.jbed.com*
- Insignia Solutions' Jeode, *www.insignia.com*
- Internet Engineering Task Force, *www.ietf.org*
- Microsoft Millennium project, *www.microsoft.com/presspass/features/1999/02-29mill.htm*

Although Jini doesn't go quite as far as Millennium, the Java platform has a way of evolving into whatever Sun needs it to be. Remember that Java first attracted attention as a cute way to enliven static Web pages with animated graphics; only later was it discovered to be a powerful language for writing server-side enterprise software. Don't be surprised if Jini becomes a similar vehicle for Sun's grander ambitions to prove that "the network is the computer."

It's Hard To Buck Microsoft

The immediate goals of Jini and UPnP are certainly worthwhile. IT departments could no doubt save significant money and downtime if networks could automatically reconfigure themselves after the arrival or departure of new users, devices, and services. Workplaces would be more flexible if networks could dynamically adapt to the needs of part-time employees, on-site contractors, temporary work groups, and other transient demands of modern business. Retailers could sell more merchandise if home networks were as easy to install as telephones. And, of course, chip makers and developers could ship a lot more hardware and software to power all of those products.

Jini and UPnP appear equally capable of making these goals a reality. Although a device could support both (as well as other mechanisms, such as JetSend, Service Location Protocol, and Salutation), this ability would increase costs. Ultimately, Microsoft's control over the PC platform gives UPnP an undeniable advantage. UPnP also sits better with those who can't abide Java.

But neither technology will change things overnight. The original Plug and Play took years to make an impact, even though its scope is limited to the relatively controlled internal environment of Windows PCs. Indeed, Windows NT still doesn't support PnP. USB's gestation period has been similarly protracted. All connectivity technologies depend on the industrywide cooperation of hardware designers, software developers, and a critical mass of vendors. Jini and UPnP are only beginning to herd those cats. M