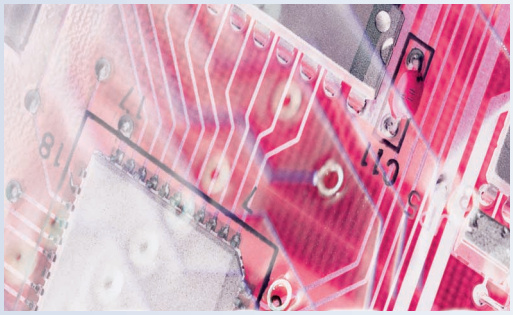
Jini Home Networking: A Step toward Pervasive Computing

Jini connection technology forms a network of devices on the fly, without manual connection or configuration. It can also complement other technologies that strive for "anytime, anywhere" connectivity.



Rahul Gupta
Sumeet Talwar
Dharma P. Agrawal
University of Cincinnati

Internetworking services have come a long way since their inception in basic communication between two homogeneous computers located not far from each other. Today, intelligent devices can interact with each other anytime and anywhere in the world. Recent advances in technology—especially in wireless communication—have fueled the market for such devices. At the same time, decreasing processor costs and size let engineers endow ever more devices with application-specific processing power. These trends are moving toward the vision of pervasive computing.

Home networks represent a step along this path. As the name suggests, a home network interconnects various electronic devices within a house. In cases where the home network connects to a broadband local loop, it can also make these devices accessible over the Internet, opening the door to new applications that rely on remote network administration.

The requirements that drive a home network differ from those for an enterprise network. For one thing, a home network must handle interference from household appliances such as microwave ovens and cordless phones. Second, given the lack of system administrators in the home, the network must operate with little or no user intervention. At the same time, it must meet the challenge of structure and usage variances among in-home devices. Current devices vary greatly from each other and from one household to another with respect to models, make, and quantity. Their purposes range from washing clothes to controlling room humidity.

A home network allows all such appliances to communicate with one another. Thus, it needs a technology that can seamlessly integrate assorted devices into a monolithic communication network.

JINI OVERVIEW

Jini network technology is middleware that provides a set of application programming interfaces (APIs) as well as network protocols that can meet home network requirements. It establishes a software platform enabling all devices that form the network to talk to each other, irrespective of their operating systems or interface constraints. In a Jini environment, each device provides a service to other devices in the network. Each device publishes its own interfaces, which other devices can use to communicate with it and thereby access its particular service. This approach ensures compatibility and standardized access among all devices.

A Jini environment is not housed on a single computer nor is it a network of computers. Jini technology provides a distributed environment for devices to communicate with each other. Each device provides a set of services to the network, creating a *federation* of services for the constituent devices. No central authority controls federation services.

As Figure 1 shows, a Jini environment implements the connection technology below the network application layer, building on top of the Java platform. The technology accepts all sorts of

0018-9162/02/$17.00 © 2002 IEEE

devices, including electronic home appliances, musical instruments, and other devices that are not part of a conventional computer network. Each device can act as a client or server depending on whether it is requesting a service or providing one. A service in a home network might be as small as requesting the room temperature from an air conditioner or as big as transferring a file from a laptop to a printer.

A Java object represents each device. The object's interfaces expose the services offered to the network. Accessing a particular service involves using the published interfaces to invoke a remote procedure on the appropriate device object.

Jini uses Java's remote method invocation for accessing a particular service. RMI, the Java equivalent of a remote procedure call, enables clients to obtain handles to the desired remote objects. Jini can also use RMI to pass objects as arguments and to return values, making it easy to move code as well as data across the network.

Figure 1 also shows Jini's two basic services: lookup and discovery. These services manage the processes of first making a network service available to devices and subsequently using it.

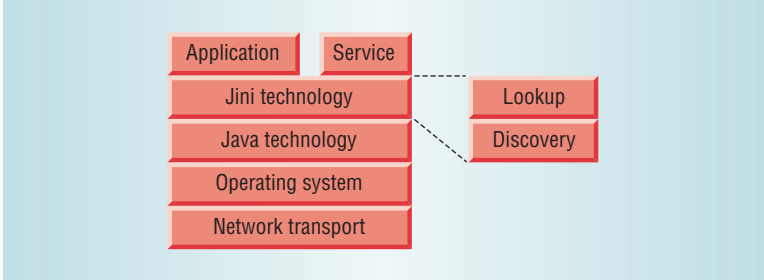


Figure 1. A typical Jini network architecture implements the connection technology below the application layer and atop a Java platform.

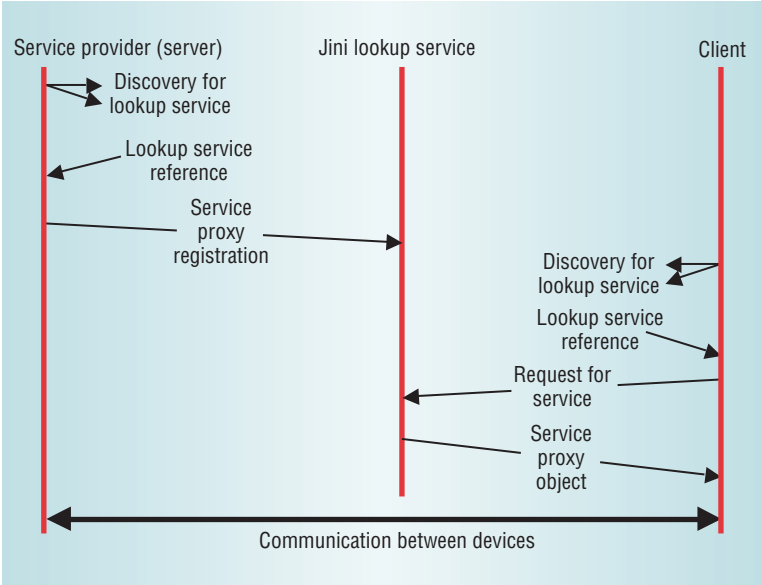


Figure 2. Jini event sequence diagram. The Jini lookup service is the common repository of services offered by the network.

Lookup and discovery services

Figure 2 shows the event sequence for Jini lookup and discovery services. Each server creates a remote object, essentially a software implementation of services being offered. A device wanting to offer a service first announces its presence to the lookup service. In this service registration process, the device uploads the serialized Java object, called a *service proxy*, to the Jini lookup service. Thus, the lookup service is the common repository of the services offered by a network. Clients use this repository, which can run on any device in the network, to gain access to a particular service.

The basic discovery service lets a new device obtain a reference to the lookup service. The discovery service identifies a lookup service that can handle the particular client request. When the client device wishes to use the network service, it sends the request to that lookup service, which in turn sends a service proxy object to the requesting client. Once the client obtains the proxy, it interacts directly with the network service via the proxy, thereby establishing a client-server communication channel.

Reliability and scalability

To handle network failures, Jini *leases* a resource to a client for a fixed amount of time. After this period expires, the client must renew the lease to continue accessing that service. The lease automatically expires for all authorized users when a service goes down.

Jini also supports redundancy in the infrastructure and resilience against failure. The network may have several lookup services distributed throughout it. Servers register their proxy service objects with all the lookup services they can discover using the discovery protocol. Clients may obtain the reference of the desired service from any of those lookup services. This keeps services available, even if key machines crash.

Jini addresses scalability by incorporating the notions of "communities" and "federations." Groups of Jini services, which are completely aware of each other, come together to form a community. Jini communities can link together, or federate, into larger groups. Ideally, a Jini community is about the size of a work group. For example, in an office environment a community might include the printers, PDAs, and other such devices needed by a group of 10 to 100 people.

The Jini lookup service for a particular community can register itself in other communities, thereby

JiniME addresses the limited size, computing power, and storage of wireless mobile devices.

acting as the interface for sharing its resources with other communities' clients.

JINI ARCHITECTURAL REQUIREMENTS

In other distributed computing technologies like the common object request broker architecture (Corba) and the Distributed Component Object Model (DCOM), a client-side stub and server-side skeleton support communication between the client entity that requests a particular service and the server entity that provides it. The stub and skeleton agree upon a protocol for information exchange—arguments from the client and return values from the server. A programming-language-neutral interface definition language specifies the interfaces for accessing a service. The Interface Definition Language compiler produces the stub and skeleton source code. A native compiler then compiles the stub and skeleton on the machine.

The main disadvantage of this approach is the tight coupling between the stub and skeleton. Any change in one must be appropriately reflected in the other. Java RMI overcomes the problem by allowing a client to obtain the stub from the server at runtime. In Jini, this stub is the service proxy object that the server uploads to the lookup service. Hence, the server implementation can be altered automatically and transparently to the client. This concept of downloading code at runtime gives Jini a significant advantage over Corba and DCOM, and it resembles object-oriented programming's dynamic binding.

A Jini network is Java-centric. It relies on Java features such as object serialization and code portability to provide a distributed computing environment. Developers can write service implementations in other programming languages, but they must encapsulate each object in a Java Native Interface wrapper so that the Java environment can still dynamically load the objects. Work is also under way to implement RMI over Corba's Internet inter-orb protocol.

This approach requires that each device run a Java virtual machine (JVM), which in turn requires processing power and memory that may not always be feasible. Nor do all devices necessarily have direct access to the Jini network—for example, a printer connected to a computer via a universal serial bus. This configuration precludes moving an object across the network and further limits this approach.

SERVICE FOR NON-JINI DEVICES

The Jini architecture allows non-Jini-capable devices to join the service federation, even if they lack enough memory and processing power to implement a JVM. There are several alternatives, but the Jini surrogate architecture is a prominent method that lets limited-capability devices deliver their code to an entity called a *surrogate host*.

A surrogate host has the memory and computing power to support a full Java 2 Standard Edition (J2SE) environment. Each device has a surrogate Java proxy object, similar to the service proxy. These surrogates represent and act on behalf of the device. When a non-Jini device joins the Jini network, it first registers its surrogate with the surrogate host, which in turn registers it with the lookup service on the device's behalf. Communication with other services takes place through this surrogate, thus integrating the device seamlessly into the system. The device communicates with its surrogate using a predefined protocol that is private and transparent to other entities.

The Jini surrogate architecture can be suitably extended to support wireless mobile devices. When two non-Jini-capable mobile devices want to communicate, each device registers its respective surrogate with the surrogate host. The surrogate host, in turn, registers the device's surrogate with the lookup service. Each surrogate object then downloads the other device's proxy object using the lookup service. The devices and their respective surrogates communicate wirelessly with each other, whereas the surrogates talk to the lookup service over the wired network.

The Jini surrogate architecture's main limitation is its dependence on the surrogate host and the lookup service, which require a complete Jini runtime environment. To operate a full JVM requires substantial computing resources, prevalent only in fixed devices such as workstations or PCs. Such an approach is not helpful in ad hoc networks that use, for example, Bluetooth or IrDA transport and peer-to-peer communication with no fixed infrastructure.

JINI MOBILE EDITION FOR WIRELESS DEVICES

Researchers at the Rochester Institute of Technology developed a mobile edition of Jini that targets wireless mobile devices.[1] JiniME addresses the limited size, computing power, and storage of such devices, as well as their independence from any fixed infrastructure. The most significant change is in the JiniME-capable device architecture. Instead of J2SE, JiniME devices use an environment based on Java 2 Micro Edition, Connected Limited Device Configuration, and Mobile Information Device Profile (J2ME CLDC MIDP). In J2SE, moving an object from one JVM to another requires *marshaling* the object—that is, serializing its state

into a sequence of bytes and annotating the serialization with the object's Java codebase URL. At the destination, a special Java classloader accesses the object's codebase URL, downloads the object's Java classfiles, if necessary, and loads them for the destination JVM to use. J2ME CLDC lacks object serialization, marshaled objects, and classloaders. Rather than rely on the Java libraries to marshal an object automatically, JiniME connection technology relies on the programmer to do it manually. Hence, JiniME incorporates some additional classes and interfaces to accommodate the essential Jini technology features.

To obviate infrastructure dependencies, each JiniME device houses its own lookup service and classfile server. Figure 3 shows a complete device architecture with a bridge between two federations. Figure 4 shows the services in a JiniME-capable device. Each device registers all its offered services with its own lookup service. The device can add or delete services anytime. Its lookup service has a service proxy object that other devices can download for access to the lookup service and, subsequently, to the network services offered.

Clients can search for a service according to a service identifier or attributes. A JiniME mobile device acts as its own classfile server. This avoids the need for a fixed-infrastructure HTTP server for exporting object codebases.

A Jini bridge joins a standard Jini federation with the JiniME federation, allowing smooth service exchanges between the two network types. The bridge provides the glue between mobile and fixed hosts, talking to the mobile devices on one side and to the fixed devices on the other. It runs a full J2SE environment to host the standard Jini federation and also the J2SE implementation of the CLDC generic connection framework. A component of the Jini bridge periodically looks up the services available on the wireless side and creates a corresponding service object—that is, one that has the same interface. The created object transfers the remote fixed host request to the mobile host service proxy object.

The bridge maintains similar service objects for mobile host requests.

JINI IMPLEMENTATIONS

The market does not yet include many Jini-enabled products, but many companies use Jini connection technology for application-specific needs. For example, Eko Systems (http://www.ekosystems.com) uses Jini in its Frontiers anesthesiology information management system for connecting to a

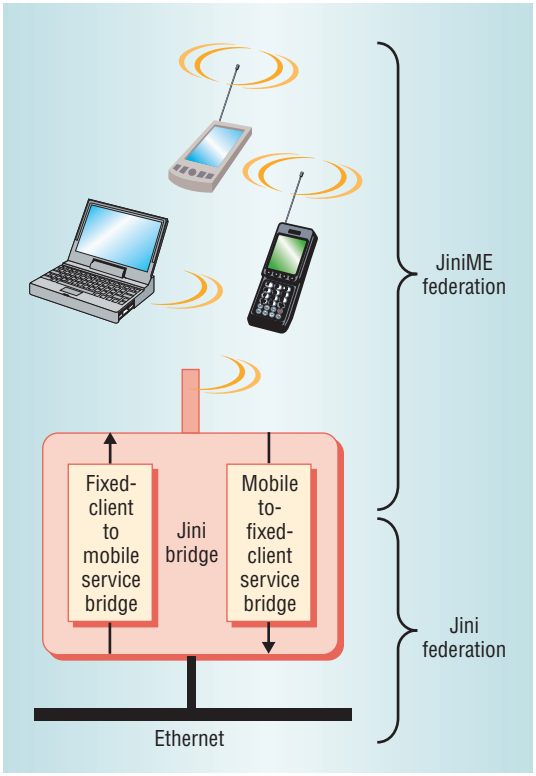


Figure 3. JiniME federation with a Jini bridge to a standard Jini federation. A bridge component creates and maintains service objects for available wireless devices.

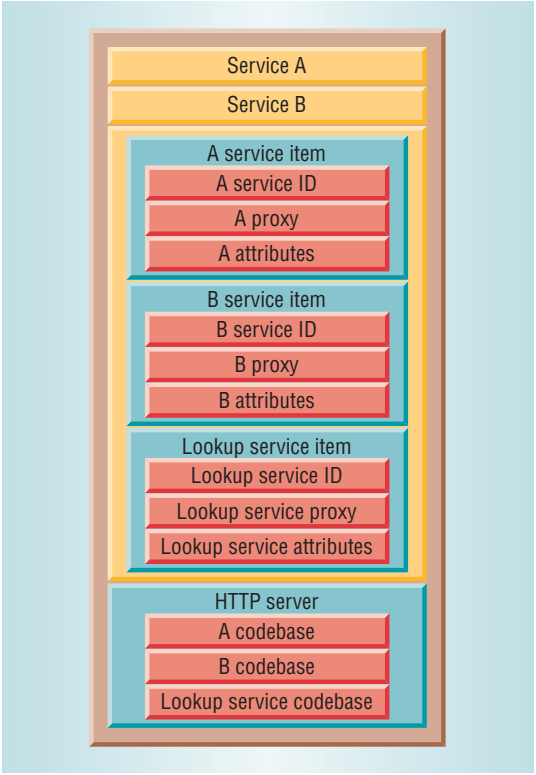


Figure 4. Services in a JiniME-capable device. Each device houses its own lookup service and classfile server.

The main distinction between UPnP and Jini lies in the API strategy.

range of medical equipment such as monitors and ventilators.

Zucotto Systems (http://www.zucotto. com) develops semiconductor solutions for wireless Internet applications. The company's Xpresso Java native processors target embedded systems in the network service and consumer device markets. Xpresso combines Sun's CLDC K virtual machine for resource-constrained devices with Zucotto's own extensions to significantly improve performance compared with traditional JVM architectures. It also provides a strong platform for the Jini surrogate architecture, as it supports both Java and Bluetooth.

With Jini technology at these higher speeds, wireless devices natively executing Java programming language code can spontaneously communicate with other Java technology-enabled devices through various protocols. Bluetooth, for example, provides a way for mobile devices to discover and communicate with other nearby devices, thus meeting the Jini surrogate architecture requirement to provide a means of detecting when a device is added to or removed from the network.

PsiNaptic (http://www.Psinaptic.com) has implemented a small footprint (100 Kbytes) Jini implementation for standalone embedded processors. The software is called JMatos and has passed the Jini Technology compatibility tests. It extends Jini technology to J2ME CLDC MIDP resource-constrained devices. Also, PsiNaptic is working with Symbian to port this software to the Nokia 9200 Communicator series of mobile phones (which use the Symbian OS). A Software Development Kit for JMatos is also available for Windows and Linux platforms.

Other such applications include Echelon's LonWorks products for home networking (http://www.echelon.com).

COMPETING AND RELATED TECHNOLOGIES

Jini has come a long way since it was officially introduced in the beginning of 1999. Jini has already been seen as a solution to a number of challenging issues. Though Jini technology is relatively mature, it is not the only technology available to meet home networking requirements.

UPnP

Universal Plug and Play is Microsoft's peer-to-peer networking initiative. The UPnP Forum (http://www.upnp.org) currently consists of more than 470 companies from the home consumer electronics, computing, and network industries. Recently, Sun Microsystems also joined the forum.

UPnP is an open, distributed architecture being developed for *proximity networking*—networks available to all clients in the same geographical area as the service they desire. Like Jini, UPnP requires no service-specific code on the client to use the service. Independent of the media and network technology below the transport layers, UPnP is transparent to the operating system implementation as well. Further, it has minimal configuration requirements, and its automatic discovery process uses IP addresses.

The UPnP strategy hinges on the use of existing standards wherever possible, especially Internet and Web protocols such as TCP/IP, UDP, HTTP, and XML.

The main distinction between UPnP and Jini lies in the API strategy. Jini uses its APIs as a contract between vendors. UPnP allows vendors to build their own APIs, modeled on protocol standards and targeted for specific features of the operating system. Also, Jini follows a code download model, which connects devices through a common application that developers must use to test their devices' operation.

HAVI

Leading audio-video electronics manufacturers have developed the home audio-video interoperability specification. The HAVI open-license specification focuses specifically on the transfer and processing of digital audio-video content among in-home digital appliances. It does not address home networking functions such as controlling lights or monitoring temperatures.

Based on the IEEE 1394 bus standard, HAVI supports isochronous communication, which guarantees packet delivery at fixed intervals. Hence, HAVI meets the real-time constraints of audio and video streams.

HAVI categorizes each home appliance according to its resource capacity. It distinguishes controllers from controlled appliances. A controller has enough network resources to store data and home networking applications. For example, an intelligent TV could contain a software program that builds a customized user interface for a controlled appliance such as a washing machine.

HAVI models home networking services as software elements. Each of these objects has a unique name and identifier. An object makes itself known to other objects via a *registry*, a systemwide naming service that stores information about the HAVI objects. Objects use messaging to request services from other objects. The messaging mechanism is based on a suite of network and transport layer protocols that provide HAVI software elements with communication

facilities. The actual messaging system implementation may differ from vendor to vendor.

HAVI allows devices to join and leave the network on the fly through an event manager that detects and announces changes in the network configuration.

In a HAVI network, a *device control module* represents an interface to a physical appliance. The DCM, a HAVI object stored in the registry, uses the messaging system to communicate with other objects. Each DCM has one or more *functional control modules* associated with it. FCMs represent different functional components within a networked device.

HAVI has specified the Java programming language for DCM development and in-home appliance applications. This makes it easy for HAVI services to work with Jini technology. A bridge protocol could provide a way for HAVI and Jini devices to share services. Applications using Jini connection software could access HAVI devices such as VCRs. Likewise, a television on the HAVI network could connect to remote services enabled by Jini technology, such as video-on-demand.

JetSend

Hewlett-Packard developed JetSend, a peer-to-peer communication protocol that lets two devices connect, negotiate data types, and exchange information. JetSend fully describes the content being exchanged so that the data transfer is both machine- and operating-system-independent.

Jini has focused on coordinating device-to-device communication. By contrast, JetSend concentrates on data content encoding, negotiation, and conversion. This difference reveals a major distinction between the two technologies. Jini does not define the data content types nor the negotiations that can take place between devices. A JetSend-enabled device can communicate with any other JetSend-enabled device, negotiate and encode data in the best form of representation, and transfer this data over the network. However, JetSend does not support federation.

A Jini-JetSend Gateway can bridge the two technologies.[2] Its main task is to register the JetSend devices with the Jini lookup service and enumerate these devices from the lookup service to other JetSend devices. Hence, once Jini technology connects the two devices, they can use the JetSend protocol to exchange information with each other.

TSpaces

The IBM Almaden Research Center is developing the TSpaces technology (http://www.almaden.ibm.com/cs/Tspaces), which it describes as "intelligent connectionware." The technology essentially combines communication middleware with database functionality. The middleware helps resolve format differences, mask the periodic disconnection of network components, and synchronize multiple components. TSpaces is implemented in Java, which restricts its portability to Java-supported devices and environments. Developers must code TSpaces services, which limits the technology's scalability.

Applications using Jini connection software could access HAVI devices such as VCRs.

Proposed TSpaces services include universal printing services, URL-based file transfer, indexing, group communication, and database services. The universal printing service, for example, supports the "any printer to any computer" concept, regardless of platform, operating system, file format, or printer language. TSpaces does not require device drivers, thus obviating the problem of installing them for every new device discovered. When a client computer wants to print, TSpaces sends driver information from the printer space as an XML file to the client.

Inferno

Lucent Technologies' Bell Labs created the Inferno network operating system for distributed computing. Currently, Vita Nuova, a United Kingdom-based company, has the exclusive global rights for Inferno (http://www.vitanuova.com/inferno/). Bell Labs, however, continues to support Inferno's evolution.

The operating system's base is written in Limbo, an efficient programming language with features such as advanced interprocess synchronization and communication facilities. The real advantage of Inferno lies in its 500-Kbyte kernel, making it highly scalable. It can operate in two modes: host and native. As a host, Inferno runs as an application over a base operating system, like Windows or Unix, providing an environment suitable for rapidly developing distributed systems. In the native mode, it runs as a complete operating system on embedded systems, requiring as little as 1 Mbyte of memory.

Inferno's design is hierarchical, where resources appear as files. Devices access these resources via a secure network-level protocol called Styx, which hides service locations from the user. Current Inferno services support public key infrastructure, POP3, SMTP, HTML, Ethernet, cable, satellite, sound cards, and video cards.

Bluetooth

A computing and telecommunications industry specification, Bluetooth (http://www.bluetooth.com) describes how mobile phones, computers, per-

Enhancing Bluetooth with Jini gives it the security features that accompany the Java Development Kit.

sonal digital assistants (PDAs), and other portable devices can interconnect using a short-range radio link connection. Bluetooth's key features are robustness with low complexity, low power requirements, and low cost. Bluetooth technology can combine a cellular phone, pager, and PDA into a three-in-one device that doubles as a portable phone at home or in the office. Users can quickly synchronize with information in a desktop or notebook computer, send or receive a fax, initiate a printout, and, in general, coordinate all mobile and fixed computer devices.

Bluetooth provides a global specification for short-range wireless technology based on a frequency-hopping protocol in the Industrial Scientific and Medical frequency band (2.4 GHz). Bluetooth provides short-range wireless connectivity in three areas: data and voice access points, cable replacement, and ad hoc networking. Bluetooth specifications address hardware, software, and interoperability system requirements.

In the simplest sense, the relationship between Bluetooth and Jini is one between hardware and software. Bluetooth can enable communication, whereas Jini defines the data flow in the communication pipe. The Bluetooth model fits the OSI seven-layer network reference model; Jini, when incorporated, sits at the session and presentation layers.

Enhancing Bluetooth with Jini gives it the security features that accompany the Java Development Kit. The combination of Jini and Bluetooth technologies under the Jini surrogate architecture can enable the sharing of services between wireless devices.

Jini's strengths lie in its object-oriented roots: It transports code and data together to perform tasks. Jini integrates distributed computing, network-based services, and reliable smart devices in a scalable network without administrative overhead. It provides simple mechanisms for devices to form an instant community without planning, installation, or human intervention.

Sun makes Jini source code available to the developer community at no cost under the Sun Community Source License (http://wwws.sun.com/software/communitysource/). The Jini community (http://www.jini.org) is currently working on new projects, including the Davis project on security. Links to university projects worldwide are available through Jini in Academia (http://www.ecs.soton.ac.uk/~ra00r/jinidemia/).

Jini has reached the implementation phase of its technological development, where manufacturers must come up with ways to use it effectively. Despite Jini's advantages, Microsoft's UPnP remains a strong competitor offering a powerful vehicle for technology transport—the Windows operating system. The marketplace will ultimately determine the direction home networking products take toward pervasive computing. ■

References

1. A. Kaminsky, "JiniME: JiniTM Connection Technology for Mobile Devices," white paper, Information Technology Laboratory, Rochester Inst. of Technology, Aug. 2000; http://www.cs.rit.edu/~anhinga/whitepapers/JiniMEWhitePaper/.
2. J. Rekesh, "Integrating Disparate Communication Technologies," white paper, California Software Co. Ltd., Oct. 1998; http://www.calsoft.co.in/techcenter/hp/sunhp. html.

Rahul Gupta *is a graduate student in computer science at the University of Cincinnati. His research interests include ad hoc routing protocols, TCP over wireless, mobile wireless networks, and sensor networks. He received a BE in electronics and communication from the University of Roorkee, India. Contact him at rgupta@ececs.uc.edu.*

Sumeet Talwar *is a graduate student in computer engineering at the University of Cincinnati. His research interests include mobile and wireless ad hoc networks and MAC-layer protocols. He received a BE in electronics from the University of Mumbai, India. Contact him at stalwar@ececs.uc.edu.*

Dharma P. Agrawal *is the Ohio Board of Regents Distinguished Professor of Computer Science and Computer Engineering at the University of Cincinnati. He is the founding director of the Research Center for Distributed and Mobile Computing. His research interests include energy-efficient routing and information retrieval in ad hoc and sensor networks, effective handoff and multicasting in integrated wireless networks, interference analysis in piconets and routing in scatternets, use of directional antennas for enhanced QoS, scheduling of periodic real-time applications, and automatic load balancing in heterogeneous workstation environments. He received a DSc in electrical engineering from the Federal Institute of Technology, Lausanne, Switzerland. He is a Fellow of the IEEE and the ACM. Contact him at dpa@ececs.uc.edu.*