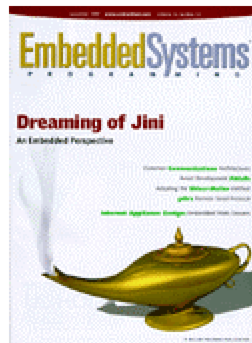
Jini In The Box

Paul J. Perrone <pperrone@assuredtech.com>
Venkata S.R.Krishna Chaganti <chaganti@erols.com>

Cover story featured in *Embedded Systems Programming*; November 1999; vol 12, num 12; pp 55-64.



What is Jini?

Building embedded systems is a tough problem. Providing a clean and flexible way to distribute the services of an embedded system is even tougher. Many of the problems that typical embedded system programmers face involve issues of memory, storage, processing power, and determinism. When you throw on top of that the issues that arise from the need to connect your embedded device to a network, such as selecting reliable communications infrastructures, standard communications interfacing, and dynamic broadcasting of device availability to potential network clients, you have a nice set of problems certain to fill your schedule between those all-too-frequent status meetings.

Jini is a collection of new Java APIs and components that is currently being heavily marketed and touted by Sun Microsystems as the next-generation technology for building distributed systems. Sun is specifically targeting the embedded systems marketplace as an industry that can use Jini to solve its unique set of problems. This article will describe the Jini architecture and provide a look at the potential architectural options currently available for using Jini in embedded environments.

Jini is a lightweight layer of Java code that rests atop the Java 2 platform. The Java 2 platform consists of the Java Virtual Machine and a set of Java APIs. The Java 2 platform is currently deployed as the Java 2 Standard Edition (J2SE) with APIs suitable for a wide range of application environments but typically ranging from very high-end embedded platforms to high-end desktop platforms. Sun has also announced the Java 2 Enterprise Edition (J2EE) with APIs useful in scaleable middle-tier server platforms and the Java 2 Micro Edition (J2ME) with APIs useful on embedded platforms.

Jini depends on Java's Remote Method Invocation (RMI) distributed communications infrastructure. Jini and RMI together offer a set of APIs and infrastructure for building distributed services that can dynamically register traits about themselves and register their availability to the Jini community such that Jini network clients can dynamically discover and make use of those services. The most significant types of distributed Jini-based services that Sun has in mind are those services offered by embedded devices on a network. Thus, with proper Jini-enabling, devices such as printers, phones, PDAs, pagers, and storage devices can all offer up their services to a dynamic network in a standard fashion. Jini clients such as other embedded devices and PCs can plug into this dynamic network and immediately obtain access to the Jini services made available to the network community. Jini and RMI together fulfill a role similar to that of other distributed paradigms such as the Object Management Group's Common Object Request Broker Architecture (CORBA). CORBA services such as the CORBA Trading service and the CORBA Orb infrastructure both drive toward satisfying goals similar to those being satisfied by Jini. Even though

Copyright 1999; Assured Technologies, Inc.; All Rights Reserved

CORBA is language-independent and an industry standard, the fact remains that Jini and RMI's ease of API usage and RMI's ability to better support mobile Java code make Jini and RMI an attractive alternative.

The Jini Architecture

The basic logical architecture of Jini and its system context is shown in Figure 1. As depicted in the diagram, Jini is dependent upon the Java 2 platform. Because Sun had not yet released the J2EE and J2ME platforms at the time of this writing, Jini is currently dependent on use of the J2SE.

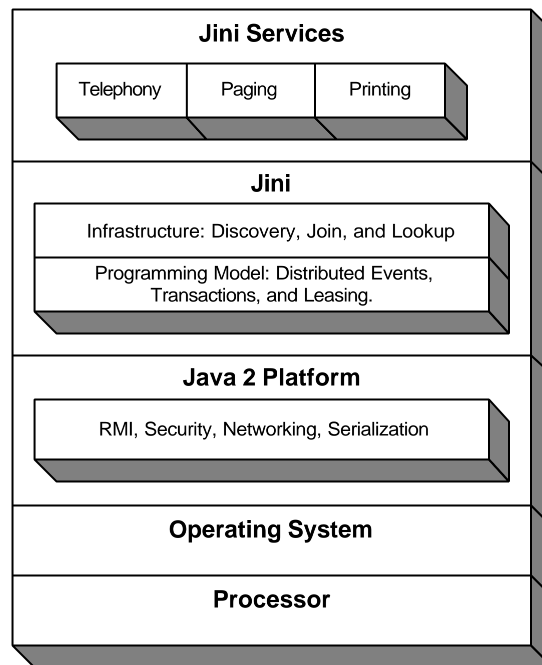


Figure 1: Jini Logical Architecture

The Java 2 platform offers a host of new security and distributed communication mechanisms on which Jini depends. RMI provides the underlying distributed communications framework for Jini. RMI provides a simple and extensible model for implementing distributed servers and handling client connectivity to these servers, passing Java objects between client and server either by value or by reference. RMI is implemented over a proprietary remote method protocol and has recently been extended to operate over the Internet Inter-ORB Protocol (IIOP). RMI/IIOP promises to offer CORBA connectivity to RMI clients and servers. RMI also provides a means to customize the underlying socket transport. The Java 2 security model provides a set of fine-grained access control mechanisms on which both RMI and Jini depend. Finally, a set of basic networking, object serialization, and other core Java language constructs are all also utilized by the Jini platform.

The Jini programming model comprises three distinct API collections for distributed events, distributed leasing, and distributed transactions. The distributed event programming model offers a set of interfaces which extend the JavaBeans event model to provide a means for remote objects to be notified of asynchronously generated events from other distributed objects. The distributed leasing model provides a set of interfaces which can be used to require distributed objects to obtain a lease for use of a particular distributed resource. Leases have a timeout associated with them before which they either must be renewed or cancelled by the holder. If the lease is not renewed or cancelled, access to the distributed resource expires and the resource is released from the holder. Leasing is used by higher-level Jini services to clean up resources no longer in use by potentially failed distributed objects. Finally, the distributed transactions

Copyright 1999; Assured Technologies, Inc.; All Rights Reserved

programming model is used to provide services for coordinating distributed operations that either all must occur atomically (commit) or guarantee that none occur at all (rollback).

The Jini infrastructure represents the quintessential layer of Jini utilized by distributed objects to first discover a community of Jini services, join a Jini community, and look up a particular Jini service. Objects first must locate a Jini community using a discovery protocol. The discovery process may involve simply finding local Jini communities via a multicast request, receiving notification from newly formed communities via multicast announcements, or simply binding to a known Jini lookup service. Each Jini community may have a group name and is made up of one or more Jini lookup services belonging to that group.

Jini services join those Jini communities that they have discovered and have an interest in joining. When a Jini service joins a Jini community's lookup service, a service item is provided by the Jini service to the lookup service. These service items contain a collection of meta-data attribute information describing the Jini service, as well as a service proxy object. The service proxy object is simply a serializable Java object that is downloaded to clients of the Jini service. Operations invoked on the service proxy object may tually be handled by the proxy object itself or marshaled as requests to a remote backend device proxy and marshal any responses (for example, service proxy as an RMI stub).

Lookup services are used by Jini clients to locate Jini services. The clients first fill out a service template scribing the service of interest. The service template can be filled out with a service identifier, a set of base Java class types, and a set of service meta-data attributes. A lookup call using this service template is invoked on the lookup service which subsequently returns a set of service matches containing one or more service items that matched the service template. With a set of matching service items in hand, the client can then extract the service proxy item of interest and begin distributed computing with that object.

Building Jini-enabled systems requires a couple of tools. You should be aware, however, that Sun charges minimal licensing fees for any usage of the Jini technology in commercial products (see http://www.sun.com/jini/licensing/). The Jini System Software Starter Kit is downloadable from developer.java.sun.com/developer/products/jini and the J2SE SDK is downloadable from java.sun.com/products/jdk/1.2. Jini communities require a run-time infrastructure to be set up before Jini services can even begin joining those communities. Setting up a run-time environment for Jini involves the following basic steps:

- Configure and start simple HTTP servers on hosts that will serve downloadable Java code. The Jini Starter Kit (JSK) comes equipped with a simple HTTP server.
- Configure and start an RMI activation daemon on hosts that will serve activatable Java code for automatically starting Jini services upon client requests. The RMI activation daemon is equipped with the J2SE SDK.
- Configure and start a lookup service. The Jini lookup service must have an RMI activation daemon running on the same host machine as it is running. A Jini lookup service is equipped with the JSK.
- You can also optionally configure and start a Jini transaction manager service that comes packaged with the JSK

Creating a Jini service is a relatively straightforward process. The basic steps for creating a Jini service are:

- Define a *Jini service interface*.
- Implement a *service proxy* class that is serializable and implements the service interface.
- Provide any *Jini client service code* that may be called by the service proxy object along with an HTTP server to serve such code.
- Provide any backend *device proxy* software or hardware devices to handle any network requests from the service proxy object.
- Implement a *Jini service registrar* which:
    - Creates an instance of a service proxy object.
    - Creates a service item containing a reference to the service proxy object and a description of the Jini service via a set of attributes.
    - Initializes a security manager.

Copyright 1999; Assured Technologies, Inc.; All Rights Reserved

- Implements a discovery process to locate one or more lookup services on the network.
- Implements the joining process by registering the service item with a lookup service of interest.
- Optionally implements use of a richer set of distributed event, leasing, and transaction services.

Jini clients are even simpler to create:

- Implement the discovery process to locate a lookup service as was done with the Jini service.
- Implement the creation and initialization of a service template.
- Implement the lookup of a Jini service using the service template.
- Implement the handling of service matches from a lookup to obtain a reference to the service proxy object.
- Implement usage of the service proxy object.

Embedding Jini

As we mentioned, Sun has been targeting the Jini platform for use in embedded environments. Those environments, of course, involve issues of memory, storage, processing power, and determinism that Jini services must address. The Jini Device Architecture Specification (http://www.sun.com/jini/specs/deviceArch.pdf) defines some general architectural options for dealing with these issues. Since the architecture options described in that specification are somewhat vaguely defined, we shall clarify what these options are, along with some refinements and an additional option.

The first (Type 1) architectural option, shown in Figure 2, houses a Jini service, Java 2 platform, and target hardware device all on the same host platform. Jini client APIs and Jini service registrars are needed on the service host platform for performing Jini discovery and registration of the service proxy along with an HTTP server to serve any code that must be downloaded by Jini clients when using the service proxy object. The device proxy object communicates with the service proxy object downloaded to the client. Distributed communications between the service proxy and the device proxy can make use of any distributed communications protocol and the device proxy can be implemented in any language. However, RMI and Java will typically be a good communications model and language of choice, respectively, because the service host in this case already needs RMI and Java for general Jini infrastructure communications. Of course, the protocol between the target hardware device and the device proxy can be of any nature.

This Type 1 option requires a minimum service/device host platform that can suitably host the J2SE, which usually means high-end embedded or low-end desktop platforms. That is, a service host here using the J2SE typically would assume a 32-bit processor, at least 32MB of RAM, and at least 24MB of ROM. While a complete J2SE environment provides maximum flexibility and richness of feature support for creating Jini applications, the service/device host platform needed here would clearly not be suited as an embedded platform itself, but rather as a larger platform hosting a smaller embedded hardware device.

Copyright 1999; Assured Technologies, Inc.; All Rights Reserved

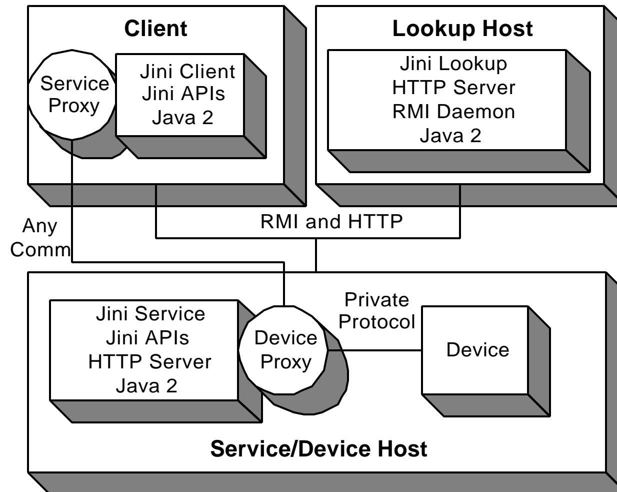


Figure 2: Service/Device Host Architecture Type 1

The Type 2 option depicted in Figure 3 is very similar to Type 1 with the exception that a pared-down, specialized Java Virtual Machine (JVM) is used instead of a full-blown Java 2 platform. This option may be pursued to overcome some of the memory and storage pitfalls associated with Type 1 architectures. The pared down Java 2 platform need only contain the classes used by the Jini client APIs, by the classes downloadable to the service host itself, and by any classes utilized by the Jini service registrar and device proxy. RMI would also need to be bundled with this option. However, dependencies on Java security libraries could be reduced if a scaled-down Jini lookup service is implemented on the service host platform. By implementing a scaled-down lookup service on the platform itself, no code would be downloaded from a remote lookup host and thus alleviate the need for a full-blown security manager. Of course, a major con of this approach is this option's code dependency on a specialized Java platform whereby any processor and Java platform modifications would cause a maintenance issue that is less of an issue with Type 1 architectures.

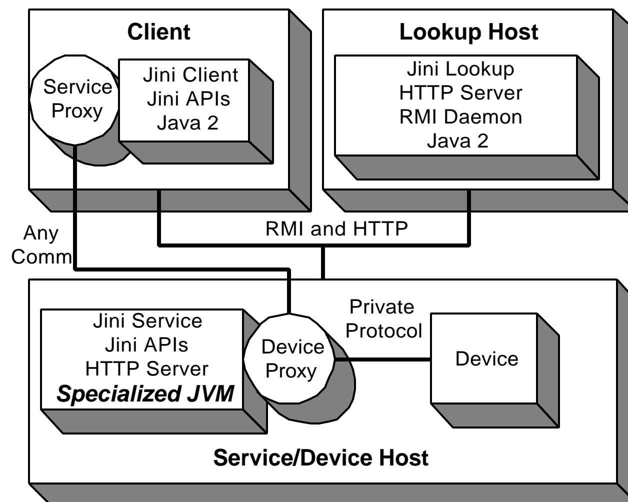


Figure 3: Specialized Java on Service/Device Host Architecture Type 2

The Type 3 option is shown in Figure 4. In this option, a number of devices are arranged in a device bay which also houses a service host platform. The service host platform houses all of the code for the Java platform, Jini platform, and Jini service server. A set of device proxies that communicate with the devices via a private protocol can be implemented in either Java or another language. Both non-Java and Java-

Copyright 1999; Assured Technologies, Inc.; All Rights Reserved

based device proxies will be able to use communications paradigms such as CORBA to communicate with the service proxy, whereas Java-based device proxies can also use RMI. The devices themselves can rest in a device bay such as a VME-based card cage and not have any additional requirements imposed on them, while still being able to utilize the connectivity to the Jini community provided by its service host in the same device bay. Thus, many existing devices with simple communications capabilities can be plugged into this configuration without further modification. However, the service host is still required, as well as a device bay to house the devices and service host. The service host and device bay then become single points of failure for multiple devices, as well as imposing additional costs for setting up such an infrastructure (if such a configuration was not already available).

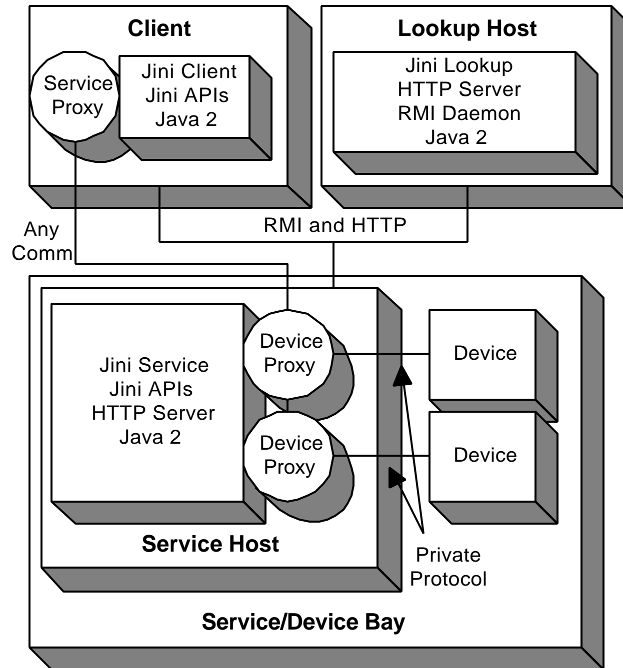


Figure 4: Service Host with Device Bay Architecture Type 3

Figure 5 depicts the Type 4 architectural option which is very similar to the Type 3 architecture with the exception that the devices are now available over a standard network (for example, the Internet) instead of resting inside of a device bay. The service host again houses all of the memory and processing power to support the Java and Jini platforms. Additionally, the device proxies may be non-Java or Java-based, as long as they can establish a communications session with the service proxy. The device proxies additionally must be capable of communicating with the devices themselves via some network protocol. This option has the advantage that devices will be usable off the shelf without further modification to support Jini. Redundant service hosts can be configured in this option to alleviate the single point of failure problems inherent in the Type 3 architectures.

Copyright 1999; Assured Technologies, Inc.; All Rights Reserved

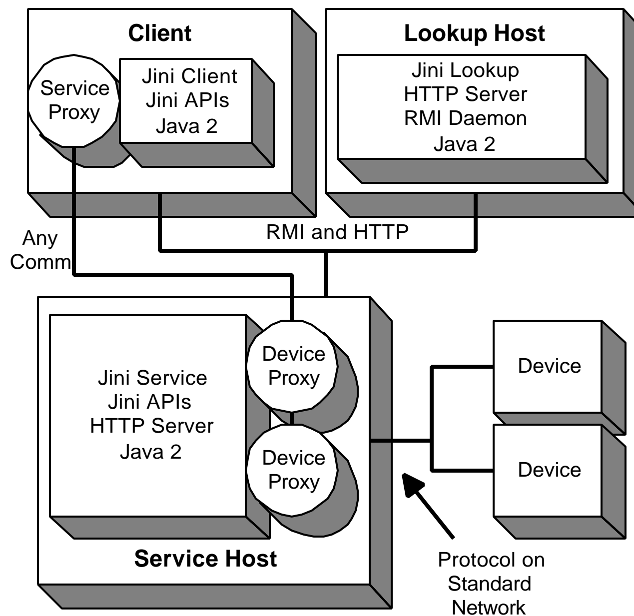


Figure 5: Service Host with Dependent Networked Devices Architecture Type 4

Finally, Figure 6 depicts the Type 5 architectural option, which illustrates how the service host independently manages the Jini service server environment on behalf of a device sitting somewhere on the network. All "device proxy" interfaces are embedded directly in the device platforms. The service host contains all Jini service server-side and client service proxy code that may need to be downloaded to the Jini client via RMI and an HTTP server. The only requirement on the device proxy is that it be able to speak with the service proxy object via some networking protocol. In the event that no networking capability was pre-built into your embedded device, you could conceivably use EmbeddedJava here to implement the device proxy, which communicates with the service proxy via a socket-based protocol. You could really use any language to implement such networking connectivity.

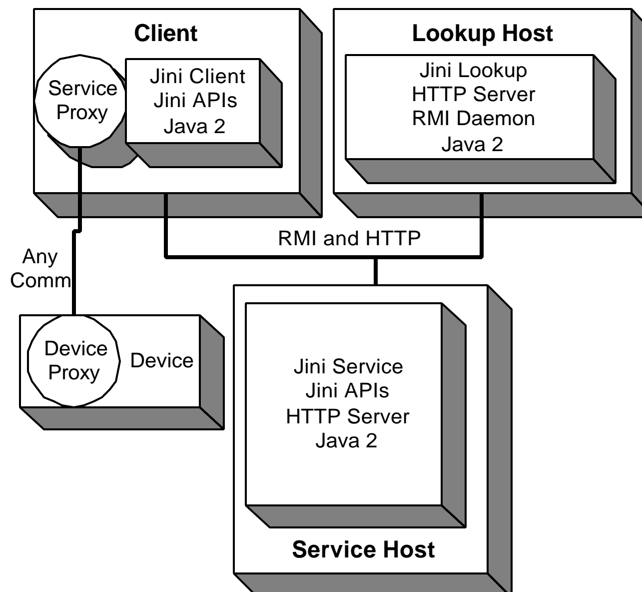


Figure 6: Service Host with Independent Networked Devices Architecture Type 5

Regardless of the various architectural options, the service host always assumes use of the J2SE. At the time we wrote this article, Sun's other embedded technologies, such as EmbeddedJava and PersonalJava, could not support the Java 2 platform required by Jini. Furthermore, the J2ME is currently not available for

Copyright 1999; Assured Technologies, Inc.; All Rights Reserved

use at all. Sun has made various announcements and is currently working toward enabling such embedded technologies usable in what we have called the service host platform. EmbeddedJava is expected to be integrated with the Java 2 platform by the year's end. PersonalJava is also expected to support RMI, other required Java 2 platform features, and Jini services by the year's end. When the J2ME becomes available, PersonalJava will simply be considered one "profile" of the J2ME geared for personal embedded device markets.

All of these options would enable a smaller service host platform and thus enable the added advantage of embedding all necessary Jini service infrastructure directly into your embedded devices. Until that time, the sole model for using Jini in embedded environments revolves around the use of a larger platform as service host acting on behalf of the smaller embedded devices.

The Next Step

Jini is being touted by Sun as the next-generation platform for building distributed systems with a specific eye toward usage in embedded markets. The model for use of Jini in embedded environments currently revolves around a fairly large service host platform capable of running the J2SE and the Jini infrastructure. Such a platform must always be present and configured to act with the Jini network community on behalf of an embedded device. Since Jini depends on the Java 2 platform, all of the architectural options outlined in this article currently require Jini service host platforms with enough memory, storage, and power to host the J2SE. The EmbeddedJava, PersonalJava, and J2ME platforms must be made compatible with Jini in order for Jini to be embeddable into the devices themselves. Of course, many of the architectural options we've outlined will still be applicable even when the J2ME becomes available and when EmbeddedJava becomes compatible with the Java 2 platform. When such events occur, the service host platforms present in the architectures defined here simply won't have the heavy memory, storage, and processing power requirements currently required to house the Jini service infrastructure.

RESOURCES:
Jini Licensing fees: http://www.sun.com/jini/licensing/
Jini System Software Starter Kit: http://developer.java.sun.com/developer/products/jini
J2SE SDK: http://java.sun.com/products/jdk/1.2
Jini Device Architecture Specification: http://www.sun.com/jini/specs/deviceArch.pdf

AUTHORS:
Paul J. Perrone is the founder, president, and a software consultant for Assured Technologies, Inc. Paul has been a key player in the architecture, design, and development of numerous large-scale n-tier distributed enterprise systems and products. Paul's key software technology expertise areas are object-oriented development, Java, CORBA, and C++. Contact Paul via e-mail at pperrone@assuredtech.com.

Venkata S.R. Krishna Chaganti is a software designer and developer with many years of experience in developing software solutions for a variety of corporations and organizations. Krishna's key experience is with Java, C++, distributed computing, and database systems. Krishna has also served as an instructor of Java programming techniques for two years.

Copyright 1999; Assured Technologies, Inc.; All Rights Reserved