Jini™ Device Architecture Specification

The Jini ™ technology is a Java™ platform-centric distributed system designed around the goals of simplicity, flexibility, and federation. The Jini architecture allows a group of machines or programs to enter into a federation, offering resources to other members of the federation and using resources as needed. These resources appear to the client of the resource as an object in the Java programming language, but can be implemented in either software or hardware. This document discusses some alternative ways of accomplishing such implementations in hardware, showing some of the tradoffs that can be made between functionality and device complexity.

THE NETWORK IS THE COMPUTER™

901 San Antonio Road
Palo Alto, CA 94303 USA
415 960-1300
fax 415 969-9131

Revision 1.0
January 25, 1999

Copyright © 1999 Sun Microsystems, Inc.
901 San Antonio Road, Palo Alto, CA 94303 USA

All rights reserved. Copyright in this document is owned by Sun Microsystems, Inc.

Sun Microsystems, Inc. has patent and other intellectual property rights relating to implementations of the technology described in this Specification ("Sun IPR"). Your limited right to use this Specification does not grant you any right or license to Sun IPR. A limited license to Sun IPR is available from Sun under a separate Community Source License.

THIS SPECIFICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY YOU AS A RESULT OF USING THE SPECIFICATION.

THIS SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE SPECIFICATION. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE SPECIFICATIONS AT ANY TIME, IN ITS SOLE DISCRETION. SUN IS UNDER NO OBLIGATION TO PRODUCE FURTHER VERSIONS OF THE SPECIFICATION OR ANY PRODUCT OR TECHNOLOGY BASED UPON THE SPECIFICATION. NOR IS SUN UNDER ANY OBLIGATION TO LICENSE THE SPECIFICATION OR ANY ASSOCIATED TECHNOLOGY, NOW OR IN THE FUTURE, FOR PRODUCTIVE OR OTHER USE.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-1(a).

TRADEMARKS

Sun, the Sun logo, Sun Microsystems, Jini, JavaSpaces, JavaSoft, JavaBeans, JDK, Java, HotJava, HotJava Views, Visual Java, Solaris, NEO, Joe, Netra, NFS, ONC, ONC+, OpenWindows, PC-NFS, EmbeddedJava, PersonalJava, SNM, SunNet Manager, Solaris sunburst design, Solstice, SunCore, SolarNet, SunWeb, Sun Workstation, The Network Is The Computer, ToolTalk, Ultra, Ultracomputing, Ultraserver, Where The Network Is Going, Sun WorkShop, XView, Java WorkShop, the Java Coffee Cup logo, and Visual Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Contents

1. Introduction **1**

1.1 Requirements from the Jini Lookup Service 2

1.2 Comments 4

2. Basic Device Architecture Examples **5**

2.1 Devices with Resident Java Virtual Machines 6

2.2 Devices Using Specialized Virtual Machines 8

2.3 Clustering Devices with a Shared Virtual Machine (physical option) 9

2.4 Clustering Devices with a Shared Virtual Machine (network option) 11

2.5 Jini Software Services over the Internet Inter-Operability Protocol 13

Introduction 1

The Jini technology infrastructure is built around the model of clients looking for services. The notion of a service encompasses access to information, computation, software that performs particular tasks, and in general any component that helps a user accomplish some goal. Services can themselves be clients of other services, and can be grouped together to provide higher-level functionality.

The Jini architecture requires a service to be defined in terms of a data type for the Java programming language, that can then be implemented in different ways by different instances of the service. A service can be a member of many different types, allowing a single service instance to provide a variety of functionality to clients. This is a standard practice in object-oriented software. However, the distributed nature of the Jini system allows data types for the Java programming language to be implemented in a combination of software *and hardware* in a way that is unique.

The core of the idea that enables this implementation flexibility is quite simple. Services are defined via an interface, and the implementation of a proxy supporting the interface that will be seen by the service client will be uploaded into the lookup service by the service provider. This implementation is then downloaded into the client as part of that client finding the service. This service-specific implementation needs to be code written in the Java programming language (to insure portability). However, since this code comes from the actual instance of the service being used, it can know in great detail the specifics of the particular service implementation for which it is the proxy. Not only can the code that is downloaded know about the software used to

implement the service, the code can know specifics about the hardware on which the service resides. In the limit case of this, the hardware could be all that there is to the service, and the downloaded software could act as a network-level device driver, taking method calls in the Java programming language from the client and generating specific, hard-coded requests to the hardware on the other end of the network wire.

This approach to services requires that there be a piece of code written in the Java programming language that can be downloaded by the client of the service, and some hardware that ultimately runs the service. Between these two points, however, there are a number of options concerning the software structure, hardware structure, and location of components that can be chosen by the service provider. These options allow trade-offs to be made in the functionality provided and the cost of the underlying hardware.

In what follows, we begin by discussing in more detail the requirements placed on a service to be part of the Jini system. We then discuss some examples of combinations of software and hardware that can be used to implement Jini-capable services once the specialized implementations in hardware begin to play a role.

1.1 Requirements from the Jini Lookup Service

The actual offering of a service places very few requirements on the entity making the offer; indeed, it is possible to implement a device using Jini software services that offers a service in such a way that the code written in the Java programming language downloaded by the client transmits bit patterns to the hardware that are directly interpreted. In such cases, the amount of intelligence needed for a Jini device is minimal. The code written in the Java programming language could talk directly to the device controller in much the same way that the device would be talked to if it were on the local computer's bus (with, of course, some modifications for dealing with the network-centric aspects of the communication).

Unfortunately, providing a service is only part of what is needed to be a Jini service. To be part of a Jini system grouping, a service must also be able to participate in the Jini Discovery protocol, and register itself into the local Jini Lookup service. This is how a service makes itself known to the djinn, and how the service is accessed by other members of the djinn.

These two requirements are intimately connected. The major goal of the Jini Discovery protocol is to allow a device or service to obtain a Java Remote Method Invocation (RMI) reference to the local Jini Lookup service. Once this reference has been obtained, the service needs to register itself in that Jini Lookup service, allowing other participants in the djinn to find and use the service.

The interface to the Jini Lookup service is a full RMI interface, and the implementation of that service uses all of the mechanisms of RMI, including the distributed garbage collection and the dynamic downloading of code. As such, there is an implicit assumption that the service that holds a reference to the Jini Lookup service lives inside a full Java™ Virtual Machine (JVM), at least capable of running the full RMI system.

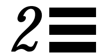
This assumption is most evident if we consider the possibility of alternate implementations of the Jini Lookup service, which might support remote interfaces beyond that specified by the Jini Lookup service itself (currently the interface `net.jini.core.lookup.ServiceRegistrar`). Such an implementation would have a different RMI proxy than the current implementation, which would be downloaded if the device had a full JVM and RMI runtime. Devices without a full JVM and RMI runtime would need a different way of dealing with such implementations of the service.

In addition to the need to download the stub code for the Jini Lookup service, registering with the service requires the creation of an object of type `net.jini.core.lookup.ServiceItem`, which is itself made up of a set of objects in the Java programming language. Maintenance of these entries in the Jini Lookup service can require the creation of other objects in the Java programming language of the type `net.jini.core.entry.Entry`. All of these objects are most easily constructed by using a running JVM.

Finally, registrations with the Jini Lookup service are leased, with the lease that is returned requiring renewal for the service to continue to be shown in the lookup service. The specification of the lookup service does not include a specification of the lease object that is returned by a registration. All that is specified is an interface written in the Java programming language that must be supported by the (local) object that is returned as the lease. Thus the design of the Jini Lookup service requires that the code that implements the class that in turn implements the `net.jini.core.lease.Lease` interface be downloaded into the service that registers so that the lease can be renewed.

1.2 Comments

Please direct comments to `jini-comments@java.sun.com`.

Basic Device Architecture Examples 2 

In this chapter, we will look at three different approaches for implementing a Jini service in hardware. Each of the approaches will look the same to a client of the service. Each approach takes a different route to interacting with the Jini Lookup service and in providing an interface written in the Java programming language to clients of that service. In each case, a different trade off was made between the complexity of the device, the flexibility of the device, and the directness of the communication between the client wanting to use the service and the device that implements the service.

All but the first of the examples make use of *interposition*, that is, the ability of a service to add a proxy between itself and the client of the service. This proxy can be used by the service as an agent to the Jini technology infrastructure, off-loading from the service some of the work needed to join the Jini system federation.

The examples given in this chapter are not the only options open to the service designer who wishes to produce a service that includes a hardware component. Rather, the examples are meant to show some samples of the range of implementation possibilities that are open to such designers. In effect, this document is meant to show that, within the overall Jini architecture, there is no single Jini device architecture. Instead, the device space is freed up allowing different services to have hardware implementations with different price, performance, functionality and flexibility design points.

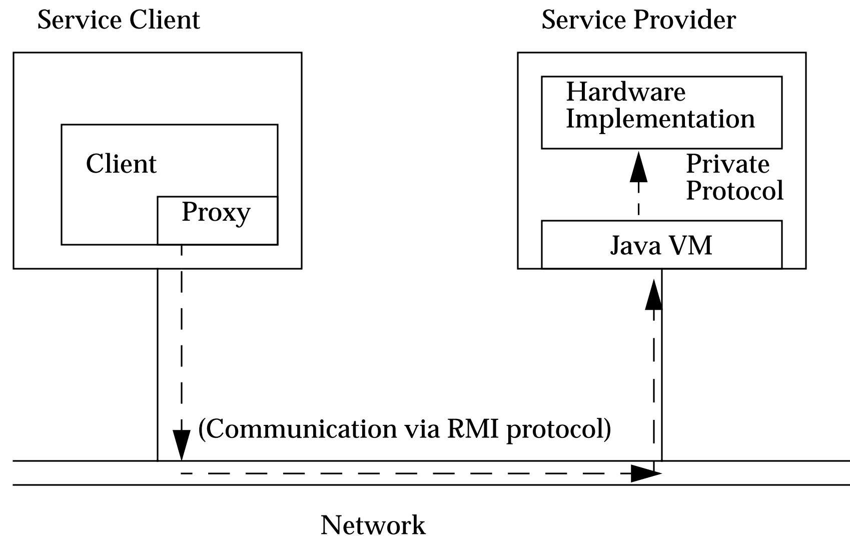


Figure 3.1

2.1 Devices with Resident Java Virtual Machines

An obvious design for a device that can become part of a Jini system federation is one that includes the computing power, memory, and non-volatile store necessary to have a full JVM and those parts of the Java application environment necessary to support the Jini infrastructure (in particular, those parts needed for code loading, RMI, and any required security). This would make the device into a specialized computing entity, with part of the device dedicated to the parts of the Java API required by the Jini architecture. On this approach, the hardware implementation is abstracted behind a device-local software abstraction, which in turn is abstracted behind the proxy code used by the client to contact the service. This sort of architecture is shown in figure 3.1.

Such a device would be able to make full use of Jini and Java technology, uploading code that is used to communicate with the device and downloading code that might be needed for the service provided by the device. Such a device can make use of the native RMI protocol for communication over the

network, and has a loose tie between the communication protocol and the particular software protocol governing the running of the device itself. On this approach, the device becomes a specialized network appliance offering a particular service (or set of services) via an embedded Java platform.

In effect, this approach uses a hardware implementation for the local implementation of an RMI server, isolating the hardware behind two levels of indirection. The first is that provided by the local proxy code that is uploaded into the Jini Lookup service and then downloaded into the client of the service. Additionally, the local JVM and code written in the Java programming language resident on the service device allow mediation between the client proxy and the hardware itself.

A device which took this approach could easily have multiple services implemented on the device in a way that was mediated by the JVM on the device. Further, such a device could be evolved with no impact on the client or the network protocol used between the client and the service, since any change in the hardware would only be seen by the JVM and any server-side code that directly talked to the hardware.

While simple and flexible, this approach does add some cost to the device. In particular, the device would need to have a microprocessor capable of running the JVM, some memory in which to create and store classes, and some non-volatile store (either disk or NVRAM) from which to load the JVM and Java™ Development Kit (JDK) software classes. All of these are in addition to the hardware needed to implement the Jini service that the device provides. This extra hardware will increase the cost of producing the device.

Meeting these requirements does not call for a hosted version of the JVM, or a full version of the JDK running on the device. The JVM could run on any form of micro-kernel or directly on the hardware of the device. Further, there are large parts of the JDK that would not be required for the minimal device—such things as the graphics and UI classes would not be needed, which form a significant chunk of the current release. Other parts of that release could also be dropped, allowing a “stripped down” JDK to suffice for Jini devices.It would be worthwhile to determine the exact definition of such a subset of the JDK and size that component; it would be something close to the definition of embedded Java technology with the additional classes needed to support RMI.

What is important for this kind of approach is for the device to be able to download any code written in the Java programming language (although whether that code is run could depend on the local security manager), utilize

the RMI communication system, and handle the requirements of a general virtual machine. By presenting a standard JVM, the device gets full membership in a Jini system federation and complete flexibility in the ways in which the machine communicates between the proxy it provides other members of the federation and the device itself.

2.2 Devices Using Specialized Virtual Machines

We can lower the barrier to entry for a device manufacturer if that manufacturer is willing to give up some of the flexibility given by the Jini distribution architecture. This can be done by allowing the device to become part of a Jini system federation with a specialized virtual machine that is tuned to allow only those operations needed by the Jini Discovery protocol and Jini Lookup service.

To do this, the device manufacturer would need to implement the interfaces to the Jini Discovery and Jini Lookup service in the device itself, include specialized knowledge of the kind of leases that are handed out by the Jini Lookup service and be able to renew those leases directly, and have sufficient functionality to download and use the stubs for these services. This is a particular set of functionalities that is considerably smaller than required by the whole of the JVM, and should be capable of being implemented in much less code. For example, such a JVM would not need to contain a security manager, a code verifier, or a number of the other components that are required for a full JVM.

Such a device would contain a JVM specialized for the Jini environment, allowing the Jini Discovery and Jini Lookup services to be accessed and leases of a particular sort to be renewed. This would limit the flexibility of such a device, as the device would not be able to have software changes made over time to the protocol used by the proxy for the device. The specialized knowledge of the kind of lease that is handed out by the lookup service would also tie such a device to a particular implementation of the lookup service. However, this penalty in serviceability may not outweigh the simplicity of the overall device.

2.3 Clustering Devices with a Shared Virtual Machine (physical option)

A third approach uses a full JVM, but amortizes the cost of the JVM (both software and hardware) over a number of different devices. On this approach, a group of devices each uses a physically co-located JVM as an intermediate layer between the device and the Jini system grouping. The device loads code written in the Java programming language into this local virtual machine allowing that local machine to interact with the device, and then delegates to the local JVM the requirements of interacting with the Jini Lookup service, Jini Discovery, and Jini Leasing.

This approach is very much like the first one discussed in this section, except that the JVM used by the devices is shared. It is still a full JVM, allowing the downloading of code and complete Java platform functionality. However, the most likely implementation of such a device would allow multiple (and perhaps different) kinds of physical devices to be plugged into the overall device to get the sharing of the Java application environment.

Such a device might best be thought of as a “Jini device bay.” This bay could provide power, a network connection, and a processor running a JVM and appropriate parts of the JDK. Physical devices used to provide a particular kind of Jini service could be plugged into the device bay and announce themselves to the bay in whatever way the two decided was appropriate. This could be using a proprietary protocol (allowing a device manufacturer to produce both the basic device or devices and the device bay) or some other industry standard, local-device identification scheme.

As part of the local announcement, a new device would tell the device bay where to find the code written in the Java programming language that is needed by a client of the service, and (possibly) where to find code that would allow the device bay to interact with the device. This allows devices to carry their own “drivers” both for the local machine and at the network level.

Upon detection of the new local device, the Jini device bay would register the services provided by the new device (previously known by the device bay) with the Jini Lookup service. It would be the role of the device bay to renew leases on the Jini Lookup service entries, and to detect removal any of the devices for which it was acting as proxy. The device bay would provide the Jini Lookup service with the code handed to it by the device so that service clients could download that code.

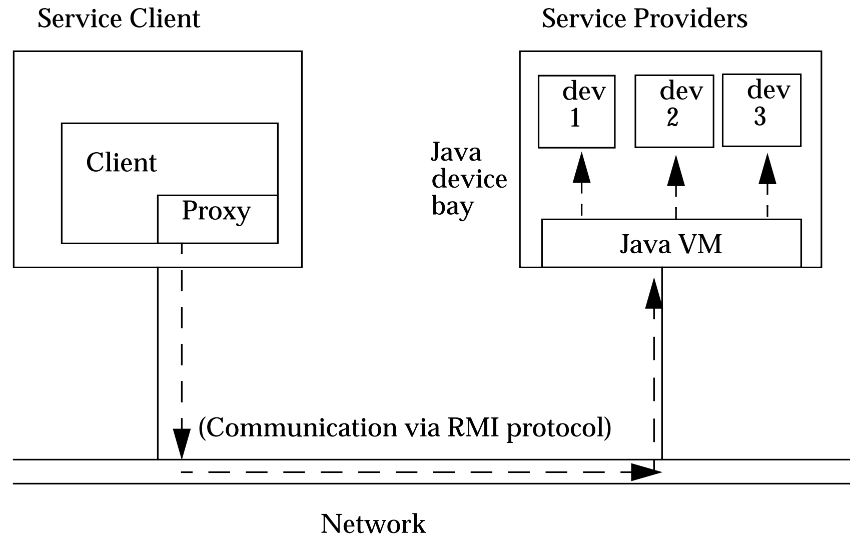


Figure 3.2

The client of the device service would believe that it is talking to the device registered in the Jini Lookup service, but would actually be talking to the device bay. The device bay would act as a dispatcher to the particular device for which it was acting as a proxy, along with any translation of protocol between the network protocol used by the service proxy and the protocol used between the device bay and the actual device. Graphically, the architecture of such an approach is shown in Figure 3.2.

The savings for the device manufacturer in this case comes from the ability of multiple physical devices to share a device bay, which contains the intelligence, memory, and perhaps other components (such as the power supply). By sharing these resources among multiple devices, the extra cost and engineering needed to interact with the Jini system federation can be amortized over a large number of devices.

The cost of this approach to the device manufacturers is that the protocol between the device acting as the Jini device bay and the devices that are placed in that bay must be defined in advance and cannot change over time. Because

there is no way of introducing dynamic behavior in the particular devices, the pairing of device and Jini device bay must be controlled and known beforehand.

It should be noted that the Jini device bay itself is a Jini device, which can be thought of as providing services to those devices housed within it. As such, it could be a revenue item in its own right. Variations in the implementation could be provided to support various internal announcement protocols (device bay, jetsend, etc.) or hardware busses (including “network”-like busses such as firewire).

2.4 Clustering Devices with a Shared Virtual Machine (network option)

A variation on the device bay approach utilizes the network rather than a physical enclosure and backplane. On this alternative, a proxy for the JVM used by the various service devices would exist on the network. Service devices could be added to the network, discover the existence of such a proxy device, and register with that proxy. Such a registration could include the code written in the Java programming language needed by a client of the device (either directly or as a URL to use to obtain the code) and code needed by the proxy to communicate with the service device.

When a service device registers with such a network proxy, the proxy device would register with the Jini Lookup service on behalf of the service device, thus allowing the service device to become a part of the Jini system federation. Requests to the new service would go first to the proxy for that device, which could then forward the requests (after appropriate protocol translation) to the

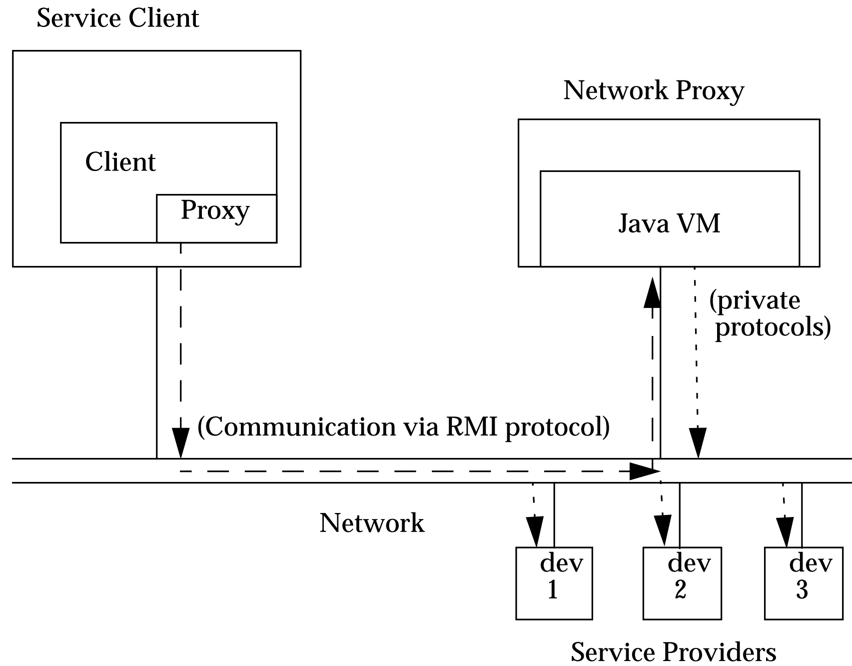


Figure 3.3

particular service device. In addition, the proxy could handle the Jini-specific tasks such as renewing leases for the service. This alternative is shown in Figure 3.3.

This alternative requires somewhat more hardware for the individual device, as it requires each service device using such a proxy to be able to be placed on the network and have its own power supply and network connection. However, the devices would not need individual CPUs, memory, or persistent store; all of that would be provided by the networked Jini device proxy.

Devices using this option would need to have a protocol parallel to the Jini Discovery protocol between the individual service devices and the network proxy for those devices. This could be a specialized code on the network, known in advance, that the devices can use to identify themselves to the network proxy. This will have to be particular to the device and the proxy for that device. However, once this protocol has been decided upon, no other intelligence needs to be built into the device. All of the intelligence can be built in to the network proxy, perhaps uploaded into the proxy by the service device

(which could easily carry code written in the Java programming language, even though it cannot execute that code). The protocol that is used by the network proxy to talk to the devices for which it is a proxy also needs to be statically defined in advance, and cannot be changed. However, it can be any protocol that is needed by the particular device.

On this approach, the individual devices will be more complex than they would be on the Jini device bay approach. However, the number of devices that can be serviced by a network available proxy is not limited by the physical constraints of the proxy device. Nor is there any requirement that the devices and the proxy device be co-located, which is a requirement on the physical clustering scheme.

This is also the approach that can be taken to build “gateways” between the Jini devices and other network-managed devices. Such devices, that already speak a particular protocol, can be spliced into the Jini system federation by providing a network proxy that speaks the Jini protocol on behalf of such devices, and the existing specialized protocol to such devices. This is the approach that can be used to add consumer electronic devices, factory controls, or home environment controls into the Jini system grouping.

2.5 Jini Software Services over the Internet Inter-Operability Protocol

A final method for connecting devices or services that are not purely Java software based into Jini federation centers on using the Object Management Group (OMG)’s Internet Inter-Operability Protocol (IIOP). This protocol defines a standard for data transmission that will be supported by a subset of RMI.

This approach relies on the ability of a device to read an IIOP stream directly, either because the device includes an implementation of a Common Object Request Broker Architecture (CORBA) Object Request Broker (ORB), or because the device knows what IIOP streams to expect and can interpret those known-form streams directly.

This approach requires that the Jini Lookup service supplies implementations of its interfaces over both the native RMI protocol and the IIOP protocol. This is supported by RMI over IIOP as long as the interfaces conform to any subsetting requirements established by the OMG. At the current time, it appears that the Jini Lookup service interfaces are in conformance with the RMI over IIOP subset.

Devices that contain a CORBA ORB could directly interact with the Jini Lookup service, using the IIOP protocol. The fact that the Jini Lookup service generated this protocol via RMI would be transparent to the service itself, and the fact that the service was using a method other than RMI to reply to the Jini Lookup service (to renew leases, for example) would be transparent to the Jini Lookup service. Current differences between the RMI programming model and the CORBA programming model would need to be dealt with by the device itself; for example, the device would not be able to download the implementation of the stub for the Jini Lookup service, and would need an implementation of the Jini Lease class used by the Jini Lookup service.

Devices that do not include a CORBA ORB could directly interpret the IIOP stream and attempt to interact with the Jini Lookup service. This approach requires very little software support on the side of the device (since the bitsteam from the wire is being directly interpreted). However, it is an approach that will only work with known versions of the Jini Lookup service that exports known implementations of a Jini Lease. Any alteration of either the Jini Lease implementation or the protocol used by the Jini Lookup service, even those that would be invisible to other clients of the service, would make it impossible for the device directly interpreting the IIOP protocol to interact with the new version of the service. Hence this alternative, while lowest in cost with respect to the hardware and software needed by the device, is also the least reliable in the face of implementations that can change over time or which are open to alternate implementations.