Connecting non-Java devices to a Jini network

Charles E. McDowell and K. Shankari
Computer Science Department,
University of California, Santa Cruz, CA 95064.
Email:{charlie,shankari}@cse.ucsc.edu

Abstract

Recently, several schemes have been proposed to interconnect extremely small devices in a plug and play *manner. One of these is Sun Microsystems' Jini^{TM} – which defines a* federation *of entities which can communicate with each other. Once a device joins a Jini federation, client programs anywhere on the network can look it up in a central directory and use it by dynamically downloading a* proxy *which represents the device.*

Jini uses the Remote Method Invocation(RMI) mechanism provided by Java at its heart and any entity which wants to participate in a Jini federation requires an RMI-enabled Java Virtual Machine. Because embedded devices generally lack the resources to support an RMI-enabled JVM, some other method has to be found to hook them up to a Jini federation. In this paper, we explore the issues involved in connecting non-Java devices to Jini federations by using a Java-based surrogate running on a more powerful machine in the same local network.

1: Introduction

In a world where electronically controlled devices are fast becoming a way of life, interest has increasingly focused around the methods to connect these small devices together with a minimum of effort. Several attempts to produce a solution have already been proposed, including CORBA[1], Inferno[2] and DCOM[3].

Jini technology from Sun Microsystems is the latest entrant into this field[4]. It eliminates the issues of manual installation and configuration associated with *plug and play* by using Java's platform independence, security and code mobility.

Installation: Drivers are downloaded from a code server when required, as described in Section 3 on page 7.

Configuration: A proxy which represents the device, and which stores all the configuration parameters for the device, is downloaded from a central **lookup server**.

Obsolescence: Because drivers are downloaded on demand, changing the drivers on the code server will automatically ensure that the correct drivers are used the next time the device is connected to a client.

Different systems: Because of Java's platform independence, only one driver has to be written for all systems.

All these benefits accrue to the user of the driver – the client program. There is no *fundamental* reason why the code used to control the device should be written in Java. On

the other hand, any program which wishes to interact with a Jini *federation* must contain Java code because Jini is inextricably linked to Java.

The complete Java Runtime Environment(JRE) 1.2, which contains a full-featured virtual machine, but no compilers or other tools, is around *20 MB.*[1] This is too big for most embedded devices, especially ones with no secondary storage devices. For these devices, it would be best to write the code which controls the device in a compact language such as C or assembler, and to delegate the task of interacting with the Jini federation to a Java-based surrogate which can run on a machine with more resources. This paper discusses the issues involved in designing such a surrogate – the Registration Proxy Server(RPS), and defines a protocol for communication between the RPS and the device.

The source for the standard Jini specifications is the Sun Microsystems' Jini site [4]. There are also third-party web-sites like Artima [5] which provide resources for developers. This field is relatively new, and contains very little published literature. Some of the more important papers include an overview by Jini's lead architect [6] and another on the security implications of Jini [7].

The four main components of a Jini system are described briefly below.

Service: A software or hardware component which can be used by other components. Here, the embedded device provides the service.

Lookup Service: A centralized service which maintains a list of all services on the local network. Services *register* themselves with the lookup service and clients *lookup* the services they are interested in.

Client: The program which utilizes the service.

Proxy: The Jini equivalent of a driver – a piece of code which abstracts the functionality of the service and provides a uniform interface to the client.

The three methods suggested by "The Jini Device Architecture Specification"[8] for connecting non-Java devices to a Jini network are described below.

Device Bay: The embedded devices are physically plugged into a Java-enabled *device bay.* Subsequently, the proxy communicates with the device through the device bay.

Network Surrogate: The embedded devices contact a Java-enabled *network surrogate*[2] over a network. Subsequently, the proxy communicates with the device through this surrogate.

CORBA: The embedded devices use IIOP(the CORBA communication protocol) to directly interact with a lookup service which can communicate using both RMI and CORBA. This can be viewed as Jini implemented over CORBA rather than RMI.

We considered a fourth approach – that of a Registration Proxy Server (RPS). This is very similar to the network surrogate approach, the difference being that the non-Java device utilizes the RPS only to perform the Jini interactions on its behalf. The proxy and the device communicate directly once these interactions are complete.

We preferred the **Registration Proxy Server(RPS)** method for the following reasons:

➤ If a device bay is used, the number of non-Java devices which can be connected is limited by the slots on the device bay. On the other hand, if the RPS method is

[1] Jini can run only on JDK1.2 or greater

[2] This is called a *Network Proxy* in the Jini documentation. We have chosen to call it the *Network surrogate* to minimize confusion with the proxy code for the service.

used, the number of non-Java devices connected is limited only by the number of simultaneous connections the RPS can handle.

- If a device bay is used, the client talks to the device bay which then talks to the service. Since the device bay is programmed in advance, it can only implement a fixed set of network protocols. Therefore, the client and the service are also restricted to communicating through that fixed set of network protocols.
  On the other hand, if an RPS is used, the client and the service communicate directly with each other after the initial download is completed. This allows for greater flexibility in the choice of network protocols for communication.
- Unlike in the network surrogate approach, once the client downloads the proxy, the Java-enabled device has no role in subsequent interaction between the client and the device. This has the advantages of a lower load on the Java-enabled device, and a more robust system because the crash of the RPS need not affect existing client – device connections.
- Unlike the CORBA approach, this does not require complex code to implement an ORB in the device, and makes no assumptions about the lookup service implementation.

The simplest form of interaction between a non-Java device and a Jini federation, using a Registration Proxy Server is outlined below.

1. The non-Java device starts up and tries to locate RPSs.
2. After locating an RPS, the non-Java device sends the URL where the class used to communicate with it (its *proxy*) can be found.
3. The RPS finds the Fully Qualified Class Name of the proxy from the contents of the className.class file located at the proxy URL. It downloads the proxy from this location and instantiates it.
4. The RPS attempts to *discover* Jini lookup services. After discovering such a service, the RPS *registers* the proxy for the device.
5. The client starts up and tries to *discover* a Jini lookup service.
6. Once a Jini lookup service is found, the client *looks up* the service it wants to use. For example, the client may try to find all washing machines registered with that Jini lookup service, or it might look for the specific washing machine in the basement.
7. The client downloads the proxy for the service it wishes to use and uses it to communicate with the service using a (possibly) proprietary protocol.

We now consider the design of the service and the RPS, and of the communication protocol between them.

2: RPS protocol design

The protocol used to communicate between the proxy and the service has to consist of at least 4 states. We do not have the address of the RPS and there could be multiple RPSs on the local network. Therefore, the protocol has to have at least 2 phases – one to find RPSs and the other to select one in case several are found. Considering the standard request/response mechanism for each phase, we see that we need a minimum of 4 states.

Rather than designing a protocol from scratch, we decided to modify an existing protocol to obtain one that meets our specifications. We chose to modify the Dynamic Host Configuration Protocol (DHCP)[9], a 4-stage protocol which allows computers to discover DHCP servers, select one of them, and obtain an IP address from it.

2.1: Service-RPS protocol

Since we need a way for the service to contact the RPS without knowing its address, we used UDP broadcast over a local network. However, UDP provides no service guarantees, and lost packets or errors in transmission could easily lead to *half-open* connections. We could handle all the errors in the same way that TCP does, but this would lead to unnecessary duplication of code and an increase in the complexity of the client. Therefore, we decided to confine UDP to the initial broadcast and switch to TCP subsequently. This initial broadcast does not suffer from the possibility of a half-open connection because it is used to probe for connections, and if the packet is lost, neither side would assume that a full connection has been established.

Two parameters of this protocol remain to be standardized. Standardization is required to allow devices from different manufacturers to use the same RPS, instead of having a different RPS for every manufacturer. Standardization is especially important in this case, because the programs which interact with the RPS run on embedded devices, and should not require user configuration. The parameters to be standardized are described below.

protocolPort: This is the default port (similar to port `80` for HTTP) at which the RPS can be contacted to initiate the protocol. We have currently set this parameter to `2326`.

tcpPortFunction: As we have discussed, the first packet, `DISCOVER`, is sent by the service using UDP broadcast, but RPS sends the next packet, `OFFER`, using a TCP connection. This means that the RPS must have some method of determining the TCP port number given the UDP port number, or in other words, we need a function f such that

$$\texttt{tcpPort} = f(\texttt{udpPort})$$

We have currently set this function to $f(x) = x + 1$, or in other words, $\texttt{tcpPort} = \texttt{udpPort} + 1$.

Since we use UDP broadcast to send the first message, we would like to transmit a message which can fit within one UDP packet, so that we can avoid broadcasting multiple packets. Hence, like DHCP, we chose a message length of 440, which is well within the minimum IP datagram size a host must be prepared to accept[10].

Figures 1 and 2 show the various stages in the service and RPS protocols. A brief description of the various states, and the actions performed in each follows. Further details can be found in the associated Technical Report[11].

INIT: In the `INIT` state, which is not shown in the figure, the service and RPS perform initialization tasks. The RPS also begins listening to the `protocolPort` for connections from services. When a connection is received, the RPS determines the sender's address and port from the UDP header, and moves to the `DISCOVER` state.

DISCOVER: In the discover state, the service attempts to contact RPSs on the local network by broadcasting a `DISCOVER` packet to the standard broadcast address of

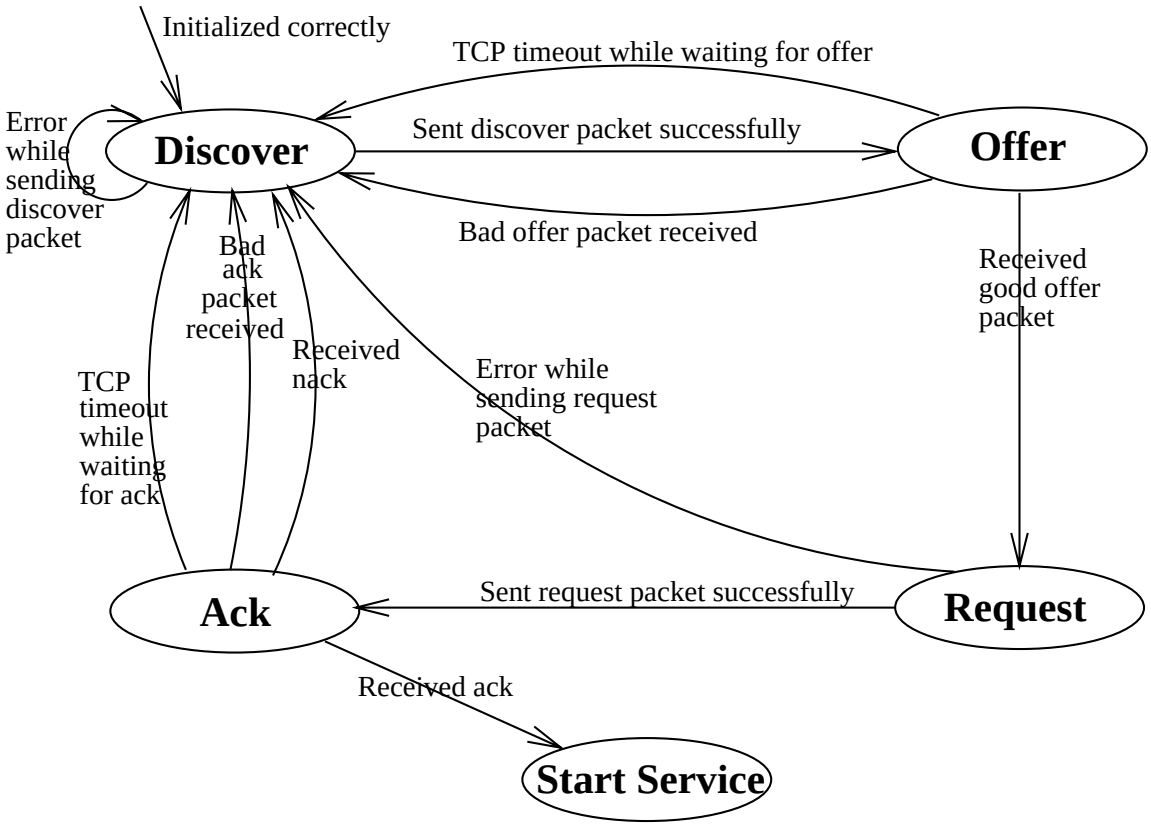

Figure 1. Finite state machine description of the protocol on the service side

`255.255.255.255`. The `DISCOVER` packet contains

lease length: the length of time the service wants to be registered at the lookup service,

refresh length: the time after which the service wants the RPS to check if it is alive, and

proxy URL: the location where the `.class` file for the proxy associated with this service is stored.

The RPS simply parses the three parameters sent by the service.

OFFER: In this state, the RPS first determines the TCP port to which the `OFFER` packet will be sent. This calculation is done using the received `udpPort` value (`tcpPort` = f(`udpPort`)). Next, it determines whether or not to register the service based on some combination the service's IP address, the TCP port, the proxy URL and the lease and refresh lengths requested. This combination can be part of the RPS configuration. If it does decide to register the service, it sends an `OFFER` packet containing the lease and refresh lengths it is willing to handle. These might be less than or equal to the values requested by the service, but they cannot be greater.

The service may receive responses from different RPSs on the local network. Depending on the implementation, the service can choose to respond either to one or to all. Because the protocol still has two states left to complete, it might be advisable to use multi-threading if it is desired to respond to all RPSs.

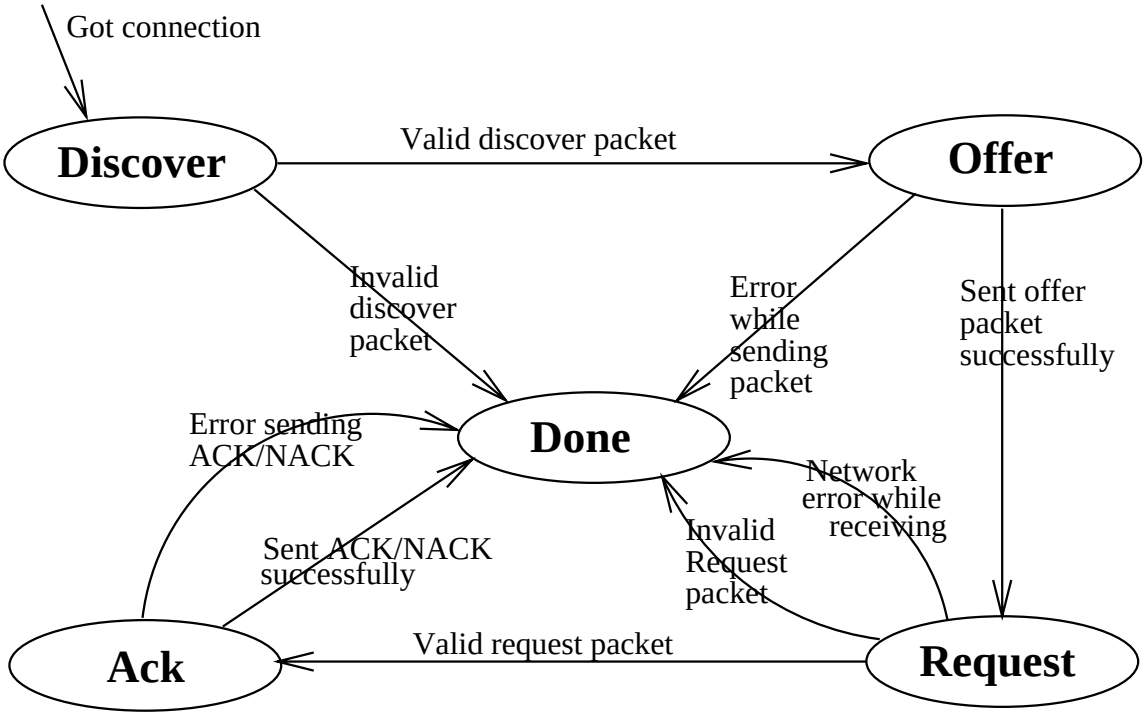

Figure 2. Finite state machine description of the protocol on the RPS

REQUEST: In this state, the service evaluates the received offers, and requests one or more RPSs to register it. In the `REQUEST` packet, the service can send a parameter list which specifies values which differ from service to service. The parameter list is encoded as a string and is meant to be passed into the proxy during initialization. If the RPS receives a `REQUEST` packet, it knows that the service wants to be registered using *this* RPS. It then reads in the parameter string, and stores it for use while initializing the proxy. The RPS does not attempt to parse the parameter string, so the encoding of the string needs to be known only by the service and its proxy.

ACK: In this state, the RPS registers the proxy for the service with the Jini lookup service, and sends an acknowledgment to the service. Because only objects can be registered, the RPS has to download the proxy class from the `proxyURL` specified by the service, instantiate it, pass it the service parameters, and then register the proxy object with the lookup service. In order to download and instantiate the proxy class, a `ClassLoader` which allows accessing code from a URL has to be used. Because the security manager of the RPS has to be set to `RMISecurityManager`(see Section 3), the only classloader suitable for use is `java.rmi.server.RMIClassLoader`, which provides static methods to load classes from URLs, using the standard RMI mechanisms.

If the registration is successful, the RPS sends an ACK to the service, while if there is any error during registration, it sends a NACK. This may be extended in future versions to return a value which indicates the nature of the error which cause the registration to fail.

The service simply waits for acknowledgment from the RPS that the registration has been completed. On receiving an ACK, the protocol module now starts up the

actual service being provided.

3: Code download

The mechanism used by the client to download the proxy in Jini is based on the RMI code download mechanism and is explained in detail in the RMI documentation [12]. A summary of its usage in Jini is shown below.

1. Downloading the proxy can be broken into two operations – obtaining the serialized proxy object and obtaining the class file for the proxy.
2. The proxy object is obtained from the lookup service where it had been placed by the RPS.
3. RMI automatically downloads the class file from the codebase of the JVM that created the remote object. In this case, the proxy class file is downloaded from one of the URLs specified in the `java.rmi.server.codebase` property of the RPS.

We see that the value of the codebase associated with the service proxy is the same as the codebase property of the RPS. Traditionally, this should be set to the location from which the **RPS classes**, **not** the service classes, can be downloaded.

In this extended version of Jini, each service has a different codebase, which it passes to the RPS when it registers. When the client wishes to download the class file for a proxy, it has to know the correct codebase for *that* proxy. The codebase is automatically associated with the proxy by the RMI runtime when the proxy is serialized. Because the proxy is serialized when it is registered, the easiest way to associate the correct codebase is to temporarily change the value of the codebase property just before the service is registered. This temporary value would be the `proxyURL` supplied by the service as part of the registration process.

However, the solution is more complex than that. In order to interact with the lookup service, we need to download its stub. This implies that the security manager has to be set to `RMISecurityManager`[3]. Unfortunately, `RMISecurityManager` is more restrictive than the standard security manager. More specifically, any attempt by the program to tamper with the codebase property throws a `SecurityException`.

In order to allow modification of the codebase, we need to modify the default restrictions of the Java RMI security sandbox. In Java 1.2[4], this can be done by specifying a security policy file. This file is supplied as a command line argument while starting the RPS and sets up the security environment under which the RPS runs. In order to allow modification of the `java.rmi.server.codebase` property, we added the following line to the policy file.

```
permission java.util.PropertyPermission "java.rmi.server.codebase", "read, write";
```

4: Pinging the service

The preceding sections talk about the procedures to be followed when the service is plugged *into* the network. We now consider the issues involved when the service is removed.

[3]The `RMISecurityManager` performs authentication while using the Java RMI subsystem.

[4]Jini can run only on JDK1.2 or greater

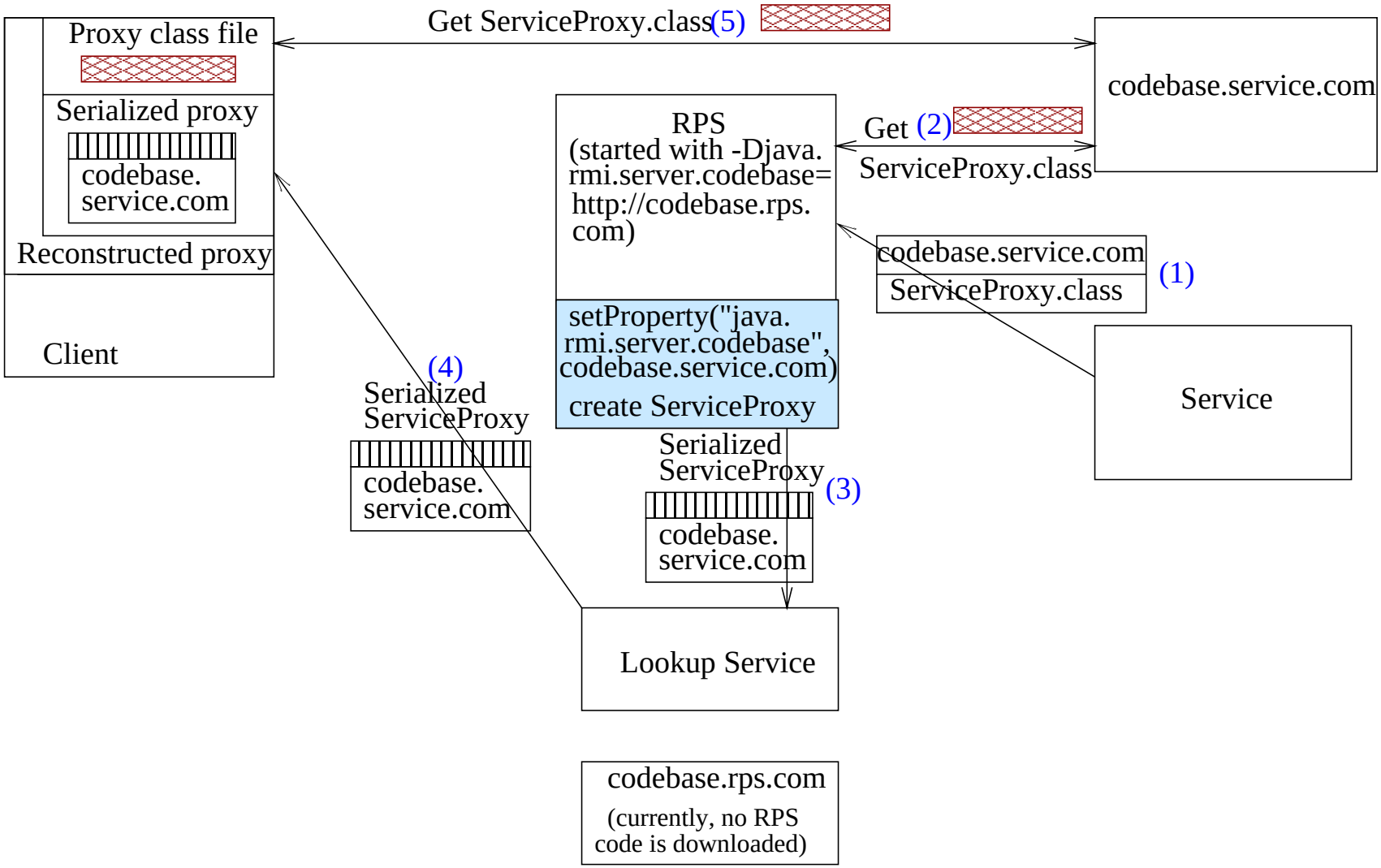

Figure 3. Associating appropriate codebases with proxies

When this happens, the service proxy has to be deregistered from the lookup service so that no clients can attempt to use an unavailable service. We considered three possible approaches to implementing deregistration.

Forced deregistering

The simplest approach to implement requires the service to inform the RPS when it leaves the network by a procedure similar to registration. However, this is impractical, because of the asymmetry between activation and deactivation.

For example, consider a network device being activated by being plugged into a network. The device can detect that it has been connected to a network because it receives packets on its interfaces. It can now communicate with the other machines on the network and inform them of its presence.

Now, consider the reverse case of a network device being deactivated by being unplugged from the network. The device can again detect that it has been removed from the network, but because it is no longer on the network, it cannot communicate this information to the other machines.

A similar argument can be applied to the power on/off process. Therefore, the *other machines* on the network have to detect the absence of the service and act accordingly.

Service list concept

Of the other machines involved in the Jini system, the RPS is the natural choice to detect the absence of the service from the network, since it detects its presence. The RPS can detect the absence of a service by maintaining a list of services which have used it to register, and at selected intervals, poll all of them. However, this method has the drawback that methods for manipulating the list have to be implemented. These methods have to be synchronized to allow access to the list by both the main server thread and the poll thread. In short, we would have to duplicate a lot of the functionality of the lookup service.

Independent `Service` object concept

In order to poll a dynamic set of services which have different refresh lengths, we create a `Service` object for every service at the time of registration. This object is initialized with the proxy and the refresh time for the service. The object then spawns a new thread which uses the proxy to check the service every *refresh length* milliseconds. If the service does not respond, it is assumed to be dead. The dead service is deregistered from the lookup service and the refresh thread is terminated. This is the method we chose for our implementation of the RPS.

4.1: Pinging methods

Since we decided to use a scheme which periodically pings the service to check if it is alive, we need to consider methods in which this pinging can be done. Three possible methods are described here in brief.

Use the service: This is the simplest method, and can be used even with simple services which perform no processing of the input. The RPS simply uses the service, and if the usage is successful, then it assumes that the service is alive, otherwise it assumes that it is dead. This mechanism has the disadvantage that the service might be resource-intensive, potentially leading to performance degradation if the refresh length is low.

Ping messages: This is a slightly more complex method in which the RPS contacts the service with a special *ping packet*. On receiving the ping packet, the service sends back a simple *alive* packet, rather than performing the actual service tasks. In this case, the service has to implement an input parser which can distinguish between *service packets* and *ping packets*.

Ping port: This mechanism is designed to be used on embedded systems whose underlying OS can support multi-threading. The embedded system can then open a *ping port* distinct from the *service port* and listen to both simultaneously. If a connection is received on the *ping port* then a simple *alive packet* is returned, but if the connection is to the *service port*, then the service task is performed and the result returned. Therefore, by using multi-threading, this mechanism can provide a non-resource-intensive ping without any input parsing.

As we can see, each of the mechanisms outlined above have different strengths and are suitable for use on different systems. Also, it is conceivable that other ping mechanisms may become available in the future. Ideally, the device designer should be able to choose the mechanism which is most suitable for the device being designed. If the ping mechanisms are implemented directly by the RPS, then all devices interacting with that RPS would be restricted to using the implemented methods. Also, if better methods are developed in the future, they cannot be easily adopted because the RPS would have to be changed.

We decided to avoid this limitation by integrating the ping mechanism into the proxy. Every proxy which wishes to be registered by the RPS has to implement an `isAlive` method which returns `true` or `false` depending on whether the service is alive or dead. The RPS can then use this method in the `Service` object described in Section 4 to determine the status of the service.

Because both the service and the proxy are written by the service developer, the developer has the freedom to use the mechanism which is most appropriate for the service. Also, because the proxy is dynamically downloaded by the RPS, and used for pinging, newer ping methods can easily be adopted by modifying the proxy to match the service.

4.2: Proxy-service communication

We are trying to connect non-Java systems to a Jini network, and so, the service will generally be implemented in a language other than Java. Therefore, a protocol implementable in native code has to be used for communication between them. In our test service, we used one built on top of TCP/IP because TCP/IP is the most widely implemented network protocol.

It must be remembered, however, that protocols other than TCP/IP can also be used as long as both the proxy and the service implement the protocol. The service *must* implement both TCP and UDP in order to use the registration protocol defined in Section 2 to communicate with the RPS.

Generally, the proxy opens a socket connection to the service, and communicates with it using a proprietary protocol (`SecretProtocol`). This protocol can be designed using standard client/server interaction design techniques. Figure 4 illustrates how the client, the proxy and the service communicate using this model after the proxy has been downloaded to the client.

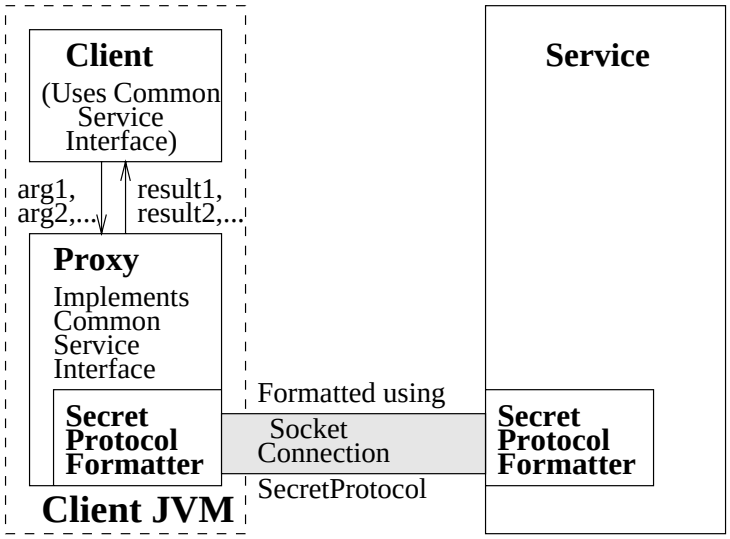

Figure 4. Communication between the different components of a C-based Jini system

5: Conclusion

Jini is a set of protocols which runs on top of Java, and it is designed to make it easy for small devices providing *services* to communicate with programs running on another computer (the *client*). It does this by allowing the clients to download the device *proxy*. The proxy is similar to the driver for a device and hides the complexity of the device from the client.

A central directory of device (or *service*) proxies is maintained, and services can be *looked up* by any client. In order to ensure that the service and the client do not need to be configured with the address of the directory, the *discovery* mechanism allows them to discover it using multicast.

Although Jini is heavily based on Java, it is possible to have non-Java enabled devices participate in Jini federations by delegating the Jini interaction to a Java-based Registration Proxy Server(RPS) running on a more powerful machine on the same network. In this paper, we presented the design of such an RPS and the protocol used by a non-Java device to contact it. We enumerated some of the implementation issues involved in using an RPS and presented easily extensible solutions to them.

We have evaluated this design by implementing a simple service which uses the RPS to interact with Jini clients. The `MessageService` service returns a fixed text string when contacted, and the pinging is done by using the service. The service was implemented on a NS486SXF running the lightweight Java Nanokernel developed at UCSC[13] as the OS.

The protocol was also formally verified using the Communicating Finite State Machine (CFSM) method described in Holzmann's book[14] and Bochmann's paper[15]. It has been verified that the protocol has no deadlocks or livelocks, and is stable. The interesting result is that although TCP has been used for ease of implementation, the formal verification shows that the protocol will also work for UDP if timeouts are used to detect errors.

The formal verification reveals only one problem with the use of UDP, and that is when the protocol terminates. It is possible that the final ACK packet from the RPS to the

service is lost. The service times out eventually and closes the connection without starting the actual service. However, the RPS has no way of knowing this, and will continue to assume that the service is alive.

This leads to a *half-open* connection, where the RPS assumes that the connection is valid, and the service assumes that it is not. However, because of the fact that we periodically ping the service to check whether it is alive, the connection can remain half-open for at most *refresh length* time. Therefore, this protocol can safely be used over both TCP and UDP.

Acknowledgments This work was partially supported by grants from MICRO, National Semiconductor and Bay Networks.

References

[1] Object Management Group, "The OMG's site for CORBA and UML Success Stories." `http://www.corba.org`.

[2] S. Dorward, R. Pike, P. Winterbottom, E. Grosse, J. McKie, D. Presotto, D. Ritchie, K. Thompson, and H. Trickey, "Inferno." `http://inferno.bell-labs.com/inferno`.

[3] Microsoft Corporation, "Distributed Component Object Model (DCOM) - Downloads, Specifications, Samples, Papers and Resources for Microsoft DCOM." `http://www.microsoft.com/com/tech/dcom.asp`.

[4] Sun Microsystems, "Jini(tm) Connection Technology." `http://www.sun.com/jini/`.

[5] Siliware Inc., "Jini Corner at Artima.com: resources for Jini developers." `http://artima.com/jini/`.

[6] J. Waldo, "The Jini Architecture for Network-Centric Computing," *Communications of the ACM*, vol. 42, July 1999.

[7] C. Crichton, J. Davies, and J. Woodcock, "When to trust mobile objects: access control in the Jini^{TM} Software System," in *Proceedings of Technology of Object-Oriented Languages and Systems - TOOLS 30, Santa Barbara, CA, USA, 1-5 Aug. 1999* (D. Firesmith, R. Riehle, G. Pour, and B. Meyer, eds.), pp. 116–125, Aug. 1999.

[8] Sun Microsystems, "Jini Device Architecture Specification." `http://www.sun.com/jini/specs/devicearch101.ps`.

[9] R. Droms, "Dynamic Host Configuration Protocol," *RFC 1541*, Oct. 1993. `http://cis.ohio-state.edu/htbin/rfc/rfc1541.html`.

[10] R. Braden, "Requirements of Internet Hosts – Communication Layers," *STD 3, RFC 1122*, Oct. 1989. `http://cis.ohio-state.edu/htbin/rfc/rfc1122.html`.

[11] K. Shankari, "Connecting non-Java devices to a Jini network," Tech. Rep. UCSC-CRL-00-05, University of California, Santa Cruz, Apr. 2000.

[12] Sun Microsystems, "Dynamic Code Downloading using RMI." `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/codebase.html`.

[13] B. R. Montague, "JN: OS for an Embedded Java Network Computer," *IEEE Micro*, vol. 17, May 1997.

[14] G. J. Holzmann, *Design and Validation of Computer Protocols*. Prentice Hall Software Series, 1991.

[15] G. V. Bochmann, "Finite State Description of Communication Protocols," *Computer Networks*, vol. 2, pp. 361–372, 1978.