UNIVERSITY of CALIFORNIA
SANTA CRUZ

HOW TO CONNECT NON-JAVA DEVICES TO A JINI NETWORK?

A dissertation submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

K. Shankari

June 2000

The dissertation of K. Shankari is approved:

Professor Charles E. McDowell, Chair

Professor Ira Pohl

Professor J. J. Garcia-Luna Aceves

Dean of Graduate Studies

Copyright © by

K. Shankari

2000

Abstract

How to connect non-Java devices to a Jini network?

by

K. Shankari

Recently, several schemes have been proposed to interconnect extremely small devices with the minimum of effort. Ideally, the user needs to exert *no* further effort beyond making the physical connections. 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, clients anywhere on the network can use it simply by 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(JVM) to run on. However, extremely small devices generally do not have either persistent storage or large amounts of RAM, and thus lack the resources to support an RMI-enabled JVM.

Thus, it is anticipated that, at least for very small embedded systems, Java will not be the programming language of choice unless a method to run Java natively can be developed. This is practical only with the development of Java chips, which can directly execute bytecode, or through better Java to native compilers.

In this thesis, we explore the issues involved in allowing devices without JVMs (non-Java devices) access to Jini federations.

Contents

List of Figures **vi**

Dedication **viii**

Acknowledgements **ix**

1 Introduction **1**

2 Introduction to Jini **7**
2.1 Jini components . . . 7
2.2 Interactions and their protocols . . . 9
2.2.1 Discovery and Join . . . 9
2.2.2 Code download . . . 11
2.2.3 Overview of the RMI protocol . . . 14
2.2.4 Leases . . . 18
2.2.5 Remote Event Handling . . . 19
2.2.6 Javaspaces . . . 19

3 Design of Java-based Jini applications **21**
3.1 The need for `CommonServiceInterface` . . . 21
3.2 Remote or Local Interface . . . 22
3.2.1 Remote . . . 24
3.2.2 Local . . . 26
3.2.3 Design decision . . . 26
3.3 Communication with a Java service . . . 27
3.4 Complete overview . . . 28

4 Extension to non-Java devices **32**
4.1 Requirements . . . 34
4.1.1 Service Requirements . . . 34
4.1.2 Protocol Requirements . . . 35
4.1.3 RPS Requirements . . . 35
4.1.4 Client Requirements . . . 36

4.2 Protocol Design . . . 37
4.2.1 Service-RPS protocol . . . 37
4.3 Code Download . . . 42
4.4 Pinging the service . . . 45
4.4.1 Pinging methods . . . 47
4.4.2 Proxy-service communication . . . 49
4.5 Complete overview . . . 51

5 A Sample Application 56
5.1 The `ServiceProxy` common interface . . . 56
5.2 The `MessageSource` Interface . . . 57
5.3 The `MessageCProxy` proxy class . . . 58
5.4 The `MessageCSource` service . . . 61
5.5 Invoking the service . . . 61

6 Analysis and Verification 64
6.1 Formal verification . . . 64
6.1.1 Construction of the graph . . . 65
6.1.2 Verification of properties . . . 68

7 Conclusion 73

Bibliography 77

List of Figures

2.1 A graphical description of mobile code 12
2.2 Stubs and Skeletons - the middlemen in RMI 16
2.3 Stub download in RMI 17
2.4 The various layers of the Jini protocol 20

3.1 Example class hierarchy for printer devices 23
3.2 Communication between different parts in a Jini system 24
3.3 Jini operation with a remote interface 25
3.4 Communication between the service and the proxy using RMI 28
3.5 Overview of the interaction between components in a Jini system 31

4.1 Finite state machine description of the protocol on the service side 39
4.2 Finite state machine description of the protocol on the RPS 40
4.3 Associating appropriate codebases with proxies 44
4.4 Using independent `Service` objects to handle different devices and abstracting different ping methods using proxies 48
4.5 Communication between the different components of a C-based Jini system 50
4.6 Overview of the interaction between components in a JiniLite system 52

5.1 The `ServiceProxy` interface 57
5.2 The `MessageSource` interface 58
5.3 The `MessageCProxy` service proxy 59
5.4 The `MessageCProxy` service proxy 60
5.5 The `MessageCSource` service 62

6.1 Formal verification of the protocol 67
6.2 Transitions due to message transfer 70

List of Tables

3.1 Services currently registered with the lookup service 22
3.2 Description of the various components of a Java-based Jini system 29

4.1 Description of the various components of a C-based Jini system 51

This thesis is dedicated to my parents.

Acknowledgements

The list of people I would like to thank obviously starts with Prof. McDowell, who brainstormed and proof-read and was in general, responsible for this thesis getting written. I would also like to thank Prof. Pohl and Prof. Garcia-Luna for their comments and advice. They helped plug some of the holes in the earlier versions of this work.

How can I not include the denizens of AS350[1]? The chocolate, the chips and the endless chats made it feel like home. A special vote of thanks to Ed Parrish, Osama Salem, Jim Freeby and Thomas Raffill, who listened to all my diatribes about Java quirks and faithfully "came and saw" my demos, even the dinky little "Hello World" ones.

And finally, I'd like to thank my parents, my little brother[2], and the rest of my family back in India. They not only taught me the basic life skills to get here (work hard, be determined...), but also nagged me until I finally got down to writing this.

[1]Now renamed to AS340, thus leading to immense confusion among us old-timers

[2]Provided I am allowed to call a 20-year old who is 5 inches taller than me little ;-)

Chapter 1

Introduction

We live in an electronically controlled world. Digitally operated versions of answering machines, microwave ovens, stereo systems and washing machines have slowly begun easing out their analog counterparts. These devices are called *digital* because they are controlled not by mechanical moving parts or by amplifiers, but by purely digital systems, with a dedicated microprocessor at the heart.

Since these are microprocessor controlled, we might now try to extend the benefits of networking to this area by finding a way to have all these devices interact with each other. For example, to use a futuristic scenario, we might want to have a home control computer which would allow you to check on the laundry, listen to your messages, and adjust the volume on the stereo from the comfort of your recliner.

The main drawback to this utopian ideal of an intelligent home is the fact that the architectures of the control microprocessors are optimized for their particular function, and differ from each other considerably.

This might not seem like a particularly compelling objection initially - after

all, peripheral devices with embedded microprocessors have been interfaced to computer systems since the dawn of computing. This interfacing is done by pieces of system software called *drivers*. There is a different driver for each brand of peripheral, and although they are different internally, each of them presents the same or similar interface to the computer system. Thus, the drivers abstract the complexity of dealing with different peripheral architectures.

However, there are problems with extending the notion of drivers to small and embedded systems. Although drivers hide most of the complexity of the device, they still require manual intervention to perform several tasks associated with them. A brief list of these tasks follows.

1. Drivers have to be *installed* on the computer system which interacts with the embedded system.

2. After installation, the driver has to be configured with the parameters of the device. This is even more important if the device is to be accessed over the network. At the very least, the network address and the port of the device have to be made available to the driver. Every time the device is moved, these parameters have to be changed.

3. If the manufacturer updates the drivers (possibly to fix a bug), then the new drivers have to be reinstalled manually.

4. Different drivers have to be written for different operating systems. Manufacturers, who write most device drivers, are unable to support all the different systems available today, and concentrate on a few popular ones.

For large computer systems, these tasks are performed by specialists - the system administrators, while on smaller systems, they are performed by the system owners. However, small and embedded systems, specially when used in consumer devices, will be expected to work transparently with each other, and with the rest of the system. We should be able to plug devices into the network the way we plug them into electrical outlets and have them work without any further effort.

Solutions to different parts of this problem already exist. These include CORBA[Obj], DCOM[Mica] and Inferno[DPW+].

Jini technology from Sun Microsystems is the latest entrant into this field. It uses Java, and handles all the issues mentioned above reasonably well by using the inherent platform independence, security and code mobility of Java. A brief description of the manner in which it handles these issues follows.

Installation: Drivers are downloaded from a code server when required

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.

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

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 current version of the Java Virtual Machine(JVM) is around **2.2 Mb**. This is too big for most embedded devices, specially 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 surrogate which can run on a machine with more resources.

There are 4 main ways in which the devices and the surrogate can be connected. Three of these are covered in the Jini Documentation [Sund], and the fourth is the method which we have implemented. All of these methods are described below.

Device Bay: The non-Java devices are physically plugged into a Java-capable device. All communication with the external world is done using the Java-capable device. The non-Java devices need no specialized programming, not even a network stack.

Network Surrogate: The non-Java device contains a network stack written in a compact language. When it is plugged in, it discovers the surrogate[1] on the network and uses it for all further communication with the rest of the network.

CORBA: The device uses CORBA in place of RMI[Micb] for distributed computing. The device contains a CORBA ORB implemented in native code and the lookup service provides an implementation over both IIOP and RMI. The device would then interact directly with the CORBA version of the lookup service.

Registration Proxy Server(RPS): The non-Java device finds the network surrogate

[1] 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.

as described above. However, it utilizes the surrogate only to perform the Jini interactions on its behalf. Once the driver has been downloaded into the client program, all further communication takes place directly between the client and the device.

We use the **Registration Proxy Server(RPS)** method for the following reasons.

1. Unlike in the device bay approach, there is no physical limitation on the number of non-Java devices which can be connected to a single Java enabled device.

2. Because all communication between the outside world and the non-Java device is through their respective proxies and not through a common device bay, the communication protocols can be more flexible.

   For example, network protocols other than TCP/IP (like Token Ring) could be used for communication between the client and the device.

3. 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 following advantages:

   (a) lower load on the RPS, thus reducing network congestion, and

   (b) a more robust system because the crash of the RPS would not affect existing client - device connections.

4. Unlike in the CORBA approach, this does not require complex code to implement an ORB, and makes no assumptions about the lookup service implementation.

In this thesis, we explore the issues involved in creating an RPS, define a protocol for communication between the RPS and the device, and provide a sample implementation of a device which sends out a message whenever contacted.

As you have seen, Chapter 1 is a general introduction to the problem under consideration. Chapter 2 provides a brief introduction to Jini in general and to the concepts used in this report in particular. Chapter 3 explores the issues involved in creating a Jini service. Chapter 4 discusses our extension to Jini in detail. Chapter 5 provides a simple example to illustrate the implementation of these concepts. Chapter 6 presents a formal verification of the protocol using the Communicating Finite State Machine (CFSM) approach. Chapter 7 summarizes our conclusion and suggests directions for future work.

Chapter 2

Introduction to Jini

As we have already seen, this thesis proposes an extension of the Jini protocol to include non-Java devices. In order to improve on an idea, it is essential to thoroughly understand it first. Therefore, let us briefly examine the Jini protocol - its components and their interactions.

2.1 Jini components

The main pieces which interact to form a Jini *federation* are the following.

Service: A service is a software or hardware component which can be used by other components.

Lookup Service: A lookup service is 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.

Proxy: A proxy is 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.

Client: A client is a program which utilizes the service.

An example of the way in which the various components interact is shown below.

1. The service is plugged into the network.
2. The service uses the Jini lookup and discovery protocol to contact the lookup service.
3. The service creates a proxy to represent itself. Since the lookup service is also a service, it has its own proxy. The service downloads this proxy (the `ServiceRegistrar`) and uses it to register its proxy (and by extension, itself) with the lookup service.
4. The client starts up.
5. The client uses the Jini lookup and discovery mechanism to contact the lookup service.
6. The client downloads the proxy for the lookup service, the `ServiceRegistrar`.
7. The client looks up the service it wants to use using the `ServiceRegistrar`. For example, the client may try to find all washing machines registered with that service, or it might look for the specific washing machine in the basement.
8. The client downloads the proxy for the service it wishes to use from the lookup service.
9. The client uses the proxy for the service to talk to the service.

2.2 Interactions and their protocols

Jini can be viewed as a complex set of interacting protocols which provide all the functionality required to have plug-and-play devices. The complexity of the protocols makes Jini very powerful, but makes it difficult to obtain a coherent view of the interaction between the various parts of the system. Here, we attempt to describe the protocols associated with the interactions in brief, and to highlight the elements central to our extension of Jini.

2.2.1 Discovery and Join

The Jini discovery and join mechanisms are used by the client and the service to locate the lookup service on the local area network. A detailed description of the parameters involved can be found in the Jini Discovery and Join Specification [Sune], but a brief summary is provided here.

Traditional schemes which rely on a centralized directory of services require that the programs which use them be configured with the address of the directory as a parameter. However, because the Jini protocol aims to work without configuration, it is necessary to have a *discovery* process which allows the client programs to discover the location of the lookup service.

In Jini, this is essentially done by multicasting to a specific broadcast address. As described in the Jini Discovery and Join Specification [Sune], two basic mechanisms are used.

Multicast Request Protocol: When an entity comes up, it first actively searches for

lookup services by multicasting to the well-known network address of `224.0.1.85`. If any such services are found, it interacts with them immediately.

Multicast Announcement Protocol: If no lookup services are found, it switches to a different mode and begins listening for announcements on the network.

Every lookup service periodically broadcasts its presence on the network by using the multicast announcement protocol. All waiting entities listening for such annoucements can now interact with the service.

Fortunately, Sun Microsystems also provides a `LookupDiscovery` [Sunc] class which automates this process and notifies the program when new lookup services are found. The notification contains an array of lookup service proxies (`ServiceRegistrars`) corresponding to the lookup services found.

Once a lookup service is found, a client can use lookup directly, but the service has to follow the **join** protocol to register itself with the lookup service. This protocol specifies the essential parameters which every service must provide, and deals with maintaining state across crashes and restarts. We will not cover these in detail here, but will only note that the essential parameters for registering a service are:

1. a unique service ID for the service (this can assigned by the lookup),
2. the proxy for the service, and
3. a set of service *attributes* which provide human readable information about the service. Some examples of standard attributes are name and location. Other service-specific attributes can be added, and since they can be arbitrary Java objects, can

include a GUI to administer the service.

The Join protocol is simple, and Sun Microsystems has provided a utility class called `JoinManager` [Sunc] which automates much of the process. A single `JoinManager` class is created for each service, and provided with the parameters described above. The `JoinManager` then performs discovery and joins the lookup service it finds, with the specified parameters. If the parameters of a service are to be changed, the modification should be done through the `JoinManager`. This will ensure modification of entries in *all* lookup services with which the service is registered.

If more control of the join process is desired, the `ServiceRegistrar` objects returned by `LookupDiscovery` can also be used to directly register services with the individual lookup services they represent.

2.2.2 Code download

We have seen that mobile code is at the core of the Jini protocol. The proxy has to move from the service to the lookup service and from there to the client, where it is used to communicate with the service.

The ability to create *mobile* code is an important aspect, not just of Jini, but of the JVM in general. The most conspicuous use of this feature is in the working of *applets*, where the applet code is downloaded from a web site into a browser and run locally. However, mobile code is also used in several other Java features, including RMI and Jini.

In order to clearly understand Java mobile code fundamentals, we need to realize that mobile code in Java typically has two components.

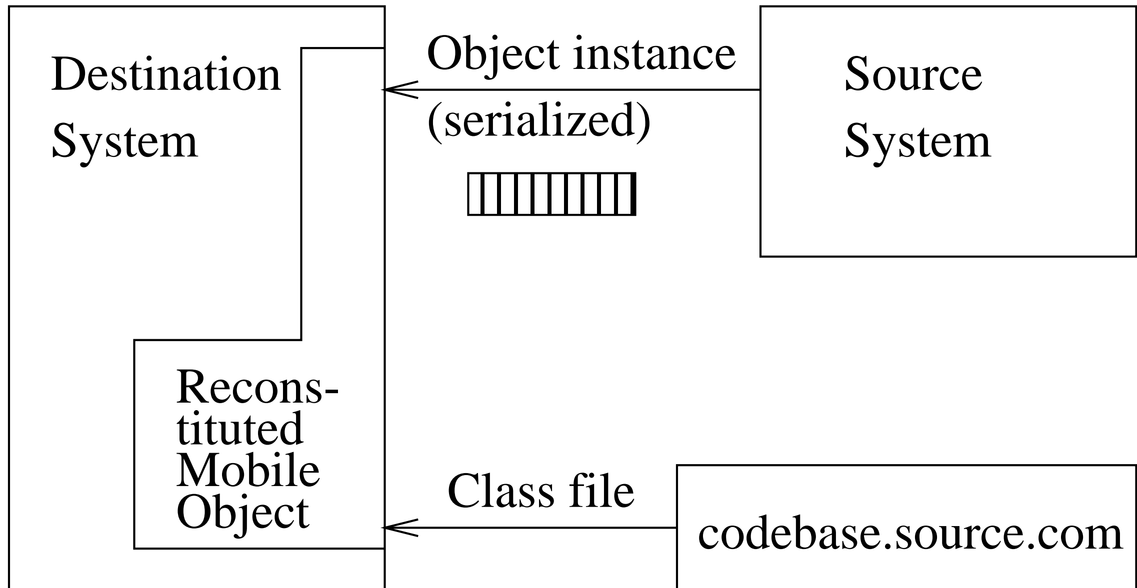


Figure 2.1: A graphical description of mobile code

Class File: This corresponds to the *definition* of a class of objects. It includes the fields associated with the class, the interfaces it implements and the methods it provides.

Object Instance: This corresponds to a single *instance* of a class. It can be represented by the values of the fields associated with the class. There can be *several* instances of a class, all of them sharing the same class structure, but with each one having different values in the fields. Obviously, since the values in the fields dictate the behaviour of the class, different object instances would react differently to external events.

Therefore, as we can see from Figure 2.1, in order to reconstruct a mobile object, the JVM needs to have both the object instance, and the class definition of the object. We now consider how the JVM might obtain each of these components.

Object Instance

Object instances are passed between JVMs by using a technique called *serialization.* Serialization is a mechanism to represent any arbitrary object instance by specifying the values of its fields [Suna]. This makes it easy to read and write object instances using standard bit streams, and allows object instances to be transferred across the network by using standard socket streams.

In Jini, the object instance for the proxy is obtained from the lookup service.

Class File

We now come to the more difficult problem of obtaining the class file. Class files for mobile code are generally downloaded into the JVM from some other location, thus allowing the class definitions to be updated easily.

The main issue for code download is the method of specifying the location of the classes to the JVM. This varies with the mechanism used for code download. In browser JVMs, for example, the `<APPLET>` tag contains a codebase parameter which performs this function.

In Jini, the code download mechanism is built on top of the RMI code download mechanism explained in detail in Section 2.2.3. Briefly, the `java.rmi.server.codebase` parameter of the service is set to the code download location. This information travels with the object instance of the proxy and reaches the client. The client JVM searches this location (or list of locations) for the proxy class file. For a more accurate and detailed description of the process, the reader is encouraged to refer to Section 2.2.3 and Figure 2.3 on page 17.

2.2.3 Overview of the RMI protocol

RMI stands for **R**emote **M**ethod **I**nvocation, and is primarily used as a method to implement Remote Procedure Calls (RPC) in Java. Just like other RPC implementations, it allows programs on one machine (the client) to call methods in code which is physically located on a different machine (the server).

A brief glossary of the mechanisms Java uses to support remote method calls is provided below.

Remote and Local methods

All classes which are to be invoked remotely have to implement a remote interface. The remote interface has to be a sub-interface of `java.rmi.Remote`, and only methods which are specified in the remote interface can be remotely invoked. Other methods in the remote object can only be invoked locally.

Stubs and skeletons

A **stub** is an automatically generated Java class which handles the RMI protocol on the client side. It is loaded into the client JVM when the remote object is first referenced. When the client calls a remote method, the method call is automatically redirected to the stub, which contains an implementation for every method in the remote interface.

The stub now implements the RMI protocol by performing the following operations:

1. marshalls the argument according to the RMI specifications,

2. transmits the marshalled argument to the remote JVM,

3. waits for the return value from the remote JVM,

4. unmarshalls the returned value and returns it to the client program.

The **skeleton** is the automatically generated Java class which handles the RMI protocol at the server side. It is loaded into the server JVM and receives method calls from the stub.

It then implements the RMI protocol by performing the following operations:

1. unmarshalls the argument according to the RMI specifications,

2. invokes the method on the appropriate object and waits for the result,

3. marshalls the returned value and returns it to the invoking JVM.

The stub and skeleton operation is described in Figure 2.2 and is easily seen to be transparent to both the client and the server.

Dynamic code download

Section 2.2.2 describes the issues involved in mobile code using Java. Here, we provide a description of how the RMI mechanisms depend on dynamic code download, and how the download is accomplished with JVM support.

When a client wants to make remote calls on an object, it attempts to get a reference to it. Internally, this reference is just the stub for the object shown in Figures 2.2 and 2.3.

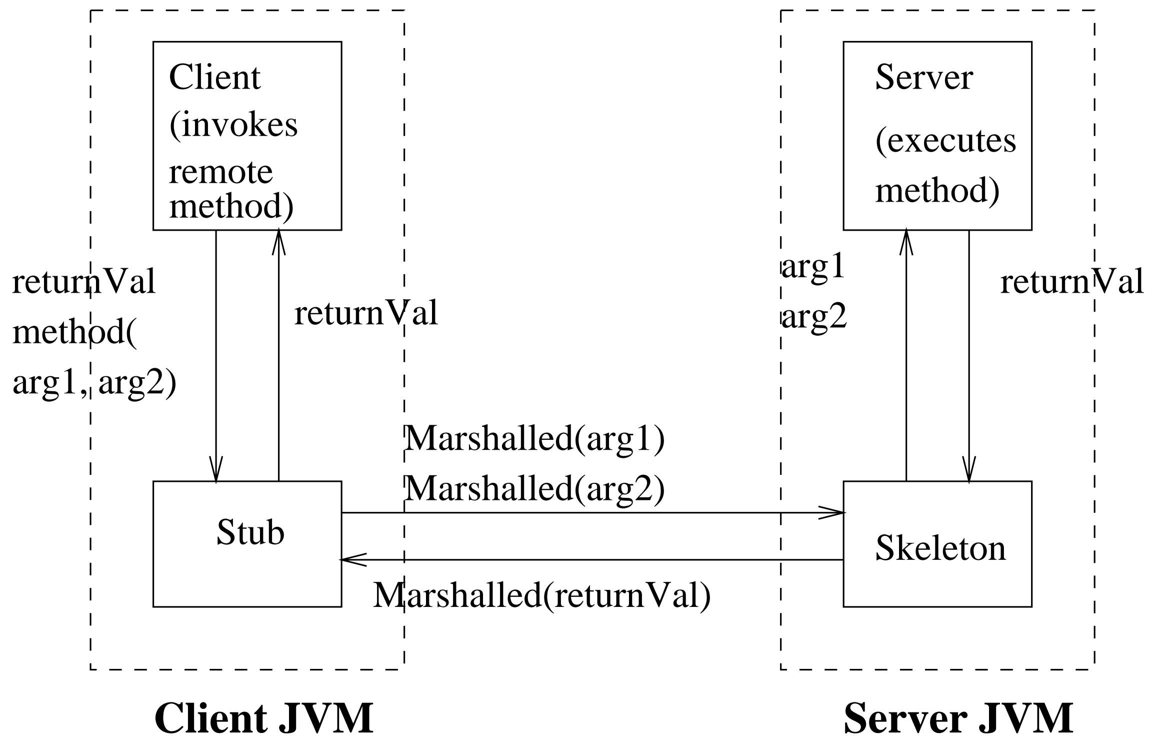


Figure 2.2: Stubs and Skeletons - the middlemen in RMI

The object instance for the stub is received from the directory JVM in serialized form. As in the case of all mobile code, in order to manipulate the stub, the client JVM needs to download its class file and create a local instance initialized with the values from the serialized stub.

This can be done by reading the codebase associated with the serialized stub, and downloading the class file from that location. This is possible because the RMI runtime stamps each object it serializes with its codebase. Since the stub was initially serialized by the server JVM, this codebase is the value of the server's `java.rmi.server.codebase` parameter.

The codebase parameter is generally a **http** URL to facilitate network download, but in those rare cases where the local file system has to be used instead, a **file** URL can

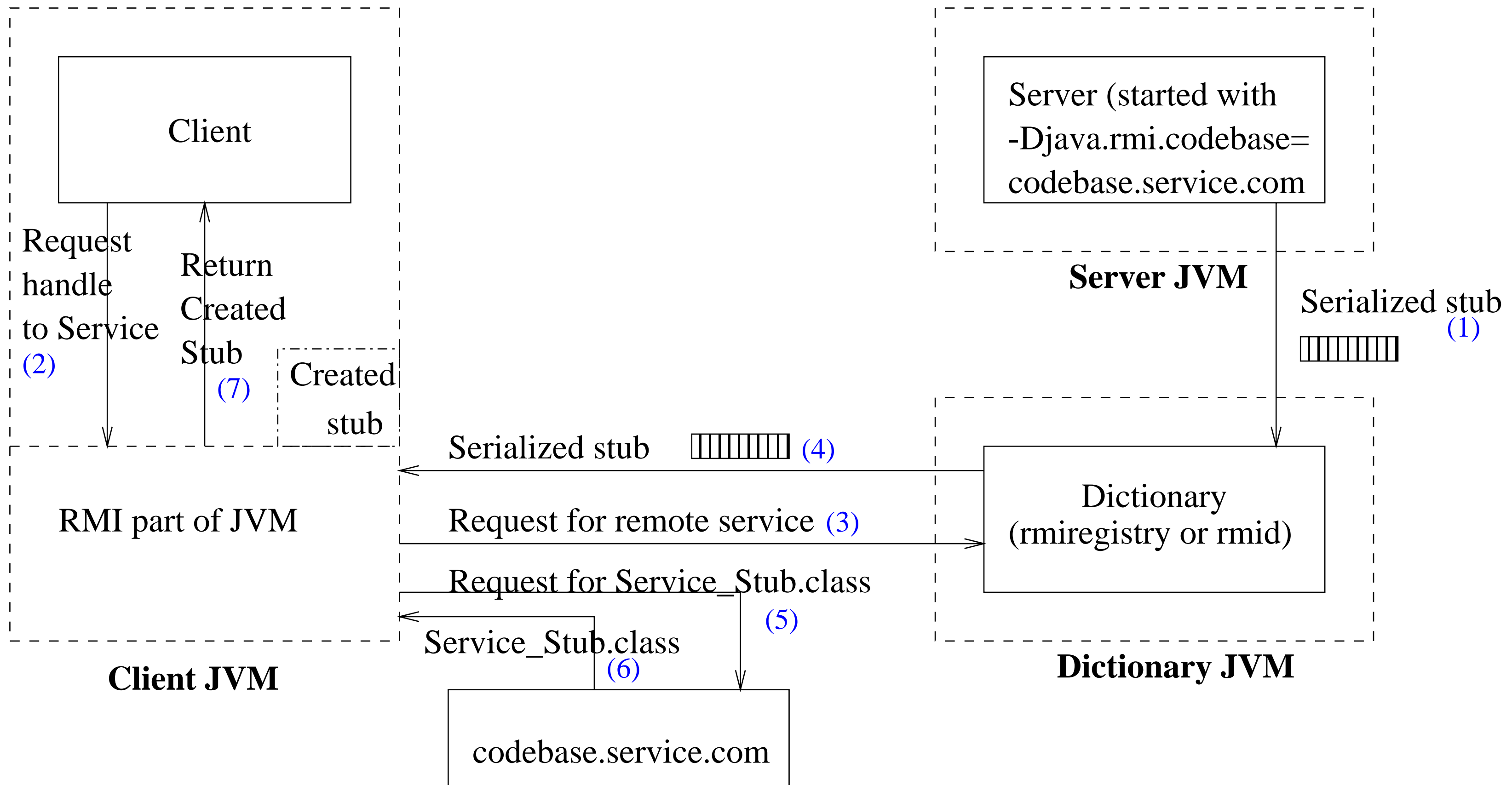


Figure 2.3: Stub download in RMI

be specified.

2.2.4 Leases

Since reliability, both of the network and of devices, is an issue in a Jini federation, allowing services to remain registered until they explicitly deregister themselves can quickly lead to obsolete services cluttering up the directory. Several mechanisms can be used to eliminate dead services, the most common ones being keepalives and ping packets, but they assume infinite registration as the default and require explicit programmer intervention to handle abrupt termination.

Jini includes an elaborate leasing scheme which makes elimination of dead services the default rather than the exception. Every time a service registers itself, it is granted registration for a specific time period. At the end of that time period, the registration expires by default. If the service wishes to remain registered for a longer time, it has to explicitly ask for renewal of the lease. If the service dies or is disconnected from the network, the renewal request never reaches the lookup service, and the registration expires automatically.

Again, Sun Microsystems has provided a utility class `LeaseRenewalManager` [Sunc] which implements this protocol and periodically renews the lease at the intervals specified[1]

---

[1] *Caveat:* At the time this report was written, the only interval supported by the default implementation is 5 minutes. Sun Microsystems is considering adding direct support for other lengths, and provides some quick fixes which are beyond the scope of this report.

2.2.5 Remote Event Handling

Local events are frequently used in Java programs to communicate state changes to other objects. The standard mechanisms for indicating interest in events and dispatching them make it easy to create loosely coupled objects which can still communicate with each other.

Since distributed computing has become popular, Java has incorporated distributed constructs like RMI to create the illusion that objects which actually exist on remote JVMs are local. However, there is no method in pure Java which would allow event passing to and from such remote objects. A simple modification of local event handling cannot be used because of the reliability and timing issues involved. Jini introduces a remote event handling mechanism to allow distributed objects to enjoy the benefits of event-based message passing.

The basic classes used are `RemoteEvent` and `RemoteEventListener`. These classes, along with several utility classes, are defined in the
`net.jini.core.event` package.

2.2.6 Javaspaces

Javaspaces is a protocol built over Jini which supports distributed computing. It is modelled on the **Linda** programming language extensions and uses the concept of atomic transactions to achieve synchronization.

However, the JavaSpaces protocol is not pertinent to our understanding of the Jini system. If we follow the traditional layered approach for designing a protocol, we can consider Jini to consist of several levels as shown in Figure 2.4 on page 20.

Javaspaces (Transactions, Entries)
Jini (Discovery, Join, Lookup)
Java extensions (Remote events, Leases)
JVM (Networking, RMI)

Figure 2.4: The various layers of the Jini protocol

We work at the Jini level to build our extension, and Javaspaces, which is present at a higher level, merely uses the Jini protocol. If our protocol does not change the interface offered by the Jini layer, it will be transparent to the Javaspaces protocols, and to all other high-level protocols which use the jini mechanisms.

Therefore, we shall not describe Javaspaces further. For further details, the interested reader is referred to the Javaspaces documentation [Sunb].

Chapter 3

Design of Java-based Jini applications

This chapter explores the various design issues which have to be resolved while creating a normal, Java-based Jini service. In chapter 4, we consider the implementation of a similar service written in C and running on an embedded system.

3.1 The need for `CommonServiceInterface`

In Jini, to make lookup easier, services are grouped by functionality, and each group has a standard interface defining it. Therefore, when the client performs a lookup, it asks the lookup service for all services conforming to a specific interface.

These interfaces are supposed to be defined by consortiums of device vendors. For example, a group of printer vendors would get together and define a `Printer` interface. Each individual vendor could then define their own printer interface, which would be a

Printer class	Number
EP3054XPrinter	3
EP7051Printer	4
HP370DJPrinter	2
HP4000LXPrinter	5

Table 3.1: Services currently registered with the lookup service

subclass of the generic printer interface. Finally, each vendor could also create a specific interface for every type of printer they produce. Each of these would allow the client to exercise varying levels of control. An example class hierarchy for the printer interface is shown in Figure 3.1 on page 23.

The client can now ask the lookup service for all printers, all Epson printers, or all EP3054X printers. Assuming that the lookup service had the registrations in Table 3.1, the Java inheritance mechanism will ensure that the number of objects returned for those queries is 14, 7 and 3 respectively.

Here, we consider the issues involved in defining such an interface, the `CommonServiceInterface`.

One of the questions to be considered while designing the interface is whether it should be remote or local.

3.2 Remote or Local Interface

Because the proxy has to implement `CommonServiceInterface`, whether it is defined to be local or remote, let us quickly recap the role of the proxy in a Jini system. The proxy is an object which implements the well-known interface, `CommonServiceInterface`, and is present on the client JVM. Since the proxy is present locally, and implements a

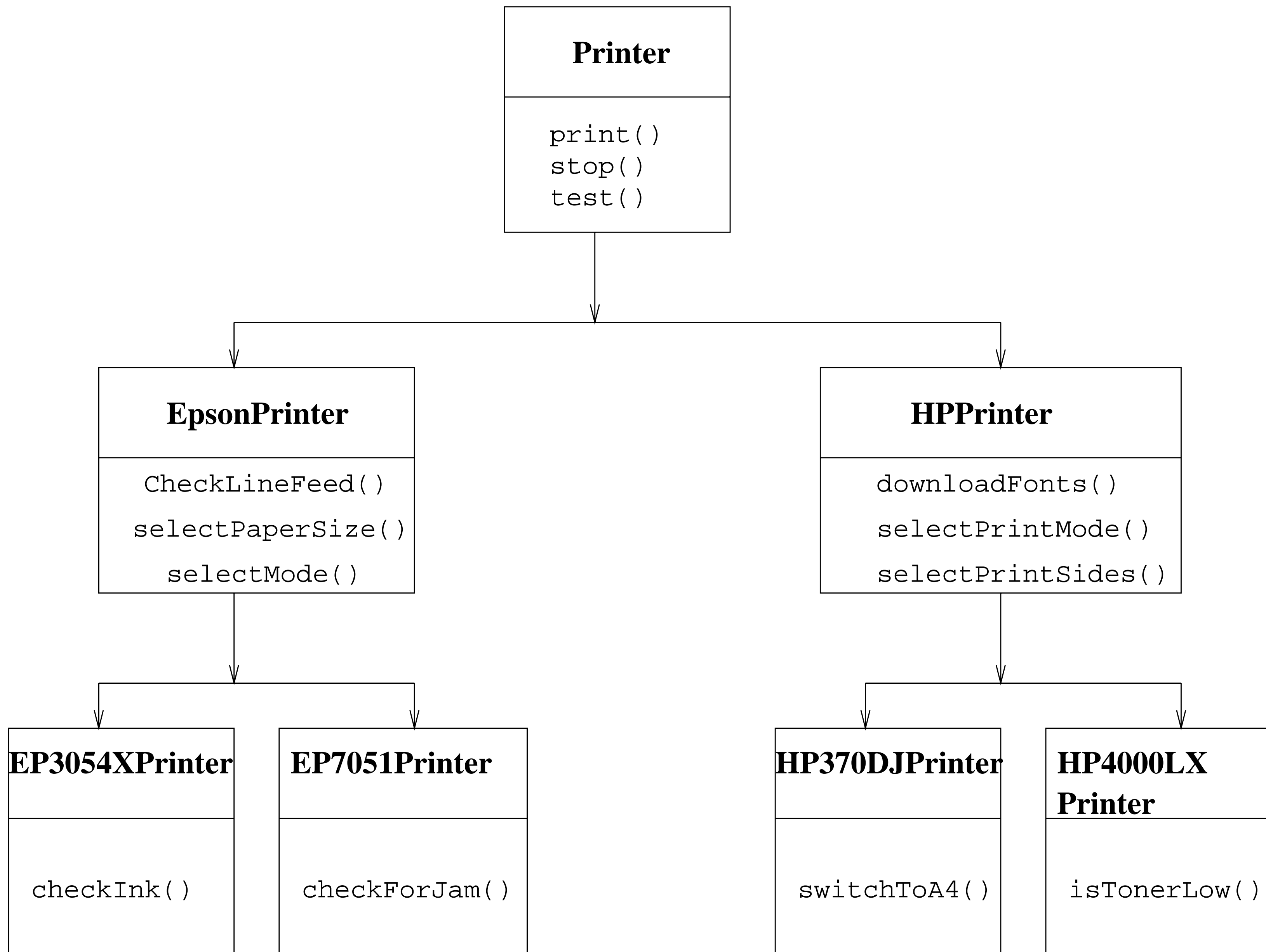


Figure 3.1: Example class hierarchy for printer devices

known interface, the client can communicate with the proxy simply by invoking methods defined in the interface.

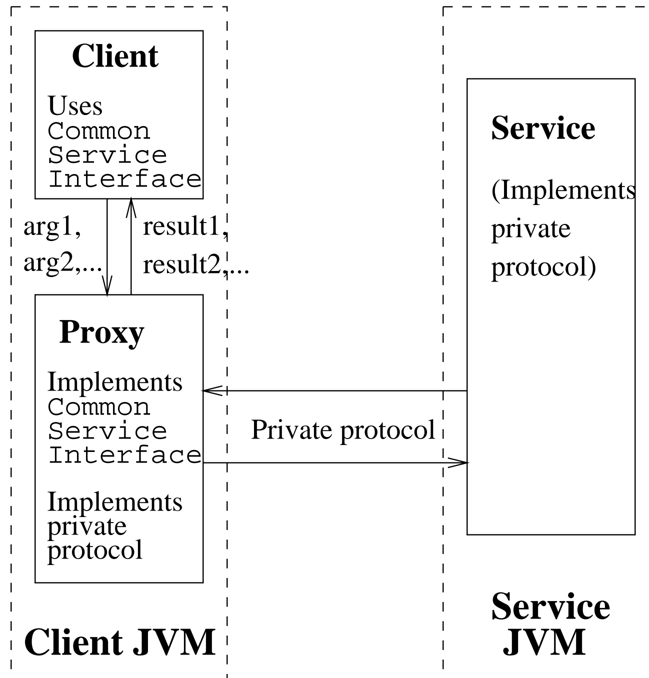


Figure 3.2: Communication between different parts in a Jini system

However, the proxy and the service can use a private protocol to talk to each other. This protocol has to be known only to the proxy and the service and is totally transparent to the client. From the client's perspective, it invokes methods on the proxy just as if it were the service itself. Figure 3.2 illustrates the interaction between the client, the proxy and the service.

3.2.1 Remote

A remote interface is one which extends `java.rmi.Remote`. Any object which implements this interface can be *exported* and accessed remotely from other machines by using the standard RMI protocol. The RMI protocol, including details about stubs and

skeletons is described in Section 2.2.3.

If `CommonServiceInterface` is defined as a remote interface, then the service can be exported to other machines and manipulated by them using the standard RMI protocol.

If we attempt to interpret this in terms of the Jini framework, we see that the stub for the service is downloaded into the client, and used for further communication with the service. This communication is transparent to the client, which merely invokes methods on the stub. Therefore, as we can see in Figure 3.3, the stub acts as the *proxy* for the service.

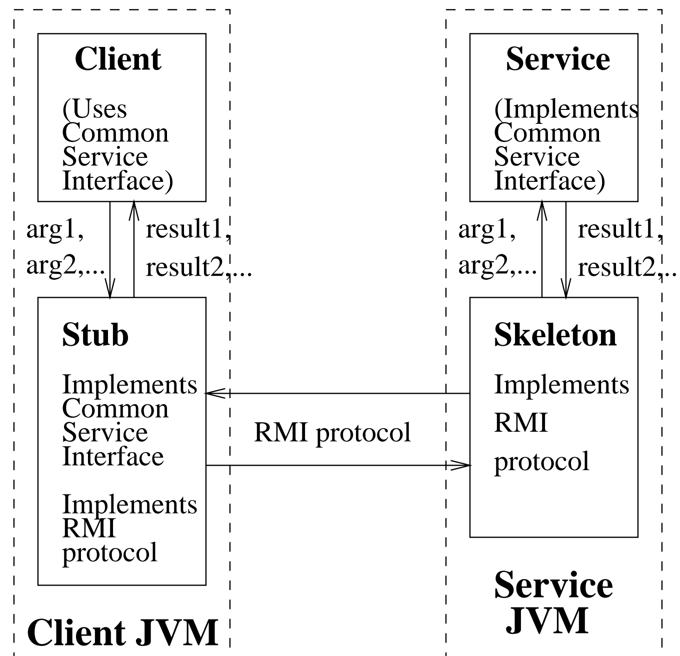


Figure 3.3: Jini operation with a remote interface

Next, the stub and the service communicate using the standard, well-defined RMI protocol. This is the private protocol described in Figure 3.2. Because the stub is automatically generated, the implementation of the private protocol occurs without the

need for explicit programming by the developer.

3.2.2 Local

In Java terminology, a local interface is simply a set of methods with no method bodies. Any object can implement such an interface. Now, instead of the proxy being an automatically generated stub, and the private protocol being the well-defined RMI protocol, the developer has to design the private proxy-service protocol, hand-craft a proxy class, and implement the protocol in both the proxy and the service.

3.2.3 Design decision

We choose the local interface definition for the following reasons specified below.

1. Using a remote interface implies that all services implementing that interface *have* to be remote. In order to achieve client-level transparency, the client should ideally interact in the same way with both local and remote services. As we have seen, using a remote interface would generate a distinction between local and remote services.

2. Since the private protocol is completely decided by the designer, it allows maximum flexibility. The designer may even choose to implement `PrivateProtocol` over some layer other than TCP/IP.

3. Finally, using RMI to generate the proxy and implement the protocol, implies that the remote service has to be implemented in Java. This might work for larger systems, but as we have seen, for small embedded systems, this is not always a viable alternative.

3.3 Communication with a Java service

Once we have decided that `CommonServiceInterface` is to be a local interface, we have to play the role of the developer and design the private protocol. If the service *is* implemented in Java, rather than design and implement a protocol from scratch, we can attempt to piggyback on the RMI protocol. This allows us to obtain the ease of implementation associated with a remote interface while retaining the flexibility of the local interface.

This approach can be implemented by having the proxy contain an encapsulated stub for the service. The Java serialization mechanisms can be used to encode the stub into a set of bytes when the proxy is created and to decode it when it is downloaded at the client. Now, when the client invokes methods on the proxy, the proxy simply invokes the same methods on the encapsulated stub. The standard RMI mechanism now allows the service and its stub to communicate.

We can see that because we use RMI, there can be a separate remote interface for communication between the proxy and the service. This interface is known only to the service and its stub and so it can proprietary. This is called `SecretInterface` in Figure 3.4.

Again, the client needs no knowledge about this proprietary interface, which is purely between the service and the proxy. Thus, the client can interact with both local and remote services from all vendors.

Figure 3.4 indicates the interactions between the client, the proxy and the service after the proxy has been downloaded to the client. The proxy contains an embed-

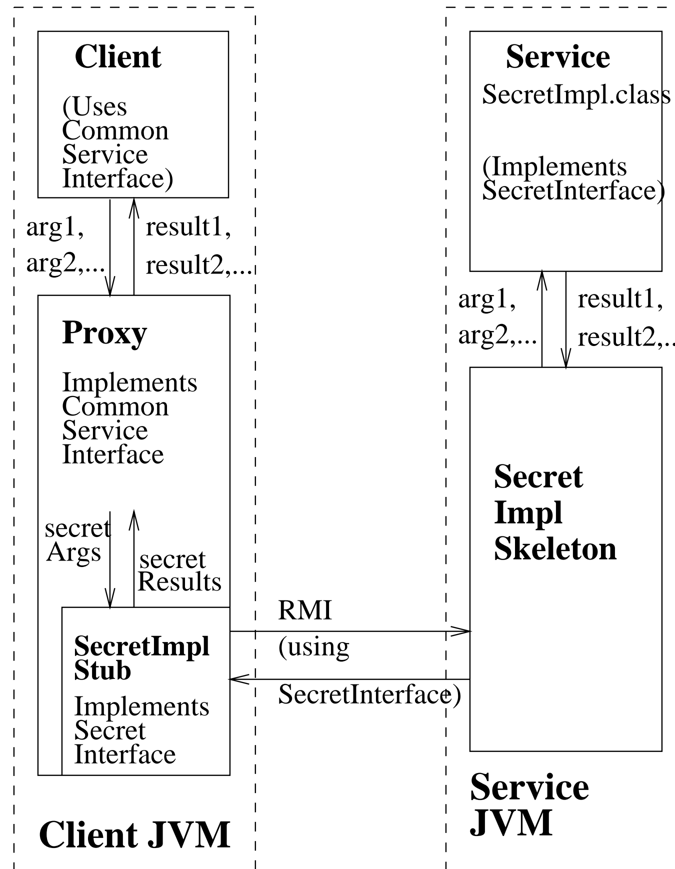


Figure 3.4: Communication between the service and the proxy using RMI

ded stub for the service, and uses it to communicate to the service using RMI and the `SecretInterface`.

Table 3.2 contains a description of the various components of this system, and the sources from which they are derived.

3.4 Complete overview

We have already considered the mechanisms for discovering services, downloading proxies, and proxy-service communication in isolation. We now present a complete overview of the system clearly illustrating the interactions between components.

Class	Supplier	Location from where class is loaded
CommonServiceInterface	Industry Consortium	Present in client JVM
ServiceJavaProxy	Service Provider	Downloaded from service codebase
SecretInterface	Service Provider	Downloaded from service codebase
SecretImplStub	Service Provider	Downloaded from service codebase
SecretImpl	Service Provider	Remains on remote JVM
Client Program	Client Provider	Present in client JVM

Table 3.2: Description of the various components of a Java-based Jini system

Figure 3.5 shows the various components - the lookup service, the service and the client.

The operations performed by each component are as follows.

Lookup service

1. The Lookup service starts up.
2. The Lookup service broadcasts its location over the network, and waits for connections from services.

Service

1. The service starts up.
2. The service discovers the Lookup service.
3. The service downloads the proxy for the Lookup service from the Lookup service codebase.
4. The service requests registration using the Lookup service proxy (the `ServiceRegistrar`).
5. The Lookup service registers the service and returns confirmation of registration.

The confirmation includes the length of the lease granted to the service. If this lease expires, the service is deregistered.

Client

1. The client starts up.
2. The client discovers the Lookup service.
3. The client downloads the proxy for the Lookup service from the Lookup service codebase.
4. The client looks up the service by specifying the interface the service has to implement, and any further attributes the client requires (eg. the printer should be in the basement).
5. The client downloads the service proxy, and any other required classes from the service codebase.
6. The client registers interest in the events pertaining to the service.
7. The Lookup service downloads the class which will be notified of service changes. This class is represented by `ClientEventListener` and is downloaded from the client codebase. Both the current and previous actions use the remote event handling mechanism.
8. The client invokes methods defined in the service interface.
9. These methods are handled by the proxy, which then uses the proxy- service protocol to communicate with the service.

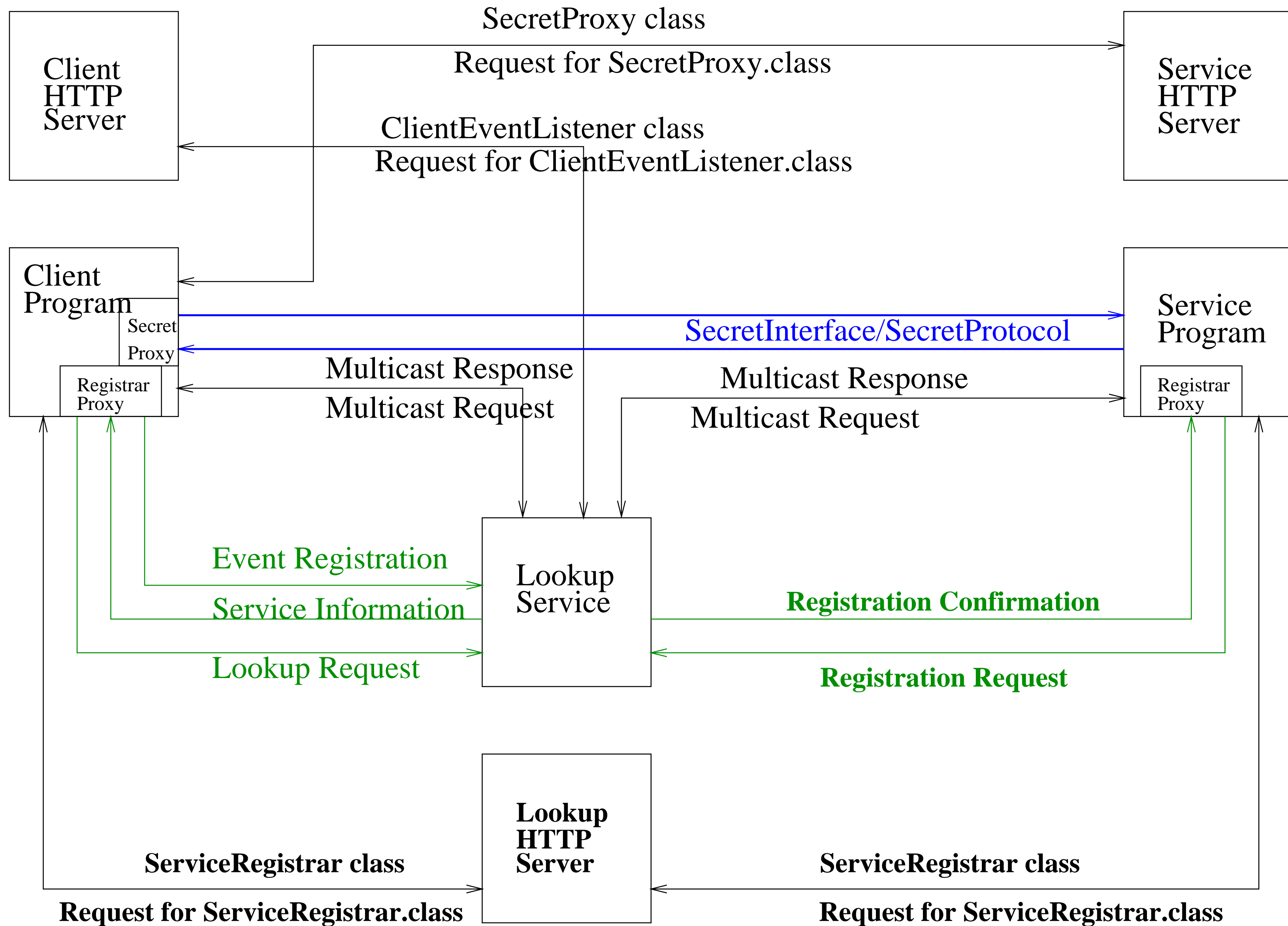


Figure 3.5: Overview of the interaction between components in a Jini system

Chapter 4

Extension to non-Java devices

The Jini Device Architecture Specification [Sund], suggests three methods for connecting non-Java devices to a Jini network. All of these involve having a Java-enabled device handle the interaction between the non-Java devices and the Jini network, and are described in page 4 of the introduction. In order to provide a quick summary, a list of these methods is presented here.

Device Bay: The embedded devices are physically plugged into a Java-enabled *device bay*.

Network Surrogate: The embedded devices contact a Java-enabled *network surrogate*[1] over a network. Subsequently, the proxy communicates with the device through this surrogate.

CORBA: The embedded devices directly interact with a CORBA based lookup service.

[1] 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.

However, each of these methods has its own drawbacks. The most important of these are described below.

Device Bay: There is a physical limitation on the number of non-Java devices which can be connected to the device bay. Also, since all communication between the outside world and the non-Java device is through the common device bay and not their respective proxies and not through a common device bay, the communication protocols are more restricted.

Network surrogate: Even after the client downloads the proxy, the Java-enabled device has to co-ordinate the interaction between them. This places a heavy load on the *network surrogate*, specially as the number of devices it handles increases. Also, this emphasis on the network surrogate decreases the robustness of the system. This is because all communication between the service and the client ceases when the network surrogate goes down, although the client and the service themselves may still be in operation.

CORBA: Implementing a CORBA ORB would be quite complicated, and the other approach of directly interpreting the bitstream will not work well for changes in implementation. Also, using CORBA would require additional programming to interact with the lookup service. In short, although CORBA can be implemented to run directly on the underlying hardware rather than on a JVM, the CORBA implementation would be too complex for a resource-starved system.

We consider a different approach - the notion of a Registration Proxy Server (RPS). This is very similar to the network surrogate approach - the non-Java device finds

the network surrogate in the same manner. However, it utilizes the surrogate only to perform the Jini interactions on its behalf. Once the driver has been downloaded into the client program, all further communication takes place directly between the proxy and the device.

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

4.1 Requirements

This section lists the requirements of each component involved in an RPS-based implementation. These include the embedded device which provides a *service*, the *RPS* used for interaction with the Jini federation, the protocol used for interaction between them, and the *client* which uses the service.

4.1.1 Service Requirements

A service running on an embedded system has the following requirements.

1. The system should contain an implementation of the network protocol over which proxy-service communication takes place. For example, if the proxy and service communicate using a TCP/IP socket connection, the system should contain a TCP/IP stack.

2. The service needs to implement a protocol designed to allow embedded system services to contact the RPS. This *registration protocol* is described in Section 4.2.

3. Since the registration protocol is built using TCP and UDP, the system should have a built-in TCP/IP stack.

4. Since the protocol described in section 4.2 uses UDP broadcast, the service has to be on the same local network as the RPS.

4.1.2 Protocol Requirements

1. The service needs to acquaint the RPS with the name of the proxy class.

2. The service needs to specify the location from which the proxy class can be downloaded. The RPS will download and instantiate the proxy prior to registering it with the lookup service.

3. Parameters describing the service have to be passed to the RPS. These parameters will be used to initialize the proxy.

4. Appropriate handshaking must be carried out to ensure that there are no *half-open* connections.

5. Ideally, the protocol should be independent of both the actual service operation and the proxy-service interaction. Any service should be able to participate in a Jini federation simply by adding in the protocol module and telling the module how to start up the actual service.

4.1.3 RPS Requirements

1. The RPS must be on the same local network as the service. It must also be on the same multicast network as the Jini lookup service.

2. The RPS must be able to download the proxy from the specified URL, instantiate it, and register the service.

3. The RPS must be able to communicate the RMI codebase of the service to the client, rather than its own. If this is not done, the client will try to download the proxy class from the RPS codebase rather than the service codebase. Further details can be found in Section 4.3.

4. The RPS must be able to "ping" the service to confirm that it is still present on the network. In case the service is not found, it should be deregistered from the lookup service.

4.1.4 Client Requirements

1. The client should be on the same multicast network as the lookup service.

2. The client should be able to download and run Java code. This is essential to run the proxy.

3. The client should have the common interface for the service installed in its codebase. The client looks up services matching this interface from the lookup service.

The rest of this chapter describes these requirements in detail, and discusses the methods we chose to meet each of them.

4.2 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)[Dro93], a 4-stage protocol which allows computers to discover DHCP servers, select one of them, and obtain an IP address from it.

4.2.1 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[Bra89].

Figures 4.1 and 4.2 show the various stages in the service and RPS protocols. A brief description of the various states, and the actions performed in each follows.

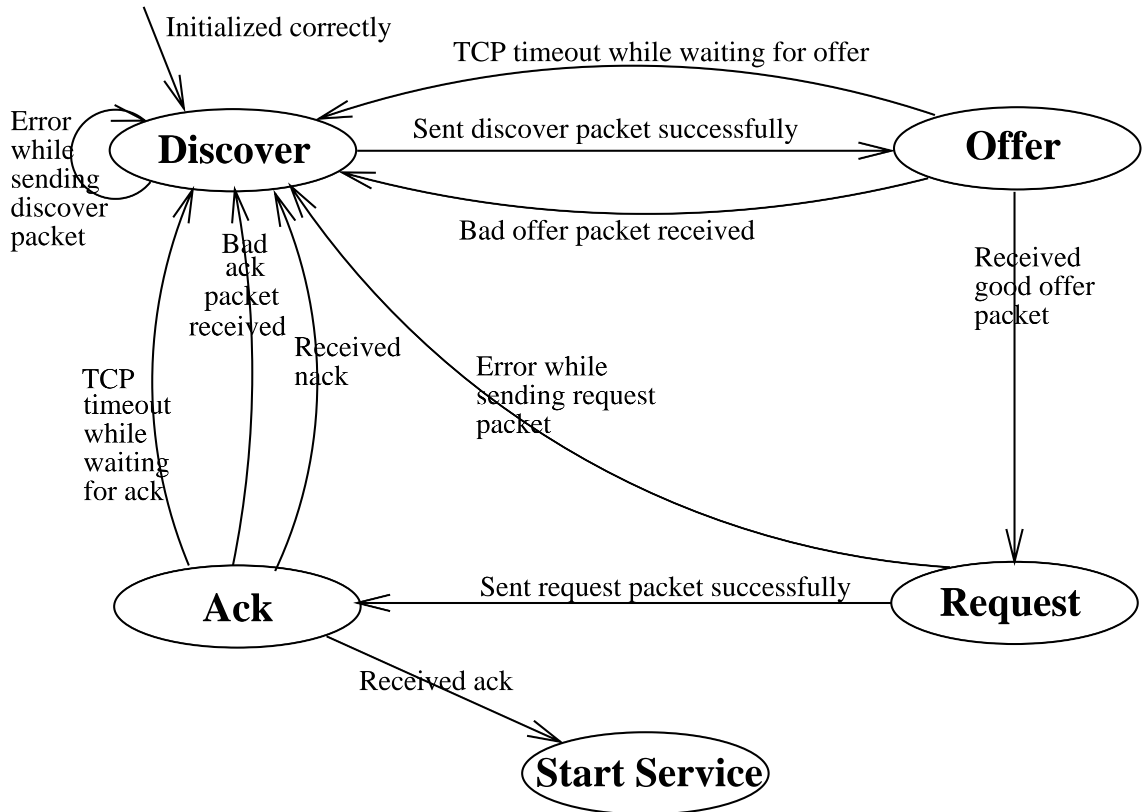


Figure 4.1: Finite state machine description of the protocol on the service side

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 `255.255.255.255`. The `DISCOVER` packet contains

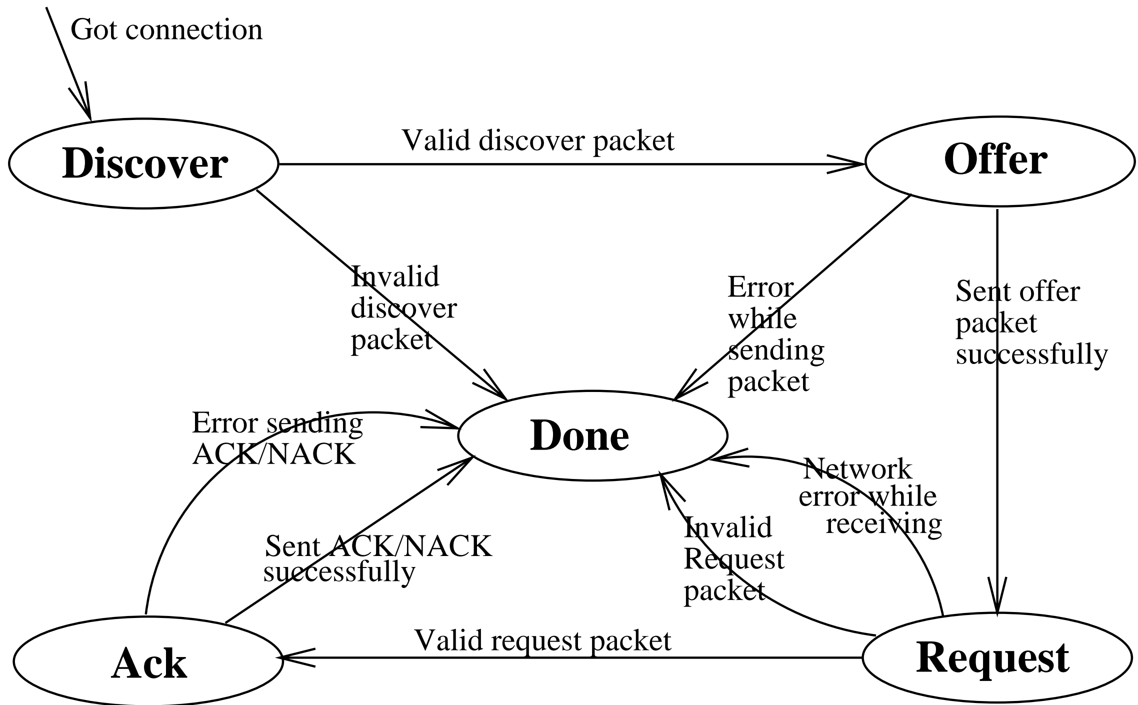


Figure 4.2: Finite state machine description of the protocol on the RPS

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

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 specific to the service.

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 2.2.2),

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.

4.3 Code Download

The mechanism used by Jini for the client to download the proxy is explained in detail in Section 2.2.2. A summary of its usage in Jini is shown below.

1. Downloading code 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. The proxy class file is downloaded from the list of 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 would be 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. Since serialization occurs during registration, the easiest solution 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 proxy (the `ServiceRegistrar`). This implies that the security manager has to be set to `RMISecurityManager`[2]. Unfortunately, `RMISecurityManager` is also more restrictive than the standard. 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[3], 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 add the following line to the policy file.

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

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

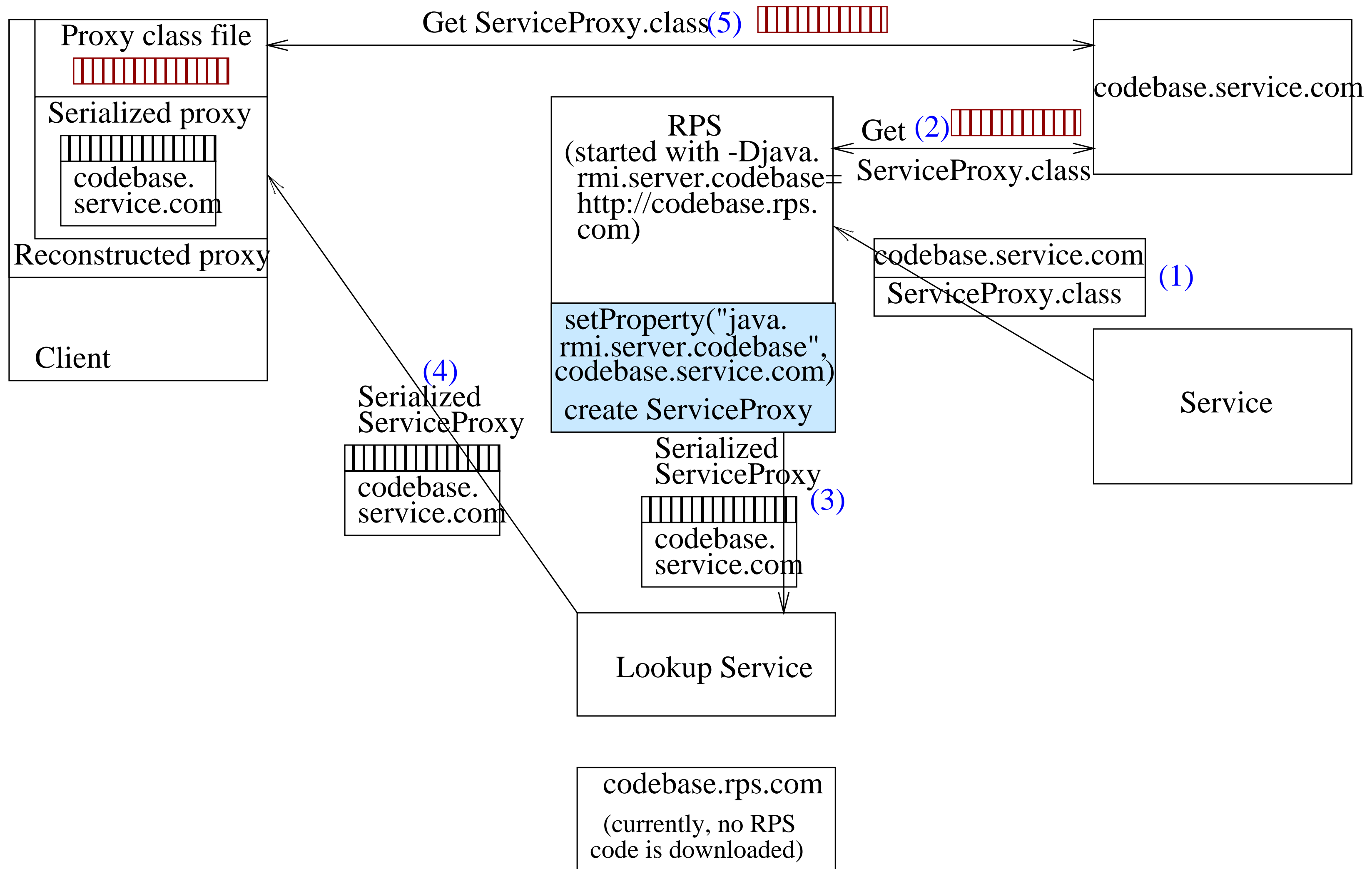


Figure 4.3: Associating appropriate codebases with proxies

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

4.4 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 from the network.

When the service is removed from the network, its 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

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

For example, consider a network device 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, the Lookup service, and the Client, 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, polling all of them. However, this method has the drawback that methods for manipulating the list have to be implemented. The manipulations which have to be implemented are basically addition and deletion. The methods have to be synchronized to allow access of 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 support different refresh lengths for different services, 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 deregistered from the lookup service and the thread is terminated.

4.4.1 Pinging methods

This section describes the three methods we considered to ping the service, and explains how the RPS can support all of them.

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, which may lead to performance degradation if the poll interval 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 RTOS 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. By using multi-threading, this mechanism can provide a non-resource-intensive ping without any input parsing.

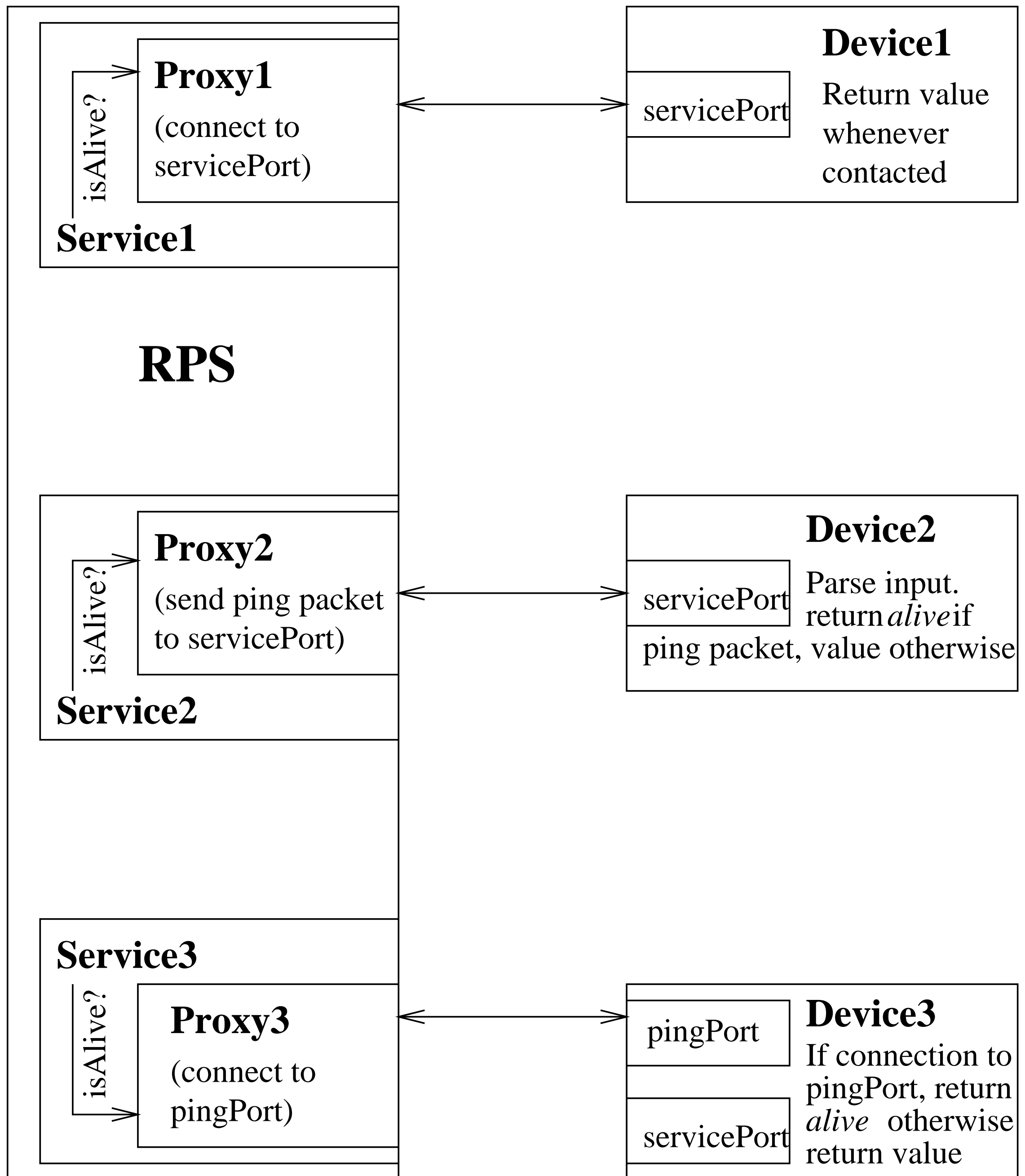


Figure 4.4: Using independent `Service` objects to handle different devices and abstracting different ping methods using proxies

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. Therefore, the RPS has to support not only these mechanisms but also be extensible to support future mechanisms.

We decided to implement this 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. For a list of other functions the proxy has to implement, please refer to Figure 5.1. The RPS can then use this method in the `Service` object described in Section 4.4 to determine the status of the service.

Since 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 be easily adopted by modifying the proxy to match the service. Figure 4.4 illustrates how one RPS can use different mechanisms to ping services.

4.4.2 Proxy-service communication

Since we are trying to connect non-Java systems to a Jini network, the service will generally be implemented in a language other than Java. This means that direct RMI communication between the proxy and the service is no longer possible. Therefore, a protocol implementable in native code has to be used for communication between them. In our test service, we use one built on top of TCP/IP to maintain consistency with the RPS registration protocol.

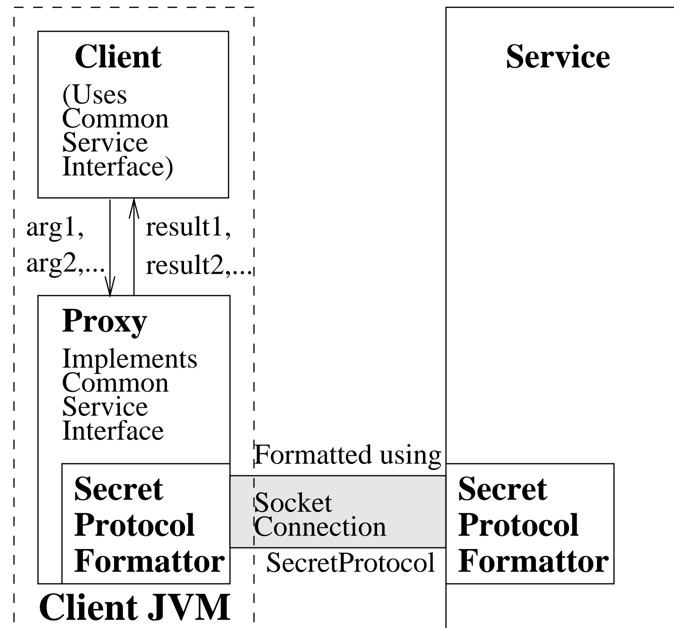


Figure 4.5: Communication between the different components of a C-based Jini system

Protocols other than TCP/IP can also be used for proxy-service communication as long as both the proxy and the service implement the protocol. However, the service *must* implement both TCP and UDP in order to communicate with the RPS using the protocol defined in Section 4.2.

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.

Figure 4.5 illustrates how the client, the proxy and the service communicate using this model after the proxy has been downloaded to the client. Table 4.1 describes the various components, the location from which they are downloaded, and the sources from which they are derived.

Class	Supplier	Location from where class is loaded
CommonServiceInterface	Industry Consortium	Present in client JVM
ServiceCProxy	Service Provider	Downloaded from service codebase
ServiceProgram	Service Provider	Remains on service host
SecretProtocol	Service Provider	Built into `ServiceProgram` and `ServiceCProxy`
Client Program	Client Provider	Present in client JVM

Table 4.1: Description of the various components of a C-based Jini system

4.5 Complete overview

We have already considered, in isolation, the mechanisms used to extend Jini to non-Java devices. We now present a complete overview of the system clearly illustrating the interactions between components.

Figure 4.6 shows the various components - the lookup service, client, the embedded device and the RPS. The operations performed by each component are described below.

Lookup service

The operation of the lookup service is identical to that of a conventional Jini system.

Embedded device

1. The service starts up. At this point, the protocol module is executing.

2. The protocol module broadcasts a request for registration on the local network. This request includes the URL where the code for the service proxy is present (the `serviceCodebase`).

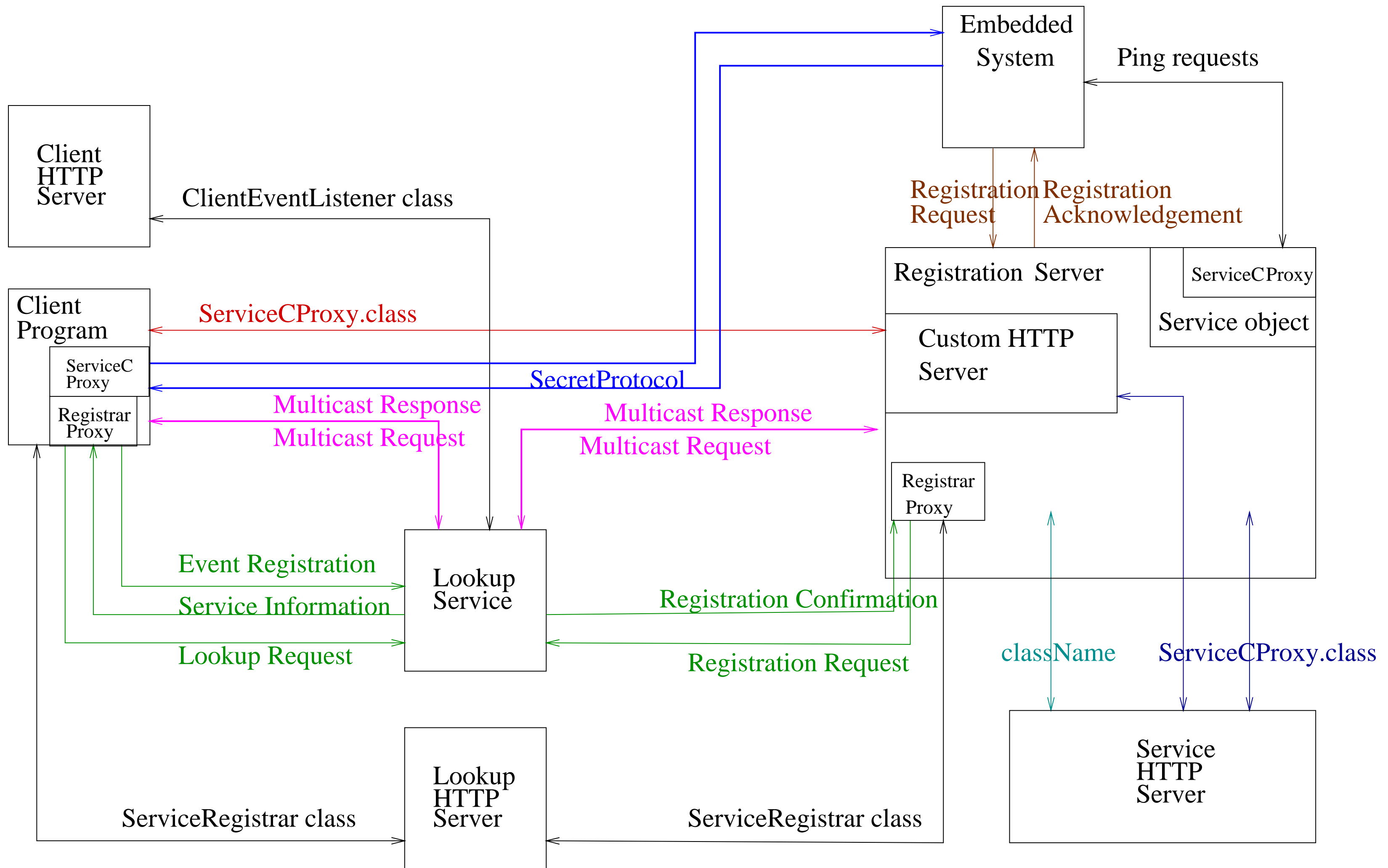


Figure 4.6: Overview of the interaction between components in a JiniLite system

3. The registration protocol module now waits for responses from RPSes on the local network.

4. The protocol module and the RPS now communicate using the registration protocol (Section 4.2). If the device was successfully registered on the Jini network, the RPS returns an ACK as the final step.

5. After recieving the acknowledgement, the registration protocol module starts up the actual service.

6. The embedded device waits for requests from its clients via the registered proxy.

RPS

1. The RPS starts up.

2. The RPS discovers any Lookup services and obtains a reference to their proxies using the discovery protocol.

3. The RPS downloads the proxy code for the Lookup service from the Lookup service codebase using RMI dynamic code download.

4. It listens to registration requests from the local network on port *2326*.

5. On receiving a registration request, it determines the origin of the request (the host address and port) using the information contained in the transport protocol packet (UDP).

6. It determines the `serviceCodebase` from the contents of the request packet.

7. It then offers the service a set of terms. If the service indicates its acceptance of the terms by sending a request packet, the RPS reads in the service-specific parameters from the packet.

8. In the final stages, the RPS contacts the `serviceCodebase` and determines the name of the proxy file. The RPS can then download and instantiate the proxy. In our implementation, the RPS requests the file `className` from the service codebase. This file contains a single line which gives the Fully Qualified Class Name of the proxy for the service. In the example shown in Section 4.4.2, this file would contain the line `com.service.ServiceCProxy`.

9. Since the Jini registration mechanism registers objects and not classes, the RPS has to instantiate the proxy.

10. The RPS downloads the proxy class (`ServiceCProxy`) from the service codebase and instantiates it.

11. Now, the RPS tries to register the `ServiceCProxy` object instance with the Lookup service.

12. After the registration succeeds and a registration confirmation is returned by the Lookup service, the RPS creates a `Service` object which tracks whether the service is alive or not. The `Service` object uses the instantiated proxy to accomplish this.

13. Finally, the RPS sends an acknowledgement to the embedded service.

Client

The operation of the client is identical to that of the conventional Jini system. When the proxy arrives, it is correctly associated with the service codebase and not the RPS codebase. This means that the fact that the proxy was created and initialized by the RPS is totally transparent to the client.

Chapter 5

A Sample Application

In this chapter, we demonstrate the concepts of Chapter 2 by the example of a simple service. This is the `MessageSource` service which broadcasts a message to the universe. This message is different for different instances of the service, and is intended to be short. For example, *"Sell IBM"* or *"Drink Coke"*. Whenever the `MessageService` is contacted, it returns the message it has to broadcast.

In this chapter, with the help of code fragments, we illustrate how such a device designer would implement such a service. This chapter does *not* examine the implementations of the RPS and the protocol module. They are assumed to have standard implementations, which can be directly used by the device designer. For an algorithmic description of the RPS and the protocol module, please see Sections 4.5 and 4.2.

5.1 The `ServiceProxy` common interface

Before the RPS registers the proxy, it has to initialize it with:

1. the address and port of the embedded service, and

2. the parameters obtained from the service during registration.

Also, the proxy has to implement an `isAlive` method which the RPS can use to determine whether the service is alive or not. It has to also specify the attributes the RPS should use while interacting with the lookup service using the *join protocol* (Section 2.2.1. It is expected that these attributes will be obtained from the parameter string for the service.

In order to ensure that every proxy which the RPS registers allows all these operations, we define an interface - the `ServiceProxy` which every proxy which wants to be registered by an RPS has to implement.

The ServiceProxy interface is shown in Figure 5.1

```
1   public interface ServiceProxy
2   {
3       public void setClientAddress(InetAddress clientAddr, int clientPort);
4       public void setClientParams(String params);
5       public boolean isAlive();
6       public Entry[] getAttributes();
7   }
```

Figure 5.1: The `ServiceProxy` interface

5.2 The `MessageSource` Interface

As we have seen in Section 3.1, every service has a common interface defined by industry consensus. This interface is implemented by all versions of the service. Since we do not anticipate that a simple message service will ever get popular enough to have an industry standard, we will define the interface ourselves.

Because the service simply broadcasts a message, the `MessageSource` interface has a single method which gets the message. The interface definition is shown in Figure 5.2

```
1  public interface MessageSource
2  {
3      public String getMessage();
4  }
```

Figure 5.2: The `MessageSource` interface

5.3 The `MessageCProxy` proxy class

We now consider the design of the proxy class. We know that the proxy class has to implement both the `MessageSource` interface for interaction with the client, and the `ServiceProxy` interface for interaction with the RPS. Since this is a simple service, the implementation of all the methods is relatively straightforward.

As we can see in Figure 5.4, the main methods are implemented as follows.

setClientAddress: The client address and port are stored in fields of the class.

getMessage: A socket connection is opened to the client using the stored parameters. The message sent out by the client is read in. Since the message is intended to be short, only 1024 characters are read in. The buffered characters are then converted to a `String` by using a `StringBuffer`.

isAlive: The service is checked to see if it is alive by simply requesting the message. Since sending the message out is not a resource-intensive task, we have chosen simplicity over efficiency.

```
1   public class MessageCProxy implements Serializable, MessageSource,
2                                       ServiceProxy
    {
4       InetAddress serviceHost;
        int port;
6       Properties clientParams;

8   /* The host ID and port of the embedded system have to be known to the proxy.
     * The serviceHost and port are generally passed in while creating the system,
10   * but can be set later. */
        public MessageCProxy(String serviceHost, int port) {
12          try {
                this.serviceHost = InetAddress.getByName(serviceHost);
14              this.port = port;
            }
16          catch(Exception e){}
        }
18
        public void setClientAddress(InetAddress clientAddr, int clientPort) {
20          this.serviceHost = clientAddr;
            this.port = clientPort+10;
22      }
    /* This method allows the parameters obtained from the embedded system to be
24   * passed in to the proxy. Currently, the MessageProxy does not use these
     * parameters. */
26      public void setClientParams(String params) {
            try {
28              clientParams = new Properties();
                clientParams.load(new ByteArrayInputStream(params.getBytes()));
30          }
            catch(Exception e) { e.printStackTrace(); }
32      }

34      public boolean isAlive() {
            String msg = getMessage();
36          if(msg == null)
                return false;
38          return true;
        }
```

Figure 5.3: The `MessageCProxy` service proxy

```
40      public String getMessage() {
            byte[] buffer = new byte[1024];
42          int c = 0;
            try {
44              System.out.println("Trying to open connection to "+serviceHost+":"+port);
                Socket s = new Socket(serviceHost, port);
46              InputStream in = s.getInputStream();
                in.read(buffer,0,1024);
48              String message = new String(buffer);
                System.out.println("Got message "+message);
50              s.close();
                return message;
52          }
            catch(Exception e) {
54              e.printStackTrace();
            }
56          return null;
        }
58  // Setting the message is not yet supported.
        public void setMessage(String message) { }
60
        public Entry[] getAttributes()
62      {
    /*  There are certain standard attributes for every service defined by SUN.
64      It is assumed that other industries will define standard attributes
        for their services, along with standard interfaces.  */
66
            ServiceInfo info = new ServiceInfo( "Message Service",
68                              "UCSC", "UCSC", "0.1", "basic", "0001");
            Name name = new Name("C Slug Message");
70          Comment comment = new Comment("This generates a silly message");
            Location location = new Location(clientParams.getProperty("floor"),
72                  clientParams.getProperty("room"),
                    clientParams.getProperty("building"));
74          Entry[] attrSets = new Entry[4];
            attrSets[0] = info;
76          attrSets[1] = name;
            attrSets[2] = comment;
78          attrSets[3] = location;
            return attrSets;
80      }
```

Figure 5.4: The `MessageCProxy` service proxy

setClientParams: The parameters received from the client are parsed. This implementation uses an encoding of the form *¡key¿=¡value¿* so that it can easily be read into a java property list.

getAttributes: Here, the parameters read in from the client are used to create the standard attributes defined by Sun Microsystems. We expect that proxies for more commerical services will have a wider variety of attributes, including some defined directly by the vendor.

Operations which are not required for this simple service, are implemented using placeholders. The complete implementation of the proxy is shown in Figure 5.4.

5.4 The `MessageCSource` service

The `MessageCSource` service is written in C and is also very simple. It simply initializes the TCP/IP networking, opens a socket, and waits. Every time it receives a connection to the socket, it outputs its message. The relevant code fragment is shown in Figure 5.5

5.5 Invoking the service

The `MessageSource` service has to be automatically started up by the protocol module after it recieves the acknowledgement of registration (the ACK message) from the RPS. The method in which this is done depends on the embedded system.

Supports the `exec` command: The easiest method to start up the `MessageSource` service is by using the simple `exec` command. No `fork` command is needed because

```
1   void runEmbeddedSystem(int port, char *message)
2   {
3       int s;
4       struct sockaddr_in from;
5
6           // Set the port of the address to the one specified
7       sin.sin_port = port;
8       messageLen = strlen(message);
9       s = socket(AF_INET, SOCK_STREAM, 0); // Create the socket.
10      if(s<0)
11          printf("Error creating socket %d\n", s);
12
13          // Bind the socket to the address
14      if(bind(s, &sin, sizeof(sin)) < 0)
15          printf("Error binding socket %d\n", s);
16      listen(s,3);
17      for(;;) // Wait forever
18      {
19          int ns, len = sizeof(from);
20          ns = accept(s, &from, &len); // Wait for a connection
21          if(ns < 0)
22          {
23              printf("Error accepting %d \n", ns);
24              close(s);
25          }
26          write(ns, message, messageLen); // Send the message
27          close(ns);  // Close the temporary socket
28      }
29  }
```

Figure 5.5: The `MessageCSource` service

the protocol module does not need to continue running. This method achieves the minimum coupling between the protocol module and the `MessageSource` service, but can be used only if the embedded system supports the `exec` command.

No support for `exec`: For systems which do not support filesystems, the code might have to be linked in directly to the protocol module. This leads to close coupling between the protocol module and the `MessageSource` service, because any change to

either part needs a recompile of the entire system, but cannot be avoided if the underlying OS requires it. This is the approach we had to follow in our implementation of the system.

Chapter 6

Analysis and Verification

The design process for the RPS-service registration protocol in Section 4.2 on page 37 was quite ad-hoc. It consisted of finding a similar protocol, DHCP[Dro93] and modifying it to meet the service needs. This chapter evaluates the protocol using formal verification techniques.

6.1 Formal verification

There are two main methods for the formal verification of network protocols. The first one constructs a reachability graph of the protocol and checks for safety and liveness conditions which can be expressed as graph properties. The second one represents these conditions as temporal logical assertions and verifies them using automated program verification techniques.

The temporal logic method is likely to express more conditions than pure reachability because it can reason on arbitrary sequences of states. However, the reachability

graph method is more popular among the network community because it can easily be automated. Brief descriptions of the two methods can be found in Sunshine's overview of the subject[Sun79], and chapter 11 of Holzmann's book[Hol91]. A more detailed description of the reachability analysis model can be found in Bochmann[Boc78], while verification of using temporal logic is discussed in Lamport and Lynch[LL90].

We have used the reachability analysis method in this report. The various steps in the verification are described below.

6.1.1 Construction of the graph

First, the processes at both ends have to be represented as Communicating Finite State Machines (CFSMs). CFSMs are similiar to regular Finite State Machines, except that instead of reading characters from a string, they transition between states based on certain *actions*. These actions typically deal with the transmission or receipt of messages.

In order to represent the Finite State machines representing the device and RPS sides by CFSMs, we had to make some changes to the machines shown in Figures 4.2 and 4.1. These changes are listed below, and are also shown in Figure 6.1.

1. In order to make the verification apply to both TCP and UDP, we added timeouts to the FSMs. Processes can now either detect errors during transmission by being informed from the lower level (TCP) or by timing out when no answer is received (UDP).

2. Ten actions were identified: S1-S4 to send messages 1-4, R1-R4 to receive messages 1-4, E to represent an error and T to represent a timeout.

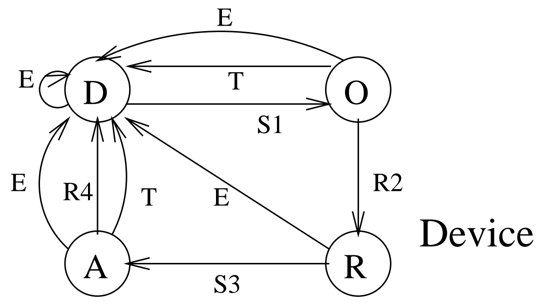
3. In Figure 4.2, we represented the FSM as terminating whenever there was an error. This was because we implemented the RPS with the standard multi-threaded behaviour - when it recieved a connection, it spun off a separate thread. The FSM represents the operations in a single connection. This also implies that if there are errors and the connection is aborted, the device will attempt to connect again. At this point, the RPS FSM will return to the `DISCOVER` state. Therefore, we removed the `DONE` state, and had all its incoming transitions go back to the `DISCOVER` state.

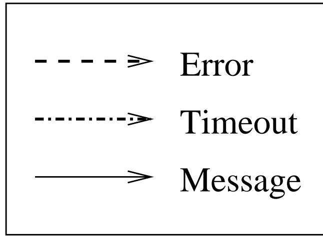
4. This change left the RPS able to handle an infinite number of connections, while the device could only handle one connection. In the interests of symmetry (to simulate multiple devices), we also removed the `DONE` state from the device FSM.

We then constructed the graph of all possible states of the system. Bochmann [Boc78] explains the construction of a similar graph which uses synchronous coupling instead of the asynchronous coupling we use. We used that work, and the brief descriptions from chapter 8 of Holzmann[Hol91] and Sunshine[Sun79] to obtain this graph.

The states of the graph are basically the cartesian product of the individual states of the FSM. Since we have only one message in the network at any time, we can use the empty medium model, and this graph provides a model of all possible states the system can be in. We now add in the transitions by using the following rule.

Consider a composite state q_0q_1. The actions out of q_0q_1 must be the union of all possible actions out of q_0 combined with all possible actions out of q_1. Let a be a action from q_0 to q_0' in the original CFSM. Therefore, the action a from q_0q_1 will lead to $q_0'q_1$. We can consider a similar case for q_1 and q_1'.





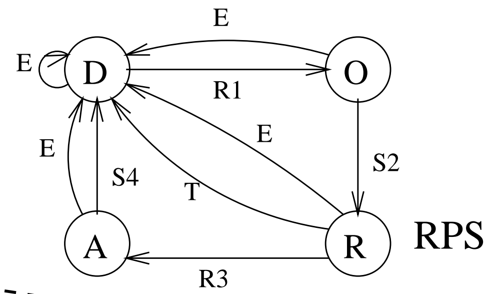


Figure 6.1: Formal specification of the protocol

We now come to the concept of enabled actions. An action a is enabled on a state q_0q_1 if the system can execute a when in state q_0q_1. Error and send actions are always enabled, but receive actions are only enabled if the corresponding send action was an input to the state, and timeout actions are only enabled if q_0 or q_1 is a state which is waiting for input. For the sake of simplicity, we have only shown enabled actions in the graph. The final graph is shown in Figure 6.1 on page 67 where message actions are shown in as continuous lines and labelled with their messages, error actions are shown dashed, and timeout actions are shown dotted.

6.1.2 Verification of properties

We now evaluate the *safety* and *liveness* properties of the protocol. Safety properties refer to the condition that nothing bad happens in the protocol, while liveness refers to the conditions that something good eventually does happen. Some examples of safety properties are the absence of livelock and deadlock, while some examples of liveness properties are proper termination and stability. Although arbitrary safety and liveness properties can be chosen depending on the requirements of the system, we have chosen to verify only the standard properties suggested in Bochmann[Boc78].

Safety properties

Absence of deadlocks: A deadlock is a state in which the system cannot proceed because all protocol processes are waiting for conditions which will never be fulfilled. A deadlock can easily be found because it would be represented by a global system state reachable from the global start state, from which there are no outgoing actions. It can be

seen from a visual inspection of the reachability graph in Figure 6.1 on page 67 that no such state exists.

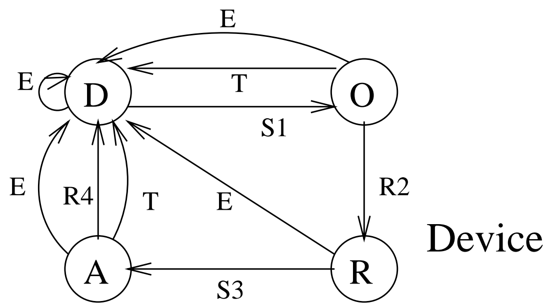
Absence of livelocks: Livelocks, also called "tempo-blockings" in Bochmann's paper, are cases in which the system cycles through a fixed set of states without ever making progress towards the final state. Livelocks are harder to detect because they show up in the reachability graph as loops. Although loops can be detected easily, the problem is that some loops are necessary for the operation of the system (for example, the loop that includes the start and final states).

In order to show that this protocol does not contain any livelocks, we first consider the reachability graph without error and timeouts. This is shown in Figure 6.2 on page 70.

This figure contains a single loop $DD \xrightarrow{S1} OD \xrightarrow{R1} OO \xrightarrow{S2} OR \xrightarrow{R2} RR \xrightarrow{S3} AR \xrightarrow{R3} AA \xrightarrow{S4} AD \xrightarrow{R4} DD$ which corresponds to the correct operation of the protocol. Therefore, if there are any livelocks in the protocol, they must be caused due to errors and/or timeouts. At this point, we make two further assumptions about the system.

1. There cannot be an infinite number of errors in a system. This is a reasonable assumption to make, unless a side has crashed. However, if a side has crashed, it is assumed to have reached its start state, and we can see that all states in which one side is in the start state eventually reach the global start state.

2. If a send or receive action is enabled in a state, it will eventually be executed. This is also a resonable assumption because these actions are almost instantaneous in practice. Because we consider an asynchronous system, we cannot assume any time limit on these actions, but we can guarantee that they will be executed if they are



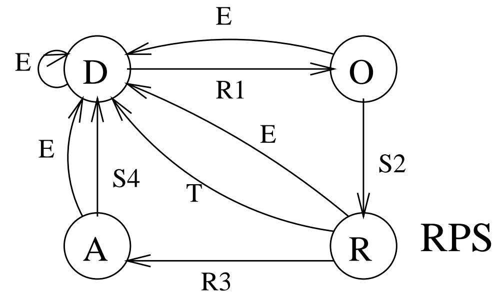


Figure 6.2: Transitions due to message transfer

enabled long enough.

With these two assumptions, and the fact that every state in the global system graph shown in Figure 6.2 on page 70 has at least one transition leaving it, the following reasoning shows that the system cannot have any livelocks.

1. The livelock cycle cannot include errors because errors cannot occur infinitely.

2. The livelock cycle cannot include timeouts because it is guaranteed that the actions will eventually take place.

3. The livelock cycle cannot consist purely of message actions because the message actions do not form a loop.

Liveness Properties

Presence of a live state: A state is live if it can be reached from all states of the overall system that are reachable from the initial state. A live state is also sometimes called the "home" or "steady" state in the literature. The state DD in Figure 6.1 is the home state of our system.

Self-synchronization and Stability: A system is self-synchronous if, started up in any possible global state, it always returns, after some finite number of operations, to the normal cycle of operation including the home state.

Of the states in our system,

DD, OD, OO, OR, RR, AR, AA and AD are part of the normal cycle of operation,

DO, DR, DA, OA, RD are reachable from DD and contain numerous transitions to the normal cycle, and

RO, AO and RA are not reachable from DD (they contain no incoming transitions), but all of them have transitions to states reachable from DD.

Therefore, the protocol is self-synchronizing, and thus, quite stable.

Chapter 7

Conclusion

Jini is a set of protocols which run on top of Java, and are designed to make it easy for small and embedded devices to talk to programs which want to use them. This is done by allowing the client to download the device *proxy.* A proxy is similar to the driver for a device and hides the complexity of the device from the client.

A directory of device (or *service*) proxies is maintained, and 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.

In the early chapters, we described the various Jini operations in brief, and then considered the design of a Java-based Jini system. We presented an overview of such a system with the components and their interactions clearly defined.

In cases when the embedded device has insufficient resources to support a Java Virtual Machine, the controlling program will have to be compiled into native code. Since Jini runs on top of Java, such devices cannot directly interact with a Jini *federation.*

Later, we dealt with an extension to Jini which allows it to be used even when the underlying implementation is not written in Java. This is done by implementing a **Registration Proxy Server** (RPS) which is written in Java, and which handles the interaction with the Jini system on behalf of non-Java devices. Unlike the approaches suggested by Sun, the RPS is not part of the communication between a client and a service once the two are connected.

In Chapter 4, we presented the design of the RPS and the protocol used by a non-Java device to contact it. We enumerated the problems involved in using an RPS and presented easily extensible solutions to them. We also stepped through the process a device designer has to follow while implementing a simple non-Java device - the `MessageSource`.

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[Mon97] as the OS.

The protocol was also formally verified using the Communicating Finite State Machine (CFSM) method described in Holzmann's book[Hol91] and Bochmann's paper[Boc78]. 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.

Future Work

There are two main directions for future work. The first deals with a further extension of Jini to support non-Java clients as well. As we have seen, the current extension allows non-Java *services* to participate in a Jini *federation*. However, because the client has to interact directly with the Jini federation, and download the Java-based proxy to use the service, it has to be written in Java. Therefore, the next logical step is to extend Jini to non-Java clients, which would then allow two non-Java devices to talk to each other without additional configuration.

The second deals with the various proposed standards for embedded system control. As we have seen, these include CORBA, DCOM and Inferno in addition to Jini. This extension deals *only* with the Jini standard. It would be interesting to see if this work can be extended to work with other standards as well. This could eventually pave the way for integation of embedded networks using different standards, and increase the potential for

interaction between devices.

As we have seen, after the embedded device contacts the RPS, the RPS downloads and instantiates its proxy before registering it with a lookup service. This could potentially be a significant security hole if the proxy contains malicious code. In general, since the device has to be plugged into the same local network as the RPS, we can estimate that it is not malicious. Therefore, the chances of the proxy being malicious are low and this protocol is relatively safe.

As Crichton, Davies and Woodcock have shown in their paper on access control in Jini [CDW99], even the standard Jini protocol has security limitations. It is essential to evaluate the security aspects of this protocol in the light of those limitations, and any further ones introduced by the interaction of the device and the RPS.

Bibliography

[Boc78] Gregor V. Bochmann. Finite state description of communication protocols. *Computer Networks*, 2:361–372, 1978.

[Bra89] R. Braden. Requirements of Internet Hosts – Communication Layers. *STD 3, RFC 1122*, October 1989. http://cis.ohio-state.edu/htbin/rfc/rfc1122.html.

[CDW99] Charles Crichton, Jim Davies, and Jim Woodcock. When to trust mobile objects: access control in the Jini^{TM} Software System. In D. Firesmith, R. Riehle, G. Pour, and B. Meyer, editors, *Proceedings of Technology of Object-Oriented Languages and Systems - TOOLS 30, Santa Barbara, CA, USA, 1-5 Aug. 1999*, pages 116–125. Computer Laboratory, Oxford University, UK, August 1999.

[DPW^{+}] Sean Dorward, Rob Pike, Phil Winterbottom, Eric Grosse, Jim McKie, Dave Presotto, Dennis Ritchie, Ken Thompson, and Howard Trickey. Inferno. http://inferno.bell-labs.com/inferno.

[Dro93] R. Droms. Dynamic Host Configuration Protocol. *RFC 1541*, October 1993. http://cis.ohio-state.edu/htbin/rfc/rfc1541.html.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall Software Series, 1991.

[LL90] Leslie Lamport and Nancy Lynch. Distributed computing: Models and methods. *Handbook of Theoretical Computer Science*, pages 1158–1199, 1990.

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

[Micb] Sun Microsystems. Java remote method invocation specification. `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html`.

[Mon97] Bruce R. Montague. JN: OS for an Embedded Java Network Computer. *IEEE Micro*, 17(3), May 1997.

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

[Suna] Sun Microsystems, Inc. Java serialization documentation. `http://java.sun.com/products/jdk/1.2/docs/guide/serialization/index.html`.

[Sunb] Sun Microsystems, Inc. Javaspaces specification. `http://www.sun.com/jini/specs/js101.ps`.

[Sunc] Sun Microsystems, Inc. Jini API documentation. `http://www.sun.com/jini/`.

[Sund] Sun Microsystems, Inc. Jini device architecture specification. `http://www.sun.com/jini/specs/devicearch101.ps`.

[Sune] Sun Microsystems, Inc. Jini discovery and join specification. `http://www.sun.com/jini/specs/boot101.ps`.

[Sun79] Carl Sunshine. Formal techniques of protocol specification and verification. *Computer*, 12(9):20–27, September 1979.