A Generic Architecture to Control Jini Services over the Internet

Brian McSweeney

A dissertation submitted to the University of Dublin,
in partial fulfilment of the requirements for the degree of
Master of Science in Computer Science

September 2001

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

____

Brian McSweeney
14th September 2001

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

____

Brian McSweeney
14th September 2001

Abstract

Distributed computer systems have brought many advantages over traditional centralised systems. However these systems have innate complications such as partial failure, lack of system wide knowledge, concurrency etc. Programming environments for these systems have typically been few and have often failed to adequately address, or even ignored, the associated complications of distributed systems. Sun MicroSystems' Jini technology [Jini Spec '99] is a new distributed systems programming technology where programmers write software as Jini *services*. It provides the mechanisms by which programmers can try to manage these distributed systems complications.

This dissertation aims to address three issues currently restricting the widespread adoption of Jini technology.

- Firstly, the usage of Jini has initially been restricted to Local Area Network (LAN) environments. This thesis addresses this issue by describing a generic architecture to allow Jini services to be accessed and controlled over the Internet.

- Secondly, the core Jini technology installation files necessitate over 3MB of memory, thus impeding Jini services on devices of limited memory capacity. This thesis investigates how Jini services can include devices of limited memory capability and describes the implementation of such a service.

- Finally, the implementation of a Jini client for a specific service currently necessitates providing the client with information about the service in advance. This is a significant restriction on Jini technology in that each service writer must provide its own client implementation for that service. This thesis describes and implements a method by which clients can control Jini services without any knowledge of them, thus providing a single Jini client to all services.

Acknowledgements

I would like to thank my supervisors Alexis Donnelly and Simon Dobson for their interest, enthusiasm and guidance throughout the course of this project. I would also like to thank all my friends, family and classmates for their friendship, support and help during the year.

Contents

1 INTRODUCTION ... 1
1.1 DISTRIBUTED SYSTEMS ... 2
1.2 JINI TECHNOLOGY ... 3
1.3 PROJECT GOALS ... 4
1.4 OVERVIEW OF DESIGN ... 5
1.5 PROJECT ACHIEVEMENTS ... 7
1.6 ROADMAP ... 7
2 BACKGROUND – JINI TECHNOLOGY ... 9
2.1 NETWORK SETUP ... 9
2.2 BASIC JINI SERVICE ARCHITECTURE ... 10
2.3 SERVICE DESIGN CHOICES ... 10
2.4 JINI SERVICE FUNCTIONS ... 12
2.5 JINI CLIENT FUNCTIONS ... 15
2.6 ADVANCED JINI CONCEPTS ... 18
2.6.1 Remote Events ... 18
2.6.2 Distributed Transactions ... 19
3 STATE OF THE ART ... 21
3.1 DISTRIBUTED PROGRAMMING TECHNOLOGIES ... 21
3.1.1 Remote Method Invocation (RMI) ... 21
3.1.2 Common Object Request Broker Architecture (CORBA) ... 22
3.1.3 Enterprise Java Beans (EJB) ... 23
3.2 WEB SERVICE TECHNOLOGIES ... 24
3.2.1 Simple Object Access Protocol (SOAP) ... 24
3.2.2 Universal Discription Discovery and Integration (UDDI) ... 24
3.3 SPONTANEOUS NETWORKING TECHNOLOGIES ... 26
3.3.1 Universal Plug and Play (UPnP) ... 26
3.3.2 Salutation ... 27
3.4 RELATED JINI RESEARCH PROJECTS ... 27
3.4.1 The SOAP-UDDI Project ... 28
3.4.2 The Surrogate Architecture ... 28
3.4.3 The Service User Interface Project ... 30
4 DESIGN ... 33
4.1 CONTROL OF JINI SERVICES OVER THE INTERNET ... 33
4.1.1 A Web Bridging Architecture ... 33
4.1.2 Reflection – A Generic Jini Client ... 35
4.2 LIMITED MEMORY CAPABLE DEVICES AS JINI SERVICES ... 38
4.2.1 A Thin Proxy Architecture to Control A Lego Robot ... 39
4.2.2 Accessing CORBA Services from Jini Clients ... 40
5 IMPLEMENTATION ... 41
5.1 IMPLEMENTING THE WEB BRIDGING ARCHITECTURE ... 41
5.1.1 A SOAP Jini Client/Web Bridge ... 41
5.1.2 A Servlet Jini Client/Web Bridge ... 44

5.1.3 Implementing A Generic Jini Client with Reflection *45*
5.2 IMPLEMENTING LIMITED MEMORY DEVICES AS JINI SERVICES 46
5.2.1 Implementing a Lego Robot Jini Service *47*
5.2.2 Implementing a Jini/CORBA Service *48*
6 CONCLUSIONS 50
6.1 GOALS ACHIEVED AND COMPLETED WORK 50
6.2 FUTURE WORK 52
7 BIBLIOGRAPHY 53
8 APPENDIX 1 56
8.1 JINI WEB SERVICES SYSTEM 56

Table Of Figures

2.1	Basic Jini Service Architecture	9
2.2	Jini Design Pattern One	9
2.3	Jini Design Pattern Two	10
2.4	Jini Design Pattern Three	10
3.1	Java 2 Enterprise Edition Technology Overview	22
3.2	Web Services Protocol Stack	24
3.3	The Surrogate Architecture	28
4.1	Web Bridging Architecture	34
4.2	Generic Client Architecture	36
4.3	Thin Proxy Architecture To Control Lego Mindstorms Robot	38
4.4	Jini Corba Architecture	39
8.1	Jini Web Services System Start	55
8.2	Stage 1: The services are discovered and Listed	56
8.3	Stage 2: The Interfaces Are Listed	57
8.4	Stage 3: The Methods Supported Are Listed	58
8.5	Stage 4: The Results Are Displayed and the Service is Offered Again	58

Chapter 1

1 Introduction

The exponential growth of the Internet has been accompanied by technologies that are designed to make distributed systems programming easier. However, fundamental research in this area [Tanenbaum '95] concludes that distributed systems are inherently unreliable and distributed programming is complex. Jini technology [Jini Spec '99] is a new technology designed by Sun Microsystems that is aimed at addressing these difficulties. It takes the new programming paradigm approach of forcing programmers to design their software systems as Jini services which must acknowledge and account for the difficulties and inherent unreliability of distributed computing.

However the adoption of Jini technology has been not as widespread as initially hoped. There are three main factors that may have hampered Jini's proliferation.

1. Firstly, the usage of Jini has initially been restricted to LAN environments.

2. Secondly, in order to be able to run a Jini service on a specific machine, that machine must have at least 3 MB of available memory to install the core Jini technology files, thus further restricting Jini's adoption.

3. Finally, in order to be able to utilise a Jini service, a client must be provided with information about the service at compile time. In this current situation, it is impossible to have a generic Jini client that can access all Jini services, which would be desirable in certain circumstances (discussed in section 3.4.3). Jini clients can currently only access services which are written before the client, and which it has information about at compile time.

This thesis aims to address all three factors. In addressing the first and third point, the thesis designs and implements a generic architecture to allow Jini services to be accessed and controlled via the Internet without any need for information about the service at compile time. To address the second point, it investigates how these services can include devices of limited memory capability. It designs and implements such a service using a Lego Mindstorms robot. It also designs and implements an architecture to allow such other services, written in languages other than Java, to be accessed from Jini clients using the Common Object Request Broker Architecture (CORBA).

This chapter provides an introduction to the area of distributed computing, followed by an overview of Jini technology. The main project goals, design and achievements are then outlined and finally a synopsis of the structure of the remainder of this dissertation is provided.

1.1 Distributed Systems

"*A distributed system is a collection of independent computers that appear to the users of the system as a single computer*" [Tanenbaum '95]. The growth of distributed systems over traditional centralised systems has been a trend in the computer industry since the late eighties which can be explained by the advantages of the former systems over the latter. These advantages include a better price/performance ratio, increased speed, the inherent distribution of some applications, reliability and the ability to add computing power in small increments.

However, design of distributed systems has led to the identification of the following seven associated fallacies: the network is reliable; latency is zero; bandwidth is infinite; the network is secure; topology doesn't change; there is one administrator; transport cost is zero. [Deutsch]. The goal of distributed programming paradigms and technologies is to make it easier to program systems in which processes on different machines communicate with each other. Unfortunately however, the current distributed object programming technologies such as Java RMI, CORBA, DCOM and EJB fail to adequately address these seven fallacies. Their architectures gloss over

issues such as partial failure and don't provide a model to the programmer as to how a distributed software system should behave when communication between its components fails. Compounding these problems is the fact that distributed algorithms are inherently complex due to concurrency, partial failure, lack of global knowledge and performance and scaling issues.

Jini technology takes a new approach to distributed systems programming. The Jini model tries to ease the administration burden of distributed systems by allowing software network services to just "plug and work". There should be no need to edit configuration files. The software (or hardware) service, once started, should automatically and seamlessly be able to be found by clients. Jini does this via "spontaneous networking" whereby the services announce themselves to the network and clients automatically find them. Jini also supports redundant infrastructure and changes in the network topology in a seamless manner. Furthermore, by making the programmers of Jini services acknowledge and deal with the inherent unreliability of distributed systems, communities of Jini services become self-healing. Given time the system will repair damage to itself. The core parts of the Jini technology model that enable these features are the topic of the next section.

1.2 Jini Technology

Three main components of the Jini technology enable Jini services and clients to spontaneously communicate, with limited need for administration.

1) A Lookup Service – this is essentially a meta-service or naming service which keeps track of all existing Jini services on the network. It is similar in function to the RMI registry, the Corba naming service etc. Sun Microsystems' default implementation of this lookup service is named Reggie.

2) Discovery – in order for services to be able to register themselves with a lookup service, they initially send multicast messages out on the network searching for one. To discover available services, clients must also do the same. This process is known as *discovery*. This has the advantage that neither services or clients need to

be aware of the location of a lookup service in advance. Moreover, this system enables several lookup services to be run independently on a network for fault tolerance purposes.

3) Proxy objects – clients use services through proxy objects which they download from the lookup service. These proxy objects provide the code needed to invoke a particular service. The proxy objects are what the Jini services register with the lookup service. There are several design decisions left to the programmer as to how the proxy object should behave, essentially whether it should be a fat or thin proxy. These design decisions are dependent on the specific service and are discussed in section 2.3.

Furthermore, being built around current Java technology, Jini technology relies heavily on Java's ability to move code from one machine to another to enable a truly de-centralised system. It builds on Java's concurrency class libraries to provide distributed transaction services and although not limited to RMI, it uses RMI to allow services to be invoked between machines.

1.3 Project Goals

This project tries to address the three factors outlined in the introduction as restricting the growth of Jini technology. With this in mind, the project has two distinct aims.

Firstly, to design and build a system to allow Jini services to be controlled over the Internet. This addresses the first factor identified as restricting Jini's proliferation. A further enhancement of this system would be to enable the client to be generic, i.e. allowing a human user to control all Jini services as they become available even though the client knows nothing about them at compile time. This addresses the third factor identified as restricting Jini's proliferation.

Secondly, to investigate how these services can include devices of limited memory capability. This will address the second factor identified as restricting Jini's proliferation.

Therefore the project can be divided into two distinct sections. The first section involves researching, designing and building the generic architecture to control Jini services over the Internet. The second section deals with researching and enabling limited memory capable devices to be run as Jini services.

1.4 Overview of Design

There are two distinct sections of design. Firstly the design of the architecture to control Jini services seamlessly over the Internet and secondly the design to control devices of limited capability. These sections are dealt with consecutively.

Initially it was hoped to be able to control Jini services over the Internet using applets by allowing the applet to download a service's proxy object and directly contact the service's machine. However, on closer examination and much research several problems were identified with this scenario. Firstly, for this scenario to be efficient the client would have to install the core Jini files. Secondly, if the proxy object used RMI to talk back to its service then the applet would have to use RMI over HyperText Transfer Protocol (HTTP). This solution was cited as not working at the time of writing this thesis [Li '00]. Finally, depending on firewall configurations it may be impossible for a client to access a service if that service is behind the firewall.

A "Web Bridging Architecture" was designed to overcome the problems of RMI over HTTP, firewalls and distribution of client files (shown in figure 4.1). This design adds an extra level of indirection to the standard Jini architecture (shown in figure 2.1). This extra level of indirection is undesirable but necessary. In the normal Jini scenario, the Jini client contacts the Jini service's machine directly without any problem. This is because they are both within the LAN. In the modified architecture, the client is outside the LAN on the Internet. Rather than the client directly contacting the Jini service's machine, it contacts a bridge which, which in turn contacts the service's machine and passes on any client requests. A bridging architecture where the bridge acts as the Jini client obviates the need for any Internet clients to install the Jini core files, allowing the bridge to do all the Jini work of discovering and invoking services. Furthermore, a client using a browser on the Internet is unlikely to be able to

access the service's machine if that machine is within a LAN unless firewall restrictions are relaxed. Finally the bridging architecture allows any services which are based on RMI to be effectively invoked since the bridge will be within the LAN and can use RMI directly.

The remaining question was choice of technology to implement the Web Bridging Architecture. An emerging Jini community project [Harrison '01] suggested a solution using the Simple Object Access Protocol (SOAP) and so a SOAP client and bridge was investigated and built. However while successfully controlling Jini devices over the Internet, many of the same problems remained. For example the client, while not needing to install the Jini core files, would still need to install core SOAP files and Extensible Markup Language (XML) parsing files and the service client. Furthermore, using SOAP means that the client cannot use a standard web browser client. These constraints definitely do not allow a generic seamless architecture to control Jini services over the Internet. Therefore a third architecture of a series of servlets using reflection in a modified servlet engine was designed and built. This has the advantage of allowing the clients to access the services without any pre-configuration or installation of files, and using any standard web browser.

The second section of the design involved investigating an architecture to control devices of limited capacity via Jini services, it soon became evident that exactly such an architecture was under construction by the Jini community. This architecture is known as the Surrogate Architecture [Surrogate Spec '01] and is explained in detail in section 3.4.2. There was little point in attempting to design any other architecture as the community process designing the Surrogate Architecture is large, experienced and in the process of adopting this architecture. A device on which to implement the Surrogate Architecture was purchased – namely, A Lego Mindstorms robot. The robot only has 32K of RAM and therefore is totally incapable of installing the Jini core files and seemed a perfect candidate to attempt to implement the Surrogate Architecture.

Unfortunately it became evident that the Mindstorms robot would be unable to implement the Surrogate Architecture without writing and downloading new firmware to the computer in the robot. The reasons for this are discussed in section 3.4.2. Writing and downloading new firmware was beyond the scope of the project,

therefore it was decided to design a service that controlled the device as a Jini service yet unfortunately would not conform to the Surrogate Architecture. A main web site resource for programmers of the Mindstorm system [Nelson 01] indentified a 100% Java API built by Dario Laverde to control the Mindstorm robot. It was decided to design the robot Jini service by utilising this package. In effect, a machine capable of running the Jini core files would have to act as a proxy to the robot, invoking the methods from the pre-written package. In order for this to be done remotely, the service would be built on RMI.

An architecture to allow CORBA services to be accessed from Jini clients was also designed and built. This enables Jini clients to access services written in languages other than Java. Because of its significant memory requirements, Java is not generally used as a programming language for devices of limited memory capacity. Allowing Jini clients to invoke services written in other languages should aid in their ability to access services on limited devices.

1.5 Project Achievements

The majority of the aims outlined in the project goals have been achieved. A system has been designed and built to allow Jini services to be controlled over the Internet. Furthermore, this system provides a generic client where it is not necessary to know anything in advance about the services available, or how to use them. A service controlling a device of limited capacity has also been created. However, although a generic architecture for this was investigated and identified, it was not possible to implement this architecture for reasons discussed in section 3.4.2. An architecture to allow CORBA services to be accessed from Jini clients was also designed and built. This should aid in the ability of Jini clients to access services on limited devices by allowing them to invoke services written in other languages.

1.6 Roadmap

A synopsis of the material covered in the rest of the chapters of this dissertation follows:

In Chapter 2 (Background) a more detailed discussion of the Jini model is presented. In Chapter 3 (State of the Art) other distributed programming technologies are discussed and their relationship with Jini examined. Alternative "spontaneous networking" technologies are also discussed and other current relevant Jini research are examined. Chapter 4 (Design) outlines the architecture to control Jini services over the Internet and explains how reflection is used to obviate the need for client programs to be aware of services' interfaces in order to control them. The identification and design of an architecture used to control devices of limited capacity as Jini services is also presented as is the architecture to enable CORBA services to be accessed from Jini clients.

Chapter 5 (Implementation) explains how the architectures specified in the design were constructed along with various other implementation specific details. It also discusses problems encountered with trying to implement the architecture identified to control devices of limited capacity and the alternative approach taken. Finally in Chapter 6 (Conclusions) the conclusions arrived at by the completion of the project are discussed and possible future work is outlined.

Chapter 2

2 Background – Jini Technology

This chapter describes in detail the primary elements of Sun Microsystems' Jini technology, the core technology on which this thesis is based. The purpose of the chapter is to familiarise the reader with the central concepts of the technology. Following a brief explanation of how to set up a Jini network, writing a Jini service is discussed. This begins with the identification and discussion of three Jini service design choices which have been identified through experience writing Jini services and from synthesis of the literature [Edwards '01] [Newmarch '00]. A more technical discussion on the core elements of the Jini API version 1.1 used to create a Jini service then follows. This is then repeated for a Jini client. These sections include discussion and explanation of the key Jini concepts of "Discovery", "Lookup Services" and "Leasing". The chapter finishes with discussion of the two advanced core Jini concepts – Remote Events and Distributed Transactions.

2.1 Network Setup

The latest version of the Jini technology (version 1.1) may be downloaded from the Sun Microsystems' web-site [Jini technology '01]. The Jini binary files are slightly larger than 3MB. Every machine that is to partake in the Jini network directly must install these files (note - machines using a proxy or bridging architecture as described previously need not install these files). At least one machine on the Jini network must run a "lookup service", or *LUS*. The core Jini download comes with a default implementation of a LUS named Reggie and comes with instructions to start Reggie. In order to run Reggie a basic HTTP server must be run on the same machine in order to export proxy objects to clients. An RMI activation daemon must also be run on the same machine so as the proxy objects can be activated only when they are needed. Thankfully, the Jini download has a simple HTTP server and RMI daemon provided.

2.2 Basic Jini Service Architecture

There are three steps to use a Jini service in a standard Jini service architecture. Firstly, the service must locate and register its service proxy object with a lookup service. To do this the service must run a small HTTP server to export the proxy object. Secondly the client must locate the lookup service and search for the relevant service. It then downloads the proxy object. The final step is when the client uses the proxy object to contact and invoke the service. This interaction can be seen in figure 2.1

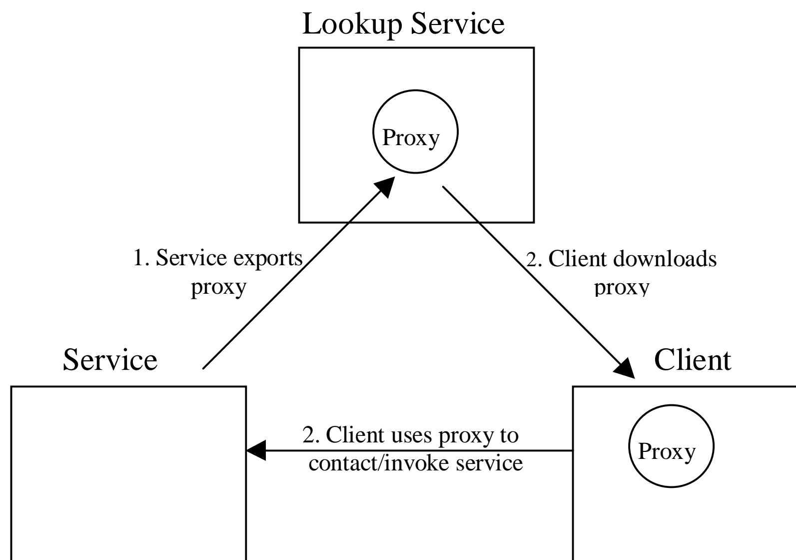

Figure 2.1 – Basic Jini Service Architecture

2.3 Service Design Choices

The scenario outlined previously is the most common Jini service design. However three distinct design choices based on the type of service have been identified from fusion of the literature and experience writing Jini services. Each of the design choices relate to the implementation of the proxy object.

The first choice is where the entire service is run in the client Java virtual machine and thus the client never contacts the service exporter machine. In fact this is just dynamic code downloading and will rarely be used. This “fat proxy” scenario can be seen in figure 2.2.

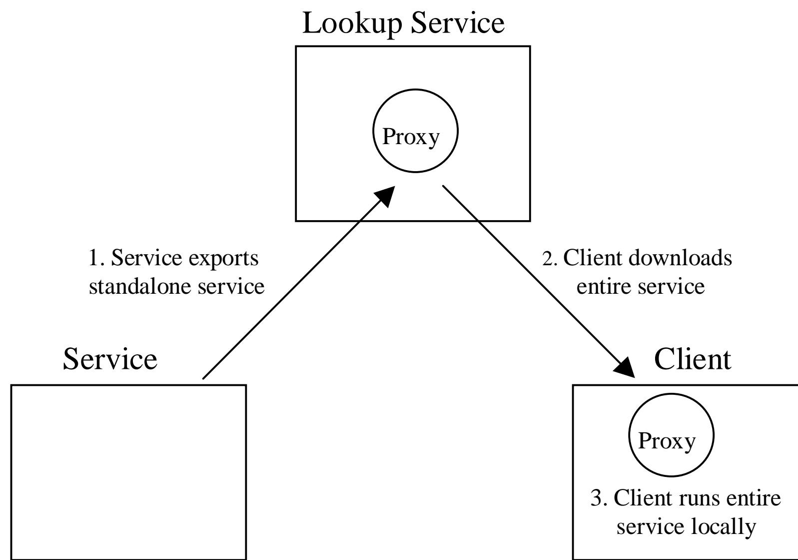

Figure 2.2 – Jini Design Pattern 1

The second choice uses a thin proxy object, which just passes on methods from the client to the server. This scenario is applicable when all the processing must be run on the server machine. A standard way to implement this is using Java RMI where the service exports the RMI stub as the proxy object. This "thin proxy" scenario can be seen in figure 2.3

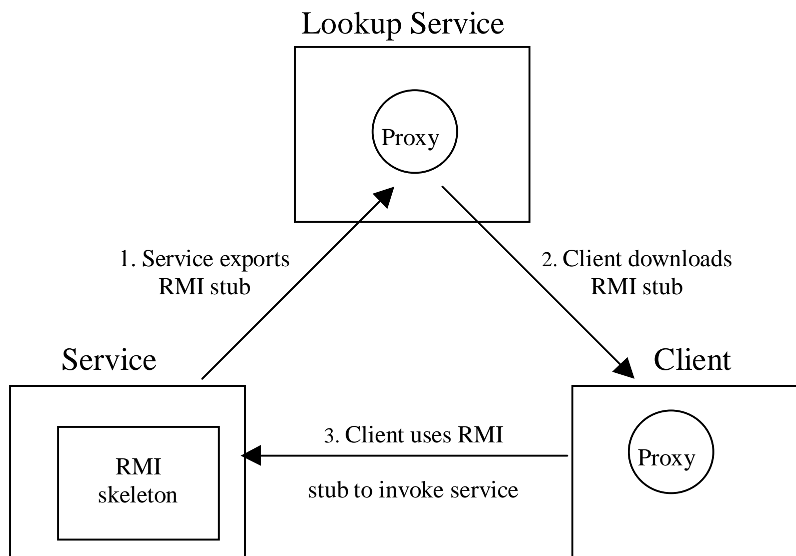

Figure 2.3 – Jini Design Pattern 2

The final scenario is where the processing is divided between both the server and the client. In this scenario the proxy object can contact the server via any protocol such as setting up sockets or using HTTP. However it also does some processing on the client Java virtual machine. This scenario can be seen in figure 2.4.

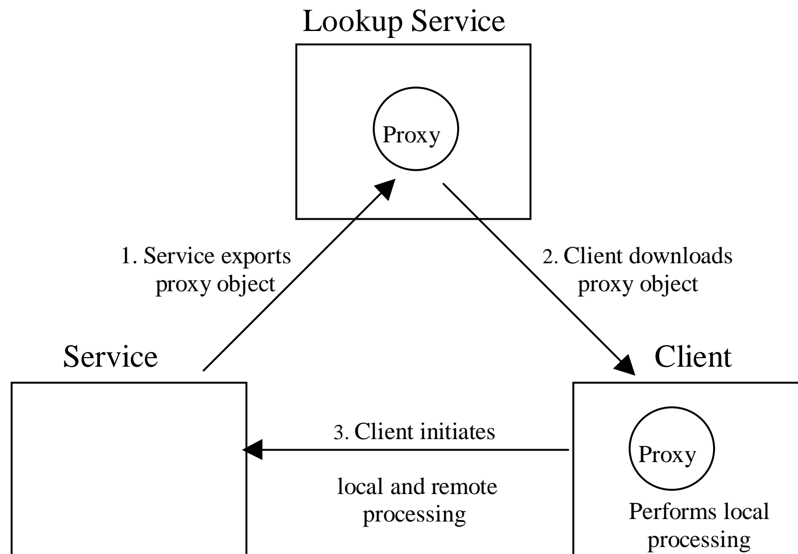

Figure 2.4 – Jini Design Pattern 3

2.4 Jini Service Functions

All Jini services must describe their service, discover one or more lookup services and register their proxy object, which contains their service description, with the lookup services discovered. As of Jini version 1.1 this process has been dramatically simplified by providing a utility class named `LookupDiscoveryManager` to take care of the discovery process and a utility class named `JoinManager` to perform the registration process. With these two classes the majority of the standard Jini work for the service developer is greatly simplified. In fact, the whole process can now occur in a single line of code. A more indepth technical overview of these two utility classes now follows.

We shall firstly examine the `JoinManager` class. The class has two constructors.
The first is used when the service is new and has not been previously registered with a

lookup service. The second is used when the lookup service has registered the service at some earlier time and has returned a service ID to the service.

```
public class JoinManager {
        public JoinManager(Object obj,
                           Entry[ ] attrSets,
                           ServiceIDListener callback,
                           DiscoveryManagement
                                      discoverMgr,
                           LeaseRenewalManager
                                         leaseMgr)
                throws IOException;

        public JoinManager(Object obj,
                           Entry[ ] attrSets,
                           ServiceID serviceID,
                           DiscoveryManagement
                                      discoverMgr,
                           LeaseRenewalManager
                                         leaseMgr)
                throws IOException;
}
```

The first parameter in the constructor is the actual proxy object to be registered. The second parameter is an array of objects that implement the `Entry` interface and are used to describe the service. The third parameter is either the `ServiceID` if it is known or an object that implements the `ServiceIDListener` if it is not known. The fourth parameter is an object that implements the `DiscoveryManagement` interface. In fact the other utility class `LookupDiscoveryManager` actually implements this interface and an object of it will usually be passed in here. The final parameter is a `LeaseRenewalManager` object.

The idea of leases is key to Jini and is the mechanism by which Jini networks become self-healing and need reduced administration. When a service registers its proxy

object with the lookup service it requests an object which implements the `Lease` interface. This is equivalent to a promise from the lookup service to keep the proxy object registered for the duration of the lease object returned. This lease object will have to be renewed by the service before it expires for the proxy object to maintain its registration in the lookup service and for the service to thus still be available. The `LeaseRenewalManager` object handles these issues for the service.

The other utility class mentioned, `LookupDiscoveryManager`, manages all discovery related issues on behalf of either a service or a client. A service or client that wants to contact lookup services whose locations are known to it can use this class' "unicast" facilities (i.e. contact them directly), yet also use whatever lookup services it can find by multicast discovery.

```
public class LookupDiscoveryManager implements
        DiscoveryManagement,
        DiscoveryGroupManagement,
        DiscoveryLocatorManagement {
        public JoinManager(String[ ] groups,
                           LookupLocator[ ] locator,
                           DiscoveryListener listener)
        throws IOException;
}
```

The first parameter is an array of Strings that identify what types of lookup services we want to locate. Some lookup services may be based on organisational departments, others may be open to the public. By passing in the constant `LookupDiscovery.ALL_GROUPS` here we would search for all available lookup services. The second parameter is an array of `LookupLocator` objects. A service or client that wants to contact lookup services whose locations are known to it can use this class' "unicast" facilities by creating `LookupLocater` objects for each known lookup service address. The final parameter is an object that implements the `DiscoveryListener` interface. This object provides methods that determine what to do when a lookup service has been discovered.

The following code fragment illustrates these concepts:

```
try {
      JoinManager joinMgr = new
            JoinManager(proxy, attrSets, this, null,
            null);
      }
catch(IOException ex) {
      ex.printStackTrace();
      }
```

It can be seen that by providing `null` as the final two parameters creates default `LookupDiscoveryManager` and `LeaseRenewalManager` objects respectively. In the above fragment of code `proxy` is the service's proxy object which is created earlier. `attrSets` is an array of objects implementing the Entry interface, describing the service.

2.5 Jini Client Functions

All Jini clients also go through a standard set of steps. They must also discover one or more lookup services, search that lookup service for the type of service they require and only then can they use the service. As with services, the discovery of lookup services can be simplified for clients by using the utility class `LookupDiscoveryManager` previously discussed. A utility class named `ServiceDiscoveryManager` is used to aid client side searching and is to the client developer as the `JoinManager` is to the service developer.

```
public class ServiceDiscoveryManager{
      public ServiceDiscoveryManager
            (DiscoveryManagement DiscoveryMgr,
                  LeaseRenewalManager leaseMgr)
      throws IOException;
}
```

The class has a single constructor that takes an object that implements the `DiscoveryManagement` interface. Again, the object usually passed here is an instance of the `LookupDiscoveryManager` class which will indicate whether the client wants to use multicast searching for lookup services, unicast searching for lookup services or both. The second parameter is an instance of the `LeaseRenewalManager` class.

Once the `ServiceDiscoveryManager` has been created, and lookup services have been located, those lookup services are typically searched by clients for services that they are interested in. The `ServiceDiscoveryManager` class provides four lookup methods to simplify this process.

```
ServiceItem[] lookup(ServiceTemplate tmpl, int
               minMatches, int maxMatches,
               ServiceItemFilter filter, long
               waitDur);

ServiceItem[] lookup(ServiceTemplate tmpl,
                    int maxMatches,
                    ServiceItemFilter filter);

ServiceItem lookup(ServiceTemplate tmpl,
                      ServiceItemFilter filter);

ServiceItem lookup(ServiceTemplate tmpl,
                    ServiceItemFilter filter, long
                    waitDur);
```

In each case the method returns either a `ServiceItem` object or an array of such objects. This object contains the services' proxy object and descriptions about the service. In each case also, the client must pass a `ServiceTemplate` object as a parameter. This object determines what type of services the client wishes to search for. The `ServiceTemplate` object is constructed in a way so as to match some or

all services based on the client's needs. One of the parameters in constructing a `ServiceTemplate` object is an array of `Class` objects of interfaces. Thus only services implementing these interfaces will be found. In this way, the client needs to know of the interfaces in advance. The other parameters relate to the number of matches that should be returned and the amount of time that should be taken to match a service.

The following code fragment illustrates these concepts. Firstly, the `ServiceDiscoveryManager` object is created. If `null` is passed as the two parameters of the `ServiceDiscoveryManager` constructor, default `LookupDiscoveryManager` and `LeaseRenewalManager` objects are created.

```
try{
      ServiceDiscoveryManager sdm = new
            ServiceDiscoveryManager(null,null);
}
catch(IOException ex){
      ex.printStackTrace();
}
```

Then we specify the types of services we are searching for by creating a `ServiceTemplate` object.

```
Class [] types = new
            Class[]{SpecificServiceInterface.class};
ServiceTemplate template = new ServiceTemplate(null,
            types, null);
ServiceItem found = null;
```

Finally we try to find a service implementing the specified interface.

```
try {
      found = sdm.lookup(template, null, 30000);
}
```

```
catch (Exception ex){
      ex.printStackTrace();
}
```

Once a matching proxy object is returned to the `ServiceItem`, it is trivial to invoke the services of the proxy object.

```
if(found!=null){
      SpecificServiceInterface jiniService =
            (SpecificServiceInterface) found.service;
//call the method now
      JiniService.arbitraryMethod();
}
```

2.6 Advanced Jini Concepts

Three main Jini concepts have been discussed so far:

- **Discovery** – both multicast and unicast discovery of lookup services.
- **Using Lookup Services** –
  Services publishing their proxies;
  Clients searching for services
- **Leasing** – used by services to confirm their availability

However there are two further concepts that are not necessary for many Jini services but are still core to Jini technology. These are:

- **Remote Events**
- **Distributed Transactions**

2.6.1 Remote Events

Jini uses events to perform asynchronous notifications of state changes just like Java. However the difference is that the Java event model was designed for delivering

asynchronous notifications within a single Java virtual machine whereas the Jini event model delivers events in a distributed environment. For example, a client may want to know if a service that it is interested in has changed in some way. It is up to the service to deliver events to the client notifying him/her of this change.

Any client who wants to be notified of these changes in services will have to register a listener for this service. This is done via the `RemoteEventListener` interface which has a `notify()` method that services can invoke. The Jini event class is called `RemoteEvent` and all remote events must use or subclass this class.

Remote events raise many problems that local events do not such as order of delivery and partial-failure. Jini leaves the handling of these issues up to the service programmers, who must decide if they need to try to re-send the remote events in the case of partial-failure and determining if clients need to receive the events in the order they were sent.

2.6.2 Distributed Transactions

Transactions are a fundamental concept in distributed systems. They are used to group several distinct operations together, so as they all occur as a single operation. They provide the *ACID* properties to data manipulations:

- **Atomicity** – either all of the transaction's operations succeed, or they all fail.

- **Consistency** – a transaction is a correct transformation of the system state. This means that after the transaction completes, the system should be in a consistent, understandable state.

- **Isolation** – partial effects of one transaction are not visible to other transactions.

- **Durability** – the effects of a committed transaction are permanent.

In practice transactions implement a protocol known as the *two-phase commit* protocol. In this protocol, all participants in a transaction are asked to *prepare* to commit to the transaction in a primary pre-commit phase. They each execute their own individual part of the transaction here and store the results in a temporary manner. If all respond that they are prepared to go ahead, then each participant is told to commit (i.e. execute) in a secondary stage. The commit effectively says that they must each make their temporary data changes permanent. This decision is final and even if a participant should fail before it can commit, it must do so when it restarts. However, if any participant should reply with an *abort* during the primary phase then each participant is told to abort and the whole transaction is thus aborted. Throughout this procedure a central entity is the destination for receiving of abort or commit messages. This entity is the transaction manager. It decides, based on all the messages it receives, as to whether or not the transaction should go ahead.

In Jini an object which implements the `TransactionManager` interface is used to manage the two-phase commit protocol. Sun Microsystems have provided a default transaction manager that implements this interface. It is a Jini service and is named `mahalo`. For an object to be a participant in a transaction it must implement the `TransactionParticipant` interface. This interface provides `abort`, `prepare` and `commit` methods which are invoked by the transaction manager.

The process for creating a transaction is as follows. Firstly a TransactionManager is located using the normal discovery process. Then the programmer creates a `Transaction` object using the `TransactionFactory` class. This `Transaction` object's operations are then defined, i.e. the programmer decides what is to comprise the transaction. The `Transaction` object is then passed to the TransactionManager who executes it to the standards of the two-phase commit protocol. Note that it is left up to the programmer to determine what actions each of the participants must take in their part of the transaction.

Chapter 3

3 State Of The Art

This chapter provides an introduction to technologies and research related to Jini and this dissertation. It is divided into four sections. The first section discusses the distributed programming technologies of RMI, CORBA and Enterprise Java Beans (EJB) along with their relationship to Jini. The second section introduces the new web technology combination of SOAP and Universal Description Discovery and Integration (UDDI). SOAP is a more recent distributed programming technology that concentrates on web services. UDDI is a technology that aims to be the Yellow Pages of web services. The relationship of these technologies to Jini is examined. The third section deals with two alternative technologies which, like Jini, are aimed to "spontaneously network" devices – Universal Plug and Play (UPnP) and Salutation. Finally, a number of fundamental research projects in the area of Jini that are relevant to the work in this dissertation are presented.

3.1 Distributed Programming Technologies

3.1.1 Remote Method Invocation (RMI)

RMI [RMI Spec '00] technology is a *distributed object* technology enabling programmers to write distributed object-oriented applications. Remote Method Invocation (RMI) technology is entirely implemented in the Java language. The technology allows clients to invoke methods on objects which reside on a remote server, as if they were invocations on a local object. A naming service, similar in function to Jini's lookup service, is provided called the rmiregistry. This acts as an information repository storing the location of all the distributed objects. Servers register their objects with this registry and provide an HTTP server whereby clients can obtain remote references for these objects. Remote references are known as client *stubs* and can be located by the clients by contacting the rmiregistry and searching for

the service. The URL locating the remote reference is then passed back to the client who thus obtains the stub. This stub is responsible for the *marshalling*, or gathering and transforming of data from the client into a valid format for transmission. A corresponding *skeleton* resides on the server, which *unmarshalls* data from the client and marshals responses back to the client.

Jini services, especially those with a thin proxy object, are often built using RMI as both technologies are written in the Java language. However, Jini provides spontaneous networking associated with its discovery protocols. It also reveals to the programmer a technology which was used behind the scenes in RMI – leasing. Leasing is the enabling technology for Jini's self-healing mechanisms.

3.1.2 Common Object Request Broker Architecture (CORBA)

CORBA (Common Object Request Broker Architecture) is, like RMI, a *distributed objects* technology. Using CORBA, a client can call a method on an object residing on a remote server, as if it were calling a method on a local object. Like RMI, interaction between the client and the server is specified by an interface. However in CORBA, this interface is written in a language neutral Interface Definition Language (IDL). The infrastructure to handle the actual passing of requests from the client to the server and its responses is hidden from the client. It consists of an Object Request Broker (ORB) residing on the client machine and another one residing on the server. The two ORBs communicate using a standard protocol known as General Inter-ORB Protocol (GIOP). This protocol allows ORBs from different vendors to communicate. The Internet Inter-ORB Protocol (IIOP) is a mapping of the GIOP specification for Transport Control Protocol/Internet Protocol (TCP/IP) networks.

The advantage of CORBA over other distributed technologies is that it is language neutral. CORBA objects may be written in many languages e.g. Java, C and C++. However, unlike RMI and Jini, CORBA objects are not mobile. They can only execute in a single machine whereas RMI and Jini objects and their data may be moved from one machine to another and executed there. There is significant value in allowing CORBA objects to be accessed from Jini clients. CORBA's language

neutrality is a significant advantage in distributed systems, especially in services of limited memory capability. A bridging architecture to enable this is described in section 4.2.2.

3.1.3 Enterprise Java Beans (EJB)

Sun Microsystems has built an infrastructure to allow programmers develop server side Java programs. This infrastructure is called the "Java 2 Enterprise Edition" or J2EE. EJB is one aspect of the J2EE architecture. The relevant technologies and their relationship with each other can be seen in figure 3.1.

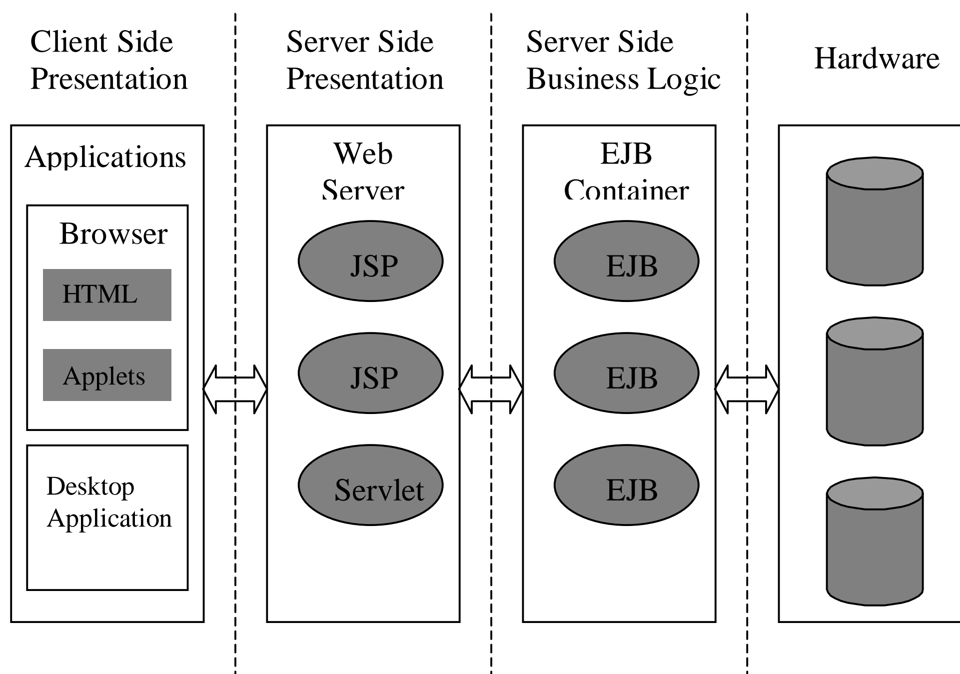

Figure 3.1 – Java 2 Enterprise Edition Technology Overview

EJBs are meant to contain the server side business logic. They do have similarities to Jini however in that EJB is another distributed programming technology, providing the notion of services on the network. EJBs often reside in different areas on the network and can access other network services via RMI over IIOP. Using IIOP adds the promise of CORBA interoperability and thus communication with languages other than Java. EJBs could be exposed as Jini services and thus be dynamically discovered once run. In fact work is being done currently to investigate the integration of these two technologies [EnterpriseWeb '01].

3.2 Web Service Technologies

3.2.1 Simple Object Access Protocol (SOAP)

SOAP is an XML based protocol used to allow distributed software applications to communicate via HTTP [SOAP Spec. '00]. Because it is over HTTP, SOAP messages have the advantage (if you are not a system administrator) of being allowed by firewalls over the standard web server port 80. SOAP essentially allows Remote Procedure Call (RPC) to run over HTTP. The SOAP protocol consists of three parts:

- an envelope describing what is in a message, who and how to process it;
- a serialization mechanism for exchanging application defined datatypes;
- a convention for invoking remote procedure calls and responses.

SOAP is a protocol specification headed by Microsoft and IBM, and submitted to the World Wide Web Consortium (W3C). SOAP is designed to be simple and therefore doesn't contain much of the functionality of other distributed object technologies such as:

- Distributed garbage collection
- Message batching
- Passing objects-by-reference
- Activation

SOAP is a protocol specification and is therefore not tied to any one programming language. There are implementations of the SOAP protocol in many languages including Java, C++, Visual Basic, Perl and Python.

3.2.2 Universal Discription Discovery and Integration (UDDI)

UDDI is a multi-vendor initiative aimed to be a Yellow-Pages type directory of businesses' services on the web. An XML file called the "business registration" is

used to describe a business entity and its web services in UDDI. This XML file consists of three conceptual components [UDDI ‘00]:

- “white pages” which consist of addresses, contact names and numbers etc.
- “yellow pages” which consist of industrial categorisations based on standard taxonomies
- “green pages” which consist of the technical information about the services that are exposed by the business

These business registration XML files are registered in a public UDDI business registry on the web. IBM and Microsoft currently run public UDDI registries on their web sites.

The UDDI business registry can be used to search for businesses in a specific industry category and see what web services they offer. It can also be used to determine the technical details of how that web service is provided in order to invoke the service.

The web services that businesses register in the UDDI registries are normally implemented in SOAP. The interaction of these protocols can be seen in figure 3.2 [UDDI ‘00].

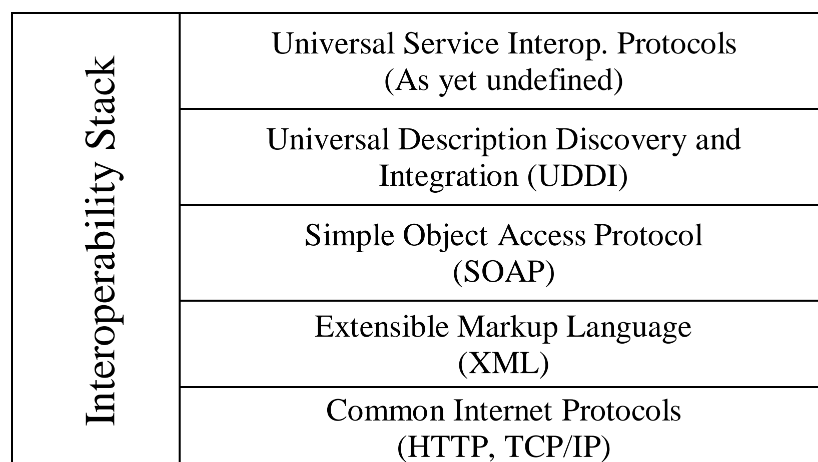

Figure 3.2 – Web Services Protocol Stack

Perhaps not surprisingly, UDDI is an industry initiative led by many of the same companies that are promoting SOAP.

The combination of SOAP and UDDI aims to be a key tool in the future of Business-to-Business technologies and web services. Their relevance to this dissertation is that SOAP and UDDI are possible technologies that could be used to expose Jini LAN based services as services on the Internet, which is the first aim of this dissertation.

3.3 Spontaneous Networking Technologies

When Jini technology was initially released, a lot of the marketing hype explained Jini as a technology that would allow spontaneous networking of home devices. This hype has not been lived up to, largely because of the memory constraints of home devices. As much of the research in this area has shown, Jini is actually a powerful software solution that doesn't have to be device centric at all. However, several other technologies were also created and marketed as solutions for the spontaneous networking of home devices. This section describes two of them – UPnP and Salutation.

3.3.1 Universal Plug and Play (UPnP)

UPnP is a technology from a consortium led by Microsoft [UPnP]. It is seen as networking extension of Microsoft's Plug and Play technology. UPnP concentrates on the TCP/IP protocol stack. It defines additions to the lower levels of the protocol stack, which can be implemented by device manufacturers.

Like Jini, UPnP implements the idea of *discovery*. It uses a protocol known as Simple Service Discovery Protocol (SSDP) that enables devices to announce their presence to the network as well as discover available devices. Note that the emphasis in UPnP is on hardware devices, whereas Jini emphasises services – both software and hardware. Discovery in UPnP can work with a lookup service, like in Jini, or without one. Again like in Jini, discovery is based on the device sending multicast messages announcing its presence.

One innovative aspect of UPnP is *self-configuration*. This allows devices that cannot be allocated IP addresses dynamically, because of a lack of any DHCP server, to use a protocol called AutoIP. AutoIP allocates an IP address for the device from a pre-

assigned range of IP addresses. However these addresses are not valid Internet routable addresses and are used only for the surrounding network of UPnP devices. Therefore if a DHCP server becomes available at any time, it will change the IP address of the device into a valid Internet routable one.

While UPnP has the backing of Microsoft and many device manufacturers, it is a very different technology offering to Jini. It does not offer a comprehensive programming API like Jini and is aimed completely at seamlessly and spontaneously networking devices together whereas Jini is aimed at services, be they hardware or software.

3.3.2 Salutation

Salutation [Salutation] is another discovery and spontaneous networking (known to Salutation as "Find and Bind" networking) technology from a consortium of computer industry companies and academics. It is based on research on intelligent agents. A device talks directly to a Salutation Manager (SLM), which may be in the same device or located remotely. Salutation assumes that this SLM is known to the device in advance. The SLMs then act on behalf of the device, discovering and coordinating with each other using Sun Microsystems' Open Network Computing Remote Procedure Call (ONC RPC).

Salutation and UPnP are aimed specifically at dynamically networking hardware devices. Jini provides far more than just the capability of dynamically networking devices. Essentially it provides the capability of dynamically networking software. Furthermore Jini has developed a strong software development community [Jini Community] due to its extensive API.

3.4 Related Jini Research Projects

The following section describes three research projects of special relevance to this dissertation. The first, "The SOAP-UDDI Project", is a project that addresses the issue of how to access Jini services over the Internet. The second, "The Surrogate Architecture", is a project which addresses how to include devices of limited memory capacity in Jini networks. The third, "The Service User Interface Project", addresses

how Jini clients can invoke Jini services dynamically even if those services are not known in advance. These three projects are especially relevant because they address the three issues that were identified in the beginning of this dissertation as impeding the proliferation of Jini technology.

3.4.1 The SOAP-UDDI Project

In section 3.2.2 SOAP was identified as a possible means of invoking Jini services over the Internet. Furthermore, UDDI was discussed as a sort of Internet Yellow Pages where these Jini services could be registered. In fact, this is exactly the aim of the SOAP-UDDI project [Harrison '01]. This research project is very new, created on the 18th of June 2001. It was the initial impetus to this thesis' investigation of SOAP as a means to accessing Jini services over the Internet. However, as stated previously, using SOAP requires distribution of the SOAP and XML parser files to all the clients for installation. Furthermore, the necessity to parse XML on both the client and server introduces noticeable reductions in operation speed. These inconveniences were found to be major disadvantages to using SOAP to access Jini clients.

3.4.2 The Surrogate Architecture

As stated previously, in order for a hardware or software component to participate in a Jini network, it must be able to run the core Jini files which occupy over 3 MB of memory. In addition it must be able to download and execute classes written in the Java programming language and it may need the ability to export classes written in Java so that they are available for downloading to a remote entity. For devices of limited capacity which cannot meet these criteria, the use of a third party or proxy which will perform these functions for the device is a solution. The Jini Surrogate project [Surrogate Spec. '01] has defined such a proxy architecture called "The Surrogate Architecture". The architecture specification is network and device independent and preserves Jini's concepts of discovery, code downloading, and leasing of distributed resources.

The basic components of the Surrogate Architecture are shown in figure 3.3. The proxy is a machine capable of participating in a Jini network directly. It shall allocate resources for the device in a framework known as the "Surrogate Host". A "Surrogate" is an object that represents, and allows control of, the device. This may be a collection of Java software components and other resources.

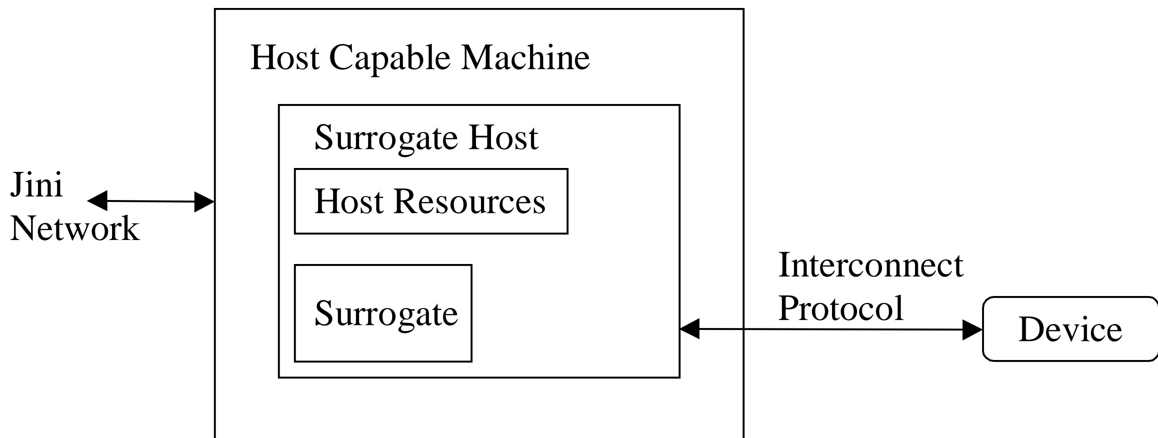

Figure 3.3 – The Surrogate Architecture

The Surrogate must be retrieved by the Host Capable Machine. This operation may be a *push* (the device uploads the surrogate to the surrogate host) or a *pull* (the surrogate host extracts the surrogate from the device or from some third party). The "Interconnect Protocol" is the logical and physical connection between the Surrogate Host and the device. The Interconnect must be specified for each network protocol. This has been done for the Internet Protocol [Interconnect Protocol '01] and it is in the process of being done for the Bluetooth network protocol and for Java smart-cards. It is foreseen that the interconnect for more network protocols shall be specified in the future.

The Surrogate Architecture solves the second problem constraining the proliferation of Jini technology identified in this dissertation – that of integrating device of limited memory capacity into a Jini network. However, for the device of limited memory capacity used in this thesis (a Lego Mindstorms robot) using this architecture was not possible. This was due to the fact that it is impossible to upload a surrogate from the Mindstorms robot without re-writing its firmware, which was beyond the scope of the project. However the Surrogate Architecture is the generic way of solving the problem of integrating devices of limited capacity into a Jini network.

3.4.3 The Service User Interface Project

The Service User Interface project [Venners '01] aims to standardise ways to attach and use any kind of user interface (text-based, graphical, voice, etc.) to Jini services. Note the difference here between the interface that the service implements which provides methods to the client to invoke on that service, and a user interface which may be graphical, voice etc. which allows control of the service. The Service User Interface project is relevant to this dissertation in that it relates to the third problem identified as restricting the proliferation of Jini technology, namely, if a client doesn't know the interface that the Jini service implements then there is no way for them to work together programmatically. However, while this is a problem for an automated client, it may not be a problem for a client used by a human. If an intuitive user interface is provided then the human may still be able to use the service. There are four choices for providing the intelligent user interface.

- The service's proxy not only implements the interface that the service is driven on, but also implements the user interface. The proxy could therefore be a subclass of `JWindow` which could display a complete user interface. However, if the client doesn't have access to the correct user interface class files it won't be able to use the user interface. Furthermore, user interface code becomes entwined with service code, which is undesirable from a design point of view.

- A better approach is that the service implements a separate interface e.g. "`Interfaceable`" which returns the GUI for the client. Clients could specify their type and the methods invoked through the `Interfaceable` interface should return the appropriate user interface. However this prohibits the addition of a user interface by the client after the service has been created.

- Another approach attaches user interfaces as attributes to the service's proxy object, by sub-classing the `Entry` interface. Thus many user interfaces can be provided, one for each client type. However, if the client doesn't have the correct class files to use the user interface it is still a problem. Furthermore, the user interface will have to be serialised to transmit it. This is a serious limitation to the user interface's capabilities.

- A fourth approach attaches not a user interface as an attribute to the service's proxy object, but an object that creates a user interface. This is a subtle but important difference. It provides several key benefits. As with the previous approach it allows clients to attach new user interfaces that the service writer may have omitted. It also allows clients to search for their appropriate user interface. However a major advantage is also that the user interface is instantiated on the client Java virtual machine, rather than on the server one and then serialised. This is a major advantage because instantiating swing components on the server side normally gives them data references that reside on the server machine. When these are serialised and sent to the client, they often won't work once de-serialised on the client.

The Service User Interface project and the Jini community have decided on the fourth approach. It defines a standard way for user interface providers to associate a user interface with a Jini service by providing three items:

- A `UIDescriptor` that describes the user interface, and is attached as an attribute to the service proxy;
- A user interface "factory" that will be instantiated on the client's Java virtual machine to produce the user interface;
- The user interface itself;

The `UIDescriptor` is the most important of these. It contains the following items:

1. A "role", which indicates how the user interface generated by the user interface factory will be used. A user interface returned by the factory object must implement a specified role. For example, two roles currently defined are:

    - `MainUI` – for a service's "main" user interface.
    - `AdminUI` – for a user interface administering the service.

2. A “toolkit”, which defines the necessary packages for the user interface. Two toolkits have been defined to date. One specifying the `javax.swing` package and another specifying the `java.awt` package.

3. A set of attributes that describes the user interface represented by the `UIDescriptor`.

4. A factory, represented as a `MarshalledObject`, which can be used to instantiate the user interface.

A client using the Service User Interface Project’s approach must use its packages in order to invoke the factory to create the appropriate user interface. However, the Service User Interface is an evolving project and at the time of this thesis it was not standardised. Therefore a fifth approach using Java’s “Reflection” capabilities was investigated.

Reflection is the ability to examine the type of an object at run time. This obviates compile-time type requirements, and enables us to call any method on any arbitrary object without having to know that object’s type at compile time [Neward ‘00]. This is an obvious solution to the problem of theJini client requiring the service’s interface in advance.

This approach has the advantage that the client doesn’t need to know of any non-standard Java classes in order to use the Jini service. Furthermore, it doesn’t need to know the interface that the service implements in advance. Moreover, in order to access Jini services via the Internet, this thesis takes the approach of using servlets. HTML is thus the graphical user interface. For these reasons the Reflection approach was used and not the Service User Interface approach. However, once its packages are defined, due to its flexibility, the Service User Interface will undoubtedly become the standard mechanism by which most human controlled clients will access services from a multitude of devices.

Chapter 4

4 Design

As stated earlier, there are two distinct sections of design. The first section deals with the architecture to control Jini services over the Internet, while also addressing the question of how to make the client generic. This in essence tries to solve the first and the third problem identified as impeding the growth of Jini technology – namely, how to access Jini services over the Internet and, how to enable a generic Jini client to control Jini services without any advanced knowledge of them. Thus the first section deals with the design of a generic Jini client to control Jini services over the Internet. The second section of the design deals with the control of devices of limited capability as Jini services. This addresses the second problem identified as impeding the growth of Jini technology – that in order to partake in a Jini network, devices need at least 3 MB of available memory to install the Jini software.

4.1 Control of Jini Services over the Internet

This section is divided into two sub-sections. The first sub-section describes the problems associated with trying to access Jini services over the Internet and provides the design of a web bridging architecture as a solution to these problems. The second sub-section discusses the necessity for a generic client and describes an addition to the web bridging architecture, which uses reflection to create the generic client.

4.1.1 A Web Bridging Architecture

As described in section 2.2, in the normal Jini scenario, when both a client and service are within a LAN, the client downloads a "proxy object" which it can use to control the service. Initially it was hoped to be able to continue to use this normal Jini architecture to control Jini services over the Internet using applets by allowing the applet to download a service's proxy object and directly contact the service's machine.

However, when the client is outside the LAN this same interaction raises problems. Firstly, the client would have to install the core Jini files. Moreover, the use of multicast to discover the lookup service cannot be used. This however can be solved by Jini's unicast lookup facility where the client specifies a known IP address of a lookup service. Unfortunately there are more serious problems. Even if the client knows the IP address of the machine running the lookup service and uses Jini's unicast facilities to try to contact it, there it is unlikely to get through to it if that machine is behind a firewall. If the firewall is configured to allow access to that relevant port on the lookup service machine, then the client will be able to access it and download the proxy object. However, in many cases the service will use the second design pattern specified in section 2.3 in which case the proxy object will then attempt to access another machine on which to run the service. Again we come across the problem of access to this machine if it is behind a firewall. We are now in the nasty position of having to open ports for every machine we run a service on. Even if this was attempted, the communication between the proxy object and the service back-end will often use RMI. RMI dynamically allocates port numbers when using communication between clients and servers and thus trying to give access to these Jini service back-end machines using RMI would be equivalent to giving access to every port on that machine.

One approach to this problem is to try to tunnel RMI over HTTP. This involves encapsulating the client RMI call in an HTTP POST request and unpacking it on the server side. However there is extensive negative experience with this on the RMI-users mailing list. Furthermore, this issue has been raised previously as the Jini community attempts to access Jini services over the Internet. Moreover, at the time of writing, the RMI over HTTP solution was cited as not working [Li '01]. Even if this solution became feasible, the client would still have to install the core Jini technology files in advance.

The solution these this problems is to add an extra level of indirection to the standard Jini architecture. This modified architecture can be seen in figure 4.1.

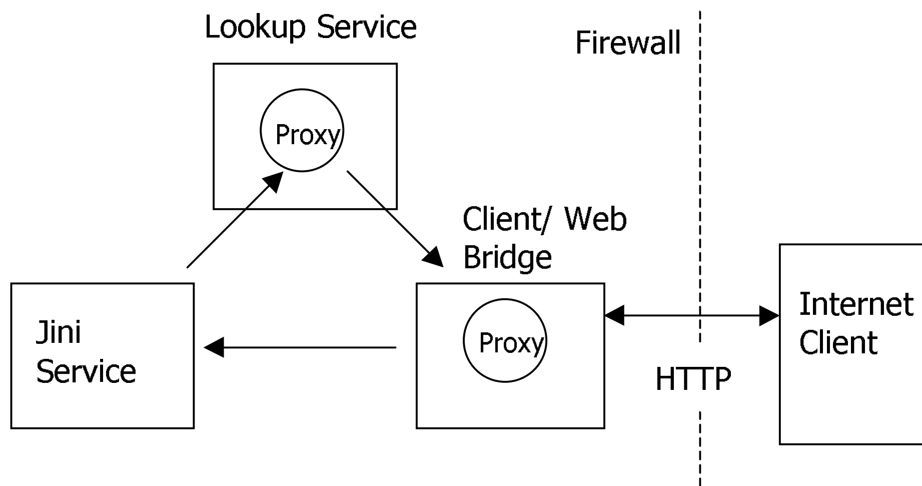

Figure 4.1 – Web Bridging Architecture

The standard Jini client now becomes a web bridge also, allowing a browser client to invoke the Jini client facilities such as discovery, searching of the lookup service and service invocations. While this design isn't as simple and elegant as the standard Jini architecture, it resolves all of the problems discussed earlier. The proxy objects can access the service back-end even if it is via RMI seeing as the Jini client/web bridge is still within the LAN. The main issue left to resolve is the type of "Jini client/web bridge" technology and thus Internet client to choose. Two alternative implementations are discussed in sections 5.1.1 and 5.1.2. While this section has identified an architecture to access Jini services over the Internet, it has not addressed the question of providing a generic client for all services. The following section provides an addition to the web bridging architecture that enables this.

4.1.2 Reflection – A Generic Jini Client

As discussed in section 2.5, in the normal scenario for a Jini client, once the lookup service has been found, it can be searched for services that implement a specific interface. Therefore, if the client is to search for a specific type of service, it must be aware of that service's interface at compile time. A client can however search for all services available implementing any interface by passing null as a parameter in the methods that search the lookup service. However, in order to invoke the methods, we do normally need the interface the service implements at compile time.

In cases where the client is automated, this is not a problem because it doesn't make sense for programmatic clients to invoke a service they know nothing about. However if there is human interaction with the client, it may be possible to provide the user with enough information about the service at run time so as he/she can sensibly invoke the service. This system would allow any new service to be accessed and invoked by the generic client, without updating or modifying the client whatsoever. Although the client knows nothing about the service, it aims to extract enough information about it at run-time, and provide this to the user, so as he/she can make a value judgement as to whether or not it makes sense to access the service.

Java's "Reflection" capabilities allow us to discover the type of an object, the interfaces it implements, and the methods those interfaces define, when we receive the object at run time. These capabilities can be used to augment the web bridging architecture. The four-stage generic client architecture using reflection is shown in figure 4.2.

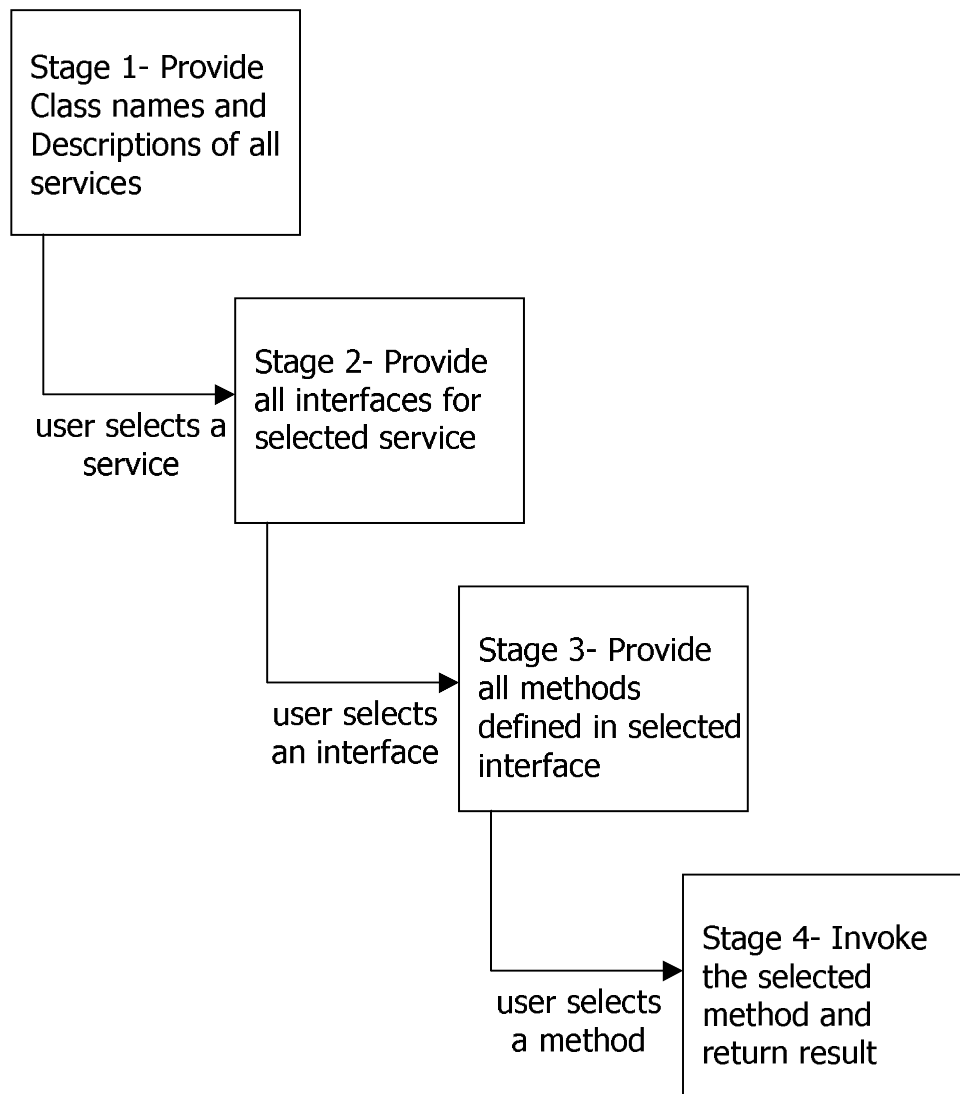

Figure 4.2 – Generic Client Architecture

As the diagram shows, the first stage supplies all discovered services, their class names and a description of each service. The user then selects a specific service that he/she wishes to access. This selection is passed to the second stage, which provides a list of all the interfaces supported by the selected service. The user then selects a specific interface. This selection is passed to the third stage, which provides a list of all the methods defined by that interface. The user then selects a method to invoke. The method is invoked on the underlying service and the final stage shows the method result. Details of the specific reflection capabilities used at each stage are described in section 5.1.3 – Implementing a Generic Jini Client with Reflection.

4.2 Limited Memory Capable Devices as Jini Services

As discussed previously, more than 3 MB of memory is required to run Jini technology natively on a device. For many devices this is not possible. Section 3.4.2 describes an architecture (The Surrogate Architecture) in the process of being defined by the Jini community to allow devices that cannot normally partake in a Jini network because they cannot supply 3 MB of memory, to partake in a Jini network using a proxy architecture. The essence of this architecture is that another machine that can supply the necessary memory will act on behalf of, and communicate directly with, the limited device.

The device of limited memory capability chosen to run as a Jini service and to implement the Surrogate Architecture is the Lego Mindstorms robotic kit [Mindstorms]. This idea of a Lego Mindstorms robot as a Jini service is relatively old. Sun Microsystems showed a series of robots controlled as Jini services from palm pilots at the JavaOne 1999 conference [JavaTanks '99]. Further literature is also available [Jini and Lego '00], [Newmarch '00]. Unfortunately during the implementation of the it became clear that the Mindstorms system cannot implement the Surrogate Architecture for reasons specified in section 3.4.2. However it can still be controlled by the general principal of the Surrogate Architecture – that a machine with the necessary memory can communicate on its behalf within the Jini network and communicate with it directly using a private protocol. The next section discusses the design of this system.

Most devices of limited capability currently do not run Java natively due to the memory requirements of a Java virtual machine. Smaller virtual machines and new smaller editions of the Java programming language are aimed to change this situation, but for the moment many limited memory devices will continue to run other less memory intensive languages.

CORBA discussed in section 3.1.2, is a distributed object technology that is language independent. It allows a client written in one language to invoke methods in an object written in another language on a remote machine. If Jini clients could use CORBA as

a means to invoke services written in other languages it could potentially give them access to services written for limited memory devices. The final section discusses the design of a Jini service that allows Jini clients to access CORBA services.

4.2.1 A Thin Proxy Architecture to Control A Lego Robot

Three possible Jini service design patterns are discussed in section 2.3. The second "thin proxy" pattern is applicable when most of the Jini service processing happens on the server and the client just fires up this processing. As discussed in section 2.3, this is often implemented using RMI where the client invokes methods on the RMI stub, which are seamlessly passed on to the RMI skeleton on the server for processing.

In the case of controlling the Lego Mindstorms robot, an infra-red transmitter tower may be attached to a computer via the serial port. Commands may then be sent from this computer over the transmitter tower to the Lego robot. In essence, that computer acts as a base-station for the robot. If we are to provide a computer program to control the Lego robot as a Jini service, it is clear that most of the processing will occur on this base-station computer. Therefore, the "thin proxy" service design pattern described in section 2.3 is applicable. This architecture can be seen in figure 4.3.

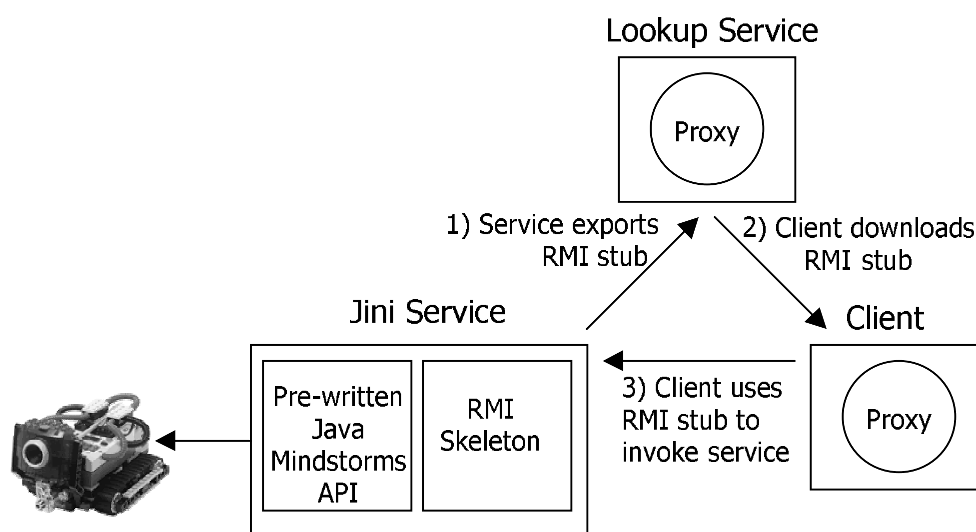

Figure 4.3 – Thin Proxy Architecture to Control Lego Mindstorms Robot

The client invokes a method on the RMI stub which seamlessly invokes the method on the server. The RMI skeleton uses a pre-written Java package [Laverde '99] to send commands to the serial port and control the robot.

4.2.2 Accessing CORBA Services from Jini Clients

When invoking any service in another technology from a Jini client, a Jini service that is a client to the other technology is written. We must however decide on the applicable service design pattern, i.e. where the processing must occur – on the client, the server or both (see section 2.3). In the case of accessing a CORBA service, the third design pattern can be used. Processing is both remote and local. Figure 4.3 shows this architecture.

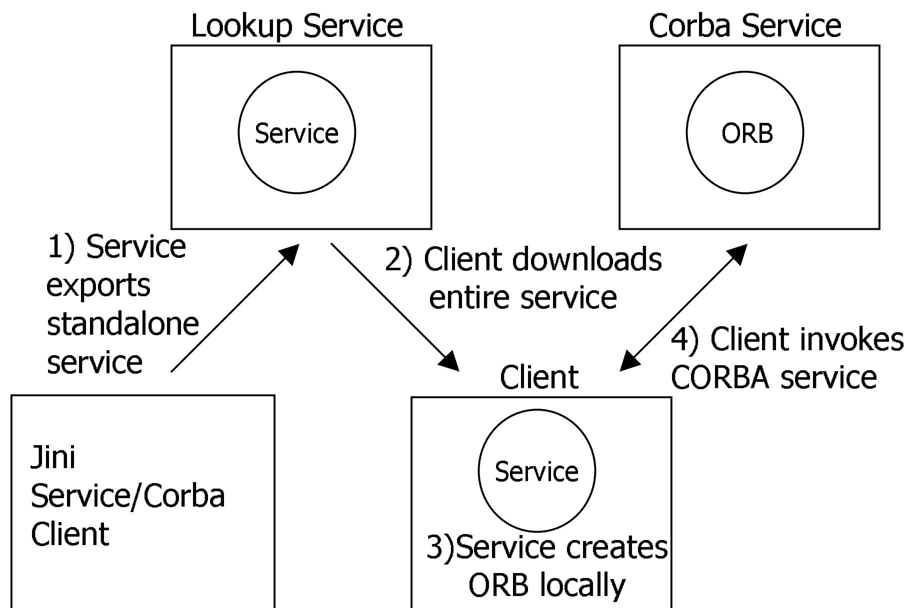

Figure 4.4 – Jini CORBA Architecture

The Jini service is initially exported to the lookup service. Once downloaded by the client the service creates an ORB locally and then contacts the CORBA service.

This chapter has provided a design to control Jini services over the Internet and an augmentation to this design to provide a generic client for these Jini services. It has also provided discussion and designs to enable devices of limited memory capacity to partake within Jini networks. The next chapter deals with the implementation of these designs.

Chapter 5

5 Implementation

This chapter is again divided into two distinct sections. The first section deals with the implementation of the web bridging architecture described in section 4.1.1 used to control Jini services over the Internet. It describes how to implement the web bridging architecture using SOAP and using Java servlets. It also describes the aspects of Java's reflection capabilities that allow us to create a generic client. The second section deals with implementing limited memory devices as Jini services. It describes the implementation of the Lego Mindstorms robot as a Jini service, designed in section 4.2.1 and then the implementation of the Jini/CORBA bridge service outlined in section 4.2.2.

5.1 Implementing the Web Bridging Architecture

This section describes issues encountered while implementing the web bridging architecture described in section 4.1.1 and in figure 4.1. The key implementation issue here was what technology to use to expose Jini services to clients over the Internet. Both a SOAP and a Java servlets version were implemented.

5.1.1 A SOAP Jini Client/Web Bridge

The SOAP-UDDI Jini research project discussed in section 3.4.1 suggests using the same web bridging architecture identified in the section 4.1.1 where the Jini client/web bridge is a Jini client whose methods are exposed as a SOAP service.

SOAP is an XML based distributed programming specification. Implementations of this specification are available in many languages. Since Jini services and clients are built in the Java programming language, an implementation of SOAP in Java is necessary in order to expose the Jini services as SOAP services that can be accessed by SOAP clients over the Internet. Apache-SOAP is one such implementation and can

be downloaded from the Apache web site[1]. Apache SOAP can be used both as a client-side library to invoke SOAP services available remotely, or as a server-side tool to expose SOAP services. The configuration of the client and server are both documented with the Apache-SOAP download. From the server's point of view, among other things, it needs a web application server that supports servlets and JSPs (Java Server Pages). The Apache-Tomcat servlet engine was used for this purpose[2]. The servlet engine, and the server-side Apache-SOAP installation resides on the client/web bridge of the modified Jini web bridging architecture in figure 4.1 in section 4.1.1.

In order to act as a Jini client and contact any Jini services on the LAN, three modifications must be made to the Tomcat servlet engine. Firstly, the Tomcat policy file must allow access to the rest of the network. Secondly the core Jini technology files must be included in Tomcat's class library. Finally, Tomcat's startup script must be modified to include all the Apache-SOAP files. Once we set up the servlet engine configured for Apache-SOAP, we can register a Jini client as a SOAP service. The process of registering a SOAP service is well documented in the Apache-SOAP download.

The Jini client, once registered as a SOAP service, acted as a means to control the Lego Mindstorms robot via a SOAP client over the Internet. Once the Jini client is initially invoked by a SOAP client, the network is searched for a service implementing a `RobotInterface`. This is the name of the interface that the Lego robot service implements. Once a service has been found, the SOAP client can move the robot "forwards", "backwards", "left" and "right" or tell the robot to "stop". These five commands are the names of five methods that have been "exposed" as methods that can be invoked by any SOAP client over the Internet. The Jini service that is located actually returns an RMI stub to the "Jini client/SOAP service". Thus any method invoked by the SOAP client on the Jini client/SOAP service actually invokes the equivalent method on the RMI skeleton residing on the Lego robot's controlling computer.

[1] HTTP://xml.apache.org/soap/index.html
[2] HTTP://jakarta.apache.org/tomcat/index.html

The SOAP client is also a Java class, which uses the Apache-SOAP implementation to create XML-based SOAP messages. The client acts in the following manner. It must first be given the location of the SOAP service. This is a two-part operation. Firstly, the location of the Apache-SOAP server-tool used to expose the SOAP services is hard coded into the client as a Java URL object.

```
URL url = new URL
     ("HTTP://wilde.cs.tcd.ie:8080/soap/servlet/
          rpcrouter");
```

Within the Apache-SOAP server, our specific service is given a Uniform Resource Name (URN). This must be supplied to the SOAP client also. The following code snippet does this.

```
String urn = "urn:Client";
```

The SOAP message can then be created and the specific method it attempts to invoke can be passed to it. In the code example below we are trying to invoke the "forwards" method.

```
Call call = new Call();
call.setTargetObjectURI(urn);
call.setMethodName("forwards");
```

The client can then attempt to make the call.

```
try {
     Response response = call.invoke(url, "");
}
```

Using SOAP, and the modified Jini web bridging architecture as the means of invoking Jini services avoids the problems of firewalls and of needing to install the core Jini technology files on the client. However in order to invoke Jini services via SOAP over the Internet, the Internet client (or SOAP client) must still install the

SOAP implementation files as well as an XML parser (Apache-Xerces was used in this case[3]). Moreover, the SOAP client is invoked using a command line interface rather than via a browser. Finally, the speed of a SOAP client-server interaction is quite slow. This is due to the fact that XML has to be parsed both on the client and on the server machines.

Only one Jini client/SOAP service was created due to the previously mentioned problems with SOAP. No attempt to make the Jini client/SOAP service work for all services using Java's reflection capabilities was made. The next section looks at a faster alternative technology, which obviates the need to distribute any files to the Internet client.

5.1.2 A Servlet Jini Client/Web Bridge

The advantages of using a series of Java servlets as the Jini "client/web bridge" are manifold. Firstly, the speed advantages of servlets over SOAP is obtained, not only because of the necessity of parsing XML at both the SOAP client and server, but also because of the multi-threaded capabilities of servlets. Secondly, the ability to maintain state between servlets is something that is impossible with many implementations of server-side SOAP tools. Furthermore, the obvious advantage of servlets over SOAP in this situation is that the Internet client can be a simple HTML page thus not needing any distribution or installation of client side files. Finally, using servlets allows easy, clean use of Java's reflection capabilities to develop a generic Jini client that can control all Jini services without any advanced knowledge of them.
While these capabilities can be used with a SOAP implementation in Java, the clean well defined model for handling data transfer between servlets simplifies this process.

The Apache-Tomcat servlet engine was again used as an application on which to deploy and run the servlets. As in the previous SOAP service scenario, Tomcat is located on the client/web bridge machine in the web bridging architecture of section 4.1.1 and figure 4.1. In order to partake in the Jini network, i.e. discover any lookup

[3] HTTP://xml.apache.org/xerces-j/index.html

services and download and run any Jini service's proxy objects, Tomcat must be modified as described in the previous section.

The following section discusses the use of reflection in the servlet web bridging architecture to create a generic Jini client, which with human interaction, is capable of controlling all Jini services without modification.

5.1.3 Implementing A Generic Jini Client with Reflection

As described previously, Java's "Reflection" capabilities allow us to discover the type of an object, the interfaces it implements, and the methods those interfaces define, when we receive the object at run time. These capabilities can be used to augment the web bridging architecture to create a generic Jini client. This Jini client should be able to access all Jini services, even ones that are created after the client has been written.

The four-stage addition to the web bridging architecture used to create the generic client is described in section 4.1.3 and figure 4.2. In fact, these four stages can be cleanly implemented by providing four servlets which can pass the user's choice at each stage to the nextservlet.

The first servlet once initiated, begins a search of the LAN for any lookup services and downloads all found services' proxy objects. The Jini utility classes `LookupDiscoveryManager` and `ServiceDiscoveryManager` discussed in section 2.5 are heavily used to perform these functions. These objects are initially downloaded as generic Java `object` instantiations. Using reflection we provide the user with the class name of every proxy object found. The Java `"Class"` class, which represents instances of Java classes running in an application, provides the reflection capabilities to implement this.

```
Class cl = unknownService.getClass();
cl.getName();
```

In the first stage of the generic client, the user selects which service he/she would like to access. This selection is passed onto the second servlet. The second servlet then displays the interfaces implemented by the selected class. Again the Java "Class" class can be used to discover these interfaces at run-time using the following method

```
Class[] interfaces = cl.getInterfaces();
```

The user then selects an interface and that selection is passed to the third servlet. This servlet shows all the methods defined by the selected interface. Again the Java "Class" class provides the reflection capabilities to enable this:

```
Method[] methods = selectedInterface.getMethods();
```

Where selectedInterface is an instantiation of the Java "Class" class. The user selects the method that he/she would like to invoke and this is passed to the final servlet. The method is then invoked on the underlying service, i.e. the service that implements the selected interface. This is done using the Java Method class

```
try {
     ReturnValue = selected.invoke(unknownService,
          null);
 }
```

where selected is a Method object and unknownService is the underlying service which is a Java object. The HTML pages returned to the user at each stage can be seen in Appendix 1.

5.2 Implementing Limited Memory Devices as Jini Services

This section is divided into two sub-sections. The first describes the implementation of the Lego Mindstorms robot as a Jini service and issues that arose during this process. The second sub-section describes the implementation of the Jini/CORBA service and associated issues.

5.2.1 Implementing a Lego Robot Jini Service

The thin proxy architecture defined in section 4.2.1 is implemented with the aid of RMI. Firstly, the interface which the service implements must extend `java.rmi.Remote` in order for the service to be invoked remotely. The service named `RobotImpl` implements this interface and extends `java.rmi.UnicastRemoteObject`. The service is then compiled using the Java RMI compiler to give the client stub and the server skeleton.

The service provides five methods: "forwards", "backwards", "left", "right" and "stop". Each of these methods creates an array of bytes and passes these to the serial port. Dario Laverde's Java package [Laverde '99] written specifically for communication with the Lego Mindstorms robot tower is used to parse string commands into these bytes arrays to be sent to the tower. In order to send messages over the serial port the Java Communications extension package must be downloaded from Sun Microsystems[4]. This is added to the Java run time environment of the machine controlling the Lego robot.

Once the service has been written, another Java class is used to add descriptions of the service, discover any lookup services on the network and register the service with these lookup services. It is also responsible for renewing the service's lease with the lookup service. The server side Jini utility classes `LookupDiscoveryManager` and `JoinManager`, which were discussed in section 2.4 – Jini Service Functions, are the main classes used in this process.

Because the service extends `UnicastRemoteObject`, when the service is being registered with the lookup service, what actually gets registered is the client stub class. RMI does this at run time.

[4] HTTP://java.sun.com/products/javacomm/

5.2.2 Implementing a Jini/CORBA Service

The CORBA server to be accessed by the Jini client is based on the simple Hello World CORBA service described in the Sun Microsystems' Java tutorial[5]. For a brief discussion on CORBA see section 3.1.2. A simple Interface Definition Language (IDL) file was written and compiled using the *idlj* compiler that comes with the download of the Java 2 SDK version 1.3 and later. The IDL file declares a module and a single interface. A method returning a string is contained within the interface. Once compiled, the module translates into a java package. This package contains Java versions of the IDL interface and methods. It also creates the server skeleton which implements the Java interface and the client stub. Auxiliary CORBA specific Java classes are also created.

The CORBA server is implemented in Java also. This is purely for convenience. Once the architecture works, it should be relatively easy to implement a server in another language, or hook up to existing CORBA services. A CORBA server manages one or more "servant" objects which actually implement the service, i.e. provide the interface's methods. The server is also responsible for creating an ORB and connecting the servant objects with it. It is also responsible for publishing the servant's Interoperable Object Reference (IOR). This is a reference to the servant object that is transparent between different ORBs. It contains the version of the transport protocol (GIOP or IIOP for TCP/IP networks) that the server supports, the IP address and port number of the server and a key used to identify the servant object by the server. The server must publish this so as the client can invoke the servant's methods. The server can publish it in two ways, by using a CORBA name-service or by physically transferring the file to the client. The server implemented used the name-service that comes with the Java 2 SDK named *tnameserv*. The server then waits for client requests through the ORB and on receiving them passes them to the servant objects.

[5] HTTP://web2.java.sun.com/docs/books/tutorialidl/index.html

In order to access this CORBA server from a Jini client we must build a Jini service, which is also a CORBA client. This Jini service, named `JiniCorbaService`, performs processing both locally (e.g. it must create an ORB locally) and remotely (e.g. it must contact the naming service and invoke a servant object's methods). The Jini service implements an interface named `JiniCorbaInterface` that defines a method that returns a string. The implementation of this method by the "Jini service/CORBA client", invokes the remote CORBA servant's method. The Jini service is registered with a lookup service using the utility classes `LookupDiscoverManager` and `JoinManager` discussed in section 2.4 – Jini Service Functions.

The first section of this chapter dealt with the implementation of the web bridging architecture described in section 4.1.1 used to control Jini services over the Internet. Implementations in two technologies, SOAP and Java Servlets, were discussed. Java's reflection capabilities used to create a generic client were also discussed. The second section of the chapter discussed implementing limited memory devices as Jini services, specifically in the context of a Lego Mindstorms robot and access to CORBA services. The next section outlines the conclusions reached, work accomplished and goals achieved during the course of this project and discusses possible further work.

Chapter 6

6 Conclusions

This chapter describes conclusions reached following the work done during the course of this thesis. The first section re-iterates the initial project goals and describes the work done to try to achieve these goals. It also gives an evaluation of if, and how, those goals were met. The second section describes possible future work and improvements.

6.1 Goals Achieved and Completed Work

Three distinct goals were outlined at the beginning of this dissertation:

1. This thesis aims to address the issue of Jini services being restricted to a LAN environment.

This was achieved by designing and describing a generic architecture to allow Jini services to be accessed and controlled over the Internet – section 4.1.1 "A Web Bridging Architecture". This has been implemented using two technologies – Java servlets and SOAP. A good understanding of both technologies was gained in order to implement these architectures. Both technologies were used in conjunction with the Tomcat version 3.2 servlet engine. Tomcat was run on Solaris so a basic knowledge of Unix was also learned. A thorough knowledge of the functions of Jini clients, especially using the Jini 1.1 API was also learned. This includes issues of discovering and searching of lookup services, as well as invoking specific services.

2. This thesis aims to investigate how Jini services can include devices of limited memory capability.

A generic solution to this problem was identified and researched – that of the Surrogate Architecture described in section 3.4.2. A device of limited memory

capability was acquired and built to test this architecture – a Lego Mindstorms robot. Unfortunately during the attempted implementation, for reasons explained in section 3.4.2, it became obvious that to implement the Surrogate Architecture using this device, the existing firmware would have to be replaced. This was beyond the scope of this project. However, a "thin-proxy" service to control the Lego robot was designed (section 4.2.1 – "A Thin Proxy Architecture to Control A Lego Robot") and implemented using RMI, which utilised the general principles of the Surrogate Architecture. A good understanding of RMI was gained during this process as well as a basic knowledge of programming a limited device via the serial port.

CORBA was identified as a key technology, which if bridged to Jini could allow many more services, perhaps on devices of limited memory capability, to be accessed from Jini clients. An architecture to enable this bridge was designed and described in section 4.2.2 – "Accessing CORBA Services from Jini Clients". A basic CORBA server that returns a string was built in Java. A Jini service that acts as a CORBA client was also built to allow access from a Jini client to the basic CORBA server. A good knowledge of CORBA was achieved in order to implement this architecture.

A thorough understanding of the server side issues of a Jini network was learned at this stage of the thesis. This includes issues of discovery, registering and leasing a Jini service with a lookup service.

3. This thesis aims to provide an architecture by which clients can control Jini services without any previous knowledge of them, thus providing a single generic Jini client to all services.

A project which goes some way to addressing this issue was identified and discussed in section 3.4.3 – The Service User Interface Project. This project is still in its early stages and at the time of this thesis, had not become standardised. Furthermore, the approach taken by the Service User Interface Project was that the client should use an additional Service User Interface Project package. For these reasons, a different approach using Java's "Reflection" capabilities was investigated. A generic client architecture using Reflection was designed and described in section 4.1.3 – "Reflection – A Generic Jini Client". This architecture was successfully implemented

using Java servlets and reflection. A good knowledge of Java's reflection capabilities was gained through this implementation.

A very good knowledge of designing and implementing distributed systems was gained throughout this project. The first and third goals outlined were achieved completely. During the implementation it became apparent that it was impossible to achieve the second goal generically. However, a limited memory capable device was successfully controlled as a Jini service. A Jini service was also built that allows Jini clients to access CORBA services. This may also simplify access from Jini clients to devices of limited capacity.

6.2 Future Work

Future work on this project could include the following:

- Register the SOAP service with a UDDI registry – the Jini client/SOAP service could be registered with one of the public UDDI registries operated by Microsoft and IBM.

- If the Service User Interface project becomes standardised as a part of the Jini technology, look at using this as an alternative to Java reflection in the generic Jini servlet client.

- Augment the Jini servlet client to be updated of remote events occurring on services.

- Investigate rewriting the Lego Mindstorms robot firmware in order to implement the Surrogate Architecture.

7 Bibliography

[Ayers ‘99] Professional Java Server Programming, Danny Ayers et al., Wrox Press, 1999.

[Coulouris ‘01] Distributed Systems Concepts and Design, George Coulouris, Jean Dollimore, Tim Kindberg, Addison-Wesley, 2001.

[Deutsch] The Seven Fallacies of Distributed Computing, Peter Deutsch

[Edwards ‘01] Core Jini Second Edition, W. Keith Edwards, Prentice Hall, Upper Saddle River, NJ, USA, 2001

[EnterpriseWeb ‘01] EnterpriseWeb Project, HTTP://developer.jini.org/exchange/projects/enterpriseweb, Dr. Teddy Achacoso, Bill Venners. July 2001.

[Freeman ‘01] JavaSpaces Principles, Patterns, and Practice, Eric Freeman, Suzanne Hupfer, Ken Arnold, Sun Microsystems, June 1999.

[Harrison ‘01] Dave Harrison, HTTP://developer.jini.org/exchange/projects/soapuddi/, June 2001

[IP Interconnect ‘01] Jini Technology IP Interconnect Specification, Sun Microsystems, HTTP://www.jini.org/standards/sa-ip.pdf , August 2001.

[JavaTanks ‘99] A Demonstration of JiniTM Technology and the K Virtual Machine, Sun MicroSystems, HTTP://developer.java.sun.com/developer/technicalArticles/jini/JavaTanks/Javatanks.html, September 1999.

[Jini Spec ‘99] The Jini Specification, Ken Arnold, Bryan O’Sullivan, Robert W. Scheifler, Jim Waldo, Ann Wollrath, Prentice Hall, July 1999

[Jini technology ‘01] HTTP://www.sun.com/jini

[Jini Community] HTTP://developer.jini.org

[Jini and Lego ‘00] Programming Lego Mindstorms Robots Using the Java Communications API and Jini Connection Technology, Dario Laverde, HTTP://servlet.java.sun.com/javaone/javaone2000/pdfs/TS-1326.pdf, 2000.

[Laverde ‘99] RCX Java API, Dario Laverde, HTTP://www.crynwr.com/lego-robotics/, 1999.

[Li ‘00] Professional Jini, Sing Li, Wrox Press Inc, August 2000.

[Mindstorms] Robotics Invention System 2.0, Lego, HTTP://mindstorms.lego.com/products/ris/index.asp

[Nelson ‘01] HTTP://www.crynwr.com/lego-robotics

[Neward ‘00] Java Server-Based Programming, Ted Neward, Manning Publication Co., 2000.

[Newmarch ’00] A Programmer’s Guide to Jini Technologies, Jan Newmarch, APress, November 2000.

[RMI Spec ‘99] Java Remote Method Invocation Specification, Revision 1.7, Java 2 SDK, Standard Edition, v1.3.0, December 1999.

[Salutation] HTTP://www.salutation.org

[SOAP Spec. '00] Simple Object Access Protocol version 1.1, HTTP://www.w3.org/TR/SOAP/ , Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, Dave Winer, May 2000

[Surrogate Spec '01] The Jini Technology Surrogate Architecture Specification, version 1.0 draft standard, Sun Microsystems, HTTP://developer.jini.org/exchange/projects/surrogate/sa.pdf, July 2001

[Tanenbaum '95] Distributed Operation Systems, Andrew S. Tanenbaum, Prentice Hall, 1995

[Tannenbaum '96] Computer Networks (3rd Edition), Andrew Tannenbaum, Prentice Hall, pp. 412-448

[UDDI '00] UDDI Technical White Paper, Ariba Inc., Microsoft Corp., HTTP://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf , September 2000.

[UPnP] HTTP://www.upnp.org

[Venners '01] The Service User Interface Project, Bill Venners, HTTP://developer.jini.org/exchange/projects/serviceui/, 2001

[Vogel '98] Java Programming with CORBA, Second Edition, Andreas Vogel, Keith Duddy, Wiley Press, 1998.

8 Appendix 1

8.1 Jini Web Services System

The following screen shots show the different stages in the servlet implementation of the generic client to control Jini services over the Internet using reflection.

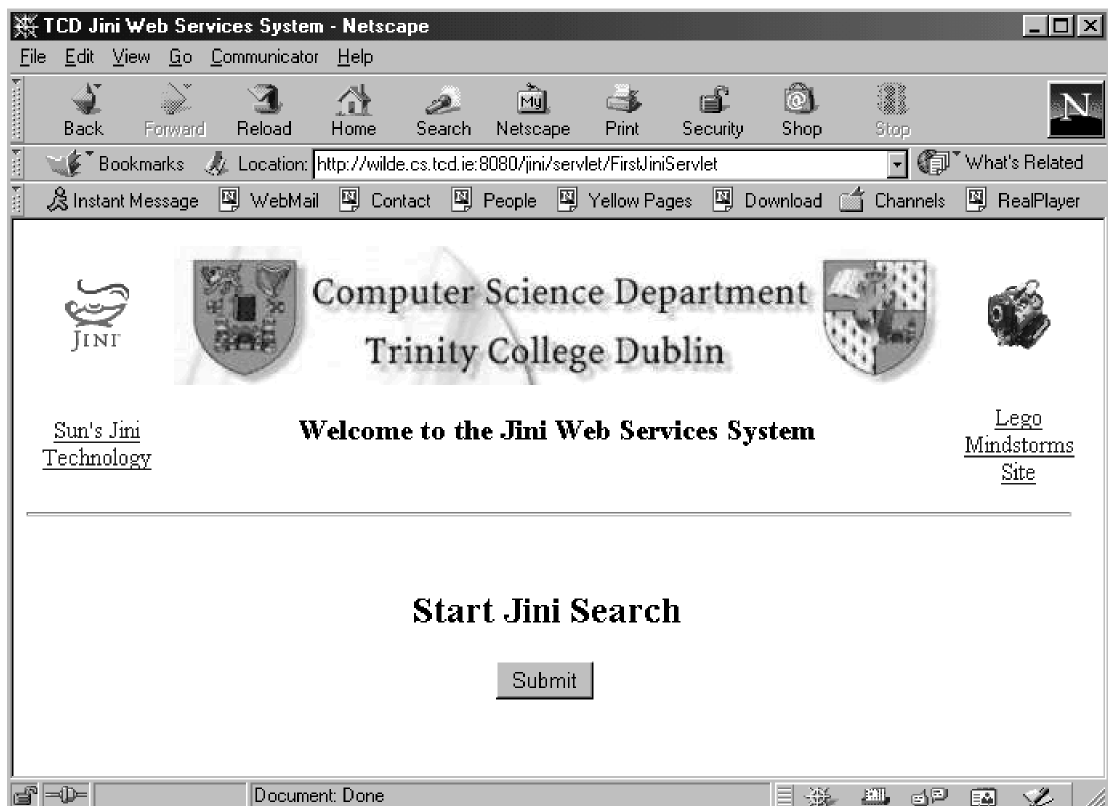

Figure 8.1 – Jini Web Services System Start

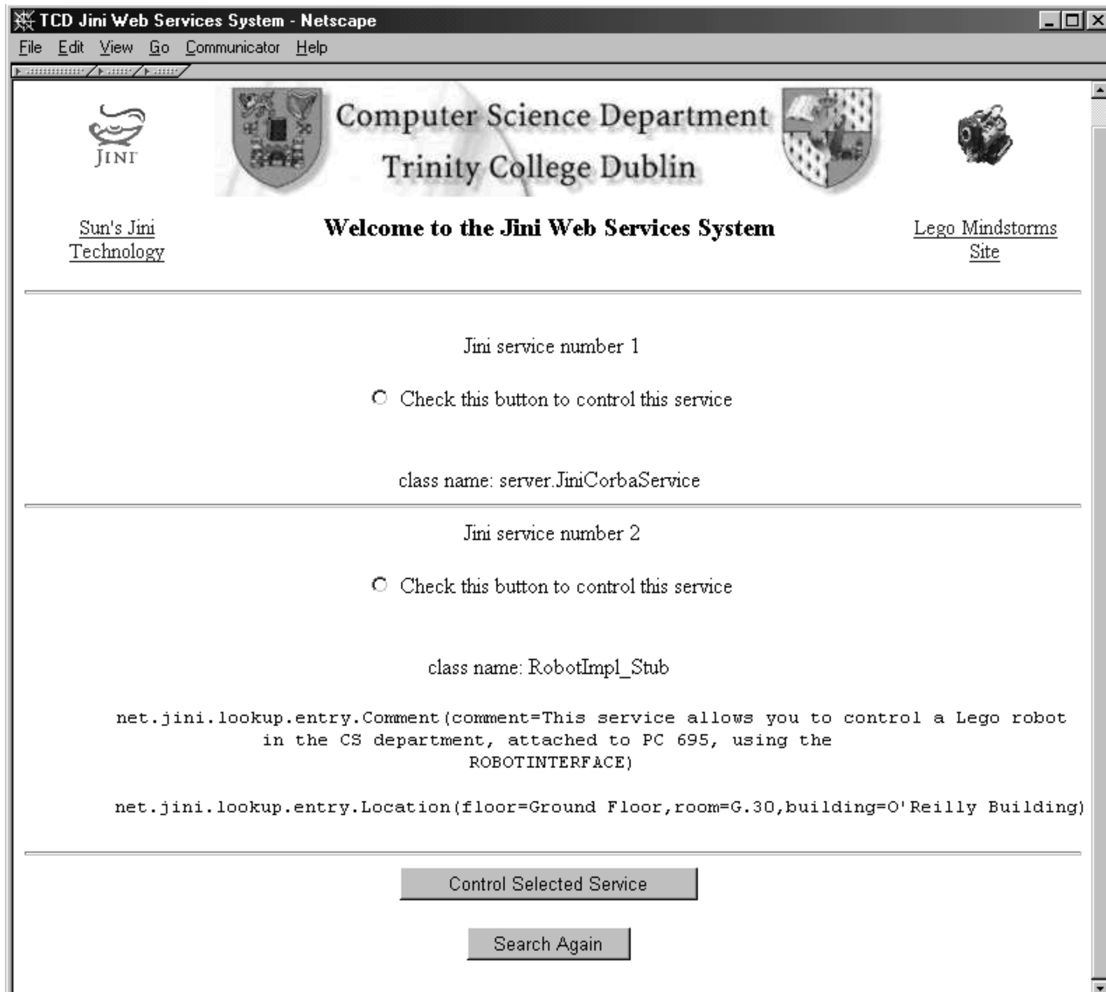

Figure 8.2 – Stage 1: The Services are Discovered and Listed

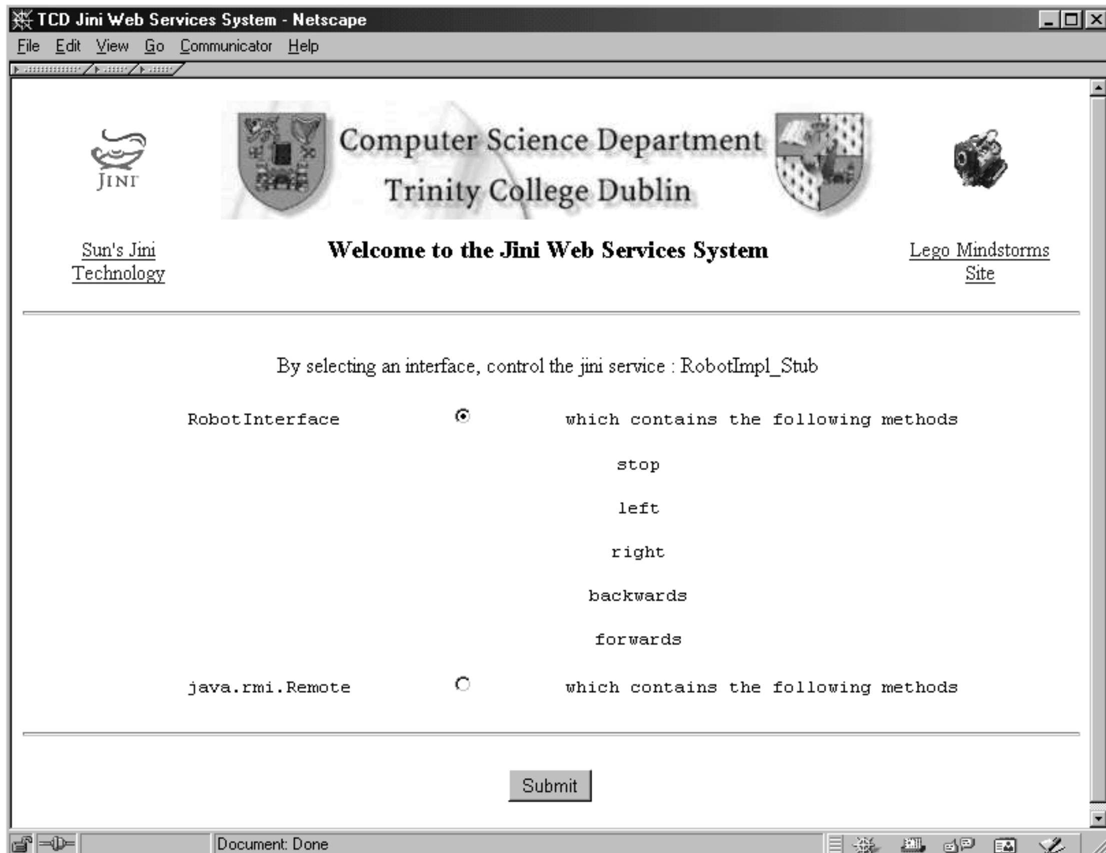

Figure 8.3 – Stage 2: The Interfaces are Listed

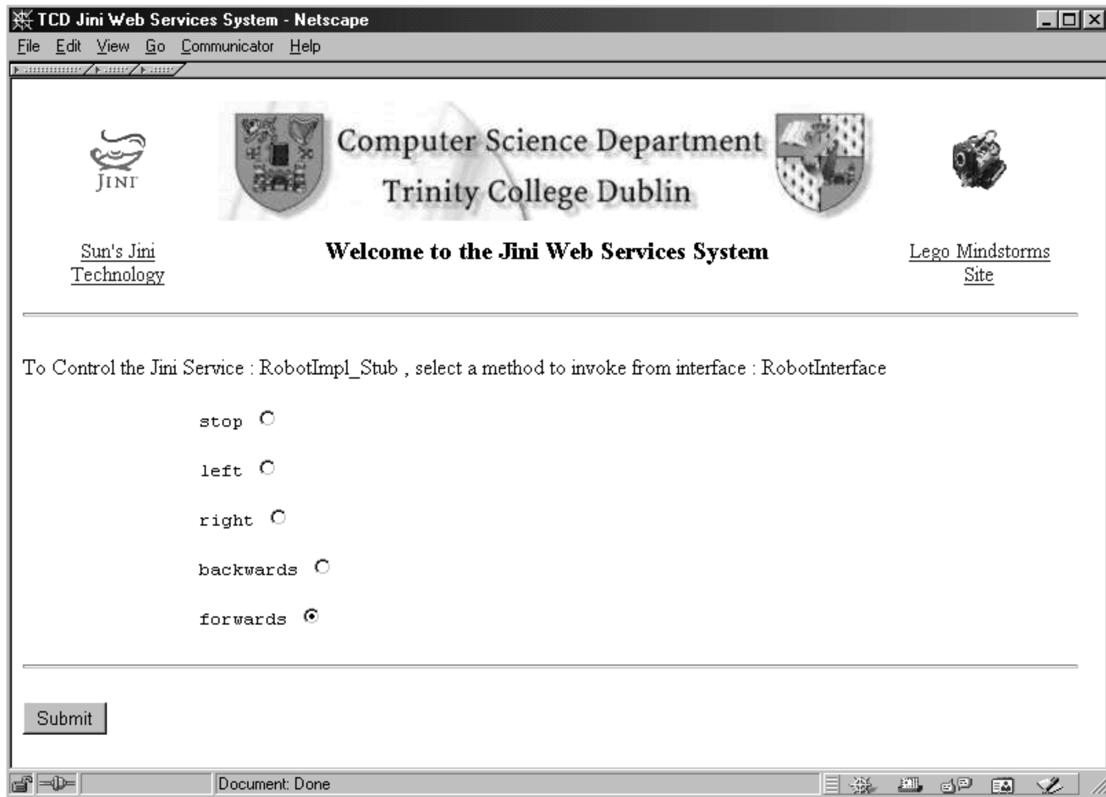

Figure 8.4 – Stage 3: The Methods Supported are Listed

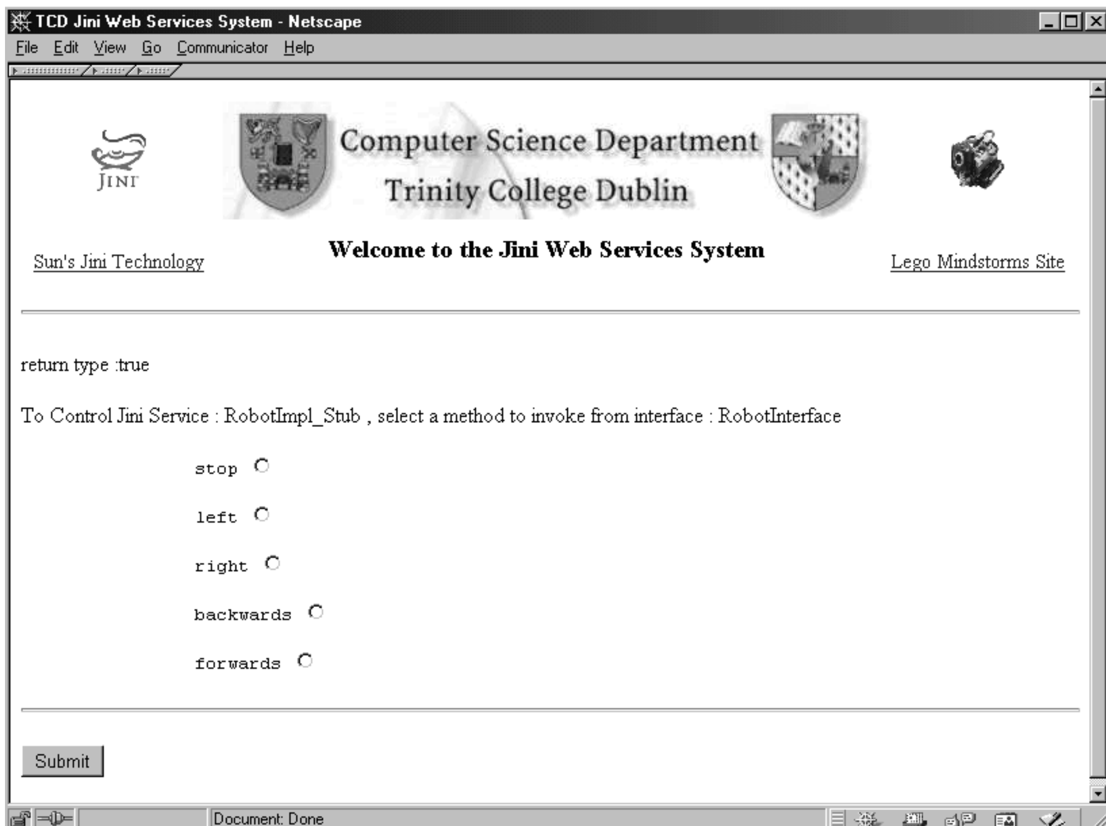

Figure 8.5 – Stage 4: The Results are Displayed and the Service is Offered Again