Using Dynamic Proxies to Support RMI in a Mobile Environment

Gregory Biegel

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 17, 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.

Signed: ____________________

Date: September 17, 2001

Permission to lend and/or copy

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

Signed: ____________________

Date: September 17, 2001

Acknowledgements

I wish to thank the following people for contributing to this project :

To Dr. Vinny Cahill and Mads Haahr, the best supervisors anyone could hope for, thank you.

To the Beit Trust, who made it possible for me to come to Ireland and conduct this research, I would like to extend my most grateful thanks.

To Norma and Richard, the best parents anyone could hope for.

To the MSc. NDS class of 2000, the best classmates anyone could hope for.

To Marci, thanks.

Abstract

The emergence of small, mobile computing devices such as personal digital assistants and cellular telephones has been driven by advances in computing and wireless communication technologies. The development of distributed applications for such mobile devices raises a number of problems not encountered in stationary computing. These problems may be divided into those posed to the management of mobile connections by intermittent network connectivity and limited bandwidth, and those posed to the location of mobile devices.

The Common Object Request Broker Architecture (CORBA) provides support for building distributed object oriented applications. The Architecture for Location Independent CORBA Environments (ALICE) provided a layered architecture for the support of CORBA objects in a mobile environment, whilst allowing mobile CORBA objects to interoperate with existing CORBA applications. Java RMI is another distributed object oriented model with poor existing support for operation in mobile environments.

This project describes the provision of mobility support to RMI applications in a mobile environment, re-using the application independent parts of ALICE. The project integrates the connectivity management offered by ALICE, into the RMI system and develops a location management scheme for RMI based on Dynamic Proxies. Collectively, these components allow the operation of RMI clients and servers in a mobile environment.

Contents

1 Introduction **1**

1.1 Mobile Computing . . . 2
1.2 ALICE . . . 3
1.3 Java RMI . . . 5
1.4 Goals of Project . . . 5
1.5 Achievements of Project . . . 6
1.6 Dissertation Roadmap . . . 6

2 Background **9**

2.1 The Java Distributed Object Model . . . 9
2.1.1 RMI Overview . . . 10
2.1.2 RMI Architecture . . . 11
2.1.3 RMI Wire Protocol . . . 12
2.1.4 RMI Fault Tolerance . . . 12
2.1.5 Dynamic Stub Downloading . . . 13
2.1.6 Custom Socket Factories . . . 14
2.2 Java 2 Micro Edition and Wireless Devices . . . 14
2.2.1 Connected Limited Device Configuration (CLDC) . . . 14
2.2.2 Mobile Information Device Profile (MIDP) . . . 15
2.2.3 J2ME and RMI . . . 16
2.3 The Architecture for Location Independent CORBA Environments . . . 16
2.3.1 The Mobility Layer . . . 17
2.3.2 The Swizzling Layer . . . 18
2.4 Summary . . . 18

3 State of the Art **20**

3.1 Mobile Computing 20

3.2 Issues In Mobile Computing 21

3.2.1 Limited bandwidth and high latency 21

3.2.2 Resource constraints 21

3.2.3 Location management 22

3.2.4 Intermittent connectivity 22

3.3 Future Trends in Mobile Telecommunications 22

3.3.1 Third Generation GSM 23

3.4 Architecture for Location Independent CORBA Environments 24

3.5 Mobile Remote Procedure Call and RMI 25

3.5.1 Wireless Java RMI 25

3.5.2 The RMI Proxy 28

3.5.3 MobileRMI 29

3.5.4 GoldRush 31

3.5.5 M-RPC 32

3.5.6 Efficient Implementation of Java RMI 34

3.5.7 A Distributed Virtual Machine Architecture For Mobile Java Applications 36

3.5.8 Interceptors for Java Remote Method Invocation 37

3.6 Summary 38

4 Design **39**

4.1 Mobility Layer 40

4.1.1 Previous Work on the Mobility Layer 41

4.2 Integrating the Mobility Layer into Java RMI 41

4.2.1 The Java Native Interface 41

4.3 Replacing the Java Socket Implementation 42

4.3.1 Creating a custom Socket Implementation 44

4.3.2 Replacing the default shared library at runtime 45

4.4 A Glue Code layer 46

4.4.1 Selected Method 47

4.5 Mobile Host as Client . . . 48
4.6 Mobile Host as Server . . . 49
4.6.1 Remote Object Referencing in Java . . . 49
4.6.2 Extending UnicastRemoteObject . . . 51
4.6.3 Perform Swizzling at the Gateway . . . 52
4.6.4 Using Dynamic Proxy Classes . . . 53
4.6.5 Selected Strategy . . . 60
4.7 Comparison with Previous Implementation . . . 61
4.8 Summary . . . 61

5 Implementation **62**
5.1 Implementation Goals . . . 62
5.2 Mobility Layer . . . 63
5.2.1 Java Mobility Layer . . . 63
5.2.2 ML_{MH} Layer . . . 63
5.2.3 Integration . . . 65
5.2.4 ML_{MG} Layer . . . 68
5.3 Invocation Redirection . . . 69
5.3.1 Dynamic Proxy Objects in Java . . . 69
5.3.2 Proposed Architecture . . . 70
5.3.3 Components Developed . . . 72
5.3.4 Problems Encountered . . . 74
5.4 Summary . . . 75

6 Evaluation **76**
6.1 Mobility Layer . . . 76
6.1.1 Performance . . . 77
6.1.2 Transparency . . . 77
6.1.3 Comparison to Previous Implementation . . . 77
6.2 Invocation Redirection (JRMP/R) Layer . . . 78
6.2.1 Performance . . . 78
6.2.2 Code Size . . . 82

6.2.3 Transparency 83
6.2.4 Comparison to Previous Implementation 84
6.3 Summary 84

7 Conclusions **85**
7.1 Completed Work 85
7.2 Remaining Work 86
7.3 Future Work 86

A The Java Remote Method Protocol **90**
A.1 Format of Output Stream 90
A.2 Types of Message 90
A.3 Format of Input Stream 91
A.4 Call Data 91
A.5 Return Values 91

List of Figures

2.1 Architecture of the RMI system . . . 12
2.2 A J2ME MIDP application running on a cellular telephone emulator . . . 15
2.3 The physical environment for which ALICE was developed . . . 17

3.1 The Monads architecture . . . 27
3.2 The MobileRMI architecture . . . 30
3.3 The MRPC architecture . . . 34

4.1 The relationship between sockets and their implementation in Java . . . 43
4.2 The `alice.rmi.ALICESocketImpl` class . . . 44
4.3 The relationship between the replacement socket classes . . . 45
4.4 The introduction of a glue code layer . . . 46
4.5 The glue code layer in detail . . . 47
4.6 Inheritance hierarchy in RMI . . . 50
4.7 The introduction of a Mobility Registry on the Gateway . . . 52
4.8 Interaction between an Object reference and a Dynamic Proxy class . . . 53
4.9 Dynamic Proxy architecture . . . 54
4.10 Reference obtained from invocation . . . 56
4.11 Reference obtained as argument . . . 57
4.12 Reswizzling . . . 59

5.1 Class diagram for the ALICENaming class . . . 70

6.1 Invocation times for 10 000 remote method invocations in one-hop RMI . 79
6.2 Invocation times for 10 000 remote method invocations in two-hop RMI . 79

List of Tables

2.1 RMI invocation characteristics 13

6.1 Size of code required for mobility support on the Mobile Host 83

Chapter 1

Introduction

As the level of cellular telephone and Personal Digital Assistant (PDA) ownership continues to rise dramatically throughout the developed world and peoples expectations and requirements for access to information increase, user requirements for such devices are becoming increasingly demanding.

This phenomenal increase in cellular and PDA ownership has been accompanied by major advances in both the computing and communication power of such devices. Advances in wireless networking technologies such as the IEEE 802.11 wireless local area network [Ins00] standard have enabled the networking of such portable devices whilst removing from them the need to be connected to a network cable. Such advances have enabled the development of mobile computing, where computing devices are small enough to make frequent physical movements and are connected to a communication network by a wireless interface.

Middleware is the term used for a distributed software layer, the role of which is to hide the complexities of the underlying physical distributed system from the application designer. The middleware layer presents a uniform programming environment independent of network, device or operating system issues. A popular example of middleware is the Common Object Request Broker (CORBA) [MG] which allows objects developed in different programming languages to communicate and invoke methods on each other. CORBA technology was not designed for operation in a mobile computing environment and its use in such an environment raises a number of problems related to location management and mobile device characteristics.

The Architecture for Location Independent CORBA Environments (ALICE) [Ray98] is an architecture developed by the Distributed Systems Group at Trinity College Dublin in order to provide support to client and server CORBA applications in a mobile environment. ALICE allows both client and server CORBA objects running on mobile devices to interact transparently with non-mobile CORBA objects without the need for a centralised location register to keep track of what address the object is accessible at.

Remote Method Invocation (RMI) [Mic99] allows transparent interaction between distributed Java objects. RMI enables a Java object to make a call on a remote object, which may be located in a different address space. RMI at present has similarly poor support for mobile environments as CORBA and the aim of this project was to investigate the provision of mobility support to RMI applications, re-using as much of the existing ALICE architecture as possible. The reason behind the re-use of existing components of the existing ALICE architecture is to reinforce the position of ALICE as a generic Architecture for Location Independent Environments, applicable across a host of application protocols.

1.1 Mobile Computing

Computing is becoming an increasingly mobile activity, beginning with the adoption of laptop computers in the 1980s and extending to the PDAs of today and most probably the cellular telephones of the future. As hardware continues to get smaller, faster and more powerful, increasingly complex tasks may be performed on smaller devices. Business benefits from this trend as they may arm their salesforce with more information to take to customers, and customers are able to get closer to the business. The advantages are by no means confined to the business world however as the explosive evolution of Short Message Service (SMS) [EV99] as a popular form of social communication and information dissemination illustrates. SMS may be considered a precursor to the instant messaging that will be available with cellular internet access.

Whilst the first attempts to bring the Internet to cellular telephones by way of the Wireless Application Protocol (WAP) [App01] have been less successful than first imagined, vendors are touting third generation networks as overcoming many of the shortcomings of WAP. Most importantly, General Packet Radio Services (GPRS) [Ina00], the

first stage in the evolution towards third generation Global System for Mobile communication (GSM) [EV99] will bring services based on the Internet Protocol (IP) [Pos81] to the handset. Whether third generation GSM will be all vendors are promising remains to be seen as third generation licenses are still to be allocated in many European countries, including Ireland, at the time of writing. What is certain however, is that the arrival of IP services to the handset or PDA will bring the possibility of development of a wide range of distributed IP-based systems aimed at mobile devices.

Wireless mobile devices typically have a number of characteristics which pose challenges to the design of software to be run on such devices. Their physical mobility and frequent movement between points of connection to the network cause addressing problems as the Internet transport protocol (IP), in its current form, is not able to direct traffic to a device which has physically moved, without significant changes being made to the protocol. Although the addressing problems posed by mobile internet hosts are solved by Mobile IP [Per98], this protocol is not widely deployed on the Internet and some other form of location management needs to be provided. The intermittent network connectivity offered by wireless networks introduces the problems of transport connections that may break frequently, a factor that has not been provided for in many internet protocols that were designed for computing systems that rarely, if ever, become disconnected from the network during normal operation. The limited resources offered by physically constrained devices themselves require the consideration of display, power and processing factors which need not be considered in the development of the majority of software for static computer systems.

Middleware, as a software layer whose function it is to hide the complexities of the underlying network infrastructure from higher levels, is the level at which the issues introduced by mobile computing are best solved.

1.2 ALICE

ALICE is an architecture initially developed to allow for mobility support in CORBA systems [Ray98] and now extended to other distributed computing technologies [Cor00], [Wal00]. The ALICE architecture was created in the environment of poor existing support

in the CORBA standard for mobile applications. ALICE solves the problems posed by the characteristics of wireless networks in a layered manner with a session level approach coupled with application support. The mobility support offered by ALICE shields higher levels from the unreliability of the wireless network by transparent re-establishment of broken transport level connections. In addition, addressing of mobile servers is supported by an address translation scheme known as Swizzling, and the transparent movement of mobile hosts between different points of connection to the network is provided for by a transport connection tunnelling scheme known as handoff.

At present ALICE is composed of three layers, not all of which need be included depending on the level of mobility support required.

The **Mobility Layer**, at the session level of the OSI model provides connectivity management to higher levels and is the minimal part of the architecture required for mobility support. The session level allows applications on different machines to establish sessions between them, allowing ordinary data transport as well as providing enhanced services, in this case connectivity management. Connectivity management consists of transparent reconnection of broken transport connections, multiplexing of transport connections, and tunnelling of open connections after the handoff of a mobile host from one point of connection to the fixed network to another.

The **Swizzling Layer**, the use of which is required if a server is to be run on the mobile host, provides location management through the 'swizzling' of server references held by clients, to the server. Swizzling refers to the alteration of the endpoint portion of an object reference to refer to an intermediate gateway rather than the host on which the object resides. The Swizzling Layer also provides invocation redirection after the handoff of a Mobile Host to another Mobility Gateway.

Finally, the **Disconnected Operation Layer** provides support for extended periods of disconnection by caching server functionality on the client side.

Through it's three constituent layers ALICE provides support for mobile applications in a manner that is almost transparent to the application programmer.

1.3 Java RMI

Like CORBA, for which the ALICE platform was initially developed, RMI is a distributed object model extending the traditional object-oriented programming paradigm to include objects in different physical locations, connected by heterogeneous communication networks. CORBA and RMI are both popular middleware technologies used in the development of distributed object systems. RMI differs from CORBA in that it is a language dependent distributed object technology. RMI applications may only be developed in Java, in contrast to CORBA which is not dependent on any particular language.

RMI does not currently provide support for mobile applications and like CORBA, makes the assumption that transport connections break infrequently and that servers do not change location. RMI uses a proprietary protocol known as Java Remote Method Protocol (JRMP) [Mic99], which has high overheads due in part to the distribution of the Java Garbage Collection mechanism, and is not at all well suited to wireless communication.

The operation of RMI within a mobile computing environment raises the same problems as those solved for CORBA by ALICE, namely those of location and connectivity management and elements of the ALICE solution are therefore equally applicable to RMI.

1.4 Goals of Project

Previous work on the generalisation of the ALICE architecture to JINI [Cor00] and RMI [Wal00] resulted in a Java implementation of the ALICE Mobility Layer [Cor00] and a means of performing invocation redirection as provided by the ALICE Swizzling Layer for CORBA [Wal00]. The invocation redirection mechanism introduced significant application level awareness of mobility and required the explicit development of mobility aware proxies for each mobile server.

The major goals of this project were to create a version of the ALICE Swizzling Layer, or equivalent, for RMI and to create an interface to the application independent Mobility Layer for RMI applications, through use of the Java Native Interface. Application level awareness of mobility should be minimised and consequently ease of development of mobile applications should be maximised. The Java equivalent of the Swizzling Layer and the

integration of the Mobility Layer should allow the operation of both client and server Java objects within a mobile environment, providing them with the same support as offered by the existing ALICE architecture.

To this end, the design and implementation of the project are divided into two major sections, one dealing with the provision of connectivity management to RMI applications by way of the Mobility Layer and the other dealing with the provision of location management to RMI applications by way of an invocation redirection scheme for RMI.

1.5 Achievements of Project

The project achieved the major goals outlined above, that is the integration of the existing application independent ALICE Mobility Layer into RMI and the development of an invocation redirection scheme for RMI.

The integration of the Mobility Layer did not require the development of an extensive 'glue-code' layer as first envisaged, but rather involved the creation of a new implementation of sockets in Java to make use of the functions offered by the Mobility Layer in place of standard socket functions.

After extensive investigation of the Java source in order to determine how Java handles remote object referencing, it was decided that due to the large number of dependencies created upon a remote reference within the RMI runtime, swizzling of such references would require substantial changes to be made to the source code, thereby rendering the solution incompatible with other RMI systems. An alternative invocation redirection solution based upon object delegation using a dynamic proxy architecture was designed and implemented. The solution implemented was transparent to the application programmer and did not require any alteration to the existing Java Runtime Environment on hosts where mobility support for applications was required.

1.6 Dissertation Roadmap

The remaining chapters of this project are laid out as follows :

Chapter 2 - Background

This chapter provides the background to the project by detailed examination of RMI, the Java Micro Edition, which permits the operation of Java on small devices, and the ALICE architecture.

Chapter 3 - State of the Art

This chapter introduces and discusses the major research projects and issues that are relevant to this work. The major issues in mobile computing are presented, followed by a discussion of work on the ALICE architecture and the use of RMI in mobile environments.

Chapter 4 - Design

Chapter 4 explains the options available for the integration of the Mobility Layer into RMI and the design of the RMI invocation redirection mechanism. Various possible solutions are discussed and the chosen alternative described in detail.

Chapter 5 - Implementation

This chapter describes how the design selected in Chapter 4 is implemented with reference to specific technologies used, components created, problems encountered and their solution.

Chapter 6 - Evaluation

The implementation is then evaluated in order to determine the relative costs of the chosen solution, the transparency of the solution and a comparison to other solutions.

Chapter 7 - Conclusions

Finally conclusions are drawn regarding the success of the completed work in the context

of the goal of the project. Finally, possible future work on the project is described.

Appendix A - The Java Remote Method Protocol (JRMP)

This appendix describes the operation of the Java Remote Method Protocol (JRMP).

Chapter 2

Background

This chapter provides the background to the project through a thorough examination of RMI, the current Micro Edition of Java and the Architecture for Location Independent CORBA Environments. The significant potential that Java has in a mobile computing environment is demonstrated, as is the capability of ALICE to provide support for mobile computing environments.

2.1 The Java Distributed Object Model

As a programming language Java is well positioned for distributed computing applications due to a well defined and tested security model, as well as platform independence allowing applications to execute unaltered on a variety of platforms. Remote Method Invocation (RMI) is part of the Distributed Object Model for Java, a model which defines interaction between remotely located Java objects at the level of a procedure call [Mic99]. The model provides the illusion of invoking a method upon an object, whilst in fact the method may be invoked on a remote object. Within the model, a remote object is defined as an object resident in a different virtual machine to the client of that object. CORBA is itself a remote method invocation model and indeed one that may be applied to the Java language, however the RMI system was designed specifically to be language dependent and hence able to take advantage of the existing Java Object Model. The advantages of aligning the Distributed Object Model closely with that of the existing Java Object Model are realised in the maintenance of the existing language semantics and associated

limitation of development complexity, as well as preservation of the well tested safety and security of the Java Object Model. Java's Garbage Collection mechanism is also extended to encompass remote objects.

2.1.1 RMI Overview

As discussed, one of the design objectives of RMI was to integrate it closely with the existing Java Object Model and to this end, the semantics of creating a remote object and making a remote method invocation with RMI are very similar to those of local object programming. RMI depends upon the notion of an interface as provided by the Java language. An interface defines the methods that any implementer of the interface must provide an implementation for, but does not itself implement the methods. A class may implement one or more interfaces, that is provide implementation for all the methods declared in the interface(s). Within RMI, remote objects are programmed against an interface, so a remote object consists of an interface and a class defining the methods within the interface. A client of a remote object is exposed to the interface of the remote object, not the implementation class. Every remote interface in RMI extends from the `java.rmi.Remote` interface which is an empty interface serving to distinguish a remote from a local object. Each method in a remote object is forced to include an extra exception of type `java.rmi.RemoteException` in its signature to provide for partial failure within the distributed system. A client of the remote object is required to handle this exception thus forcing the programmer to be aware of the potential for partial failure within the system, and to handle it accordingly. This is justified by the fact that development for a distributed computing environment is inherently different from the local case and hence the application developer should be made aware of the potential for partial failure in a distributed system.

Parameter Passing in RMI

In the Java Object Model, parameter passing is by copy for primitive parameters, and by reference for object parameters [AWW96]. Parameter passing in RMI follows the same model except for the case of non-remote object parameters to a remote call and non-remote object results of a remote call. In both of these cases, the objects are passed by

copy, meaning that a copy of the parameter is passed or returned, rather than the actual object itself. Pass by reference is maintained for remote object parameters and return types, with a reference to the stub being passed rather than to the object itself.

Distributed Garbage Collection

One of the key components of the Java Object Model is the automatic Garbage Collector that keeps track of object references and controls memory allocation, thus removing this burden from the programmer. In the Java Object Model, an object is considered 'live' (and thus not eligible for garbage collection), if there is at least one reference to it held by a client. Due to the possibility of partial failure within a distributed system, this rule is extended within the Distributed Garbage Collector to classify a remote object as having an active reference to it if it has been accessed within a specified time (the lease period). A remote object is eligible for garbage collection if all references to it have been explicitly dropped by those clients that held them, or the lease period has expired without the object having been accessed. The lease period is a variable under programmer control, but defaults to 600 seconds.

2.1.2 RMI Architecture

The RMI architecture is built around a layered model as illustrated in Figure 2.1. The *Stub / Skeleton Layer* is the uppermost layer of the architecture and the layer that client and server applications interact with. A stub is a client-side proxy and is responsible for the marshalling of parameters to an invocation made upon the remote object it represents, the dispatch of the call to the remote reference layer and the subsequent unmarshalling of return types. A skeleton is a proxy performing similar marshalling and unmarshalling of parameters and return types at the server-side. Skeletons have been deprecated in the Java 2 platform since v1.2 and are no longer required for remote method calls. RMI stubs and skeletons are presently produced by the rmi compiler tool (rmic), although it is possible to replace this tool with the use of dynamic proxy objects.

The *Remote Reference Layer* exists on both the client and the server side, is used by the Stub / Skeleton layer and uses the Transport Layer. The Remote Reference Layer is responsible for transport-independent RMI functions such as connection management,

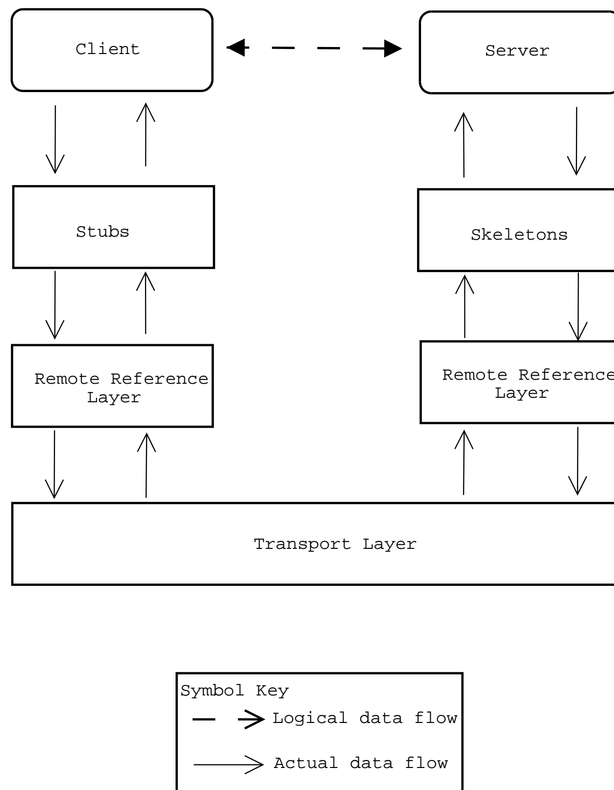

Figure 2.1: Architecture of the RMI system

unicast or multicast method invocation and object activation. At present, the Remote Reference Layer is mostly reserved for future development such as replicated objects and connection recovery, whilst multicast invocation remains to be implemented.

The *Transport Layer* is responsible for setting up and managing the connection between different Java Virtual Machines, and presents an interface to the Stubs / Skeletons Layer. At present, the only transport level protocol supported in RMI is TCP/IP.

2.1.3 RMI Wire Protocol

RMI uses a proprietary protocol 'on the wire', known as the Java Remote Method Protocol (JRMP) [Mic99]. JRMP is illustrated in Appendix A.

2.1.4 RMI Fault Tolerance

RMI provides at-most-once invocation semantics, meaning that the invoker of a method either receives a result, indicating that the method was executed exactly once, or an

Retransmit Request Message	Duplicate Filtering	Re-execute Procedure or Re-transmit Reply	Invocation Semantics
Yes	**Yes**	**Re-transmit Reply**	**At-most-once**

Table 2.1: RMI invocation characteristics

exception indicating that no result was received. In this case, the method may have been executed once or not at all. These semantics ensure that the method called is never executed more than once.

At-most-once invocation semantics provide RMI with tolerance against omission failures whereby a communication error results in the loss of a message, crash failures whereby a remote host fails for some reason and arbitrary failures which may occur for any of a number of reasons.

Table 2.1 illustrates RMI's behaviour in the face of the characteristics of partial failures.

2.1.5 Dynamic Stub Downloading

A significant feature of the Java language is the ability of a Java Virtual Machine (JVM) to dynamically download Java bytecode from any URL (usually in a different physical location), and to execute these bytecodes locally [Mic99]. This provides a substantial amount of flexibility as it removes the need for explicit code installation on the client side. This is especially pertinent to RMI, as in order for a client to make method invocations upon an RMI server it is necessary to have a server stub present on the client. The invocation then takes place upon the stub representation of the RMI server. In the case of physically remote RMI client and server, dynamic code downloading gives the client the ability to obtain the stub code on demand from the server. Server recompilation does not then necessitate explicit stub redistribution to all clients. The security issues inherent in such an approach are addressed by Java policy files which govern which permissions are available to code from different sources.

Dynamic stub downloading allows the implementation of a remote interface to be changed independently of the client and with no need for the client to have the implementation of the interface available prior to execution.

2.1.6 Custom Socket Factories

By default, RMI uses the TCP transport protocol as implemented by `java.net.Socket` and `java.net.ServerSocket` types. The RMI system uses instances of these sockets in order to communicate between clients and servers.

Additionally, RMI allows the specification of a custom *Socket Factory* that produces an application programmer defined socket type for use by the RMI transport, and allows the client to download the socket factory by way of the dynamic code downloading property of Java. Two types of Socket Factory are typically developed, an *RMIClientSocketFactory* which produces an application programmer defined subclass of `java.net.Socket` and an *RMIServerSocketFactory* which produces an application programmer defined subclass of `java.net.ServerSocket`. The custom socket factory need only be specified once to the RMI runtime for sockets of the type produced by the factory to be supplied to the system.

It is this feature of RMI that facilitated the integration the Java Mobility Layer [Cor00] into the RMI system and was retained in the integration of the C Mobility Layer into RMI in this project.

2.2 Java 2 Micro Edition and Wireless Devices

The Java 2 Micro Edition (J2ME) is a framework for the deployment and use of Java technology in what is termed the 'Post-PC world' by Sun [Micb]. The Post-PC world is envisaged as being composed of numerous small, mobile computing devices fulfilling the role of todays PC. J2ME will be supplied in different configurations aimed at various different market segments and fulfilling certain requirements. One such configuration is the Connected Limited Device Configuration (CLDC).

2.2.1 Connected Limited Device Configuration (CLDC)

The Connected Limited Device Configuration [Suna] is designed for resource-constrained devices, more specifically for devices running a 16 or 32-bit processor and having 512K or less memory available to the application environment. Target devices are assumed to have intermittent, limited-bandwidth network connectivity as well as a limited power supply. The configuration supplies a Virtual Machine, the K Virtual Machine (KVM) and an

Figure 2.2: A J2ME MIDP application running on a cellular telephone emulator

Application Programmer Interface (API). Above the configuration layer, industry groups are able to define a *profile* relevant to their particular industry. This allows various industries to define J2ME profiles containing only what they need for their particular device. The Mobile Information Device Profile (MIDP) is an example of an existing J2ME profile.

2.2.2 Mobile Information Device Profile (MIDP)

The Mobile Information Device Profile [Sunb] is a J2ME profile aimed at wireless devices such as cellular telephones and pagers and provides a set of API's for J2ME development on such devices. The profile specification was developed by an expert group composed of major players in the wireless industry such as Motorola, Oracle and Samsung and addresses interface and networking issues specific to such devices.

J2ME has already been included in certain cellular offerings such as Motorola's iDEN telephone and NTT DoCoMo's iMode phone and although there are currently only around 3 million J2ME enabled devices in the world, Nokia has announced plans to ship 100 million J2ME enabled cellular phones before the end of 2003. An example of a J2ME MIDP application running upon a cellular emulator is illustrated in Figure 2.2, and helps to demonstrate the potential graphical capabilities of J2ME.

Inclusion of Java into such popular consumer devices will result in a burgeoning market for Java applications to run on such devices and the possibility of the development of the so

called **killer app** which has thus far eluded the industry. RMI, as the de-facto distributed object technology for Java will be essential to the development of distributed computing systems on small devices.

2.2.3 J2ME and RMI

RMI is not included in the J2ME specification in its present form, but there is a Java Specification Request (JSR 66) [Mica] for a J2ME RMI Profile to be developed under the Java Community Process. The proposed final draft for JSR 66 states certain functionality present in the J2SE RMI API that will be retained in J2ME. Such functionality includes maintaining full RMI call semantics, maintaining the current RMI wire protocol and keeping the rmiregistry. Existing RMI functionality that will not be included in J2ME includes support for RMI through firewalls via proxies, RMI's multiplexing protocol and server-side support for Activatable objects. Our location management solution for mobile RMI server objects uses only functionality which will be included in the J2ME RMI profile and thus is expected to be portable to this edition.

The implementation of RMI in the Micro Edition of Java will bring the possibility of development of distributed object technologies resident on small wireless devices.

2.3 The Architecture for Location Independent CORBA Environments

ALICE is an architectural framework that provides mobility support for a range of application-level client / server protocols [Mad00]. The architecture was initially developed for CORBA's IIOP protocol and work to generalise the architecture to other protocols is ongoing. The ALICE architecture allows such application-level protocols to provide their own mobility support through location management, disconnected operation support and connectivity management which collectively address the issues discussed in Section 3.2. ALICE permits the operation of mobile servers with no centralised location register required to keep track of the whereabout of the servers. The physical environment for which the ALICE framework was developed is illustrated in Figure 2.3. Mobile Hosts (MH) are small mobile computing devices with wireless network interfaces such

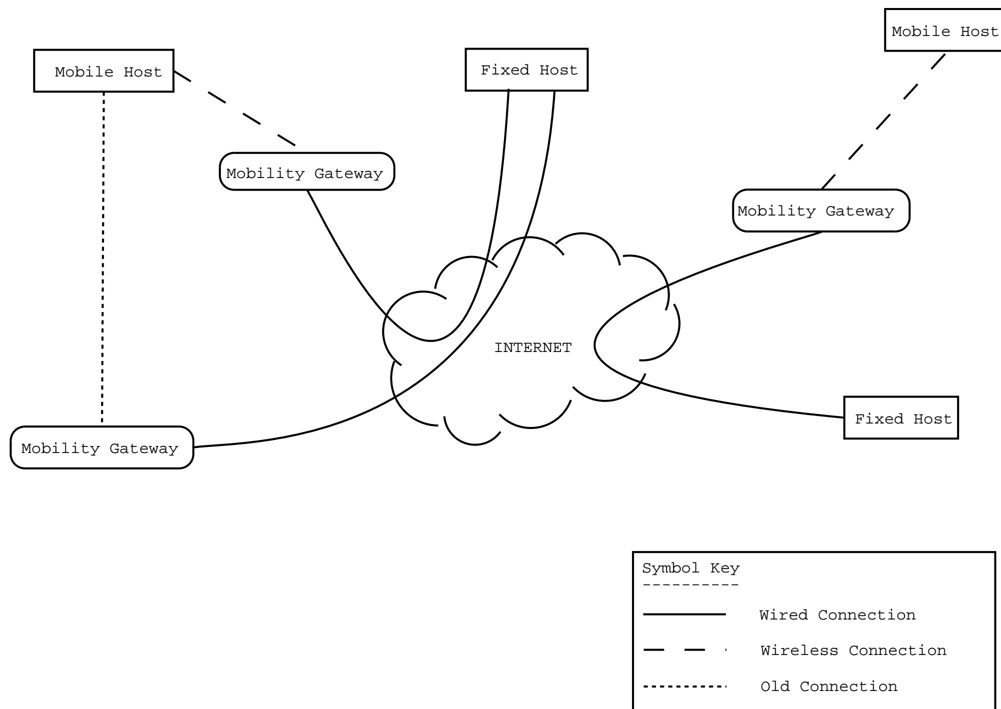

Figure 2.3: The physical environment for which ALICE was developed

as an iPaq handheld computer with an 802.11 wireless LAN interface card. These MHs are connected to static Mobility Gateways (MG) which maintain both wireless and wired network interfaces and act as the entry point from the wireless network into the wired network. Fixed Hosts (FH) are non-mobile i.e static computing devices which communicate with the MHs via the MGs. The MHs are physically mobile and may change their point of connection from one MG to another during a procedure known as handoff. ALICE caters for such movement and provides for handoff of connections to from the old to the new gateway.

ALICE provides mobility support in the form of a layered architecture, with each ALICE layer solving specific problems introduced by the mobile environment.

2.3.1 The Mobility Layer

The Mobility Layer addresses connectivity management within a mobile environment and overcomes the problems posed by the intermittent connectivity, limited bandwidth and frequent changes of point of connection to the network of wireless networks.

The layer provides transparent reconnection attempts to a Mobility Gateway when

a Mobile Host has become disconnected from the network, thus eliminating the need for application level handling of reconnection attempts. The use of limited bandwidth is maximised by the multiplexing of several logical connections onto a single physical connection. The tunnelling of open connections from one Mobility Gateway to another following the movement of a Mobile Host to a new point of connection is also handled at this layer.

Whilst the above functions of the Mobility Layer are all totally transparent to the application, the Mobility Layer also provides additional functions to applications where knowledge of the mobile environment is desirable. Such information is provided in the form of callbacks from the Mobility Layer to the application, which are invoked when there is a change in the state of the wireless connection, and provide state information about the wireless link to higher layers.

2.3.2 The Swizzling Layer

The support offered to applications by the Mobility Layer is sufficient for the operation of clients on Mobile Hosts, but does not address the problems of location management in mobile computing which occur when a server is present on a Mobile Host. For instance, CORBA uses Interoperable Object References (IORs) to uniquely identify and locate server objects. These IORs contain an IP address and port number at which the server object may be found. When there is no direct network connection between a client and the server object, these IORs are not valid as a client must rather contact the Mobility Gateway to which the server is currently attached, and any attempts to directly contact the server object will fail.

ALICE solves this problem by way of a Swizzling Layer for IIOP, which changes the IORs exported by servers resident on Mobile Hosts to refer to the Mobility Gateway to which the Mobile Host is attached rather than the Mobile Host itself.

2.4 Summary

This chapter presented the background to the project by way of a detailed examination of the RMI system and current developments in the operation of Java applications on

mobile wireless devices in the form of the Java Micro Edition.

The ALICE architecture which provides for the operation of CORBA clients and servers in a mobile environment and forms the basis of our solution for RMI was introduced and its constituent layers outlined.

Chapter 3

State of the Art

This chapter examines the issues associated with mobile computing, recent work on the ALICE architecture and research into the operation of RMI in wireless environments and on small devices. In addition, future trends in mobile telecommunications likely to drive the move towards mobile computing are also examined.

3.1 Mobile Computing

Continued advances in the fields of memory and microprocessor design combined with those in wireless networking technologies have led to a proliferation of small, portable computing devices equipped with diverse wireless communication technologies such as Global Systems for Mobile telecommunications (GSM) [EV99] interfaces and the IEEE 802.11 standard for wireless local area networks [Ins00]. At the same time, an increase in the trend towards a global economy and workforce has led to a highly mobile working environment in which access to information is the empowering factor. Such trends are not confined to the workplace either, with increased ease of information access transforming almost every aspect of our lives. Mobile computer networks based around portable devices and wireless communication technologies are facilitating this ever-increasing demand for information at any time and any place.

Such mobile networks pose certain challenges to the systems designer as factors need to be taken into account that play no part in wired networks.

3.2 Issues In Mobile Computing

This section examines some of the major constraints that exist in mobile computing environments and their effect on systems running in such environments.

3.2.1 Limited bandwidth and high latency

Wireless communication links are characterised by lower bandwidths and higher latencies than their wired counterparts. Indeed, bandwidths of 300bps to 10Kbps and round trip latencies of 5 seconds are common. Protocols such as the Transmission Control Protocol (TCP) [SI], used extensively on the Internet, may perform poorly in such an environment as they make assumptions based upon facts about wired networks. For instance, most TCP implementations interpret packet loss as network congestion and consequently reduce transmission rates to reduce the network load [1]. In the case of wireless networks, packet loss is generally due to unreliability of the wireless link and the correct course of action would be to increase packet transmission rates in order to increase the overall throughput of data [And96].

Communication protocols with high overheads (such as the Java Remote Method Protocol used by RMI) perform poorly in constrained bandwidth environments due to the large amount of protocol control information that has to travel over the limited bandwidth link.

3.2.2 Resource constraints

Mobile devices are by definition smaller and lighter than static devices. Their small physical size, combined with a limited power supply serve to constrain the processing, memory and display capabilities of such devices. Applications designed for traditionally large, high resolution computer monitors often need to be radically redesigned to run on a PDA or cellular telephone. Similarly, applications that rely upon the large amount of main memory available on a desktop computer will not run unaltered upon a Palm Pilot.

Resource constraints often mean that existing systems need to be extensively modified or redeveloped before they may run in such an environment.

[1] Due to Jacobson's slow start algorithm

3.2.3 Location management

Mobile devices change their point of attachment to the network frequently in contrast to such changes which are seldom made by static systems. The addressing scheme currently in use by the Internet Protocol (IP) assumes that a hosts point of attachment to the network (its IP address) seldom changes, and major modifications need to be made to the protocol to accommodate frequently changing addresses [Per98].

Since the required modifications have not been made on a global scale, it is the responsibility of the wireless networking architecture to provide for the change of address of the Mobility Gateway to which the Mobile Host is attached.

3.2.4 Intermittent connectivity

A mobile device may be only intermittently connected to the network for a variety of reasons. It may be that the device is out of the wireless coverage area, it's power supply may have been exhausted or it may have been disconnected for economic reasons such as the significant expense of a commercial GSM link. Systems running on such devices must be able to handle the intermittent connectivity patterns and perhaps provide for continued operation of the system whilst disconnected from the network for a substantial period of time.

3.3 Future Trends in Mobile Telecommunications

It is unlikely that the development of advanced wireless networking technologies such as those based upon the IEEE 802.11 standard will create a massive global consumer market for these technologies. Such networking technologies are simply too specialised and indeed currently too expensive for widespread consumer adoption. It is more likely that the evolution towards mobile computing will be driven by a wireless communication device that a large proportion of the developed worlds population already owns - the cellular telephone. Modern cellular telephones are small, mobile and contain increasingly powerful processors. They may be combined with portable computers or PDAs to provide such computers with a wireless connection to the network. Attempts have already been made to provide Internet content to cellular phones (in the form of WAP) and in the

process have both proved the popularity of such a service and begun to blur the distinction between voice and data communications. The incorporation of Java on the latest cellular telephone offerings will enable the development of a wide range of computing applications which may make use of the wireless communication interface of the cellular telephone and contribute significantly to the development of mobile computing.

The majority of the world's mobile telecommunication infrastructure is presently in a state of evolutionary transition as operators substantially upgrade the capabilities of their networks, particularly in terms of bandwidth, in preparation for the offering of full IP-based data services to the handset.

3.3.1 Third Generation GSM

Globally, the most widely deployed mobile telecommunications system is currently based upon the Global System for Mobile Communications (GSM) standard [EV99]. The GSM system is a second generation (2G) system, offering low-rate, circuit-switched and packet-switched time division multiple access (TDMA) based data services. International Mobile Telecommunications in the year 2000 (IMT-2000) [Int00] is the third generation (hereafter referred to as 3G) specification developed by the International Telecommunications Union (ITU) that will provide enhanced voice, data and multimedia services over wireless networks [Int00]. The current plan is for IMT-2000 to specify a collection of standards that will allow for the following data rates to be provided to cellular subscribers :

- 144 kbps at mobile speeds
- 384 kbps at pedestrian speeds
- 2 Mbps in an indoor environment

The 3G specification aims to evolve existing 2G systems to provide high-speed data access whilst attempting to preserve existing infrastructure and to this end, the evolution towards 3G is divided into three phases.

The first enhancement to the existing infrastructure is the introduction of General Packet Radio Services (GPRS) which involves the overlaying of a packet based interface onto the existing circuit-switched interface of GSM. The introduction of GPRS will enable

the use of any IP-based service such as Telnet or FTP, by a GPRS handset. GPRS will not, however, have any effect on the existing bandwidth capacity of a cell and single user throughput is likely to be around 56 kbps. Phase One is expected to be complete by the end of 2001 [Int00].

The second step towards 3G involves the move to Enhanced Data Rates for GSM Evolution (EDGE), by GSM network operators. EDGE will enable higher data rates to be achieved over existing infrastructure through different modulation techniques and an adaptive radio link protocol. It is predicted that Phase 2 will emerge in early 2002.

The final step in the evolution to the 3G network consists of a change to the air interface of the GSM communications network through the introduction of Wideband Code Division Multiple Access (WCDMA). This final step will see the arrival of the vastly increased data rates as defined above and is expected to become commercially available in 2002 / 2003.

The increased bandwidth capabilities to be offered by 3G cellular networks are likely to have significant effects in the domain of mobile computing. With current bandwidth constraints vastly improved, a large range of data services will be available to the handset. More importantly, always-on network connectivity and data-centric pricing models offered by 3G networks are likely to drive the widespread adoption of mobile computing as offered by 3G cellular telephones.

3.4 Architecture for Location Independent CORBA Environments

Since the initial development of the Architecture for Location Independent CORBA Environments [Ray98], work has been carried out on integrating it with JINI environments [Cor00], the RMI architecture [Wal00] and the development of the Disconnected Operation Layer.

The initial work on the integration of ALICE with the Java language (in the form of integration with JINI) resulted in the initial development of a Java based version of the Mobility Layer [Cor00]. This implementation of the Mobility Layer was a direct port or conversion of the existing C Mobility Layer, into Java. It was demonstrated that mobility

support may be incorporated into the Java language in a transparent manner by using the facility of RMI which allows the specification of a custom transport protocol. Since JINI uses RMI as its underlying transport, the project was directly applicable to RMI systems. The Java Mobility Layer permitted the operation of JINI clients on Mobile Hosts and provided connection management to such applications.

Further work on the Java Mobility Layer [Wal00] resulted in completion of the implementation, as well as the development of support for server objects residing on mobile hosts. Support for mobile server objects was provided for by way of a system of proxy representations of server objects which resided on the Mobility Gateway and provided invocation relaying between client and server. The mobility proxy objects were developed at the application level by the application programmer consequently removing transparency from the process. The system also placed some restrictions on the type of parameters that may be passed in RMI calls.

3.5 Mobile Remote Procedure Call and RMI

Numerous research projects have been undertaken in the field of RMI and Remote Procedure Call in wireless and mobile environments. Much of the research conducted into RMI covers the optimisation of the RMI communication protocol to improve its performance over wireless links. Research into Remote Procedure Call in a mobile environment has also examined communication protocol optimisations to improve performance over wireless links. This section examines important facets of this research, as well as projects dealing with remote object mobility in RMI and a distributed virtual machine architecture for Java on small devices.

3.5.1 Wireless Java RMI

The developers of the Wireless Java RMI architecture argue that due to high protocol overheads, RMI is not suited to operation over inherently slow wireless links. Protocol overheads are introduced by Java's Distributed Garbage Collector, as well as TCP handshaking and in experiments performed by the authors, auxiliary data related to garbage collection and TCP handshaking made up the majority of the data transmitted during a

remote method invocation. A solution, based upon performance enhancing proxies known as *mediators* is proposed to counteract such poor performance [Ste00].

Due to limited bandwidth and high latency, the amount of data sent over a wireless link is important, and should be as small as possible. RMI uses a proprietary protocol on top of TCP to provide for communication between hosts of distributed objects. Following the RMI protocol, when an RMI connection is opened the transport layer either opens a new TCP connection, or reuses an existing one if a free one is available [Ste00]. A ping message is used to test the liveness of an existing TCP connection if it has been idle for longer than a certain period of time. In a wireless network environment in which latencies are high, such ping messages to test the liveness of a connection cause the RMI protocol, and hence RMI applications using the protocol, to perform poorly. Performance of the RMI protocol also suffers from the problems posed by the slow start algorithm in the operation of TCP in a wireless environment, as discussed in Section 3.2.1. Furthermore, in JDK 1.1, the RMI protocol wrote header data one byte at a time, causing the negative effects of the slow start algorithm to be compounded, as acknowledgement of a segment containing only one byte of data is required before a second segment may be sent. Such extra data communication degrades performance over high latency wireless links and leads to poor performance of RMI applications in such network conditions. The negative effects of the slow start algorithm have, however, been minimised since JDK 1.2, as the RMI protocol no longer writes data one byte at a time.

Campadello, Koskimies and Helin analyse the traffic generated for a simple remote method invocation, and show that the actual invocation only takes up 5% of the total transmitted data, while 69% was taken up by the Distributed Garbage Collection mechanism [Ste00].

The solution proposed to the problems of protocol overhead was designed so that no modification of the existing RMI implementation would be required. To this end, a number of optimisations were introduced including the use of data compression to reduce overheads introduced by serialization, the handling of protocol acknowledgements by local mediators rather than their transmission over the wireless link, and the decoupling of distributed garbage collection.

The architecture introduces an RMI Agent on the client side and an RMI proxy on

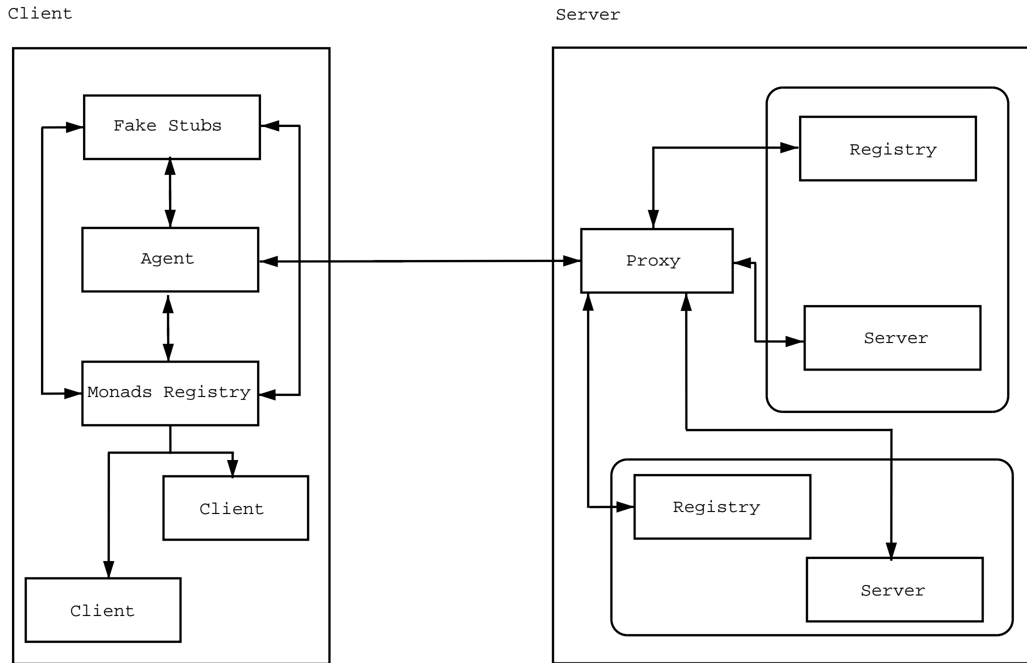

Figure 3.1: The Monads architecture

the server side as illustrated in Figure 3.1. The RMI Agent intercepts method invocations made by the client, only forwarding a lookup request to the server if no reference is already held by the client. Major improvements are realised by decoupling the garbage collection and having the RMI proxy on the server side renew leases on objects until instructed to stop by the RMI Agent. This significantly reduces Distributed Garbage Collector (DGC) messages being passed over the wireless communication link, thereby improving performance dramatically. Furthermore, all communication between client and server is compressed using a generic compression algorithm.

During field trials of the mediator-based approach not using any compression, an improvement of 417% was obtained on the time taken to perform a registry lookup and an improvement of 216% on the time taken to perform a remote invocation [2], leading to an overall improvement of 365% in the time taken to obtain a remote reference and invoke a method. It should be born in mind that a registry lookup is, however, only performed once in order to obtain the reference to a remote object and thus improvements in this area will not significantly impact the operation of RMI.

The project demonstrated some of the reasons behind RMI's notoriously poor per-

[2]Measured as the time between the client request and receiving the return value of the invocation

formance in the environment of a wireless network. The solution succeeded in reducing the protocol overhead inherent in an RMI invocation, without making any changes to the underlying RMI architecture. The ideas presented to reduce the overhead of RMI may be applied to any wireless system in which RMI is a component and demonstrate the redirection of RMI calls by an agent.

3.5.2 The RMI Proxy

The RMI Proxy is a commercial product offering a standard approach to controlling RMI traffic through network firewalls, allowing RMI applications to traverse the Internet in a controlled manner without the limitations of the RMI/HTTP tunnelling approach as offered by Java [PB01].

The RMI Proxy provides a Java application and API to the application programmer which allows controlled penetration of firewalls by approved RMI clients and servers. The RMI Proxy attempts to overcome the problems caused by the blocking of traffic, by a transport firewall, which does not originate from a trusted application firewall such as is provided for the FTP and SMTP protocols. To this end, the RMI Proxy provides an application firewall (proxy) based upon the Java Remote Method Protocol (JRMP). A similar architecture to enable CORBA applications to communicate across one or more firewalls, and known as the General Inter Orb Protocol (GIOP) Proxy is included in the CORBA 3.0 specification.

A common way for applications to traverse firewalls is through the use of HTTP tunnelling [Mik]. Since HTTP traffic is generally permitted through a firewall, the application protocol traffic is placed in HTTP request and response messages in order to gain access through the firewall, and is then unpacked, and the original Protocol Data Units reformed, at its destination.

RMI does provide a mechanism for traversing firewalls based upon HTTP tunnelling, but the scheme suffers from a number of drawbacks. The process is inefficient and up to 10 times as slow as normal RMI, and incurs a 15 second overhead on each new client connection, as the RMI runtime must first attempt a direct connection before falling back on HTTP tunnelling. More importantly however, the scheme only provides for the traversal of client-side firewalls and presently there is no support for the traversal of

server-side firewalls. Thus an RMI server must necessarily exist outside of a firewall in order to remain accessible to outside clients, a constraint that is not feasible.

The RMI Proxy is an application protocol that overcomes the constraints of HTTP tunnelling in RMI and allows the penetration of server-side transport firewalls by JRMP traffic. The proxy is able to block access by non JRMP-protocols (an essential feature of application firewalls) and permit or deny access and execution of remote methods by clients. Code mobility via the RMI codebase is also mediated by the proxy in both directions.

The RMI Proxy architecture provides insight into the use of proxies to mediate on RMI invocations and demonstrates the use of such proxies to intercept, process, and forward RMI invocations and return values.

3.5.3 MobileRMI

MobileRMI is a toolkit developed to realise the benefits introduced by the use of mobile code and mobile objects. Such benefits include the movement of code to a different execution site where it may interact with local resources, thus allowing the movement of the processing task rather than transferring potentially large amounts of data over the network. The MobileRMI toolkit extends the super class of RMI server objects, `java.rmi.server.UnicastRemoteObject`, adding methods to enable the creation of objects in a remote address space, as well as move objects between address spaces [MAne]. Although RMI supports object mobility in the way that objects may be passed as arguments to methods, the MobileRMI toolkit aims to make object mobility more explicit, arguing that mobile object system programming requires awareness of the differences inherent in programming for a distributed object system.

MobileRMI provides a number of classes to provide support for the remote creation and moving of objects and achieves this through modification of the underlying RMI implementation source. Most notably, server objects supporting MobileRMI must extend `MobileUnicastRemoteObject`, rather than `UnicastRemoteObject`.

In order to support object mobility, some form of remote reference updating is required so that clients holding a reference to an object which has since moved to a different address space may still contact it. The way in which MobileRMI deals with remote

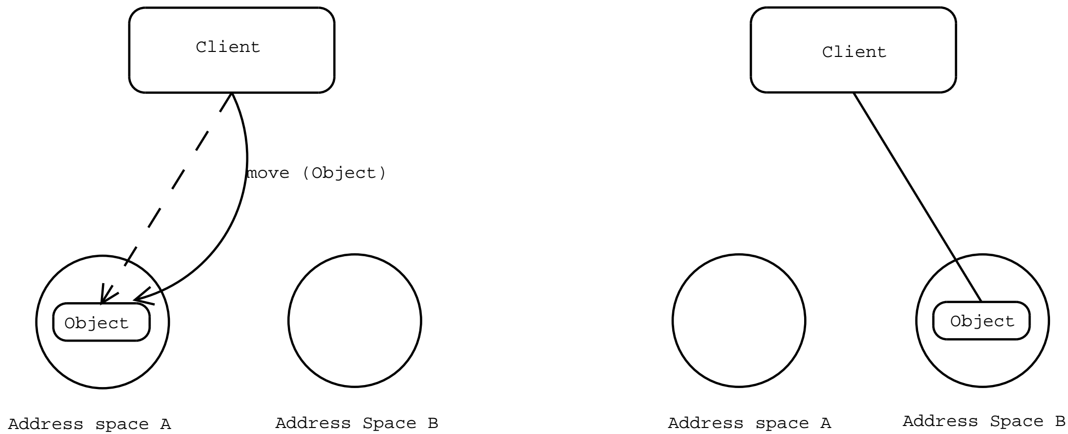

Figure 3.2: The MobileRMI architecture

reference updating is through the creation of dummy objects at the point of departure of the original object. The dummy object maintains an object reference to the new location of the object (which in turn may itself be a dummy object if the object has since moved), which is returned to a client attempting to contact the object. References are updated by way of updating the `sun.rmi.transport.LiveRef` attribute held by the client. This attribute consists of a unique object identifier as well as a [hostname, port#] pair on which the object accepts calls. Once no more clients hold a reference to a dummy object, it becomes eligible for garbage collection. The movement of an object between address spaces is achieved by invoking the `move(Object)` method, which causes the movement of the object to a different address space as illustrated in Figure 3.2. A client holding a reference to an object (of type `MobileUnicastRemoteObject`) in address space may invoke the move() method upon it and move it to address space B. This may be done in order to take advantage of spare CPU cycles on the host in address space B, for instance.

The moving of objects between address spaces requires the operation of a Mobility-Support RMI server in the destination address space which handles the creation of the object in the remote address space from a serialized wrapper class received from the originating address space. In order to maintain the autonomy associated with the definition of a software agent, RMI's default on-demand class loading strategy is replaced by sending the full class closure to the destination host when an object moves. This removes the need to load the bytecode from the originating host's codebase and ensures the mobile object is autonomous.

The MobileRMI toolkit provides mechanisms for programming mobile software agents in Java, utilising RMI for object mobility. The new and extended classes need to be imported for mobility support and a new version of the rmiregistry is required, but the original stub compiler may still be used. The toolkit is not suited to legacy applications due to slightly different syntax, which is justified by the need to make the programmer aware that she is dealing with a distributed object system.

Although the MobileRMI system deals with code mobility and the moving of objects to different address spaces potentially all on a wired network, the solution does demonstrate some important issues in remote reference updating in RMI.

3.5.4 GoldRush

Gold Rush is a middleware system developed to support Java clients that reside on intermittently connected mobile devices and which access an enterprise database on a central server [Mar97]. In addition to the challenges to wireless database access created by platform limitations and wireless communications, frequent disconnection modes create data consistency problems, as a disconnected client may have an inconsistent view of the data in the system. This may arise if a client has performed some processing on the data whilst other clients were disconnected. The Gold Rush system has overcome the challenges posed to database access by devices with limited capabilities and frequent disconnection and provides transaction-based access to data from such devices.

The Gold Rush system works by allowing part of the enterprise database to be replicated on the mobile device in the form of Java objects, allowing transactions to be carried out against the replicated data while the device is disconnected from the enterprise system. Once reconnected, any transactions performed during disconnection, are replayed against the database and any conflicts resolved using pre-defined formulae and conflict resolution strategies. As a result, the Gold Rush system is most applicable to those applications in which the client may be disconnected for the majority of the time and is not suitable for any application requiring real-time data access.

The Gold Rush system makes use of RMI between the mobile client and an Object Server, the purpose of which is to download subsets of the database to the mobile client, as Java objects. The Object Server communicates with the enterprise database via JDBC

calls. Database records are stored locally on the client in a persistent store and database actions upon the data are mapped to method invocations, each of which generates a detailed log entry for later replay.

Gold Rush aims to reduce traffic between server and client over what is likely to be an expensive and unreliable link by maintaining a directory, both at the client and the server, of objects which have been cached on the client. Before an object is transmitted to the client, a check is made against the client directory, for the object ID. If the object ID is not present in the client directory, the entire object is transmitted. If, however, the object ID is present in the client directory, indicating that the object is cached at the client, a check is made against the timestamp of the object. If the timestamps do not vary between client and server, only the object ID is transmitted, otherwise only the differences between the object to be transmitted and the cached copy are transmitted. This strategy serves to minimise traffic over the wireless link between client and server. It should be noted that this reduction of data traffic between client and server was realised without altering the internal RMI classes, although the solution is computationally expensive due to the need to maintain collections of objects at both client and server, and to perform comparisons on these objects.

The Gold Rush system uses data caching strategies and transactional properties to provide database access to Java applications on primarily disconnected mobile devices, allowing the users of such devices to maintain access to data in the absence of a communication link. This type of caching strategy falls down however when a mobile user needs to access a large dataset randomly and the working set is consequently too large to be adequately predicted and cached prior to disconnection.

GoldRush provides valuable insight into caching and replay of remote method invocations using RMI. Caching and replay play an important role in the operation of wireless systems in the absence of a communication link.

3.5.5 M-RPC

M-RPC is a Remote Procedure Call service for mobile clients developed at Rutgers [BB95]. M-RPC, in common with many architectures for wireless mobile computing, aims to minimise the amount of traffic on the wireless link, due to the limited bandwidth available

on wireless networks.

A client in a client / server system typically learns a server address by performing a name lookup from which a handle to the server is returned. Bakre and Badrinath make the point that it is unrealistic to expect mobile applications to change server bindings, that is perform another lookup on a server name to obtain a new handle (binding), after each move and an RPC system for mobile clients should perform this task transparently [BB95]. The concept of subject-based naming is introduced whereby a client does not make a binding to a particular server, but rather to a service, which may be transparently provided by one or more physical servers at any one time.

The issue of the transport layer to be used in wireless mobile communication is examined and the authors point out that whilst a connectionless protocol such as UDP allows for simple stateless servers, it is not sufficient for error-prone wireless links. TCP, as a connection-oriented protocol is not ideal for invocation-response style communication, as it requires servers to maintain the state of all open connections. In a mobile environment where the state of such connections may be changing rapidly, this is a disadvantage. In addition, the performance of TCP degrades rapidly on a wireless channel due to the slow-start congestion control algorithm as mentioned in the discussion on the Wireless RMI project. It is proposed that for RPC in a mobile environment, a transport protocol is needed which is reliable (as is TCP), but fairly simple (as is UDP) and provides the capability to dynamically bind to the closest information provider.

Another aspect of wireless mobile communication examined is that of disconnected operation, which is best solved at the transport and RPC layers. An interesting point raised by Bakre and Badrinath is that of dynamically changing the presentation of data transmitted over the wireless link, to fit with the physical characteristics of the link. For instance, if a mobile client is within an area of poor radio reception, it may want to delay certain data communication until reception improves.

The M-RPC solution is based upon an indirect client-server model, and introduces a Mobility Support Router entity between the Mobile Client and the fixed Server. This enables the operation of distinct transport protocols on the wired and wireless portions of the network.

Before initiating an RPC, a client needs to obtain a handle to the server, which it

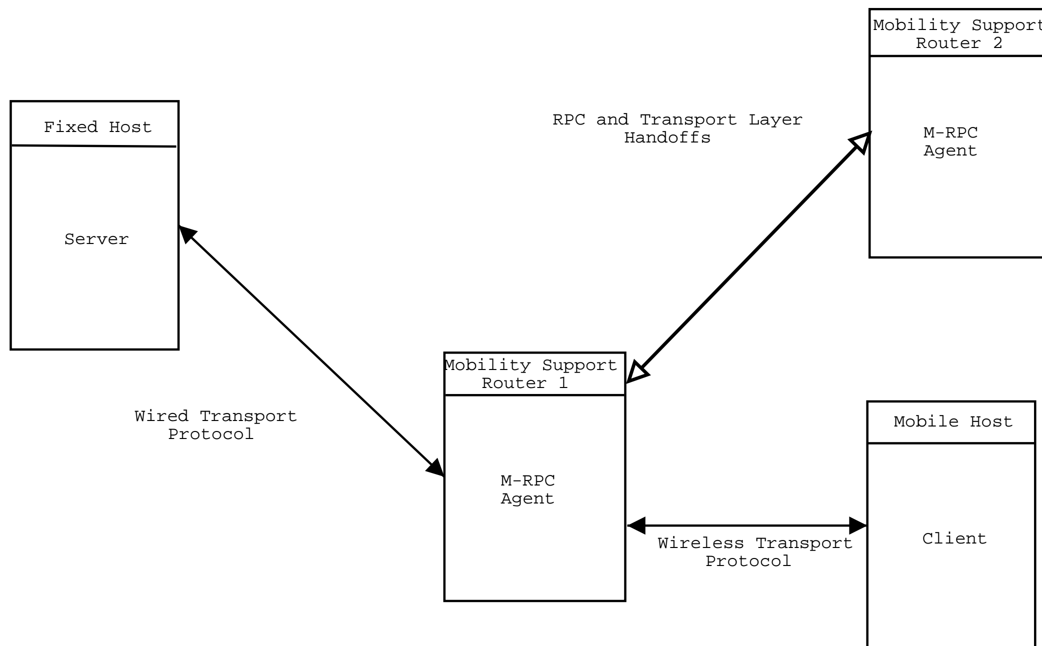

Figure 3.3: The MRPC architecture

may do by specifying the service-name or a particular server, in keeping with the scheme of subject-based naming. By binding to a service rather than the particular provider of a service, mobile applications are relieved of the need to rebind to the service after movement.

The M-RPC solution makes use of an indirect transport layer, with Reliable Data Protocol (RDP) with sequenced packet delivery being used on the wireless link, and UDP being used for communication with the wired portion of the system. RDP provides increased protection against potential errors on the wireless link and is only deployed where it is needed.

M-RPC raises issues regarding location management, through its subject-based naming approach, as well as connectivity management through the use of 2 separate transport protocols on different parts of the network.

3.5.6 Efficient Implementation of Java RMI

When applications of an interactive nature are developed using RMI, quick response times are essential. Given the poor performance of the RMI protocol, and the high latencies present in wide area, heterogeneous network environments, an efficient implementation of

RMI is needed to support such distributed, interactive applications. Such concerns are equally valid in the context of a wireless network where bandwidth is typically low and latencies high.

RMI currently employs a communication protocol that uses TCP, which, being a connection-oriented protocol, requires explicit acknowledgement of data transmissions. Krishnaswamy et al. argue that since most RMI communication falls into the request-response model, explicit acknowledgements could be avoided by a transport protocol that was aware of the underlying structure of RMI communication [KWB+]. To this end, a reliable message delivery protocol based upon UDP (named R-UDP) is developed. R-UDP achieves reliable data transfer by way of using server replies to client requests as a form of implicit acknowledgement of message receipt. Timeouts are used in conjunction with this approach to request explicit acknowledgement of message delivery if required. The implementation of such a protocol in a wireless environment would have the advantage of reducing data traffic over the expensive and unreliable wireless link.

Krishnaswamy et al. combine the R-UDP protocol with an object-caching strategy to further improve the performance of RMI. Caching is often used to improve performance within a distributed system, since local access is invariably cheaper than remote access, although caching within a wireless mobile device environment is hampered by memory constraints inherent in such devices. The caching strategy developed by the authors makes use of a reliable multicast framework developed for the purposes of consistency enforcement amongst cached objects.

The results of performance tests conducted on the transport protocol and caching strategy showed that whilst R-UDP resulted in better round-trip message times at the transport level than TCP, it did not provide better execution time at the invocation level due in part to the use of multiple synchronized threads to handle message transmission. Performance testing with regards to caching revealed that performance benefits were dependant upon the pattern of access to the remote objects. For instance, an invocation that only read a cached copy of an object resulted in performance gains, whereas a write operation caused invalidation traffic to be sent in which case performance degraded from a non-caching model.

Whilst a more efficient transport protocol is highly desirable for RMI especially within

a wireless environment, whether or not to use some form of caching strategy depends upon the typical pattern of access to objects as well as physical device considerations. In the case of clients running on mobile, resource-constrained devices, caching may not be viable.

3.5.7 A Distributed Virtual Machine Architecture For Mobile Java Applications

The Ghost Machine platform allows the operation of a limited subset of Java on mobile computing platforms such as a Palm Pilot [Sea98]. Such devices have severe resource constraints which impact upon their ability to run Java bytecode. For instance, at the time of development of the Ghost architecture (1998), Palm devices were limited to only 1 megabyte of memory for both data and applications and ran on a 17 MHz 32-bit processor. PersonalJava, the subset of the language intended for mobile devices requires system resources far in excess of those offered by a Palm Pilot in terms of both memory and processor-speed.

The Ghost architecture overcomes the limitations imposed by mobile devices, specifically the Palm Pilot, by moving some of the resource-intensive functions of the Java Virtual Machine (JVM), off the mobile device and onto a desktop system. Only those services termed 'up-front' services, indicating they are services that are only performed once prior to program execution, are distributed to the desktop environment. The up-front services removed from the mobile device in the Ghost architecture are class verification and processing. In addition, the memory footprint of a Java class is reduced through the transformation of class type information from a String representation to an integral data type. This enables the operation of a 'lite' Virtual Machine upon the mobile device, although pre-processing of the Java class is required on a desktop server prior to class installation onto a mobile device. In effect, the decoupled Virtual Machine behaves similarly to a just-in-time compiler (JIT), which compiles to machine code before execution, as it serves to compile java source to PalmOS binaries.

With the rapid advances that have been made in handheld computing hardware, such limitations as imposed by the Palm Pilot are no longer the norm. For instance Compaq's iPaq PDA boasts a 207 MHz processor combined with 32 MB of RAM and 16 MB of ROM, well within the PersonalJava system requirements. However, the Ghost machine

architecture clearly illustrates the use of Java upon a resource constrained, mobile device and lays the way for possible optimisation techniques for code in such environments. It is interesting to note that the Java 2 Micro Edition (J2ME) offering from Sun as described in Section 2.2 makes use of just such optimisation techniques in the form of bytecode preverification prior to application execution on the device.

3.5.8 Interceptors for Java Remote Method Invocation

An interceptor is a software mechanism that provides the hooks that are needed to introduce additional code dynamically into the execution path of an application [N.]. The authors use interceptors to introduce additional services such as logging, into the RMI runtime, transparently to the application and without the need for revision or recompilation of client code.

Three possible interception points, that is positions in the RMI architecture at which RMI calls may be intercepted, were identified. The first interception point occurs at the Stub / Skeleton proxy level of the RMI architecture and exploits the functionality of the Java Dynamic Proxy API. Following development of an interceptor at this point, a dynamic proxy implementing the same set of interfaces as the RMI object it is intercepting communication for, is developed. This dynamic proxy is transparently returned to a client requesting a reference to the RMI object, and all calls dispatched through it where they may be processed before being forwarded to the real object.

The second interception point identified is at the transport layer of the RMI architecture and involves the development of custom socket types which are provided to the RMI runtime in place of standard Java sockets. These custom socket types intercept RMI calls passing through them, and perform additional processing on the call. This interception point is at the Java API level (through the need to specify alternative socket factories to the RMI application) and is not transparent to the application.

The third interception point introduces interception at a lower point in the RMI architecture, that of the shared native library containing the socket functions RMI uses to facilitate communication. RMI makes use of platform specific socket implementations by way of a shared library which is loaded into the RMI runtime. By providing a custom shared library which intercepts and processes socket communication, it is possible to

provide interception of RMI calls at the shared library level. Replacement of the shared library and subsequent interception may be carried out transparently to the application.

This project comprehensively examined the interception of RMI calls using a number of mechanisms, all of which are represented in our final solution.

3.6 Summary

This chapter examined the issues introduced into a distributed system through its operation in a wireless environment and went on to examine the advances in wireless mobile telecommunications which are likely to drive the adoption of mobile computing. A number of research projects dealing with the operation of RMI in a wireless environment and on small devices were also examined.

Chapter 4

Design

This chapter describes the design of a scheme to support the operation of RMI clients and servers in a mobile environment. The design was performed with the objective of retaining as much of the existing ALICE architecture as possible, in order to reinforce the position of ALICE as a generic architecture for the operation of a range of application protocols in a mobile environment.

This chapter is divided into two major themes, one being connectivity management for RMI applications in a mobile wireless environment, and the other being location management for RMI applications in such an environment.

The design of the integration of the existing application independent ALICE Mobility Layer, which provides connectivity management is presented first. Connectivity management provides solutions for the problems encountered in a wireless network such as frequently broken transport connections, limited bandwidth and tunnelling of connections after handoff.

The way in which RMI handles remote object references is then examined in detail with a view to designing a form of the Swizzling scheme employed by ALICE, for RMI. It was found that such a scheme would require significant changes to be made to the RMI source code and an alternative location management solution based on dynamic proxy objects was developed for mobile RMI servers.

4.1 Mobility Layer

The Mobility Layer is the layer of the ALICE architecture that provides connectivity management between the Mobile Host and the Mobility Gateway and serves the functions of transparent reconnection of broken transport connections and tunnelling of existing connections after a Mobile Host has changed the Mobility Gateway to which it is connected. Implementing a superset of standard BSD sockets, it is the Mobility Layer that replaces standard TCP in mobile applications.

Currently, two implementations of the Mobility Layer exist, the original implementation in C, and an implementation coded in Java. One of the design goals of this project was to create some form of glue code layer or equivalent, through which Java applications may access and use the functionality of the C Mobility Layer. This was motivated by the desire to reinforce the position of the ALICE architecture as a generic Architecture for Location Independent Environments and to make use of the language independence of the C Mobility Layer.

The existing Mobility Layer (ML) is composed of two distinct parts, that part resident on the Mobility Gateway (ML_{MG}) and that resident on the Mobile Host (ML_{MH}). The two components communicate via a single transport layer connection and exchange Mobility Layer Protocol Data Units. These two components collectively implement *sockets+*, a superset of the BSD sockets API which overrides the existing BSD socket functions and introduces new functionality. The replacements of the standard socket functions are used in ALICE mobile systems to provide connectivity management in a mobile environment. Such support is sufficient for systems in which application level knowledge of mobility is not required. When application level knowledge of mobility is required[1], ALICE provides a facility to provide callbacks from the Mobility Layer to the application, providing higher layers with information about the current state of the wireless link. The additional socket functions allowing for the registration and deregistration of callback functions represent the + in the ALICE sockets+ API.

[1] In what are termed *mobile-aware* applications

4.1.1 Previous Work on the Mobility Layer

The Java version of the original Mobility Layer was implemented by Mark Corbet [Cor00] in his work on ALICE and JINI, and further extended by Thomas Wall [Wal00] in his work on ALICE and RMI. The Java Mobility Layer was developed to avoid the numerous complexities associated with the development of the glue code layer between Java applications and the C Mobility Layer. The Java Mobility Layer introduced significant overheads into the RMI system and required the operation of distinct implementations of the Mobility Layer for the IIOP and JRMP protocols. Operation of the Java Mobility Layer caused a remote invocation to be on average an order of magnitude slower than a normal RMI invocation. The Java Mobility Layer did, however provide the same functionality to JRMP as the C Mobility Layer did to IIOP.

This project revisited the original goal of integrating the application independent version of the Mobility Layer coded in C, into the RMI system.

4.2 Integrating the Mobility Layer into Java RMI

In order to reinforce the position of ALICE as a generic Architecture for Location Independent Environments, it is necessary to reuse as much of the original implementation as possible across different application protocols. It is thus highly desirable that the existing Mobility Layer, designed to be independent of application protocol issues, be integrated into the RMI runtime.

4.2.1 The Java Native Interface

The Java Native Interface (JNI) is an API provided by Sun which allows Java code and native code (such as C) to interact with each other. In the context of the ALICE Mobility Layer, it was decided to use JNI as a means with which to interface the RMI runtime (coded in Java), with the existing C Mobility Layer.

The JNI is normally used in one of two ways -

1. To make use of platform specific functionality that may only be provided by native code

2. To interface with legacy native code from Java

In many ways, the first use of the JNI is the preferable option. The native functionality is often not implemented prior to being required and may be developed bearing in mind that it will be accessed by Java code. In this case, the procedure for developing the native interface is as follows :

1. Write the Java method declarations, declaring the native methods with the `native` keyword and deferring implementation of the method.
2. Compile the Java class with the standard Java compiler.
3. Run the `javah` tool against the class file generated by the compiler to produce C function prototypes for the native methods. These are stored in a .h header file.
4. Write the native implementation of the function prototypes generated by `javah`.

The advantage of this approach is that the function prototypes are automatically generated and then implemented with special consideration being given to the design of data structures and data types within the native code, knowing that the structures will be accessed from Java.

The second use of the JNI, and the way in which it was used in this project, is slightly more complex due to the need to write an interface to C code that has been developed with no intention that it would be accessed from Java. The function prototypes already exist and it is necessary to construct Java method declarations around these prototypes, or to rewrite the prototypes and hence the implementation too. Additionally, if the existing function prototypes are to be retained, each prototype will need to be altered to include arguments specific to the native interface. This is an extremely difficult and error prone process.

4.3 Replacing the Java Socket Implementation

There are two ways in which the C Mobility Layer could possibly be integrated into the RMI system using JNI and although both ways rely on the ability to specify an alternative

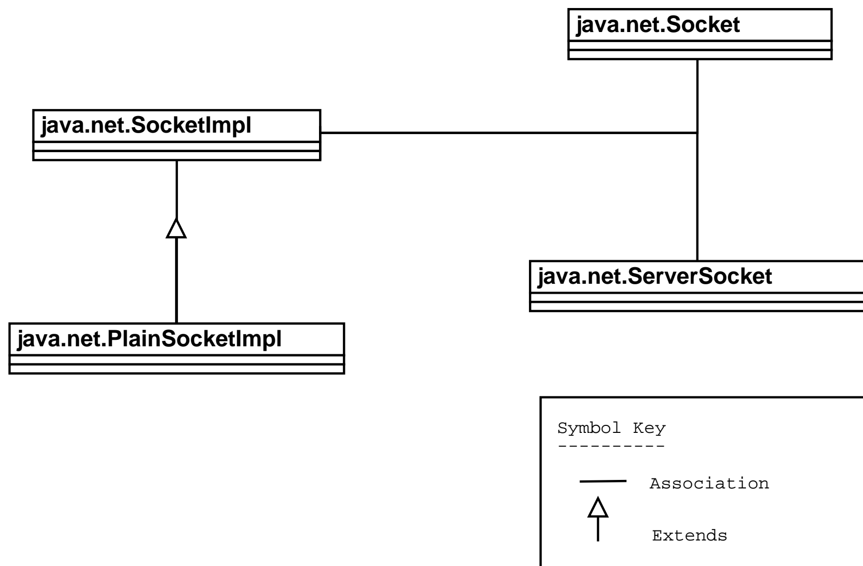

Figure 4.1: The relationship between sockets and their implementation in Java

transport to the RMI system by way of Socket Factories, there are differences in how the Java socket replacement classes are developed.

Before attempting to replace the Java socket functions, it is important that we understand how sockets are implemented in Java. The Java networking package provides the `java.net.Socket` and `java.net.ServerSocket` classes which provide client and server communication endpoint functionality respectively. By default JRMP uses instances of these socket classes to facilitate communication between remote objects, although it is possible to specify extensions of these classes to the RMI system, as discussed in Section 2.1.6.

Both the `java.net.Socket` and `java.net.ServerSocket` classes contain a `java.net.SocketImpl` attribute, which is an abstract class representing the socket implementation. The socket implementation defaults to the `java.net.PlainSocketImpl` class. This socket implementation class handles the dispatch of calls to the socket functions implemented in native code. This is done through JNI calls made by the `java.net.PlainSocketImpl` class and dispatched to a shared library, (`libnet.so` on Linux, `net.dll` on Windows).

The relationship between the client and server sockets and the socket implementation class is illustrated in Figure 4.1. Both `java,net.Socket` and `java.net.ServerSocket` have a `java.net.SocketImpl` attribute which defaults to an extension of this class, `java.net.PlainSocketImpl`.

Given the way in which sockets are implemented in Java, there are two ways in which the Mobility Layer socket functions may be integrated in Java at the native library level.

```
alice.rmi.ALICESocketImpl
+connect(host:String,port:int): void
+connect(address:InetAddress,port:int): void
+connectToAddress(address:InetAddress,port:int): void
+doConnect(address:InetAddress,port:int): void
+bind(address:InetAddress,lport:int): void
+listen(count:int): void
+accept(s:SocketImpl): void
-socketCreate(isServer:boolean): native void
-socketConnect(address:InetAddress,port:int): native void
-socketBind(address:InetAddress,port:int): native void
-socketListen(count:int): native void
-socketAccept(s:SocketImpl): native void
-socketAvailable(): native int
-socketClose(): native void
+add_callback(sockID:int,CBF:long): int
+delete_callback(): int
```

Figure 4.2: The `alice.rmi.ALICESocketImpl` class

4.3.1 Creating a custom Socket Implementation

Firstly, it is possible to write a custom socket implementation class other than the default `java.net.PlainSocketImpl`, say `alice.rmi.ALICESocketImpl`, which accesses a custom shared library, say `libALICEnet.so`. The shared library would be constructed from the Mobility Layer socket functions with appropriate JNI method signatures, to present a similar interface to the `libnet.so` library.

The class diagram for the `ALICESocketImpl` class is illustrated in Figure 4.2, and is very similar to the `java.net.PlainSocketImpl` class. One of the major differences between the implementations, and one that is not evident from the class diagram, occurs in the loading of the shared library. The call to load the shared library containing the platform-specific socket functions is altered to rather load the Mobility Layer socket functions, thus providing these to the RMI runtime in place of standard sockets. Another difference is the inclusion of the sockets+ functions in the `alice.rmi.ALICESocketImpl` class.

Custom extensions of the `java.net.Socket` and `java.net.ServerSocket` classes could then be developed which use the `alice.rmi.ALICESocketImpl` socket implementation rather than the `java.net.PlainSocketImpl`, and consequently use the Mobility Layer socket functions. RMI Socket Factories which produce sockets of these types for the RMI runtime system are the final step in this process. A class diagram representing the relationship between the replacement classes is illustrated in Figure 4.3.

This approach would require the specification of a custom RMI socket factory in the RMI application through the `setSocketFactory()` method in the standard RMI API, in order for the RMI application to make use of the Mobility Layer.

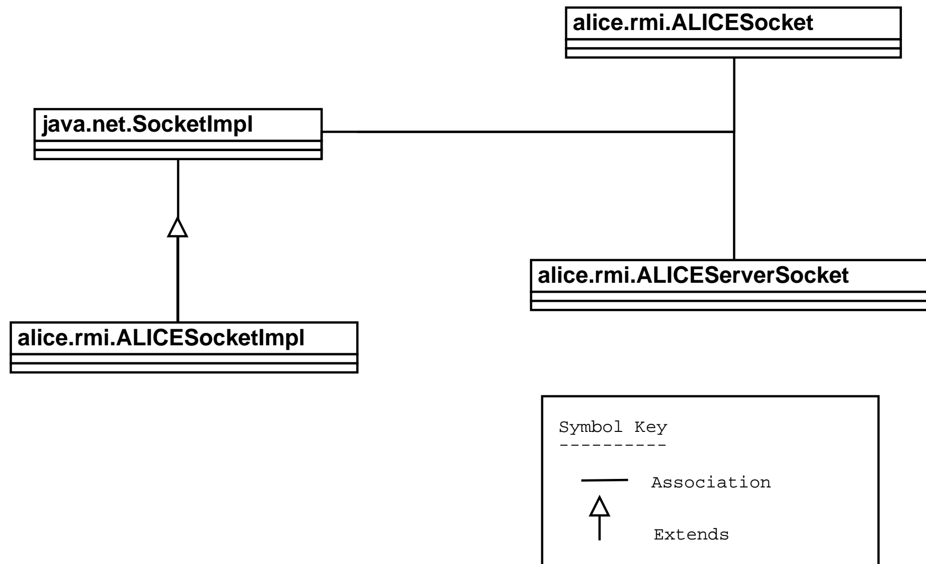

Figure 4.3: The relationship between the replacement socket classes

4.3.2 Replacing the default shared library at runtime

Another approach to replacing the standard native socket calls is to create a shared library with the same exposed external interface as the default `libnet.so` library, but which delegates calls to the Mobility Layer socket functions rather than the standard native socket functions. It is possible to specify, to the Java runtime library loader, the path from which to load libraries. By altering this path to load the altered `libnet.so` library at runtime, the standard socket functions may effectively be replaced by the ALICE socket replacement functions.

The advantage of this approach is that the standard socket functions could be replaced with the ALICE Mobility Layer socket functions transparently to the application, without the need to alter legacy code. This approach does however have significant disadvantages in that the ALICE Mobility Layer does permit for mobile aware operation and provides an extended API - sockets+ for mobile-aware operation. The sockets+ API would not be exposed by simply replacing the `libnet.so` library. This approach is not considered further for these reasons.

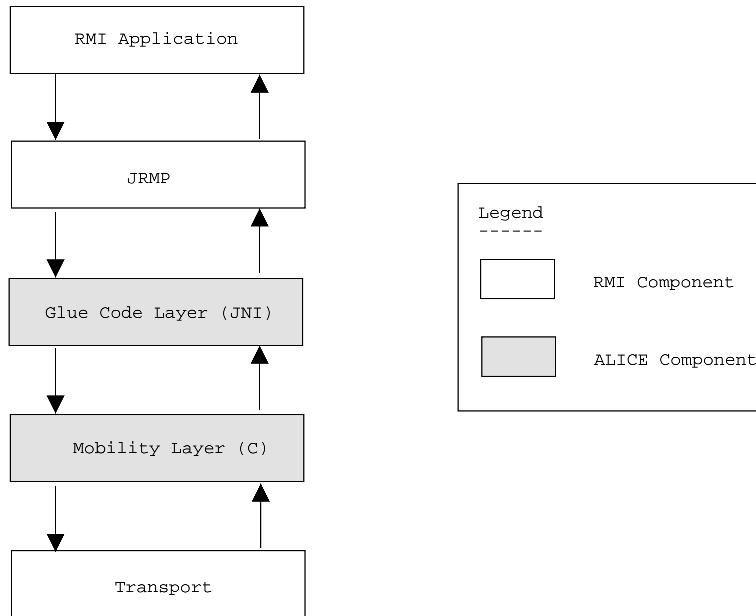

Figure 4.4: The introduction of a glue code layer

4.4 A Glue Code layer

Another way in which the default socket implementation in Java may be changed is to introduce an additional glue code layer between the JRMP and the transport layer. This would involve the development of a Java interface file to expose the existing C Mobility Layer API to Java. The Java interface file would contain native method declarations developed with JNI and would dispatch calls to these methods to the underlying C Mobility Layer socket functions. The position of the glue code layer is illustrated in Figure 4.4

A Java socket class extension could then interact with the glue code layer by way of a shared library encapsulating the C socket functions and JNI wrapper functions.

Following this approach, two distinct interface files are created from each of the C source files, a Java interface file presenting Java declarations of the native functions, and a C wrapper file that wraps each of the C functions in a form that may be called by Java code. The Java interface file accepts calls from Java code, dispatching them to the C wrapper code. The wrapper code has the responsibility of dispatching the call to the underlying native implementation. It is envisaged that this design will require minimal, if any, modification to the existing C code. The way in which the Java interface file and C wrapper file will collectively provide interaction with the Mobility Layer is illustrated

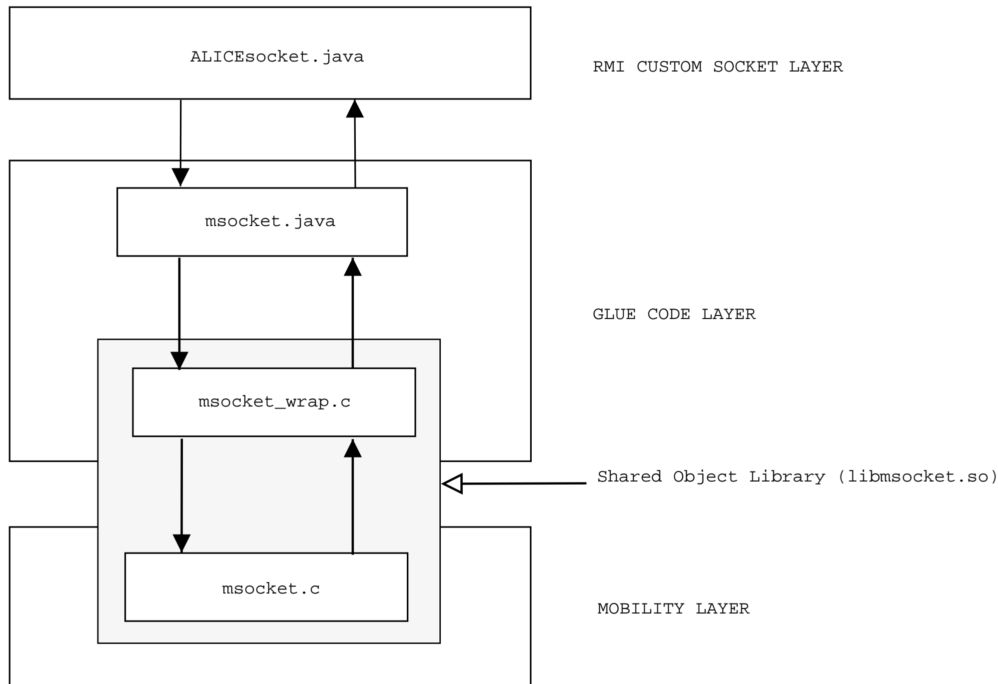

Figure 4.5: The glue code layer in detail

in Figure 4.5 for an imaginary C file called `socket.c`. In this scenario, the custom socket, `ALICEsocket.c`, interacts with the native socket implementation through the Java interface `msocket.java` which in turn accesses the shared object library, `libmsocket.so` generated from the `msocket_wrap.c` wrapper class and the native implementation itself, `msocket.c`.

The difference between this approach and the creation of a custom socket implementation lies in the placement of the calls to the native socket functions. Whilst creating an extension of `java.net.SocketImpl` keeps the calls to the native socket functions at the `java.net.SocketImpl` level, the introduction of the glue code layer moves the calls to the native socket functions to the `java.net.Socket` and `java.net.ServerSocket` level. This approach has the advantage of not requiring knowledge of the `java.net.SocketImpl` class which is not documented in the public domain, unlike the Java socket classes.

4.4.1 Selected Method

The method selected of integrating the ALICE Mobility Layer into the RMI runtime system was the creation of a custom socket implementation as discussed in Section 4.3.1. This approach was considered the most closely aligned with the goals of the Java lan-

guage, where new socket implementations are expected to be developed to fulfill certain requirements. Following this approach, the custom Socket and ServerSocket classes will maintain a socket implementation attribute of the type `alice.rmi.ALICESocketImpl`. The introduction of a glue-code layer would have required significant implementation knowledge to be introduced at the Socket / ServerSocket level through native calls, whilst in the original Java socket implementation, knowledge of native code is not introduced at this level. Changing the socket implementation rather preserves the Object Oriented goals of the Java language.

4.5 Mobile Host as Client

When the Mobile Host is acting as a client of a remote server object, the support offered by the Mobility Layer in the form of connectivity management is sufficient. The Mobility Layer support is integrated with the RMI system through specification of custom RMI client and server socket factories which manufacture socket objects which make use of the replacement shared library to access the Mobility Layer functionality. Although the specification of socket factories to the RMI system is not explicit in most applications, it is semantically correct, part of the RMI API and should not present any great challenge to the application programmer. It does require slight modifications to be made to client code to provide for mobility support.

The operation of the Mobility Layer hides the vagaries of the wireless network from higher layers by providing Connectivity Management in the form of transparent reconnection and tunnelling and optionally provides state information to higher layers.

The Java Mobility Layer operated with the proviso that the client did not pass references to remote objects on the Mobile Host, as arguments to a call made to the server [Wal00]. If the remote object received such a reference and then attempted to invoke a method on the object, the client on the Mobile Host is then acting as a server too. It is possible to remove this restriction by ensuring that any references to remote objects on the Mobile Host are changed before they are passed as arguments to the server on the fixed network. This introduces the need for some form of location management and more specifically a form of managing the references exported by remote objects resident

on Mobile Hosts.

4.6 Mobile Host as Server

Addressing problems arise when a server is resident on a Mobile Host. Servers traditionally export a reference to themselves so that clients know where they may be contacted. CORBA servers export Interoperable Object References (IORs), whilst RMI servers export `java.rmi.server.RemoteRefs`. These references contain an IP address and port number combination referring to the machine on which the server is hosted. In a mobile environment, a Mobile Host is not directly contactable and all communication to and from it must pass through a Mobility Gateway. Thus, if a server is hosted on a Mobile Host and exports a reference based upon the Mobile Hosts address, the reference is invalid (since the host is not directly contactable at its address) and any attempts to invoke the server using this reference will fail. It is necessary to change the reference exported by a mobile server to refer to the Mobility Gateway to which it is attached, rather than the Mobile Host itself. It is the process of changing the endpoint of a server reference to point to the Mobility Gateway rather than the Mobile Host, that is known as Swizzling in ALICE. Unswizzling, or the restoration of the original endpoint value to the reference is performed at the Mobility Gateway in order to determine the Mobile Host for which communication is intended.

4.6.1 Remote Object Referencing in Java

Every host that hosts objects available for remote invocations, in an RMI system, must run a registry process. The rmiregistry is a simple bootstrap name server from which a reference to a remote object on a given host may be obtained by a client The reference that a client receives from the registry is, in most cases actually to a stub and not the object itself [2]. As illustrated in Figure 4.6, the stub contains a `sun.rmi.server.UnicastRef` attribute, which in turn contains a `sun.rmi.transport.LiveRef` attribute, that contains a `sun.rmi.transport.Endpoint`, giving the hostname and port number at which the object

[2] If the server resides in the same VM as the client, a reference to the actual object will be returned rather than a stub

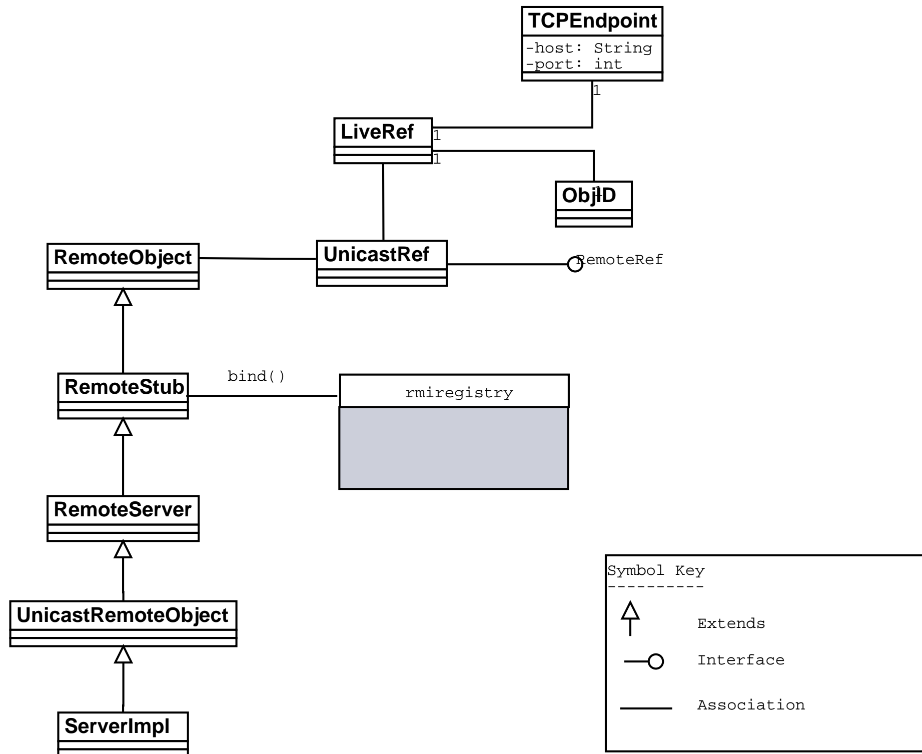

Figure 4.6: Inheritance hierarchy in RMI

resides, and a unique identifier in the form of an object ID. This is similar to the information stored in an IOR profile in CORBA. In addition to the retrieval of a remote object reference by way of lookup on an rmiregistry, there are two further ways by which an RMI client may obtain a reference to a remote object :

- **As the return value from a method invocation**

  A method invoked by a client on a remote object may return, as a result of the method invocation, a reference to a remote object. The client may then invoke methods on this object as if it resided locally. The registry lookup essentially obtains a reference as a result of the invocation of the lookup method. The reason that it has been classified separately here is that the retrieval of a reference by way of a registry lookup involves an explicit request for a remote reference being made by a client and thus is accounted for separately in the invocation redirection scheme.

- **As an argument to a method called on the client**

  The third way in which a client may obtain a reference to a remote object, is as an argument to a method invoked on the client, by the server. This case arises when

the client is itself an RMI server object too. In such a scenario, an RMI server may make a callback to the client and may potentially pass it a reference to a further remote object.

As is the case with the registry lookup, what is in most cases returned is a reference to the stub representing the object, rather than the object itself.

Consequently, the design of a location management scheme needs to take into account the three ways in which a reference to a remote object may be obtained in RMI. The way in which each of these cases is handled is discussed in the context of possible strategies.

Several strategies were considered to accomplish Location Management for mobile RMI objects as provided for in ALICE and these are discussed below.

4.6.2 Extending UnicastRemoteObject

Most server classes in RMI extend the `java.rmi.server.UnicastRemoteObject` super class [3]. An option available for the swizzling of remote object references is to enhance the `java.rmi.server.UnicastRemoteObject` class and delegate responsibility for the changing of remote object references to the enhanced class. In this way responsibility for swizzling object references is moved from the Mobility Gateway to the client side. This approach is similar to that adopted by the developers of the MobileRMI toolkit [MAne] which included a class named `MobileUnicastRemoteObject` which in turn included a method to move a remote object from one remote address space to another and obtain the new remote object reference via a `getNewLiveRef()` method. The updating of remote object references is then performed by the instance of the `MobileUnicastRemoteObject` subclass resident on the client.

This approach of updating remote object references at the client is not aligned with the ALICE approach of performing swizzling at the Mobility Gateway, completely transparently to the client. This approach would require modifications to be made to the syntax of obtaining and maintaining a remote object reference by the client, and thus is not considered further.

[3] It is possible for a server object not to extend this super class and explicitly handle exportation of itself to the RMI runtime

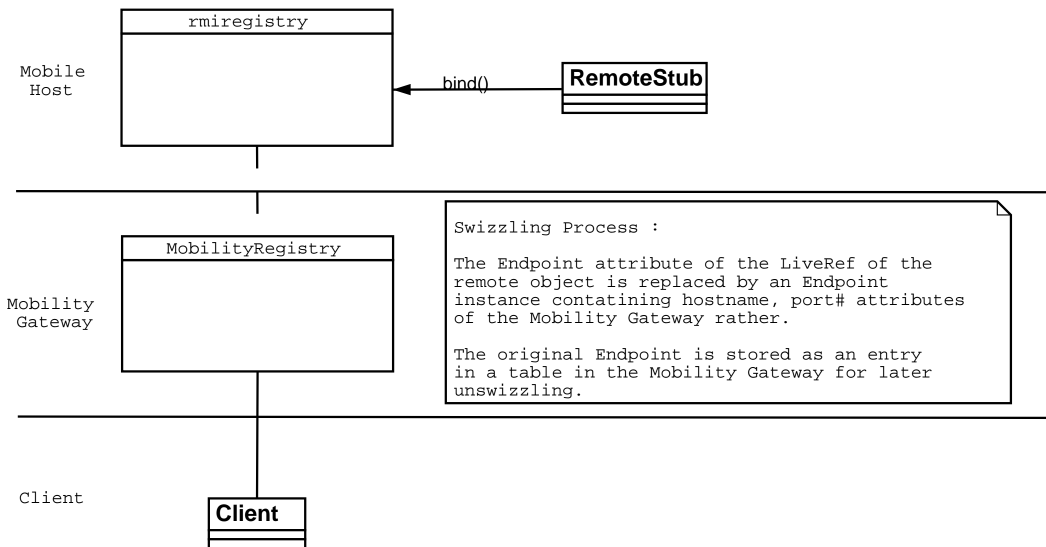

Figure 4.7: The introduction of a Mobility Registry on the Gateway

4.6.3 Perform Swizzling at the Gateway

With this option, the client would continue to obtain a reference to a remote object by way of a registry lookup on the name of the object. The lookup would necessarily be done on an rmiregistry resident on the Mobility Gateway, since the client has no way of directly contacting the server. The Mobility Gateway would run a modified version of the rmiregistry that would perform address manipulation when a reference to a remote object resident on a Mobile Host is requested by lookup. The introduction of a Mobility Registry is illustrated in Figure 4.7.

The swizzling process will consist of changing the hostname, port no attributes of the remote object's sun.rmi.transport.Endpoint attribute to refer to the Mobility Gateway instead of the Mobile Host. This swizzled reference will then be returned to the requesting client process. The modified rmiregistry will maintain an internal table matching object ID's to their original endpoint values. Upon receiving an invocation on a swizzled reference, the registry will unswizzle the reference, restoring the original endpoint value and forwarding the invocation on to the Mobile Host. The invocation response will likewise be relayed back to the client.

This approach requires a client wishing to obtain a reference to a remote object to do a lookup on a Mobility Gateway, which although syntactically identical to obtaining a reference to a 'regular' RMI object, is semantically different and may require minor

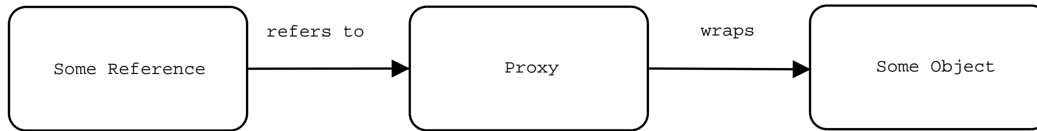

Figure 4.8: Interaction between an Object reference and a Dynamic Proxy class

modification of legacy client code that contains hard-coded registry lookups. The final solution to the problem of location management does maintain lookup support and leaves the responsibility of changing object references on the Mobility Gateway, but manipulates the address of the reference upon registration of the remote object with the gateway.

4.6.4 Using Dynamic Proxy Classes

Since the `sun.rmi.transport.LiveRef` and `sun.rmi.transport.Endpoint` types are specific to Sun Microsystems' RMI implementation, they are not guaranteed to be present in all Java implementations and any approach to swizzling relying on these types will not necessarily be compatible with other RMI implementations. Furthermore, at present only a TCP based transport implementation is available in the Java Development Kit (JDK), so the future introduction of alternative transport mechanisms may render these incompatible with the Java ALICE implementation if the solution were implemented at this level without a swizzling mechanism being developed for each new transport. At the same time, any changes made to internal Java classes would render the system incompatible with other implementations.

A third option available for the redirection of method invocations from the Mobility Gateway to the Mobile Host is in the form of dynamic proxies available in Java 1.3. Dynamic proxies provide the ability to create a class, at runtime, that implements a list of interfaces, this being a dynamic proxy class. An instance of the class is a proxy, which is able to handle invocation of the interfaces represented by the proxy class. An invocation on an instance of the class will be dispatched to another object through a uniform interface as illustrated in Figure 4.8. Method invocations are then handled by an implementation of the `java.lang.reflect.InvocationHandler` interface, which if present on the Mobility Gateway, could forward the invocation onto the Mobile Host attached to it. It is possible to remove all knowledge of creating the proxy from the client code which

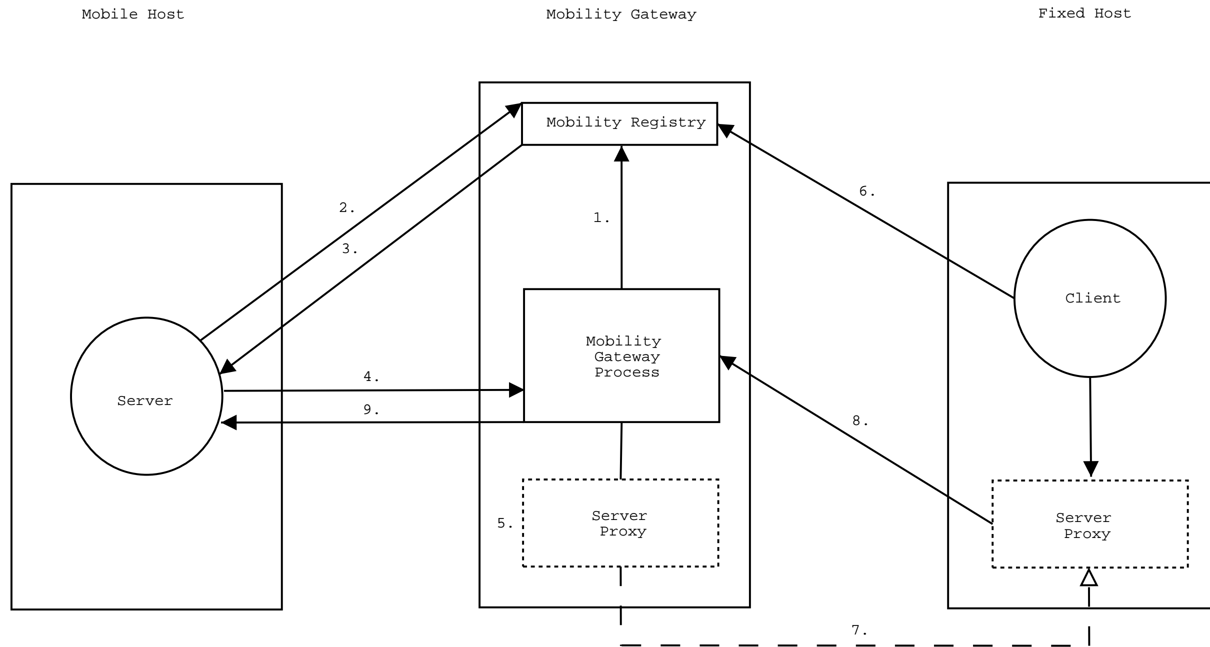

Figure 4.9: Dynamic Proxy architecture

would enable client-side transparency in the process.

Following the dynamic proxy approach, a dynamic proxy object, say a `ServerProxy`, could be developed for the Mobility Gateway. The proxy object takes a generic `Object` as an argument to its constructor and then uses reflection to determine what interfaces the object implements. Each object hosted by a Mobile Host needs to bind to the registry running on the Mobility Gateway that it is currently connected to. What is actually bound to the registry, however, is a dynamic proxy object transparently implementing the same remote interface as the remote object. This `ServerProxy` object is bound to the registry on the Mobility Gateway and is what is actually returned to clients performing a lookup against the remote object name. The `ServerProxy` also maintains a reference to the Mobility Gateway from which it originated and any invocations made on the proxy are propagated firstly to the Mobility Gateway and then on to the actual remote object, the location of which is known to the Mobility Gateway. Invocation responses are likewise propagated back through the Mobility Gateway. The operation of the scheme is presented in more detail in Figure 4.9, for the case of a client obtaining a reference to a remote object by way of a registry lookup. The dynamic-proxy based scheme operates as follows :

1. The Mobility Gateway process binds to the Mobility Registry daemon on the Mo-

bility Gateway host, using a well-known name.

2. A remote object resident on a Mobile Host performs a lookup, against the well-known name, on the Mobility Gateway to which it is attached.

3. A reference to the Mobility Gateway process is returned to the remote object.

4. The remote object invokes the `register()` method on the Mobility Gateway process, passing itself as a parameter.

5. The registration process creates a ServerProxy object on the Mobility Gateway which implements the same set of interfaces as the remote object.

6. A client performs a lookup against the server name, on the Mobility Gateway.

7. A reference to the ServerProxy, rather than to the actual remote object is returned to the client.

8. Any invocations made against this reference are forwarded to a single method within the ServerProxy and then onto the Mobility Gateway process.

9. The Mobility Gateway process then forwards the invocation onto the actual remote object resident on the Mobile Host, and returns the result along the same path back to the client.

This approach to redirection of remote invocations from Mobility Gateway to Mobile Host using proxies is achievable with no changes necessary to the existing RMI architecture and for this reason is one of the most attractive solutions to the location management problem. Interaction diagrams for a number of likely scenarios encountered using this approach are presented below

Reference obtained as return value of invocation

This scenario represents the case where a client receives a reference to a remote object by way of the return value of a method invocation, and is illustrated in Figure 4.10

1. The client invokes a method on a remote object (s1) and the invocation is forwarded to the Mobility Gateway to which the server is currently connected.

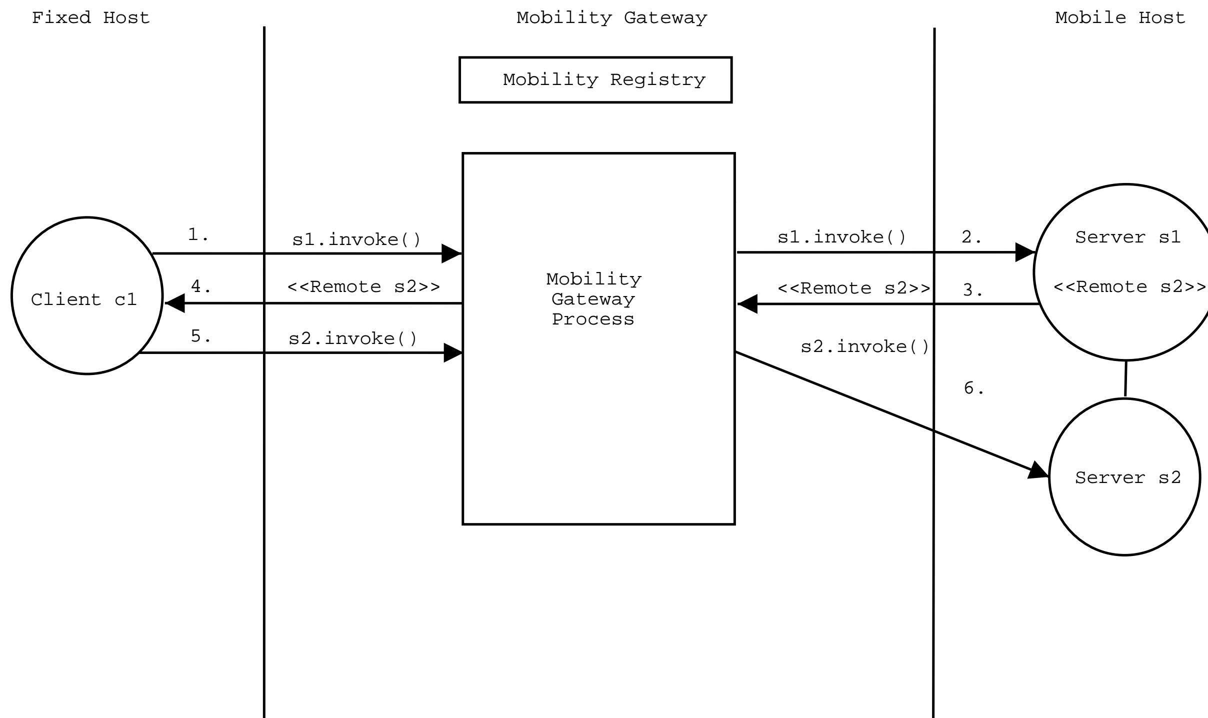

Figure 4.10: Reference obtained from invocation

2. The Mobility Gateway forwards the invocation on to the remote object (s1) hosted on a Mobile Host.

3. The return value of the invocation, a reference to another remote object (s2), hosted by the Mobile Host, is returned to the Mobility Gateway. At the Mobility Gateway, the return value is inspected and since it is a reference to a remote object, a dynamic proxy representing the object is created and registered with the Mobility Gateway (relieving s2 of the need for explicit registration with the gateway).

4. The proxy object created in 3 is returned to the client as the return value of the invocation.

5. A method invocation on the proxy representation of s2, from the client, is forwarded to the Mobility Gateway.

6. The Mobility Gateway forwards the invocation to the real remote object.

The rmiregistry process is not necessarily involved in the process, although it may have been involved in obtaining the reference to s1.

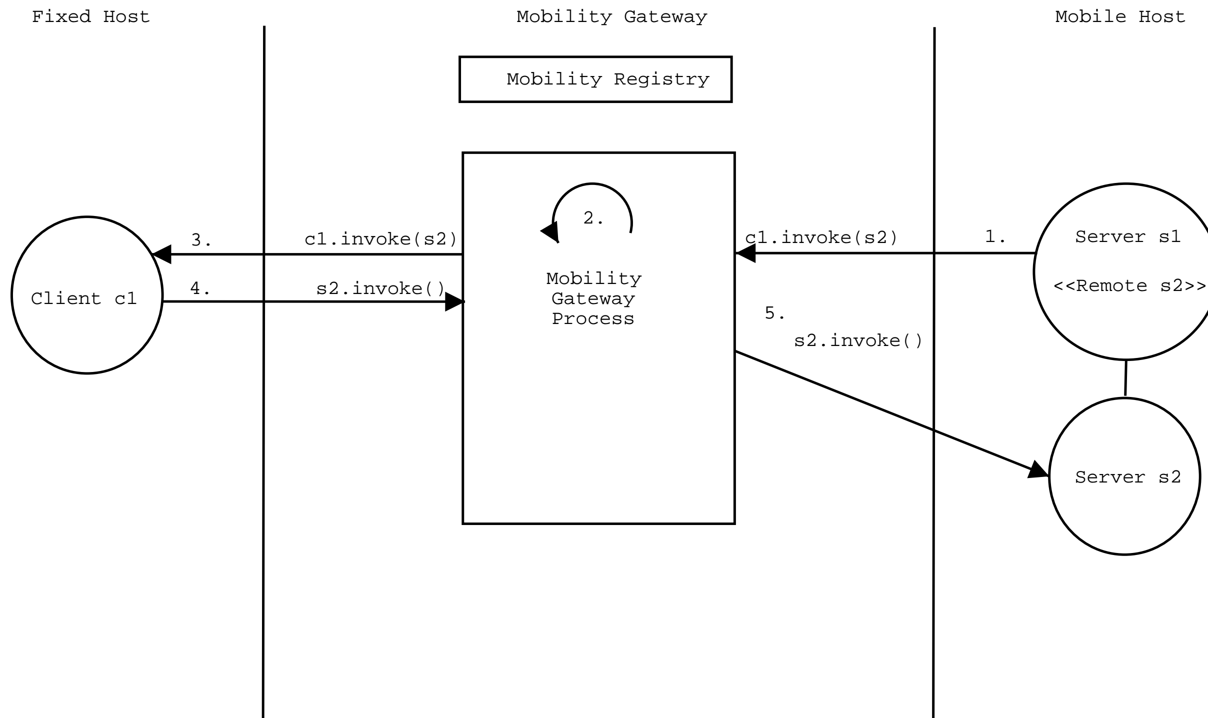

Figure 4.11: Reference obtained as argument

Reference obtained by argument

This scenario represents the third way in which a client may obtain a reference to a remote object and how this is handled by the proposed location management strategy. The sequence of events is illustrated in Figure 4.11.

1. A remote object s1 makes a callback to client c1, passing a reference to s2 as an argument and this is forwarded to the Mobility Gateway.

2. The method arguments are inspected at the Mobility Gateway, and when it is determined that a remote object is present as an argument, a dynamic proxy is created and registered with the Mobility Gateway and this proxy replaces the original argument.

3. The invocation, with the proxy argument is forwarded to the client.

4. An invocation made on this proxy (s2) is forwarded to the Mobility Gateway.

5. The invocation is then forwarded to remote object s2.

The remote object originally received a reference to the client as an argument of a remote call made by the client, on the remote object. The argument inspection scheme as described above, ensures that the client receives a proxy representation of the remote object.

Handoff

The handoff of a Mobile Host from one Mobility Gateway to another occurs when the Mobile Host moves out of the coverage range of the first Mobility Gateway and into the range of a second one. The handoff procedure proceeds at the level of the Mobility Layer and involves the setting up of a connection, from the old Mobility Gateway to the new one, for each of the logical connections open between the Mobility Gateway and the Mobile Host. Any existing transport connections between Fixed Hosts and the Mobility Gateway are then tunnelled via the old Mobility Gateway, to the new Mobility Gateway and onto the Mobile Host, for the remainder of their lifetime. Handoff is implemented in the Mobility Layer and thus is available to RMI via the integration of the Mobility Layer, with no further development required. The handoff procedure does however affect location management in that references held by clients to mobile servers need to be updated to reflect the new Mobility Gateway. The way in which this is solved is by a procedure termed Reswizzling in ALICE.

Reswizzling

Reswizzling in the ALICE architecture refers to the changing of an already swizzled object reference to point to a Mobility Gateway to which the mobile server has moved since the original swizzling of its object reference.

Reswizzling is necessitated by the handoff of the Mobile Host to a different Mobility Gateway and is initiated by way of a callback received from the Mobility Layer indicating that the address of the MG has changed. In the ALICE CORBA implementation, this callback causes the Swizzling layer on the Mobile Host to replace all endpoints referring to the original Mobility Gateway, with endpoints referring to the new Mobility Gateway. The JRMP/R layer implements the following solution which is equivalent to that posed by reswizzling in ALICE.

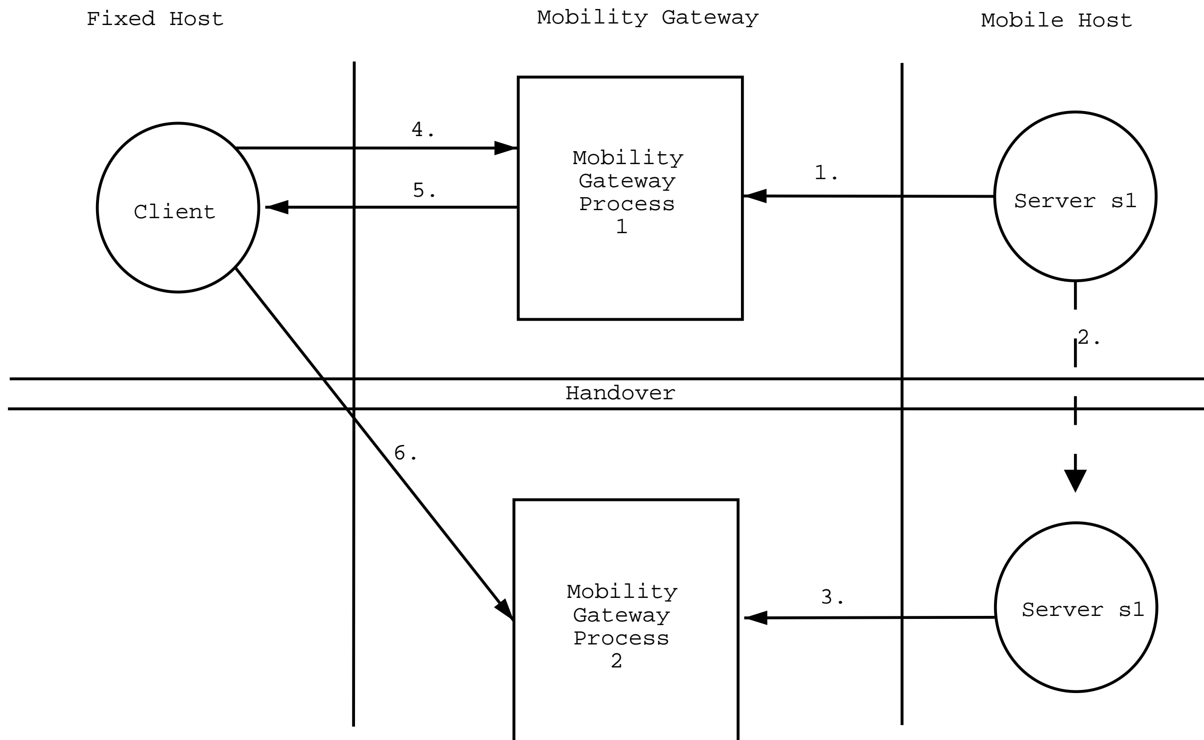

Figure 4.12: Reswizzling

- The Mobility Gateway maintains a collection of all remote objects registered with it and their corresponding proxies. Each proxy, which is returned to a client requesting a remote object reference, maintains a reference to the Mobility Gateway with which it was first registered.

- When the Mobile Host is handed over to another Mobility Gateway, all remote objects registered with the Mobility Gateway have their proxy entries updated to reflect the new point of attachment.

- Existing remote object references held by client objects are updated to reflect the new point of attachment the first time they attempt to invoke the server since handoff. In this case, the invocation will travel to the MG with which the remote object was first registered. The Mobility Gateway, aware that handoff has occurred since registration will update the Mobility Gateway reference held by the proxy stub on the Client, to reflect the new point of attachment.

The process is illustrated in Figure 4.12 and occurs in the following steps :

1. A remote object resident on a Mobile Host, registers with a Mobility Gateway.
2. The Mobile Host moves and changes its point of attachment to another MG (2).
3. The Mobile Host registers with the new Mobility Gateway and this causes a callback to the original Mobility Gateway, which changes all registered remote object proxies to now reference the new Mobility Gateway.
4. A client holding a stub to the remote object proxy invokes a method the invocation is redirected to the original Mobility Gateway.
5. The original Mobility Gateway makes a callback to the stub resident on the client, updating its Mobility Gateway reference to point to the new Mobility Gateway.
6. The invocation is then redirected to the new Mobility Gateway.

This scheme ensures that references held by clients to remote objects resident on Mobile Hosts are not invalidated by the movement of the Mobile Host and subsequent connection to a different Mobility Gateway.

4.6.5 Selected Strategy

The strategy finally selected was that described in Section 4.6.4 and does in part depend upon the maintenance of lookup support in the RMI system as described in Section 4.6.3. In contrast to the ALICE IIOP swizzling layer, this approach does not depend on changing of the original object reference exported by the server, rather relying on the construction of a proxy representation of the remote object whose reference is different to the original object reference. It is for this reason that we have decided to call this layer of the architecture the Java Remote Method Protocol Redirect (JRMP/R) layer. The functionality provided by this layer is analogous to that provided by the S/IIOP layer in the original ALICE architecture.

The selected strategy is a form of object delegation whereby a request from a client is forwarded by an intermediate host to a remote host. The intermediate host performs the redirection by maintaining, transparently to the client, a proxy representation of the object.

4.7 Comparison with Previous Implementation

The problems posed by having a server hosted on a Mobile Host and solved by the Swizzling Layer were also solved by delegation to proxy objects by Tom Wall [Wal00]. However the solution introduced significant application level knowledge into the process, with the application programmer being required to construct and compile proxy and carrier representations of each remote object with a consequent loss of transparency.

The use of dynamic proxy classes in this solution completely removes such awareness of the proxy process from the application programmer and leaves her free to develop RMI applications with the same amount of effort as for static systems. The need for an automatic code generator as described in [Wal00], is consequently removed.

4.8 Summary

This chapter examined the design of a variety of ways in which similar support could be offered for RMI clients and servers in a mobile environment as is provided to CORBA clients and servers by ALICE. Existing ALICE components were reused as far as possible resulting in the retention of the application independent ALICE Mobility Layer in the RMI solution. A method of swizzling RMI object references in the same way as is done for CORBA Interoperable Object References (IOR's) in ALICE was also examined, but it was decided to rather use a form of object delegation based upon dynamic proxy objects for invocation redirection.

Chapter 5

Implementation

This chapter describes how a Java equivalent to the Swizzling Layer, the JRMP Redirect JRMP/R layer was implemented, and the how the interface between Java and the existing C Mobility Layer was constructed using the Java Native Interface (JNI). The implementation of additional features to increase the transparency of mobility support as described in [Wal00] is also described and discussed. In areas where implementation details are platform-dependent, development has proceeded for the Linux Operating System.

5.1 Implementation Goals

The main implementation goals of the project were to provide for the operation of RMI client and server applications in a mobile computing environment. One of the major implementation goals was to provide connectivity management to RMI applications in a mobile environment. Connectivity management involves the transparent reconnection of broken transport connections, the maximum use of limited available bandwidth, and the tunnelling of open transport connections following handoff. The implementation of Connectivity Management was achieved through the integration of the C Mobility Layer of ALICE, into RMI.

Another implementation goal of the project was to create a layer which would provide location management for the operation of RMI servers in a mobile environment. The JRMP/R Layer allowed the operation of an RMI server on a Mobile Host and its interaction with clients that were not aware of the server mobility.

Additionally, implementation of the reswizzling scheme as described in Section 4.6.4 was carried out. The entire implementation was carried out with the goal of minimizing the application-level mobility awareness.

5.2 Mobility Layer

The successful introduction of the Mobility Layer into the RMI system to be used as an alternative transport mechanism to the default TCP sockets depends upon the ability to specify custom socket factories to the RMI system as discussed in Section 2.1.6. The integration of the Mobility Layer also depended upon the ability to develop a custom socket implementation which is used to create the socket classes. These features are all provided as an integral part of the RMI package.

5.2.1 Java Mobility Layer

The Java version of the Mobility Layer was completed in [Wal00] and gave valuable insight into how a different transport mechanism may be specified to the RMI runtime. The specification of a socket factory manufacturing Java Mobility Layer sockets and supplying them to the RMI runtime resulted in the successful integration of mobility support into RMI.

5.2.2 ML_{MH} Layer

The ML_{MH} Layer is the part of the Mobility Layer present on the Mobile Host. It is this layer that provides the ALICE socket replacement functions and the sockets+ API (for mobile-aware applications) to layers above it, and which replaces the TCP transport layer in mobile-enabled applications. At present the ML_{MH} Layer is implemented as a shared object file in C, against which applications needing mobility support are linked. This ensures that applications (which have no need to be aware of the mobile environment) may be provided with mobility support with no changes necessary to the existing code. All that is required is to recompile the client code, linking against the replacement socket functions rather than those found in `socket.h`. The standard socket functions replaced by the Mobility Layer are listed below :

```
int socket(int domain, int type, int protocol);
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
int bind(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, const struct sockaddr *serv_addr, socklen_t *addrlen);
```

It is through the API provided by this layer that the application programmer is able to access the functionality provided by the Mobility Layer. In terms of RMI, socket functions are not accessed explicitly by the application programmer, but rather by the RMI runtime system which creates sockets through the standard API as they are needed for communication. Consequently, it is for this API that an interface to Java needs to be constructed. Since these functions and the sockets+ functions discussed in Section 5.2.2 are the only Mobility Layer functions called explicitly from an application, the functions present in this layer are the only functions in the Mobility Layer for which the interface is needed. The other Mobility Layer modules do not require to be accessed from Java code and interactions between them may remain in pure native code.

The API is defined in the `mlmh.h` header and `mlmh.c` files of the ALICE implementation, and in order to provide the socket replacement functions to Java, an interface needs to be created into these files only.

Integrating the sockets+ functions into Java

The ALICE Mobility Layer provides a superset of standard BSD sockets, called sockets+. The downcall and tuning APIs of the Mobility Layer collectively make up the sockets+ API and allow the Mobility Layer to receive invocations from upper layers (the downcall API) and to be configured at runtime (the tuning API). The downcall API allows the creation, addition and deletion of callback functions which provide mobility information to higher layers if so required. Such a callback function can provide information about a Mobile Hosts current point of connection. The downcall API is :

```
typedef void (*CBF) (int sockfd, char *new_mg_name, int new_port);
```

```
int add_callback(int sockfd, CBF cbf);
int delete_callback(int sockfd);
```

Since the sockets+ functions are not part of the standard sockets API, use of them requires explicit awareness of mobility and hence the additional functions are reserved for use in applications in which an awareness of mobility is desirable. It is possible to integrate these functions into the custom Java socket types, giving the programmer the ability to call and make use of the functions from an RMI application. Callbacks are only used for server sockets, and the Java methods to access the functions are therefore placed in the server socket class.

5.2.3 Integration

The integration of the Mobility Layer into the RMI runtime entailed a number of a number of steps, discussed below. For each step, the way in which the standard socket accept() function and the sockets+ delete_callback() function were implemented is outlined as examples.

The Creation of a Custom Shared Object Library

The default shared object library containing socket functions in the Java runtime was created by linking the `PlainSocketImpl.c` file against the standard socket functions. The `PlainSocketImpl.c` file contains JNI methods which dispatch calls made upon them to the underlying platform specific socket functions. The method signature for the accept() method in the `ALICESocketImpl.c` file is :

```
JNIEXPORT void JNICALL Java_alice_rmi_ALICESocketImpl_socketAccept(JNIEnv *env, jobject this,
jobject socket)
```

The delete_callback() function, used to remove registered callbacks from the Mobility Layer, and not part of the standard socket functions, has the following signature :

```
JNIEXPORT jint JNICALL Java_alice_rmi_ALICESocketImpl_delete_1callback(JNIEnv *env, jint fd)
```

We created a file named `ALICESocketImpl.c`, which was functionally equivalent to the `PlainSocketImpl.c` file, but contained additional methods for the sockets+ API, and linked this file against the ALICE Mobility Layer socket replacement functions, to produce a shared object file called `libALICEnet.so`. In terms of the bind() method shown above, the method is called, via the shared object library, from the RMI runtime and is then dispatched to the ALICE implementation of the socket function in `mlmh.c`.

The Creation of a Custom Socket Implementation

The next step was to create a custom Java socket implementation type, extending from the `java.net.SocketImpl` class and making calls to `libALICEnet.so` file via the Java Native Interface. The default `java.net.PlainSocketImpl` class creates sockets in the same manner. We named this class `alice.rmi.ALICESocketImpl` and the major differences between this class and the default implementation lie in the shared library that the class uses, and the specification of the additional sockets+ functions. The shared object library is provided to the class through a call to a static LoadLibraryAction() method as follows

```
new sun.security.action.LoadLibraryAction("ALICEnet"));
```

The `alice.rmi.ALICESocketImpl` class is functionally similar to the `alice.rmi.PlainSocketImpl` class, but provides the additional sockets+ functions. Each socket function is represented by 2 methods in the `alice.rmi.ALICESocketImpl` class, a regular Java method which is called by the RMI runtime, and a private native method declaration which dispatches the call to the shared object library. For the socket accept method, the regular Java method is

```
protected synchronized void accept(SocketImpl s) throws IOException
```

and the native method declaration is

```
private native void socketAccept(SocketImpl s) throws IOException;
```

Likewise, the additional delete_callback() method is represented by 2 methods, a protected method (part of the API provided to higher layers), and a private native method which dispatches the call to the shared object library. The method signatures are as follows :

```
protected synchronized int delete_callback(int fd) throws Exception
```

```
private native int delete_callback(int fd) throws Exception
```

The protected Java methods are exposed to higher levels and these methods redirect calls made upon them to the corresponding private native methods and consequently onto the shared object library.

The Creation of Custom Socket Types

Up until this point, there has been no distinction between `Socket` and `ServerSocket` types, as both use the same implementation type. The creation of custom socket types realises this distinction and two separate socket types are created, `alice.rmi.ALICESocket` and `alice.rmi.ALICEServerSocket`.

Both the custom types have a `alice.rmi.ALICESocketImpl` socket implementation attribute, and for the `alice.rmi.ALICESocket`, this is the only difference between it and the `java.net.Socket` type. The sockets+ functions were implemented in the `alice.rmi.ALICEServerSocket` class since callbacks are only used for server sockets. The socket accept() method is also part of `alice.rmi.ALICEServerSocket` and has the following signature

```
public Socket accept() throws IOException
```

This method is part of the Java socket API and is used by the application programmer, or in the case of RMI, by the RMI runtime to instruct a socket to accept connections.

The delete_callback() method of the sockets+ API is provided by the following method

in `alice.rmi.ALICEServerSocket` :

```
public int delete_callback() throws Exception
```

This method may either be incorporated into the RMI runtime, or used separately by the programmer. This is a design decision that needs to be taken in the future.

The Creation of a Custom Socket Factory

The final step in the provision of Mobility Layer functionality to the RMI runtime, is through the creation of Client Socket Factory and Server Socket Factory types which RMI will use to supply instances of `alice.rmi.ALICESocket` and `alice.rmi.ALICEServerSocket` respectively for communication. A custom Server Socket Factory is created by implementing the `java.rmi.server.RMIServerFactory` which defines a single method `createServerSocket()`. This method is then implemented to return an instance of `alice.rmi.ALICEServerSocket` when RMI requires a new server socket. Likewise, the `java.rmi.server.RMIClientSocketFactory` interface is implemented to return an instance of `alice.rmi.ALICESocket` to the RMI runtime.

5.2.4 ML$_{MG}$ Layer

The ML$_{MG}$ Layer is that part of the ALICE Mobility Layer present on the Mobility Gateway. The ALICE ML$_{MG}$ Layer consists of a daemon, the `mlmgd`, which executes on the Mobility Gateway. The `mlmgd` daemon connects to the `mlmhd` daemon running on a Mobile Host and is responsible for relaying connections between the Mobility Gateway and the Mobile Hosts. Since this part of the Mobility Layer is a daemon process and does not require any interaction with the application programmer, there is no need to provide an interface to it from Java, and thus this component is not considered further in the implementation.

5.3 Invocation Redirection

After extensive investigation into the way in which Java handles remote object referencing, it was concluded that rather than performing address translation for location management of mobile servers as done by the ALICE Swizzling layer, a form of invocation redirection using object delegation would give a solution that is more compatible with the existing Java language.

To this end, a layer named the Java Remote Method Protocol Redirect (JRMP/R) Layer was developed and introduced to provide location management for the operation of mobile RMI servers.

5.3.1 Dynamic Proxy Objects in Java

Dynamic proxy classes, as part of the Java Development Kit, contributed greatly to the transparency of the JRMP/R solution.

A dynamic proxy class is a class that implements a set of interfaces specified at runtime in order to provide a type-safe proxy through which an invocation of an interface method, on the proxy, is dispatched to another object. The interaction between an object and a dynamic proxy representation of the same object is illustrated in Figure 4.8. Invocations on the proxy are dispatched to a single `InvocationHandler` method in the proxy class. This method is then free to do anything with the invocation, including dispatch it to another object, and returns the result of the invocation to the client.

The fact that the proxy class is developed against an interface ensures the proxy is totally transparent to the client and lends the dynamic proxy class towards use within an RMI application. As discussed in Section 2.1.1, remote objects in RMI applications are required to be coded against an interface. This means that any remote object within an RMI application is bound to present an interface API. Consequently, a dynamic proxy representation may be created for any remote object without requiring any additional representation of the remote object. Importantly, no pre-generation of the proxy class is required, further aiding the transparency of the process.

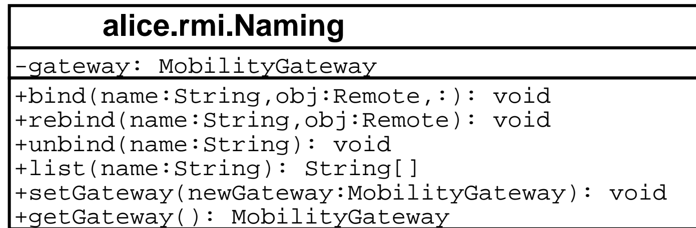

Figure 5.1: Class diagram for the ALICENaming class

5.3.2 Proposed Architecture

The JRMP/R layer consists of two components, that component resident on the Mobile Host, JRMP/R_{MH}, and that resident on the Mobility Gateway, JRMP/R_{MG}.

JRMP/R_{MH}

The JRMP/R_{MH} layer consists of a modified version of the `java.rmi.Naming` class. The modified class, known as `alice.rmi.Naming` maintains a reference to the current Mobility Gateway to which the Mobile Host is connected. When mobility support is required, RMI server objects use the `alice.rmi.Naming` class to register themselves with the RMI runtime. This object presents the same API to the application programmer as the `java.rmi.Naming` class. A design decision was made as to whether to place the additional functionality within the existing `java.rmi.Naming` class, or whether to introduce a distinct version of the naming service. It was decided that the introduction of the `alice.rmi.Naming` class would provide a less difficult wireless migration path to existing RMI server objects. Although the introduction of a new naming class does remove some transparency from the RMI server object (by requiring importation and use of `alice.rmi.Naming`), the alteration of the existing naming class would have forced mobile server objects to replace the existing Java runtime archive with a modified version.

The `alice.rmi.Naming` class replaces the `java.rmi.Naming` class, overriding a subset of the naming class methods and adding some additional methods. The class diagram is illustrated in Figure 5.1. The most important change introduced by the `alice.rmi.Naming` class is the overriding of the bind() and rebind() methods. The overridden methods still take the same arguments as the methods in `java.rmi.Naming`, that is a String name for the object, and the remote reference to the object. The `java.rmi.Naming` class causes

the binding of the remote reference and the name of the object in a table within the rmiregistry subject to the condition in RMI that a remote object may only register with an rmiregistry running within the same address space as itself.

The overridden bind method, rather than causing the binding of the name and remote reference to a registry in the same address space, causes the *registration* of the reference with the JRMP/R_{MG} layer on the Mobility Gateway. Registration with this layer effectively causes the instantiation of a proxy representation of the server and its binding to an rmiregistry running on the Mobility Gateway (in a separate address space). In this way, the semantics of RMI are changed slightly in that calling the bind() method in one address space causes the binding of the object (at least a proxy representing the object) in a different address space.

JRMP/R_{MG}

The JRMP/R_{MG} component of the JRMP/R layer consists of objects resident on the Mobility Gateway which collectively work together to transparently accept invocations intended for the server object resident on the Mobile Host, and relay them to the Mobile Host, relaying responses back to the client.

- **Mobility Gateway Process**

  The Mobility Gateway process executes on the Mobility Gateway itself and provides a set of methods to the JRMP/R_{MH} layer to allow the registration and deregistration of remote objects resident on a Mobile Host which is connected to the gateway. The Mobility Gateway process is involved with server handoff.

- **Mobility Registry**

  The Mobility Registry is an rmiregistry running on the Mobility Gateway and providing a lookup service for clients wishing to obtain a reference to a server hosted by a Mobile Host. The Mobility Registry contains the name of the server object bound to a proxy representation of the object created upon registration. Since the proxy object implements a remote interface (the same interface implemented by the remote object it represents), the original rmiregistry which stores String / remote object bindings, may be retained with no modifications necessary.

- **Proxy objects**

  Each remote object which registers with the Mobility Gateway has a dynamic proxy object, implementing the same remote interface as the object itself, created upon the gateway. The proxy object is an instantiation of the `alice.rmi.ServerProxy` class, which is part of the JRMP/R_{MG} layer.

  When a client performs a lookup against a remote object name, what is returned is the proxy object on the Mobility Gateway rather than the actual remote object itself. This is analogous to the returning of a CORBA IOR which has had its endpoint swizzled to point to the Mobility Gateway rather than the original server object.

5.3.3 Components Developed

As discussed, the implementation of the chosen solution to the problems of location management for mobile RMI servers introduced a number of components the salient functions of which are now discussed.

The ServerProxy class

The `ServerProxy` class is the dynamic proxy class, an instance of which is created for each mobile remote object. The class implements the `java.lang.reflect.InvocationHandler` interface (this defines it as a dynamic proxy class) and exposes a single method to which all invocations made on the proxy are forwarded :

```
public Object invoke(Object remote, Method m, Object[] args)
```

The object parameter to this method is the object which the proxy represents, the `Method` parameter represents the method invoked on the proxy, and the final parameter is an array of the parameters to the method being invoked. These parameters allow the proxy class to process the invocation before dispatching it to the real object.

Instantiation of a `ServerProxy` object is caused by registration of a mobile remote object with a Mobility Gateway. The constructor of the `ServerProxy` class takes three arguments namely the stringified name of the remote object, the remote object itself, and

a reference to the Mobility Gateway on which it was created. Any invocation made on the proxy is forwarded to the ServerProxy.invoke() method and from there to the Mobility Gateway (a reference to which is maintained by the proxy). The reference to the Mobility Gateway held by the proxy is updated after the Mobile Host moves to a different point of connection.

The MobilityGateway class

The `MobilityGateway` class provided the Mobility Gateway process which executes on the Mobility Gateway and provides for the registration of remote objects resident on Mobile Hosts with the Mobility Gateway. The MobilityGateway.register() method is illustrated below, showing the creation of a new instance of `ServerProxy` :

```
public void register(String name, Remote obj){

   /*Create the Dynamic Proxy object*/
   Remote serverProx = (Remote)alice.rmi.ServerProxy.newInstance(name, obj, this);
   /*Bind the proxy instance to the rmiregistry resident on the Mobility Gateway*/
   Naming.rebind(name, serverProx);

}
```

The registration process is hidden behind the bind() method presented to the application programmer by the ALICENaming class and is thus transparent.

All invocations made on the proxies held by the clients are forwarded to the following invoke() method in the MobilityGateway class :

```
public Object invoke (String objname, String meth_name, Object [] args)
```

From this method, the invocation is redirected to the Mobile Host. This method takes the method name, rather than a `Method` object due to reasons outlined in Section 5.3.4.

The ALICENaming class

In order to create a remote object binding in a different address space, the registry binding was delegated to the registration method on the Mobility Gateway, by implementing the bind() method as follows :

```
public static void bind(String name, Remote obj)
   throws RemoteException, AlreadyBoundException, MalformedURLException
{
   gateway.register(name, obj);
}
```

5.3.4 Problems Encountered

The implementation of the JRMP/R Layer posed some problems, which are outlined below, with the appropriate solution.

Non-Serialisability of the `java.lang.reflect.Method` type

Dynamic proxies in Java have a single invoke() method, to which all invocations made against the proxy are forwarded to. This method takes in the actual method invoked upon the proxy representation, as an argument of type `java.lang.reflect.Method` which represents the method invoked. Knowledge of which method was invoked is required so the proxy may forward the invocation to the appropriate method on the class which the proxy represents. The dynamic proxy is resident on the same host as the client, and thus the `java.lang.reflect.Method` type is passed as an argument to the proxy on the client.

The dynamic proxy then forwards the invocation to the Mobility Gateway which in turn forwards it to the appropriate server method on the Mobile Host. The mechanism by which the dynamic proxy forwards the call to the Mobility Gateway is by way of an RMI call to the Mobility Gateway. RMI relies upon data types being serialised before transmission, that is a data type is transformed into a representation that may be easily transmitted over the transport connection. In order to be serialised, a data type must implement the `java.io.Serializable` interface.

Unfortunately, the `java.lang.reflect.Method` data type is not serialisable and hence may not be transmitted by way of an RMI call to the Mobility Gateway.

A solution to this problem was to transmit the name of the invoked method, rather than the `java.lang.reflect.Method` data type itself. Reflection was then used at the Mobility Gateway to determine which `Method` the call should be forwarded to on the server object. This increased the cost of the invocation redirection mechanism due to the need to perform reflection at the Mobility Gateway, and meant mobile RMI servers could not

provide overloaded method declarations, as the name of the method is reflected upon.

5.4 Summary

Integration of the Mobility Layer into RMI was achieved via the Java Native Interface which allowed the RMI system (coded in Java) to communicate with the Mobility Layer (coded in C). This enabled the retention and re-use of the existing software components of the Mobility Layer and provided connectivity management to RMI applications in a mobile environment.

The implementation of the location management scheme to allow the operation of RMI servers in a mobile environment was carried out completely in Java, and with no modifications being made to the RMI source code.

Chapter 6

Evaluation

This chapter evaluates the success of the components developed in providing support to RMI applications in a mobile environment. A performance evaluation is carried out, with the cost of normal RMI between a client and a server examined first. A baseline cost for normal RMI invocations with an additional level of indirection (following the introduction of a mobility gateway) is then established. The additional cost to the process introduced by the use of dynamic proxies, and finally the full JRMP/R Layer is then demonstrated and possible explanations of the results are offered. The performance tests are carried out for a number of different remote invocation scenarios.

The degree to which transparency is obtained with the introduction of mobility support is also examined for each of the layers, as is the size of the code that is required for mobility support. The final measurement is important in the context of the limited storage capacity of mobile devices.

Finally, comparisons are drawn between this implementation and previous work done on extending ALICE to RMI.

6.1 Mobility Layer

The inclusion of the Mobility Layer is required to provide mobility support to RMI clients operating on mobile devices. Providing connectivity management in the form of transparent reconnection of broken transport connections, multiplexing of transport connections and tunnelling of transport connections after handoff, the existing C coded Mobility Layer

was retained for use with RMI.

6.1.1 Performance

A performance evaluation of the Mobility Layer has been carried out and the results presented in [Mad99]. Since such a performance evaluation effectively measures the performance of the ALICE Mobility Layer socket replacement functions over regular Linux socket functions, and since RMI running on Linux uses the default Linux socket functions, any performance evaluation carried out for the ALICE Mobility Layer socket functions is equally applicable to the use of the Mobility Layer in RMI.

6.1.2 Transparency

The operation of the Mobility Layer in RMI depends upon the specification, by the application programmer, of a different transport mechanism to the RMI system by replacement of the default RMI socket factories, with socket factories which produce custom Socket and ServerSocket types. The provision of Mobility Layer support to RMI applications is thus not transparent to the application programmer, in the sense that he must be aware of the specification of an alternative transport mechanism to the RMI system. The specification of an alternative transport to RMI, is however part of RMI and in this sense, Mobility Layer support may be considered transparent to a certain extent, as it does not require the use of any features that are not part of the RMI specification already.

6.1.3 Comparison to Previous Implementation

Previous work on the integration of the Mobility Layer into RMI resulted in a Java implementation of the Mobility Layer, [Cor00], [Wal00]. Implementing the Mobility Layer functions in Java added significant overhead to the process, resulting in applications making use of the mobility support running, on average, an order of magnitude slower than without mobility support. The performance of the C Mobility Layer far exceeds that of the Java Mobility Layer due mainly to the fact that Java is an interpreted, and thus slower, language. The integration of the C Mobility Layer into RMI resulted in mobility support being offered to RMI applications for a much lower cost than that of the Java

Mobility Layer.

6.2 Invocation Redirection (JRMP/R) Layer

The JRMP/R Layer provides location management support to RMI servers operating on mobile devices. Such location management support is required to manage the redirection of invocations from the Mobility Gateway to the Mobile Host, and managing references to mobile remote objects held by clients to ensure they are still valid after a Mobile Host has changed the Mobility Gateway to which it is attached.

6.2.1 Performance

The performance of the JRMP/R Layer is measured through a comparison of the time taken to perform certain types of invocation using the JRMP/R Layer, with the time taken to perform the same types of invocation using 'normal' RMI and using dynamic proxies. Providing mobility support to RMI applications involves the introduction of an extra 'hop' per invocation, since any communication between client and server must travel via a Mobility Gateway. The cost of an extra hop is quantified by comparison with single hop RMI.

Parameterless Invocation (Type 1)

For one hop RMI or RMI between a client and a server, the introduction of a dynamic proxy which simply forwarded the invocation to the real remote object, resulted in a marginal increase of 3.6% in the time taken to perform a remote invocation as illustrated in Figure 6.1. This increase is due to the use of reflection by the dynamic proxy.

For RMI in a mobile environment, as would be expected the parameterless, void invocation was the least expensive type of invocation, in terms of time taken in all cases except for that of RMI using dynamic proxies where the object return type of invocation was marginally quicker. This difference is possibly due to differences in network performance.

This type of invocation is the least expensive due to there being no need for RMI to serialise either arguments or return values. Out of the three architectures evaluated for this type of invocation, the baseline 'normal' RMI was the least expensive which

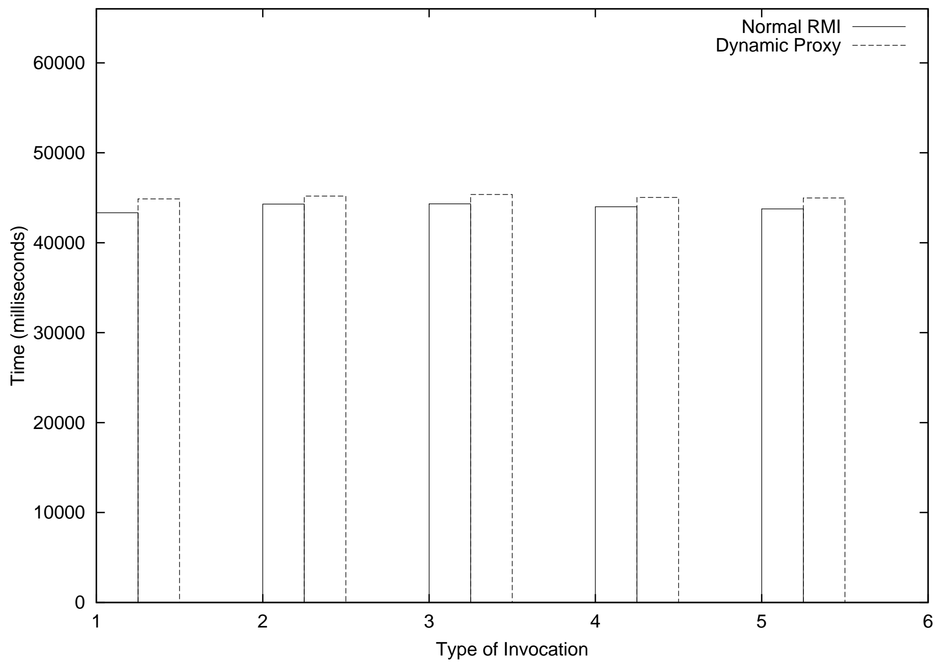

Figure 6.1: Invocation times for 10 000 remote method invocations in one-hop RMI

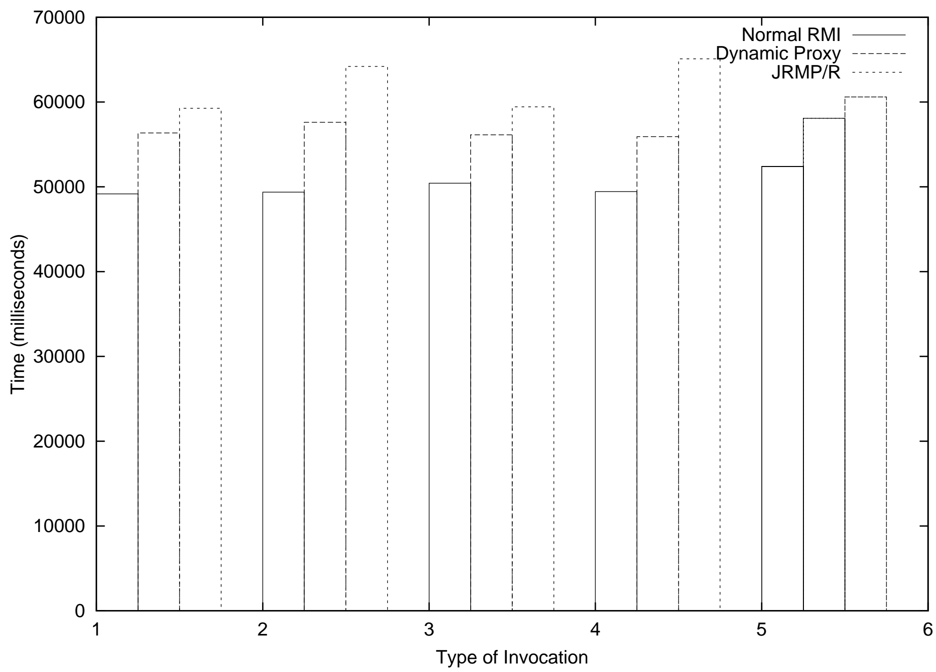

Figure 6.2: Invocation times for 10 000 remote method invocations in two-hop RMI

was as expected, as both the other architectures introduce additional processing into the invocation.

The use of dynamic proxies introduced additional overhead into this type of invocation, leading to an increase of 14.6% in the time taken to perform this type of invocation, over that of 'normal' RMI. The extra time is due to the high costs of reflection which is used by dynamic proxies in Java to determine which method has been invoked on the proxy.

The operation of full JRMP/R support for this type of invocation led to an increase of 5.1% in the time taken over that of using just dynamic proxies, and an increase of 20.5% in the time taken over normal RMI. This is illustrated in Figure 6.2. The additional time taken to perform an invocation using full JRMP/R support is introduced by the need to use additional reflection at the Mobility Gateway in order to determine the `Method` to be invoked at the server due to the issues discussed in Section 5.3.4. Since the method being invoked is both void and parameterless, there is no need for the replacement of parameters or return types with proxy representations at the gateway.

Primitive Parameter (Type 2)

Once again, the introduction of dynamic proxies into normal RMI between a client and a server led to a slight increase of 2% in the time taken to perform an invocation, as illustrated in Figure 6.1.

For RMI in a mobile environment, the invocation of a void method with a primitive parameter (an integer in this case) was once again least expensive at the level of 'normal' (not proxied) RMI. The introduction of dynamic proxies led to an increase in the time taken to perform an invocation of 16.6% which is similar to the increase observed in the void parameterless invocation and has the same explanation.

Full JRMP/R operation in this type of invocation led to an increase of 11.4% in the time taken over that using only Dynamic Proxies, and an increase of 30% over that using 'normal' RMI. The increase is illustrated in Figure 6.2.

The significant increase in the time taken over that using 'normal' RMI is due in part to the need to perform reflection at the Mobility Gateway in order to determine the method type, and due in part to the need to inspect the method parameter at the Mobility Gateway. The inspection of the parameter is needed in order to determine

whether a reference to a remote object is being passed as a parameter, in which case a proxy representation is required (see Section 4.6.4). The process of inspection utilises reflection which accounts for the additional time introduced into the invocation.

Object Parameter (Type 3)

A marginal increase of 2.3% in the time taken to perform a remote method invocation is observed with the introduction of a dynamic proxy in one-hop RMI as illustrated in Figure 6.1.

Similarly, for RMI in a mobile environment, the passing of a Java object as a parameter to a method invoked upon a server is least expensive when using 'normal' RMI. The use of dynamic proxies in the architecture led to an increase of 11.3% in the time taken to perform an invocation, which is similar to the previous two scenarios.

Additional overhead is introduced by reflection at the Mobility Gateway in order to determine both the method type and in order to inspect the parameter type to see whether a reference to a remote object is being passed. In this case, the method type determination and parameter inspection together resulted in an increase of 5.9% in the time taken to perform an invocation, over that of just using dynamic proxies, and an increase of 17.8% over 'normal' RMI (see Figure 6.2). This is significantly less than the overhead introduced by the inspection of primitive parameters at the Mobility Gateway. This may be explained by the fact that object creation is an expensive operation in Java. For the primitive parameter scenario (Type 2), an object representation of the primitive data type needs to be constructed so that it may be serialised for the RMI call. For example, if the parameter is an `int`, then an object of type `Integer` must be created to represent this integer.

Primitive Return (Type 4)

For one-hop RMI in this type of invocation, an increase of 2.3% in the time taken to invoke a remote method is observed (Figure 6.1). This increase may be attributed, once-again, to the reflection performed by the dynamic proxy.

For RMI in a mobile environment, the invocation of a method with no parameters, but a primitive return type (an integer in this case) was least expensive under 'normal' RMI,

with an additional 13.1% being introduced into the time taken to perform an invocation with the introduction of dynamic proxies.

The operation of the full JRMP/R Layer led to an increase of 16.4% in the time taken over that of RMI using dynamic proxies, and an increase of 31.6% over 'normal' RMI. This is illustrated in Figure 6.2.

Once again, part of the additional cost of making an RMI call when using full JRMP/R Layer support may be accounted for by reflection at the Mobility Gateway in order to determine the method type. With no parameters requiring inspection at the Mobility Gateway, a portion of the additional cost may also be apportioned to the need to inspect the return type of the invocation at the Mobility Gateway as discussed in Section 4.6.4. The inspection process involves reflection and consequently increases the time taken to perform an invocation of this type.

Object Return (Type 5)

The introduction of a dynamic proxy in this type of invocation increased the time taken to invoke a remote method by 2.7% as shown in Figure 6.2.

For RMI in a mobile environment, the invocation of a method with no parameters, but which returns a Java object is the most costly type of invocation in terms of 'normal' RMI. This is due to the need to serialise the return type of the method, a process which introduces overhead into making an RMI call.

The use of dynamic proxies for this type of invocation increases the time taken to invoke such a method by 10.8% due to the cost of reflection.

Full JRMP/R Layer support increases the time taken to invoke a method of this type by 4.3% over the use of dynamic proxies and an overall increase of 15.6% over 'normal' RMI (see Figure 6.2). Once again, the overhead introduced by inspection of object return values is less than that introduced by inspection of primitive return values and the need to create an object representation of these values as explained in Section 6.2.1.

6.2.2 Code Size

An evaluation of the size of code that is required on the Mobile Host for the provision of mobility support is important, as Mobile Hosts typically have severely constrained

Software Component	Size of Code (kB)
ALICENaming	3.54
MobilityGateway_Stub	6.53
Total	**10.07**

Table 6.1: Size of code required for mobility support on the Mobile Host

storage resources. As illustrated in Table 6.1, the amount of code required on a Mobile Host to provide location management to a mobile RMI server is 10.07 kB. This is 1.96% of the 512kB of memory required by J2ME, and is considered an acceptable overhead for mobility support.

6.2.3 Transparency

The transparency of the solution offered is important in the evaluation, as transparency of mobility support was a goal of the original design.

Client Side

The incorporation of the location management support offered to mobile RMI servers by the JRMP/R Layer is completely transparent to the client of a mobile RMI server, barring the need to perform certain bootstrap remote reference lookups on a different host (the Mobility Gateway, rather than the host itself). Whilst this may require the alteration of hard-coded host addresses in certain legacy applications, in most cases it should simply require the change of the host parameter provided at runtime to the client.

Server Side

The introduction of the JRMP/R layer at the server side is not (and should not be) completely transparent to the application programmer. The use of an alternative to the `java.rmi.Naming` class is required for Mobile Servers, but the alternative Naming class does present the same API and use of it is syntactically and semantically identical to the normal RMI Naming class.

6.2.4 Comparison to Previous Implementation

The previous implementation of support for mobile RMI servers [Wal00], also made use of invocation redirection via a proxy representation of the remote object at the Mobility Gateway. The solution required the development of an additional proxy representation of each mobile remote object, by the developer and thus was a less transparent solution. Although a performance evaluation of the scheme in [Wal00] was not carried out, it would be expected that the performance would have been better than that in our use of dynamic proxies, since there was no need for reflection.

6.3 Summary

This chapter evaluated the success of the components developed in order to support mobility for RMI, in terms of the additional cost introduced into making a remote method invocation. Tests showed that the introduction of dynamic proxies into normal one hop RMI between client and server led to marginal increases in the time taken to invoke a remote method of between 2 and 3.6%. The additional costs were attributed to reflection performed by the dynamic proxies. It was shown that the JRMP/R Layer does introduce additional cost into the process of between 15.6% and 31.6% over normal two hop RMI. These costs were attributed to reflection and the need, in some cases, the need to create object representations of primitive data types.

Chapter 7

Conclusions

This chapter describes the work completed and the conclusions drawn during the course of this project. Remaining work and possible future directions for the project are also outlined.

7.1 Completed Work

The first major piece of work completed was the integration of the ALICE Mobility Layer socket functions into Java. This enabled the use of existing Mobility Layer functionality from within Java code and allowed the C Mobility Layer to be retained for use in mobility enabled RMI applications. The integration required an in-depth knowledge of the design and implementation of socket functions in the Java language, as well as a good understanding of how the Mobility Layer works. Experience in the use of the Java Native Interface (JNI) API was also required. This part of the project provided connectivity management to mobile RMI clients and servers and permitted the full operation of RMI clients on mobile devices.

The second major part of the project was concerned with the provision of location management to mobile RMI servers. This involved extensive investigation of the RMI source code with a view to providing a form of the ALICE Swizzling Layer for RMI. It was rather decided to provide location management through the use of invocation redirection using dynamic proxies, as this approach would not require any changes to be made to the RMI source code. A system which performed redirection of invocations from a mobility

gateway to a mobile host was implemented for RMI. This system allowed clients to invoke methods on servers resident on hosts which repeatedly changed their point of connection to the network. The process was completely transparent to the client.

The project succeeded in the original goal of retaining the application independent ALICE Mobility Layer for use in RMI and provided a way of providing location management to mobile RMI servers. We found dynamic proxies to be a valuable tool in the processing of remote method invocations transparently to the client at an acceptable cost to the application.

7.2 Remaining Work

A version of the Swizzling Layer for ALICE remains to be implemented for RMI. Extensive investigation into remote object referencing in RMI in this project has laid the foundation for the development of this layer.

7.3 Future Work

Future work in the area addressed by this project could involve the integration of the mobility support offered by the developed components, into the RMI profile to be developed for the Java 2 Micro Edition (J2ME). Initial work carried out in the development of a distributed application to run on a Java enabled cellular telephone was promising, although in the absence of an RMI profile, communication with remote objects was via HTTP. Since our solution does not make use of any RMI feature that will not be present in the J2ME RMI profile, this work is not expected to be extensive.

Bibliography

[And96] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall, Third edition, 1996.

[App01] Wireless Application Forum Ltd. Wireless Application Protocol Architecture Specification WAP-210-WAPArch-20010712. `http://www.wapforum.org`, July 2001.

[AWW96] Roger Biggs Ann Wollrath and Jim Waldo. A Distributed Object Model for the Java System, June 1996.

[BB95] A. Bakre and B. Badrinath. M-RPC: A Remote Procedure Call Service for Mobile Clients, 1995.

[Cor00] Mark Corbett. A Feasability Study of the Implementation of ALICE in a JINI-based Environment, July 2000.

[EV99] Jorg Eberspacher and Hans-Jorg Vogel. *GSM switching, services and protocols.* John Wiley, 1999.

[Ina00] Inacon GmBH. *GPRS from A-Z.* Karlsruhe : Inacon GmBH, 2000.

[Ins00] Institute of Electrical and Electronics Engineers Inc. IEEE 802.11 Standard. `http://www.ieee802.org/11/`, 2000.

[Int00] International Telecommunication Union. IMT-2000: The global standard for Third Generation Wireless Communications. `http://www.itu.int`, 2000.

[KWB+] Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient Implementations of Java Remote Method Invocation (RMI). pages 19–36.

[Mad99] Mads Haahr, Raymond Cunningham and Vinny Cahill. Supporting CORBA Applications in a Mobile Environment. In *MobiCom '99: 5th International Conference on Mobile Computing and Networking*, August 1999.

[Mad00] Mads Haahr, Raymond Cunnigham and Vinny Cahill. Towards a Generic Architecture for Mobile Object-Oriented Applications. In *SerP 2000: Workshop on Service Portability*, December 2000.

[MAne] Alessio Vecchio Marco Avvenuti. MobileRMI: a toolkit for enhancing Java Remote Method Invocation with mobility. 6th ECOOP Workshop on Mobile Object Systems , 2000 June.

[Mar97] Maria A. Butrico et. al. Gold Rush : Mobile Transaction Middleware with Java-Object Replication. June 1997.

[MG] Object Management Group. Full CORBA 2.4.2 Specification. `http://www.omg.org/technology/documents/formal/corba_iiop.htm`.

[Mica] Sun Microsystems. J2ME Pofile. `http://jcp.org/jsr/detail/066.jsp`.

[Micb] Sun Microsystems. Java 2 Micro Edition. `http://java.sun.com/j2me/`.

[Mic99] Sun Microsystems. Java Remote Method Invocation Specification Revision 1.7. `http://java.sun.com/rmi`, December 1999.

[Mik] Mike Spreitzer. Overview of HTTP-NG. `http://www.w3.org/Protocols`.

[N.] N. Narasimhan, L.E Moser and P.M. Melliar-Smith. Interceptors for Java Remote Method Invocation. to be published in Concurrency and Computation: Practice and Experience.

[PB01] Esmond Pitt and Neil Belford. The RMI Proxy White Paper. `http://www.rmiproxy.com`, March 2001.

[Per98] Charles E. Perkins. Mobile Networking Through Mobile IP. IEEE Internet Computing, January 1998.

[Pos81] J.B. Postel. Internet Protocol. IETF RFC 791, September 1981.

[Ray98] Raymond Cunningham. Architecture for Location Independent CORBA Environments. Master's thesis, Trinity College Dublin, September 1998.

[Sea98] Sean McDirmid. Ghost Machine : A Distributed Virtual Machine Architecture for Mobile Java Applications. in Handheld Systems 6.5 Sept/Oct 1998, 1998.

[SI] Information Sciences Institute. RFC 793 Transmission Control Protocol, Protocol Specification. `http://www.ibiblio.org/pub/docs/rfc/rfc793.txt`.

[Ste00] Stefano Campadello, Oskari Koskimies, Kimmo Raatikainen. Wireless Java RMI. In *4th International Enterprise Distributed Object Computing Conference*, pages 114–123, September 2000.

[Suna] Sun Microsystems. Java 2 Platform, Micro Edition Connected Device Limited Configuration Version 1.0.2. `http://www.sun.com/software/communitysource/j2me/cldc/`.

[Sunb] Sun Microsystems. Mobile Information Device Profile. `http://www.sun.com/software/communitysource/midp/`.

[Wal00] Thomas Wall. Mobility and Java RMI. Master's thesis, Trinity College Dublin, 2000.

Appendix A

The Java Remote Method Protocol

A.1 Format of Output Stream

```
<Out> ::= <Header> {<Messages> | <HttpMessage>}

<Header> ::= 0x4a 0x52 0x4d 0x49 <Version> < Protocol>

<Version> ::= 0x00 0x01

<Protocol> ::= <StreamProtocol> <SingleOpProtocil> <MultiplexProtocol>

<StreamProtocol> ::= 0x4b

<SingleOpProtocol> ::= 0x4c

<MultiplexProtocol> ::= 0x4d

<Messages> ::= <Message> {<Messages>|<Message>}

<Message> ::= <Call>|<Ping>|<DgcAck>
```

A.2 Types of Message

```
<Call> ::= 0x50 <CallData>

<Ping> ::= 0x52

<DgcAck> ::= 0x54 <UniqueIdentifier>
```

A.3 Format of Input Stream

```
<In> ::= [<ProtocolAck> <Returns>] |  [<ProtocolNotSupported>] | [<HttpReturn>]

<ProtocolAck> ::= 0x4e

<ProtocolNotSupported> ::= 0x4f

<Returns> : <Return> {<Returns>|<Return>}

<Return> ::= <ReturnData> <PingAck>

<ReturnData> ::= 0x51 <ReturnValue>

<PingAck> ::= 0x53
```

A.4 Call Data

```
<CallData> ::= <ObjectIdentifier> <Operation> <Hash> {Arguments}

<ObjectIdentifier> ::= <ObjectNumber> <UniqueIdentifier>

<UniqueIdentifier> ::= <Number> <Time> <Count>

<Arguments> ::= <Value> {<Arguments>|<Value>}

<Value> ::= <Object> <Primitive>
```

A.5 Return Values

```
<Return> ::= <Return Code> <UniqueIdentifier> [<Value> | <Exception>]

<Return Code> ::= 0x01
```