Java RMI in a Mobile Environment

Paul Mac Sweeney

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

September 13th, 2004

Declaration

I, the undersigned, declare that this dissertation is entirely my own work, except where otherwise accredited, and that it had not been previously submitted for a degree at this or any other university of institution.

Paul Mac Sweeney
September 2004

Permission to Lend and / or Copy

Paul Mac Sweeney
September 2004

Acknowledgments

I would first like to thank my supervisor Mads Haahr for his constant support, wisdom and guidance during my work on this project.

Secondly, to Greg Biegel who previously worked on ALICE and was very generous with his time in answering my questions.

To Marisa, thank you.

Abstract

There are many problems that must be addressed when attempting to enhance a particular middleware programming framework, in this case Java RMI, to allow it operate effectively in a mobile environment.

The Architecture for Location Independent Computing Environments provides for the addition of mobility to such a framework in a set of reusable components. In this thesis I have outlined the problems that Java RMI faces and have implemented the components to allow it operate under mobile conditions.

Using a layered approach, I tackled common mobility issues like disconnection, relocation and reference management with specific reference to Sun Microsystems' implementation of Java RMI. I have provided a detailed design and implementation written in Java while also utilizing an enhanced version of the standard Berkeley sockets API with the Java Native Interface.

This updated API now facilitates the creation of mobile friendly RMI applications and can be utilized with the maximum amount of transparency available to the application programmer. The platform used to implement the design was JBuilderX running on Fedora Core Linux.

Contents

List of Figures

List of Tables

CHAPTER 1

-

INTRODUCTION

1.1 Introduction

In the recent past, probably the biggest growth area for personal computing has been that of mobile devices that provide functionality and services combined with a simple user interface e.g. hand held telephones, personal digital assistants (PDA's), pocket PC's and others.

For the majority of these devices and others like personal computers and workstations which are connected over wireless links, their connectivity to services and other networks is based on the interoperability of the underlying architecture that they are running on. This underlying architecture is most commonly known as middleware, which can be defined as:

“Software that sits between two or more types of software and translates information between them. Middleware can cover a broad spectrum of software and generally sits between an application and an operating system, a network operating system, or a database management system.” [1]

Some examples of it are Java Remote Method Invocation [2] or the Common Object Request Broker Architecture [3]. A common problem associated with object oriented middleware frameworks is that they do not provide for mobility comfortably. The frameworks were designed to operate well in fixed networks but now the demand for mobility ensures that they must be adapted to operate in such environments.

This thesis will describe and implement, specifically for RMI, the Architecture for Location Independent Computing Environments which allows mobility to be added to any middleware framework that can support some fundamental requirements, for example, operating over the transmission control protocol specification [4].

This implementation of ALICE for RMI will demonstrate that the key challenges of mobility like managing network connections over wireless links, maintaining service for periods of disconnection and reconnecting to the network are all transparently available to the application programmer when using ALICE. The development of each layer is presented and an evaluation is provided at the end of the project to judge the success of ALICE in the RMI environment.

1.2 Java RMI

The popularity of the Internet has seen many new technologies being introduced to the software development community. One of the most widely accepted and used languages which is still relatively new is Java.

Java has become so popular for two simple reasons:

- It is a fully object oriented language and hence it fully complies with one of todays most popular development models, object oriented design (OOD).
- It is completely platform independent. When building a Java application, the source code is compiled into bytecode and then a Virtual Machine (JVM) on the host machine interprets this bytecode into native machine code, hence allowing the code to run on any platform.

In terms of middleware development, Java also has a comprehensive API known as remote method invocation.

This API allows a Java application to connect and invoke methods on objects in a completely different address space or computer / device, over a network. For a Java specific distributed system, RMI is arguably the most suitable option for development as recommended by William Grosso in his overview of the framework.

“Java RMI is a robust and effective way to build a distributed framework in which all participating programs are running Java”.[5].

RMI programs communicate by attaining a reference to an object that has been “exported” (made available for invocation). As Java is a fully object oriented language, the usage of objects and classes is mandatory in its development. When a program, which has implemented certain interfaces that are required by RMI, attempts to invoke a method on a different machine, it must get a reference to the (server) object via a number of different ways. Once a reference is attained the client can proceed to invoke methods on the server object , hence allowing middleware operation to be achieved by the application programmer relatively simply.

1.3 ALICE Overview

Initially, the ALICE project was aimed at providing mobility features to the Common Object Request Broker Architecture middleware framework.

The design attempted to address the most common problems that CORBA and distributed object computing environments tended to endure, especially considering that many were largely developed with fixed networks in mind. The architecture is based on a stack design, not dissimilar in appearance or inspection to that of the popular Internet Protocol Stack [6]. There are six main components, each tackling different mobility challenges, and each providing its own functionality and communication with its closest layers.

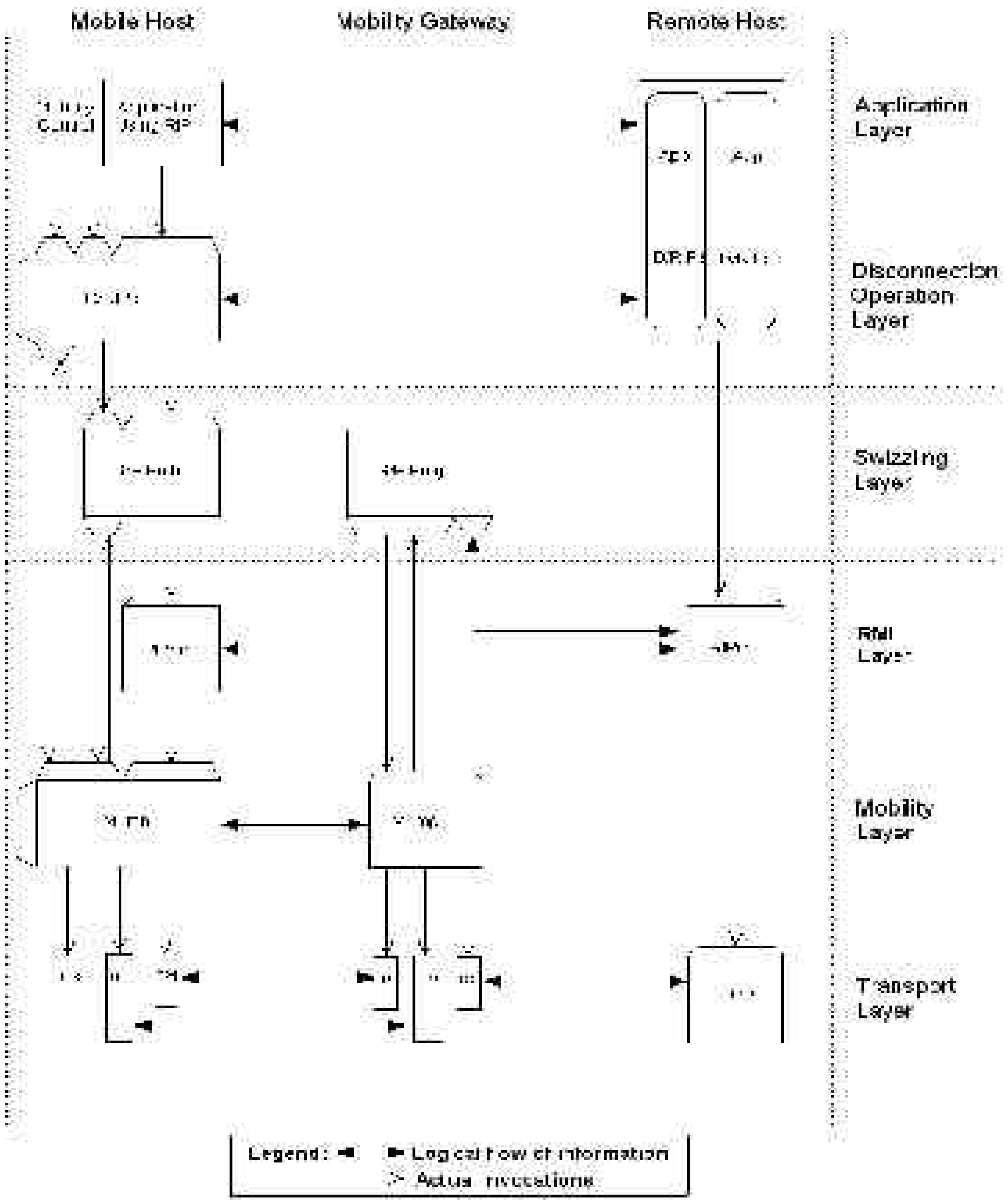


Fig 1.1 ALICE Architecture for RMI

- Application Layer – services defined by the programmer.
- Disconnected Operation Layer – long term disconnection management.
- Swizzling Layer – address management / redirection for mobile objects.
- Remote Invocation Protocol Layer – RMI invocations.
- Mobility Layer – connectivity management between MG and MH
- Transport Layer – Bluetooth, Infrared, Serial Line etc...

These six layers make up the abstract ALICE architecture and provide for mobility support based on the many challenges presented in distributed object environments. The main challenges that are addressed in this thesis specifically relate to RMI and come under the areas of address management during relocation of server objects, managing periods of client or server disconnection (whether short or long) from the network, socket relocation.

The underlying concept of the architecture is to introduce a proxy known as a mobility gateway (denoted onwards as MG) to manage connections from a mobility host (denoted onwards as MH) which is typically in a wireless, mobile environment.

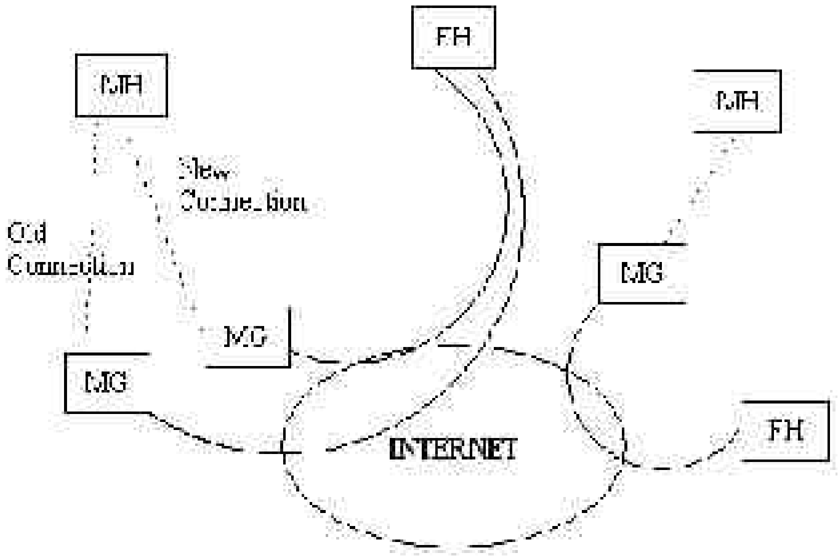


Fig 1.2 The Mobile Environment

The MG controls all access from the MH to other elements in the network and also processes invocations made on services offered on the MH, should it be hosting any server applications. Essentially the MG acts as a proxy to control the communications to and from the MH. This departs from the standard RMI client / server architecture but is necessary to aid the mobility of MH's in a wireless environment as will be explained in future chapters.

1.4 Project Objectives

The overall objective of the project is to allow Java RMI to operate successfully in a mobile environment. This can be evaluated using different criteria:

1. Maintaining Java RMI transparency. The application developer should be able to create ALICE enabled applications with no difference or the least amount of changes possible to the standard development procedure.

2. Integrating the ALICE mobility layer (denoted onwards as ML) into the RMI runtime. This allows the ML to transparently manage connectivity between the MG and ML, and is essential for mobile servers.

3. Implementation of the swizzling and disconnected operation layers. These layers tackle the common problems associated with mobility and are implemented specifically for RMI.

1.5 Thesis Roadmap

Chapter 1 - Introduction

Introduction to the framework of RMI, what is it, how does it operate, and also a brief explanation of ALICE.

Chapter 2 – Java in a Mobile Environment

An analysis of current standards in middleware and how these relate to RMI's operation. The chapter also gives a history of the area and explains some problems in the area and demonstrate some middleware systems.

Chapter 3 - State of the Art

This will focus specifically on RMI related systems and developments. I will explain some major research topics associated with RMI and how they relate to RMI in an ALICE environment.

Chapter 4 - Design

This contains an explanation of each layer in the architecture and of how it is to be integrated with RMI to enable mobility.

Chapter 5 - Implementation

This explains how I implemented the design described in the previous chapter. It details classes, operations and code samples that demonstrate how the design works in reality.

Chapter 6 - Evaluation

This chapter will give a discussion on the performance of the enhanced architecture and how it compares to RMI in the standard format. I will provide statistical analysis on round trip times for invocations and on the code sizes for enhanced RMI.

Chapter 7 - Conclusion

This will give a discussion on the results of the evaluation and where the future work lies in this area.

CHAPTER 2
-
JAVA IN A MOBILE ENVIRONMENT

2.1 Introduction

This chapter will provide a brief history and understanding of the functions performed by middleware in computing. I will look at the main challenges facing the development of middleware software in today and give an overview of the advancements that this area has made in recent times.

2.1.1 A Brief History

The term middleware first appeared in the 1980's to describe software that managed network connectivity but did not come into full utilization until the 1990's when network technology had achieved good reliability and customization. By then it had evolved into a paradigm that allowed distributed applications to be built and customized and the Internet revolution in the late 1990's resulted in middleware being a core requirement when offering new services and computing products.

Cronus [7] was one of the first main object oriented distributed middleware systems in the early 1980's and this introduced the remote procedure call (RPC) paradigm to computing. Other similar systems introduced at the time were Sun's Open Network Computing (ONC) and Apollo's Network Computing System (NCS).

In 1989 the Object Management Group [8] was formed and was the largest group dedicated to the development of middleware technology. Message oriented middleware became a standard in the 1990's and HTTP allowed for the operation of middleware easily with firewalls and the web.

2.1.2 Remote Procedure Call Middleware

Arguably the most used and well understood architecture in distributed computing today is that of the client - server model. It is the bedrock of the Internet revolution and is a well-defined method of efficient communication between machines across networks i.e. middleware. The basic premise is of a client who requests, by means of a marshalled message, some type of information from the server. To produce this information the server executes a service of some kind and returns the result to the client in a response message. The important idea of an interface is introduced here as it is one that is commonplace in middleware systems.

An interface is simply a formalized list of services that is provided by a server and guarantees that a client can expect these services when programming. In the remote procedure call framework, the client machine invokes a service made available at the server and by doing this is placing the idea of the interface at the core of the framework. This idea is now also a fundamental one when speaking about CORBA and also Java RMI. In CORBA, the Interface Definition Language (IDL) ensures that the specification of the services is provided and also combines this with the idea of a "broker" to implement the client – server model. It is a non-language specific middleware framework so it allows the integration of various back-end systems.

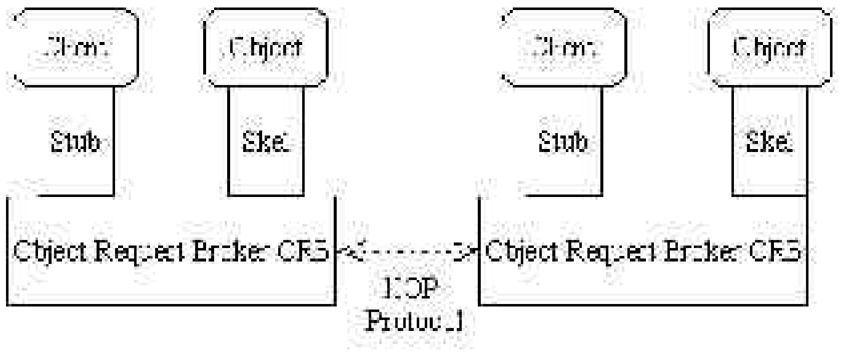


Fig 2.1 CORBA Operation

Now to look at the RMI model which is also a client – server one. The same idea of defining and programming to the interface is present in RMI. The server is a Java object which has implemented the methods specified in the interface. For a client to get access to the server it must obtain a reference to it, this is handled by the remote reference layer of the RMI framework. Stubs and skeletons are produced by the RMI compiler,`rmic`, and they manage the actual invocations between clients and servers.

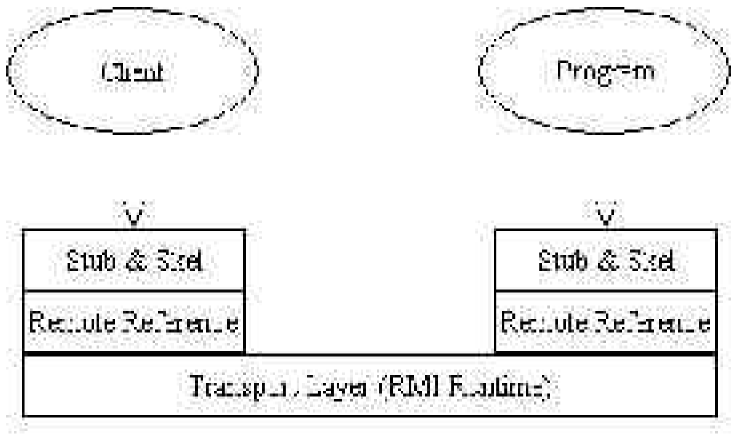


Fig 2.2 Java RMI Architecture

The concept of a remote procedure call has been used by all the major software vendors in recent times and has allowed the distributed computing environment to flourish, especially the world wide web.

2.1.3 Other Middleware Frameworks

There are many types of middleware frameworks that are very different from RPC middleware. Some of these are message oriented middleware (MOM) or event driven middleware. MOM has developed the concept of a common communications mechanism allowing programs to exchange data without knowing the basic primitives involved. The idea of a communications bus is a central one in message based middleware. The queuing system allows applications to run independently as the message can be sent and the queuing system will ensure that it is delivered, even if the sending application has no knowledge whether the receiver is active or not at that time. This allows the communication to be synchronous or asynchronous, which is the main difference from RPC based middleware. An example of a message based middleware system is the Java Message Service [9].

I have described one alternative area in the overall middleware domain but this thesis will now focus on RPC based middleware operating in a mobile environment and Java RMI in particular.

2.2 Middleware Challenges

The aim of any middleware system is to allow different applications to communicate over a network transparently. As it is a layer of software placed between the operating system and the application to facilitate communication across many platforms, it therefore has inherent problems that standalone applications will not incur. I will now outline some of the major problems that object oriented middleware systems operating in mobile environments face. The purpose of this list is to show the challenges that the design and implementation of ALICE for RMI faces and how to best tackle these problems.

2.2.1 Address Migration

The overall objective of the ALICE architecture is to enable mobility for middleware frameworks which support basic requirements. One problem facing the standard RMI structure is that it strongly based on the client – server architecture.

To facilitate RMI with ALICE on a MH there needs to be a mechanism to tackle the problem of when a MH moves between different MG's. If a server resides on a MH connected to MG1, then after the MH moves to MG2, the clients that have already invoked methods on the server will still be pointing to MG1 as the most up to date gateway in the invocation process. This gateway address will be old and there needs to be some mechanism to address this problem. This is tackled by the S/RMI$_{MG}$ component which is explained in chapter four. It facilitates the movement of MH's between gateways and provides a forward pointer algorithm for when invocations are made on servers that have since moved to a different gateway.

2.2.2 Disconnection

In any wireless network, a disconnection does not necessarily mean that an error or fault has occurred in the network but only that a client or device cannot at some time connect to a transmitting station or server.

It is important to categorize the different types of disconnections that can occur so as to get a comprehensive understanding of the disconnection area. In one sense, a user may choose to disconnect from the network because of battery levels or they may be approaching an unserviced area, this is a voluntary one and can be preemptively handled by some type of caching / saving mechanism. On the other hand, the most important area to manage is when a disconnection occurs due to some failure in the architecture, whether a hardware or transmission failure. This will occur suddenly and unexpectedly and it is vital that the middleware has some predictable response to ensure that the result is a valid state.

In ALICE connectivity management is handled by the mobility layer which implements a round robin reconnection algorithm, trying all transports in turn to reconnect. This comprises of an attempted reconnection, which if unsuccessful delays for a time k then attempts again with a different transport mechanism.

When a disconnection occurs the data is queued by the ML for transmission to the ML_{MH}. Also the disconnected operation layer can handle caching of RMI objects to ensure a cached version can be used when an invocation is requested, this replica object is then reconciled with the actual server object to provide genuine operation to facilitate operation in a disconnected environment.

2.2.3 Device Constraints

As the beginning of the thesis, small hand held devices such as personal digital assistants are listed as appropriate devices that are considered for RMI in a mobile environment. These devices have many technical requirements that a software component must meet in order to operate properly in the context of the systems capabilities.

Many of these requirements are based on the typical amount of memory available and the processor that the device in question uses. For example a typical J2ME application which is using the CLDC profile (explained in section 3.2.2) is designed to operate in an environment where there are 128KB of memory for running the KVM (k virtual machine, smaller version of standard JVM) and 32KB of available memory available during the running of the application for the allocation of objects. This is the standard type of Java for use in the mobile phone environment and represents the constraints that must be met for these target devices in mobile environments.

CHAPTER 3

-

STATE OF THE ART

3.1 Introduction

This chapter will describe some popular RMI based technologies and how they are mutating from the standard version of Sun's RMI to address mobility challenges in different ways.

3.2 Java for Wireless Devices (J2ME)

The Java 2 Platform, Micro Edition, is a flexible platform for the development of Java applications to run on resource constrained devices such as mobile phones, PDA's and similar devices.

Like J2SE and J2EE it provides objected oriented development to applications running on these devices, but also minimizing the resources that it uses. It has a much smaller memory footprint on the target device, usually operates in a battery powered environment and expects a small processor. Its main drawback is that it does not support all the Java API's.

3.2.1 Architecture

The J2ME architecture is built from many configurations, profiles, and optional packages that developers choose from. The developer combines the configurations to match the resources of the device in question and constructs a runtime environment that matches its requirements. Each combination is optimized for the memory, processor, and I/O capabilities of a related category of devices. The result is a common Java platform that takes advantage of each type of device to provide maximum efficiency.

3.2.2 J2ME Profiles & Configurations

A configuration provides the basic set of libraries and virtual-machine features that is needed in each implementation of a J2ME environment. When built with one or more profiles, the Connected Limited Device Configuration [10] gives developers a platform for creating applications for many devices. It is the most basic layer of a J2ME application and the profile for the target application is then placed on top of this layer.

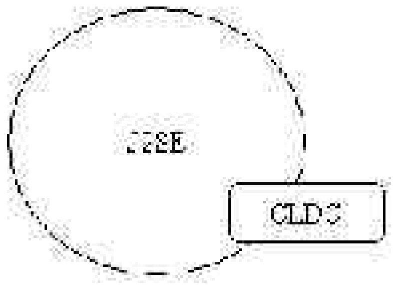


Fig 3.1 CLDC Functionality

A profile supports a smaller category of devices within the framework of a chosen configuration. A common use is to combine CLDC with the Mobile Information Device Profile (MIDP) to provide a complete Java application environment for cell phones and other devices.

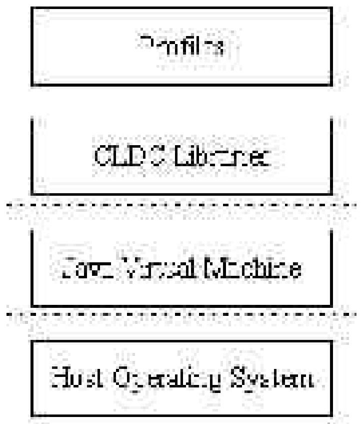


Fig 3.2 J2ME Architecture

3.2.3 RMI Optional Package

As RMI was always an important feature of the standard Java products it was expected that a version of it would be developed for J2ME, even when device constraints like memory or processing power on phones or mobile assistants is taken into place. This was achieved due to Java Specification Requests JSR-000036 and JSR-000046 [11].

With the advent of the optional package, not only do desktop and server systems, but now smallhandhold and embedded devices have interoperability with J2SE RMI systems. The package is a subset of J2SE RMI that can be used on devices that support the Connected Device Configuration (CDC) and the Foundation Profile (FP). The package profile supports the following minimum device capabilities:

- 2.5 MB minimum ROM available.
- 1 MB minimum RAM available.
- TCP/IP connectivity to the network.

This now allows for constrained devices like mobile phones to communicate using RMI. For Sun Microsystems, the penetration that their Java games now have in the mobile phone market will allow it now to combine these games, based on J2ME, with other games, possibly over an RMI framework, opening up a potentially lucrative market for mobile handset gaming.

3.2.4 RMI Remote Reference Layer

To invoke on a remote object in RMI, there is a standard infrastructure available to attain object references. Firstly, the stub created by the object must be attained using the standard lookup procedure, the rmiregistry. This is a lookup server operating on port 1099 of all machines running it. When the `lookup()` method is called on a registry object, the returned object is of type `Remote` and is a stub object which acts as a proxy on the client side and implements all the methods that the actual server does. To now make an invocation, the client invokes on the stub which handles the marshalling of parameters and transport layer, Java Remote Method Protocol (JRMP).

This stub of type `Remote`, contains a number of objects which are all used to point to the correct object on the server machine. A RMI object when exported is assigned an `ObjID` which points to the position of the object in the server address space, it also contains a `LiveRef` object, which contains a `TCPEndPoint` object which contains the IP address and port number of the object. These are all unique to the server machine over time and are used to allow the stub contact the actual server and differentiate between objects of the same type on the server machine.

3.2.5 J2ME Restrictions

As I outlined in the project objectives section in chapter one, the main goal was to develop a working implementation of ALICE for RMI. Due to this, I chose to develop under the standard J2SE environment because of the full gambit of API's available to that platform. If one wished to develop ALICE for RMI under the J2ME architecture there would have to be some core Java programming constraints that must be met. These are:

- **No Java Native Interface:** A Java virtual machine supporting the CLDC does not implement the Java Native Interface (JNI) primarily for security reasons. Also, implementing JNI is considered expensive, given the memory constraints of CLDC target devices. The mobility layer that I was working with was implemented in C and so I would have been unable to interface with it, in a J2ME environment.
- **No reflection:** Because reflection is not supported, there is also no support for RMI or object serialization (but this is tackled by the new

RMI optional package for J2ME).

- **No thread groups or daemon threads:** While a Java virtual machine supporting the CLDC implements multi-threading, it cannot support thread groups or daemon threads. This can limit the implementation style of the developer as the ALICE architecture is a design, not an implementation, so the developer can choose to implement threading or daemon groups as he pleases.
- **Poor error handling:** The CLDC defines only three error classes: `java.lang.Error`, `java.lang.OutOfMemoryError`, and `java.lang.VirtualMachineError`. Non-runtime errors are handled in a device-dependent manner that terminates the application or resets the device. The latter option is unacceptable for a middleware environment where maintainability and transparency are paramount, so for this reason, J2ME was unsuitable for the ALICE implementation.
- **No support for RMI Multiplexing:** This represents a problem for the RMI runtime as the sockets are multiplexed over existing connections. To have to create new sockets for all would enforce a change on the existing architecture of the runtime as in J2SE.
- **No support for JDK1.1 stub / skeletons:** This removes the backward compatibility of any RMI system built with this package.

3.3 Mobile RMI

In any typical mobile environment, the ability of the servers to move around is critical, as periods of disconnection and other network related problems can occur at any time. In these cases, the server must be able to anticipate and respond to these failures sensibly. One of the problems associated with this type of movement is the updating of server references held by clients.

In standard RMI, the `sun.rmi.server.UnicastRef` class contained in any stub issued from the rmiregistry on the server, controls a method invocation from a client on the server object. So, if a mobile server were to relocate from machine A to machine B, the previously issued stubs from that server would now contain incorrect server references.

The key feature of MobileRMI is the automatic updating of remote references to mobile objects, necessary to support remote method invocation even in the presence of mobility. The extensions to Java RMI in MobileRMI only effect the remote reference layer while they leave the transport layer and bytecode generation process unchanged.

3.3.1 Embedding Mobility

The idea of code mobility is a central one to MobileRMI. One type of code mobility paradigm is that of the mobile agent. A mobile agent is a program that can, at a time of its own choosing, migrate to a different machine and continue processing transparently.

There are many Java-based systems proposed in order to support code mobility, Aglets [12] and Cougaar [13] for exampleThe Aglets Software Development Kit is a framework and environment for researching and developing mobile agents whereas Cougaar is a Java-based architecture for the construction of large-scale distributed agent-based applications.

Both of these developments tackle the idea of mobility in their own ways but MobileRMI has tackled the problem differently, by embedding it. The idea is to embed mobility primitives into a particular programming environment, Java RMI. It allows integration of mobility primitives with the Java language and takes from Sumatra [14] the idea of interacting with mobile objects by means of method invocation.

3.3.2 Mobile RMI System

In Java RMI, the basic component of a remote object is the `java.rmi.server.UnicastRemoteObject (URO)`. This class is a remote object implementation that uses a TCP based stream protocol for carrying out method invocations from clients to servers and must be extended to create a remote object. In MobileRMI, the URO is extended to create the MobileUnicastRemoteObject (MURO) which can be moved across different JVM's, and its remote references are updated according to its migration path, so that clients holding a reference to it can continue to invoke regardless of its current position.

The MobileRMI systems allows a MURO to be created on a different JVM by invoking the `create()` method, and a client can trigger the movement the MURO by invoking the `move()` method on it. When a client or a MURO itself invokes the `move()` method, it serializes itself and transfers the bytestream to the receiver JVM, where it is made available for method invocations. If the relocation is successful then the remote references are updated to the new location. From now on, new invocations are executed in the new address space.

3.3.3 Remote Reference Updating

In order to preserve remote method invocation on a mobile server object, it is vital to update all remote references held by clients which point to that server. To achieve this, in MobileRMI, the `sun.rmi.server.UnicastRef` class was extended to allow the operation of the `move()` method. After the migration has completed successfully, the `move()` method then updates the reference held on the client side. As soon as the migrated object has been reconstructed properly, the 'Mobility Daemon' returns the new reference to the `UnicastRef`, this ensures the process is completely transparent to the application programmer.

For clients that did not initiate the migration, their reference updating takes place when they attempt an invocation on the now relocated object. When the object migrates it leaves a dummy object in its place. The dummy objects that are formed as an object migrates form a chain and each object holds a reference to the next in the chain, following that chain, the objects will eventually get their references updates to the now accurate location.

3.3.4 Using MobileRMI

When compiling the MURO class, it is only necessary to put the MobileRMI package in the classpath. The stub compiler,`rmic`, can still be used but an enhanced version of the rmiregistry must be used. So, MobileRMI doesn't require modifications to the Java interpreter and so a programs' portability remains intact.

3.4 Cajo

This project is a free library enabling machine cooperation, both within, and between Java applications [15]. It provides an easy-to-use and understandable framework to simplify the use of RMI.

3.4.1 Operation

In standard Java RMI, a server object is exported by extending the `sun.rmi.server.UnicastRemoteObject`class. This enables clients to invoke methods on the object, assuming the server has registered with the bootstrap rmiregistry. But in Cajo, every remotely available server in the framework is known as an`'item'`. Items can call other items, and an item can also be called by others. A special type of item is known as a proxy, these are items sent from one virtual machine, to execute in the context of another. Proxies can also call, or be called, by other items and even by other proxies.

In the standard RMI framework, each remote object must have a unique stub compiled for its callable methods. This can sometimes cause unwanted overheads on the running program due to argument marshalling and only works when method signatures / hashes are known at compile time. To address this issue, the framework defines only one functional interface as every item inherits the `'Invoke'` interface. It defines a single method, to represent all object methods; it accepts a method name, the arguments to be provided to it, and the data to be returned:

```
Object invoke(String method, Object args);
```

To actually invoke upon an RMI remote object we must use the `Remote` class which implements the `invoke` interface. The `Remote` class takes any object, and places a remotely invokable wrapper around it. When remote objects execute the invoke method, the wrapper uses reflection to find the method on its internal object, and matches the method name and argument types against those contained in the object. If it is found, it executes that method, and returns the result, if any. If no matching method can be found on the internal object, a Cajo exception is thrown. This technique makes all public methods of the internal object remotely callable.

3.4.2 Managing References

As I stated in chapter 2, one of the most difficult areas of mobility and middleware is that of reference management. In ALICE, the reference management system is that of a chaining of old references from the mobility gateways to update a mobile host when invoking on an object, but in Cajo it is slightly different. Obtaining initial references is done via two standard mechanisms: First, it is generally useful to bind each item with the `ItemServer` class. `ItemServer` will automatically create an rmiregistry, and bind the item under the name provided. Other clients can now contact the registry, and obtain a remote reference to the item by name.

The rmiregistry, and the item communicate using the same TCP port. Therefore, all that is required for a client to obtain a reference to an item using this mechanism is the server host name, port number, and the name under which the item is bound. The default item for any server is bound under the name "main". Obtaining a reference to a bound remote item is accomplished using a static getItem method of the Remote class. This mechanism is called linkingstatically.

The second mechanism to obtain a reference to a remote item is using the Multicast class. Unlike the previous method which uses TCP/IP to obtain the reference, this technique uses UDP/IP. Instead of needing to know the server's TCP address and port number, the client needs to know the UDP address and port number the server will be using. The difference is that servers can pick their UDP addresses independent of their TCP addresses.

Also, a server can use multiple multiple UDP addresses if needed. Multiple servers will share the same address, to form groups. A server broadcasts a remote reference to an item by calling the announce method of Multicast. This will send the reference to all clients listening in the group. Items listen for these announcements using the listen method of multicast.

3.5 Ninja RMI

Ninja is a free implementation of RMI which allows Java code to invoke methods on objects running on remote machines using a network connection. It is an independent implementation of RMI which was developed at UC Berkeley for the 'Ninja' project there [16].

3.5.1 Project Overview

The Ninja project aims to develop a software infrastructure to support the next generation of Internet-based applications. The concept of a service is at the center of this project. It is an Internet-accessible application which is scalable, fault-tolerant and must be robust for the high number of clients.

3.5.2 Differences with Sun's RMI

Their version of RMI is not meant to be received as an alternative to Sun Microsystems standard RMI. It was designed to provide extra functionality that the standard version does not provide. These new added components are:

1. Multiple communication protocols, allows for TCP, UDP, and multicast
2. Reliable, unreliable, one-way, and multicast communication semantics.
3. API methods to allow both client and server objects to determine the peers' hostname and port addresses.
4. The ability for server code to register callbacks on specific events

3.5.3 Operation

In the standard RMI, a remote server is created by extending the `java.rmi.server.UnicastRemoteObject` This allows the object become available for invocation and is the standard procedure. In the case of Ninja, the new remote server is created by extending `NinjaRemoteObject`, and the remote reference layer is also altered in the sense that the stubs and skeletons are created using a special `ninjarmic` compiler. It also comes with its own `ninjaregistry` instead of the standard rmiregistry that is part of Sun's package.

Once these changes are made with a small change to the transparency of standard RMI, the Ninja version now has access to these valuable added features as listed above. An interesting feature which is not dissimilar to the ML area in ALICE is that of the callbacks available to the client in

Ninja. Here, the developer can register for callbacks and this can be utilized at the socked level so in a mobile environment the application can update a server or client as the case may be, which similarly is implemented in ALICE.

3.6 Ka RMI

Ka [17] is a replacement for the standard RMI package. It is based on an efficient object serialization mechanism called `uka.transport` that replaces regular Java serialization from the `java.io` package. KaRMI and `uka.transport` are implemented completely in Java without native code. KaRMI also supports non-TCP/IP communication networks. It can also be used in clusters interconnected with heterogeneous communication technology.

3.6.1 Differences with RMI

When a standard RMI object is exported, it is bound to a specific port on a machine, this is in line with the TCP/IP protocol. As KaRMI does not run over TCP then this has no meaning so the principle of “exporting” has no basis. In KaRMI, the new runtime configuration replaces ports with export points. The principle is the same except that an export point can be configured to mean a special transport protocol using a KaRMI configuration file. This configuration is then used to setup the transport involved and access the endpoint of the RMI application.

3.6.2 Using Ka with ALICE

As ALICE is based on the standard implementation of RMI using TCP/IP, the idea of a Ka alternative is an interesting one. If a version of ALICE over UDP or a different transport was required, then Ka would provide an interesting starting point from a research perspective.

3.7 JavaBT

This is an experimental technology to see how Java RMI can be implemented to run in a wireless Bluetooth [17] environment. A set of layers for the Bluetooth stack were implemented in Java and then RMI is implemented on top of this layer to test it over a wireless connection [18].

3.7.1 Customized Bluetooth Sockets

Similar to the idea of the mobility layer in ALICE, here a set of customized sockets are written in Java to provide functionality for the Bluetooth driver. They serve both the standard `java.net.Socket` and `java.net.ServerSocket` objects and also inherit from the standard `java.io.InputStream` and `java.io.OutputStream`. Now using the RMI socket factory facility, the Bluetooth sockets are supplied to the runtime using the `setSocketFactory` static method. This achieves the correct alterations to the runtime as the new sockets are supplied and multiplexed between connections in the Bluetooth environment.

3.7.2 Operation

When running the RMI applications over Bluetooth, it was found that its performance and management of bandwidth are far superior in a WLAN environment. Also the support of Java in the Bluetooth environment gives more high-level support to the developer and this is not dissimilar to my objectives as ALICE is supposed to provide a generic architecture for the middleware, not a specific one for specific underlying transports.

The same principle is used in the mobility layer to alter the runtime and provide the sockets+ API. This enables the enhanced sockets to be supplied and allow relocation from one ALICE MG to the next. It is explained in section 4.3.2 as part of the integration of the mobility layer with the RMI runtime.

CHAPTER 4
-
DESIGN

4.1 Introduction

This chapter will detail how Java RMI can be enhanced by the ALICE design to allow for operation in mobile environments. I will explain a layer by layer approach and detail thespecific changes that need to be made to classes that represent the structure of RMI and how these objects then operate to provide for mobility.

The model for communications used by ALICE is the gateway (or proxy) model, this encompasses the MH, MG and FH that are explained in section 1.3.2. The benefit of the architecture is that the API's implemented in Java for RMI allow for a transparent integration of mobility for the application developer.

4.2 API Notation

All of the layers that I have implemented have their own components and share a common language of communication. The interaction that exists between the layers is vital to the operation of the architecture and is split into three main areas. There are 'upcalls' and 'downcalls'.

If a layer performs a service or function for the layer above it then it is deemed to have a 'downcall' API through which invocations from upper layers are received. Conversely, the layer that makes the invocation then has an 'upcall' API to talk to the layer directly below it. The interaction between these layers is very important especially when the system is dependent on mobility updates from the mobility layer (ML).

In case the component requires some time of runtime configuration then the 'tuning' API would be invoked as it provides for this type of architecture specific configuration.

Apart from the API's that are available for implementation, there are also different aspects to the layers involved in ALICE. For example, the mobility layer exists across an ALICE enabled framework but different actual ML components can reside on the different devices e.g. the ML_{MH} component would reside on a MH but not on a MG, and represents the part of the ML that resides there.

4.3 Mobility Layer

As the mobility layer is written in C, it needs to be integrated into the above swizzling and disconnected operation layers to enable the application layer callbacks. At the start of the project I received the mobility layer package as written in C. This consisted of a number of files, each of which contributes to the sockets+ API which is the enhanced version of the Berkeley Sockets API used in ALICE. To utilize the mobility that the sockets+ API contained, I had to supply these custom sockets to the RMI runtime.

4.3.1 RMI Runtime

In the standard operation of RMI, the runtime plays a fundamental role in the supply of both client and server sockets to the transport layer. A quick review of RMI tells us that a stub object on the client side interacts with the skeleton on the server side to provide the abstraction from the actual server object. The basis for this communication is a stream-based sockets underlying transport protocol and the most important role played by the runtime is that the sockets are reused where possible.

When a new invocation is made on a remote server object, the runtime (also known as the transport layer) checks to see if an already existing socket is available, and if so, reuses it. One of the benefits of the RMI system is that when required, the application programmer can supply custom sockets to the runtime instead of the default ones. This can be done using the `RMISocketFactory` class. Connections from the MH to the gateway are multiplexed over a single connection in order to conserve the limited and expensive bandwidth available to the device and make the tasks of handoff and connection re-establishment easier.

4.3.2 Integrating the Mobility Layer with RMI

The RMI runtime attains its sockets functions using the Java Native Interface (JNI). This allows it to access native code and invoke the standard Berkeley socket functions that are contained in the `libnet.so` shared library on Linux. To access these, it makes the calls using JNI from the `java.net.PlainSocketImpl` class.

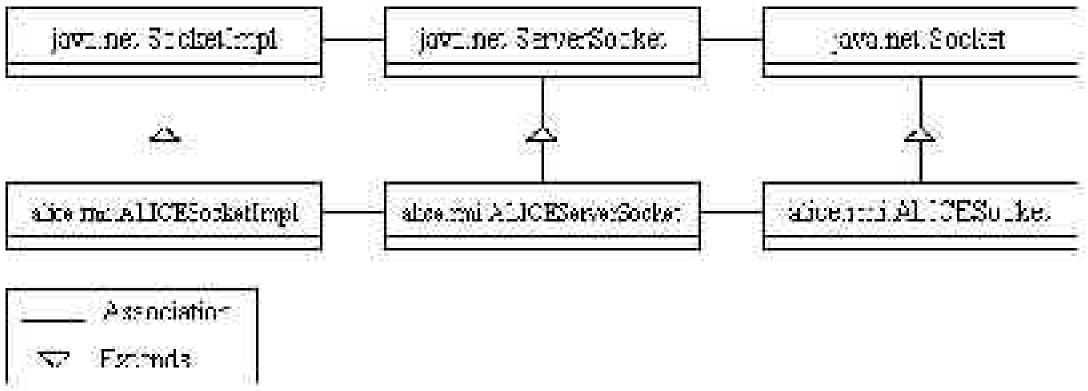


Fig 4.1 ALICE Replacement Socket Classes

To alter this process, a custom socket implementation was created by Biegel et al [19]. They created a file named `ALICESocketImpl.c` with additional methods for the sockets+ API and linked it against the ALICE ML socket replacement functions to create the shared library `mlmh-ma.so`. This filename stands for '**m**obility **l**ayer **m**obile **h**ost – **m**obility **a**ware'. This new shared library contains the enhanced sockets+ version of the Berkeley Sockets and hence allowed calls to be made from an enhanced version of the custom Java socket implementation `java.net.SocketImpl`, it is called `ALICESocketImpl.java`. Also a new `ALICEOutputStream` and `ALICEInputStream` are extended from the standard `java.io.OutputStream` and `java.io.InputStream` respectively.

The final step was to create a custom socket factory which would provide the new mobility layer sockets to the RMI runtime instead of the default sockets. This would now ensure that the ALICE enabled sockets+ were supplied to the RMI runtime, enabling the runtime to cooperate with the ML_{MG} or ML_{MH}, depending on the particular configuration.

4.4 Swizzling Layer (S/RMI)

There are three main problems associated with mobility that are addressed specifically in the swizzling layer. These are:

1. **Gateway Storage of Hosts**: One of the functions of the S/RMI_{MG} daemon is to maintain a list of the MH's that have, in the past, or are presently connected to this MG. The purpose of this data structure is to ensure that the forward pointer mechanism of address migration is maintained correctly for when an out of date invocation should occur.
2. **Client Redirection**: This occurs when an invocation takes place and the server is no longer connected to the MG contacted by the `SwizzledUnicastRef` object in the client stub. We need some mechanism to ensure that the client receives a response and update and can continue the invocation with the new information provided. This is managed by the forward chain of pointers that a server leaves behind

when it relocates from one MG to another.

3. **Server Reference Management**: This occurs when a MH moves from one MG to another MG. After this change takes place the MH must then update the `SwizzledUnicastRef` class to ensure that any future ALICE enabled servers will insert stubs with the correct address at that time. This is performed by the `handoff()` method in the S/RMI$_{MG}$ daemon which manages connection status of ALICE enabled servers. This update if performed transparently to the server and the result is that the server always serves an updated reference.

The three problems listed previously are solved by the different S/RMI components implemented at different places in the architecture. One of the benefits of the redirection scheme is that the home-agent (HA) represents a fall back point for the address management of any ALICE server object. It also maintains the transparent nature of RMI as the management of server references is completely hidden from the application developer.

4.4.1 Gateway Storage of Hosts

Another key area to think about when implementing the forward pointer scheme is to understand how the details regarding the hosts connected to gateways are stored and how they are accessed. Any host that is connected to a gateway must have some key pieces of information stored in a tabular fashion on the gateway. For example, the RMI server below in Table 4.1, “server1” is currently connected to a MG located at 134.226.51.195, it has also specified its home-agent address when it registered with the gateway and its symbolic name, this is useful when the RMI registry is required.

SYMBOLIC NAME	MG	HA
server1	134.226.12.34	134.226.56.78

Table 4.1 Server Address Management

The management of these pieces of information is important as the S/RMI$_{MG}$ needs access to the table when an invocation should occur. If the server has since moved then the MG variable will have been updated and the daemon can reply with the updated address i.e. implementing the redirection scheme discussed in section 4.4.2.

4.4.2 Client Redirection

The nature of mobility allows any MH to move between different MG's in the ALICE architecture. The purpose of the swizzling layer in an instance of relocation is to ensure that the relative server management components are updated in the correct fashion. This enables the redirection scheme to operate accurately and be updated when required. A chaining system and forward pointers are implemented on all server references (`SwizzledUnicastRef`) to ensure that the invocation can be rerouted to a more accurate gateway for processing when required. For example, if a client attempts an invocation on a server object at MG location cs.1 and the server has since moved to MG location cs.2, therefore we need some system to allow the invocation to proceed in light of this change.

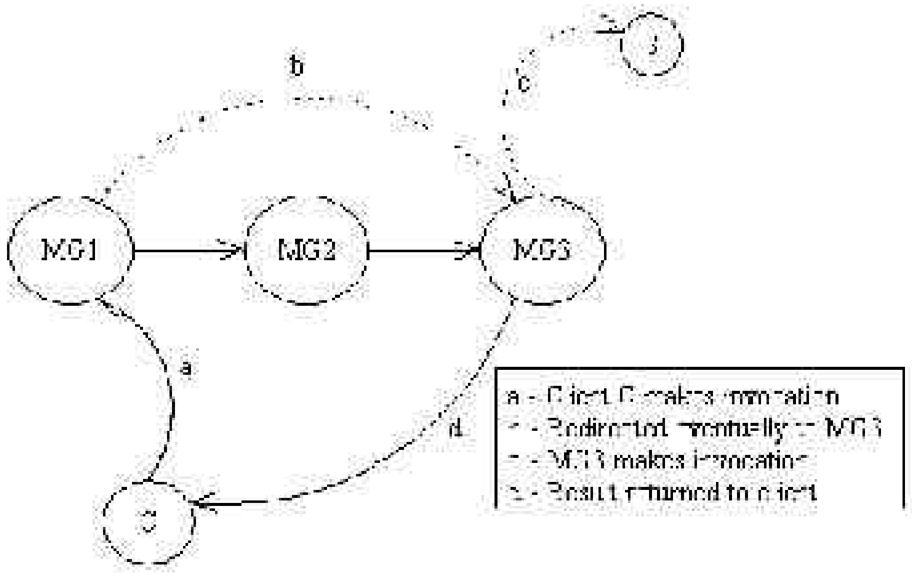

b
c
S
MG1
MG2
MG3
a
d
C
a - Client C makes invocation
b - Redirected eventually to MG3
c - MG3 makes invocation
d - Result returned to client

Fig 4.2 Chaining pointers in ALICE

This is managed using the chaining system illustrated above in Fig 4.2. Here it can be seen that the MG will reply to the invocation with a message either specifying that the server is currently connected to the gateway and the invocation is taking place, or it replies that the server has since moved and sends back the updated MG address, or else it has lost contact with the server currently and tells the client to contact the servers HA.

4.4.3 Server Reference Management

I have already discussed the need to manage the references that a server has in section 4.3.1. This is a core requirement for the success of mobile servers and is managed by the swizzling layer components on the mobile host i.e. S/RMI_{MH}.

This layer manages the addresses held by the host when connected to a gateway. It is vital that when a reconnection should occur, the address of the new gateway is stored and the swizzled unicast references are updated to ensure that new references when published are accurate. This is achieved by integrating the ML upcalls with the swizzling layer component on the MH and causes “reswizzling” to occur when required. The new state is saved internally and the forward pointer on the previous S/RMI_{MG} is updated to preserve the forward pointer chaining algorithm.

4.5 Disconnected Operation Layer

Above the swizzling layer resides the disconnected operation layer. This layer is responsible for long term disconnection management and provides caching functionality on the client side and replication and reconciliation on the server side.

4.5.1 Cache Management

As outlined before, one of the most common problems associated with mobility in a wireless environment is that of disconnection, either short or long term. In short term circumstances, the ML is responsible for the transparent reconnection of sockets but for long term, the management of server objects is handled by implementing caching mechanisms.

Any server object can be cached by a client assuming it implements the `DRMI_Server` interface. This interface consists of the following core methods to facilitate caching and restoration of objects.

- **IsCacheable()** – allows a client to check if a server object is cacheable. Returns a boolean to the client whether true or false. This can depend on the type of object in question, whether the members are serializable or also on security issues like whether is should be available for replication or not.

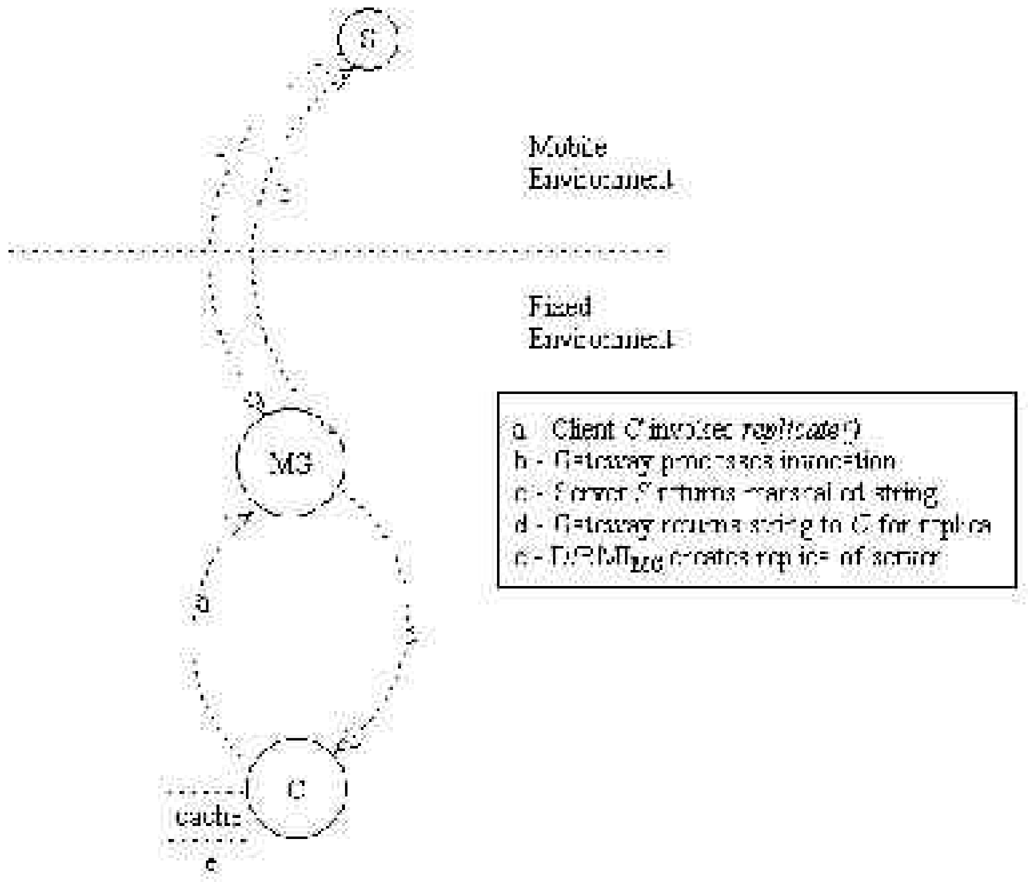
- **Replicate()** – caches the object by serializing it to a string, then returns the string to the client (configured where the client is on the MH). The client then proceeds to construct a replica object using the unmarshalled string from the real object. This is then added to a table of replica object hosted by the D/RMI$_{MH}$ component. From then on, when the client attempts to invoke the server, the invocation is caught by the D/RMI$_{MH}$, executed on the local replica and returned. This continues until the `Reconcile()` method is called and the replica is removed.

- **Reconcile()** - this method when called by the client allows the replica held locally to be reconciled with the real server and all future invocations are then processed correctly by the real object. It has no return type, but it marshalles the replica member values to a string and invokes the `Reconcile()` on the actual object, passing the string as a parameter.

- **Register Callback()**- allows a callback to be registered on a specific server object where required.

4.5.2 Replication

Once a server object has been cached and returned to the client, it must be started so as to allow invocations from clients be caught and carried out on the locally held copy. This is achieved by the D/RMI$_{MH}$ as follows.



S
Mobile Environment
Fixed Environment
MG
C
cache
a
b
c
d
e
a - Client C invokes replicate()
b - Gateway processes invocation
c - Server S returns marshalled string
d - Gateway returns string to C for replica
e - D/RMI$_{MH}$ creates replica of server

Fig 4.3 Replication of RMI servers.

1. When the gateway receives a request to replicate the object from the client, it invokes the `Replicate()` method which is implemented on all ALICE server objects as part of the `DRMI_Server` interface This returns a string representation of the object held on the real server.

2. Once the string is returned, the MH instantiates a replica object locally and inserts the stub into the local registry. It also adds the name of the object to a local caching table, a simple data structure to store objects.
3. From then on, when a client invokes a method on the stub, the invocation is caught by the D/RMI$_{MH}$, the caching table is checked to see if the object is cached locally and if so, a local invocation occurs, otherwise the MH proceeds with a standard invocation via the MG.

4.5.3 Reconciliation

When the client wishes to reconcile the replica object stored locally with the real object it simply invokes the `Reconcile()` method. The S/RMI$_{MG}$ invokes the `Reconcile()` method on the real object implemented via the `DRMI_Server` interface. As a parameter the S/RMI$_{MG}$ passes a marshalled string containing the contents of the replicated object members to the real server. The real object is then makes the update to reflect the changes made during the replicated period.

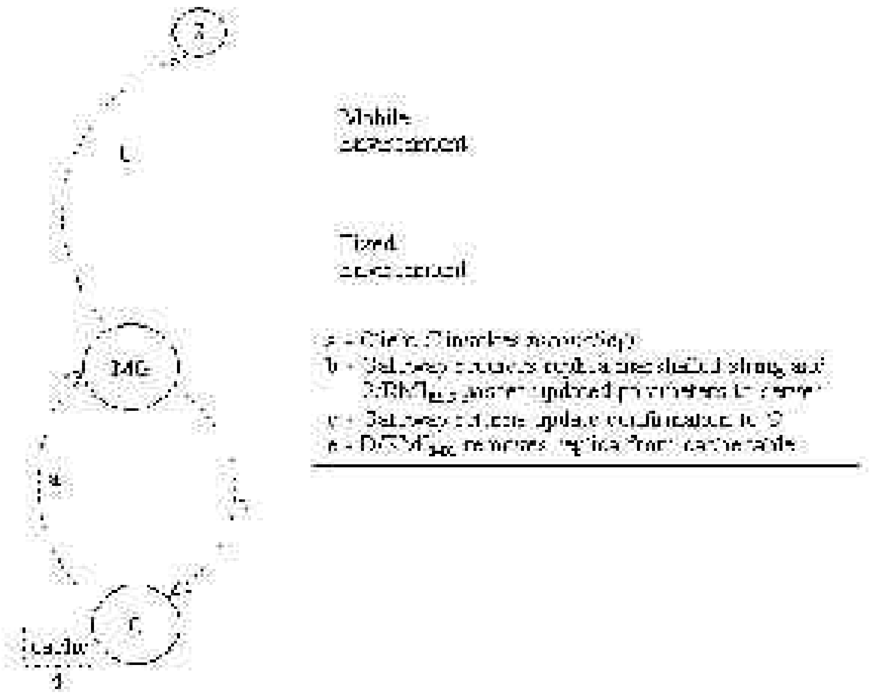


Fig 4.4 Reconciliation of replica servers.

This provides for operation during a period of disconnection. One possible improvement could be a synchronized S/RMI$_{MG}$ component for times when several caches of the same server object are attempting a reconciliation.

CHAPTER 5

-

IMPLEMENTATION

5.1 Introduction

In this chapter I will describe and explain the actual classes and Java code written to implement the ALICE design for Java RMI, as described in the previous chapters. I will explain the Java components in each layer of the architecture and how each class contributes to providing the functionality that the layer is responsible for in the overall architecture.

5.2 Mobility Layer

This layer was already written in C by my supervisor as part of his thesis in 2003 [19]. The purpose of the mobility layer was to allow callbacks on all the swizzling and disconnected operation layer components of the architecture. My role here was to allow the C mobility layer which I had received be integrated with the RMI runtime. This was done using JNI and allowed my code to access the sockets+ API.

5.2.1 ALICE Socket Factory

In a standard RMI application, the sockets provided to the runtime are Berkeley sockets, but there is an alternative to providing these. A developer can supply new specific sockets based on its requirements in the system, for example, there may be a need to manage a firewall constraint. This is done using the RMI socket factory and it is how the ML is integrated with the runtime. Instead of using the standard `Socket` object, I used an `ALICESocket` object to create the socket objects. It extends from the standard socket but it returns a new `ALICESocketImpl` object, which is the actual implementation of the socket and comes from the sockets+ API.

```
public class ALICESocket extends Socket
{
    /* The implementation of this Socket. */
    ALICESocketImpl impl;


    /* Creates an unconnected socket, with the
       new type of ALICESocketImpl. */

    protected ALICESocket()
    {
       impl = (factory != null) ? factory.createSocketImpl() : new
       ALICESocketImpl();
    }
}
```

The following section is taken from ALICESocketImpl.java. It specifies which shared object library is loaded into memory. This specifies that the sockets+ API is loaded for ALICE functionality instead of the standard library. This allows every socket to connect with the ML components and register callbacks should a reconnection occur.

```
/* Enable ML functions (sockets+ API).
   Load mobility aware shared library (libmlmh-ma.so) into runtime. */

static
{
  java.security.AccessController.doPrivileged(new
  sun.security.action.LoadLibraryAction("mlmh-ma"));
  initProto();
}
```

5.2.2 Supplying ALICE Sockets to Runtime

To supply these new ALICE enabled sockets to the runtime I had to specify a `ClientSocketFactory csf` and a `ServerSocketFactory ssf` when creating any remote objects. The following section of code is taken from ALICESocket_RMIClientSocketFactory.java which supplies the new ALICE enabled sockets to the RMI runtime.

```
public Socket createSocket(String host, int port) throws IOException
  {
    InetAddress local = InetAddress.getLocalHost();
    String localHost = local.getHostName();

    if (localHost.compareTo(host) == 0)
      return new Socket(host, port);
    else
      return new ALICESocket(host, port);
  }
```

The following section shows how an ALICE server socket is supplied to the runtime. Taken from ALICEServerSocket_RMIServerSocketFactory.java.

```
public ServerSocket createSocket(String host, int port) throws
IOException
  {
    InetAddress local = InetAddress.getLocalHost();
    String localHost = local.getHostName();

    if (localHost.compareTo(host) == 0)
      return new ServerSocket(port);
    else
      return new ALICEServerSocket(port);
  }
```

5.2.3 Enabling Callbacks

Every time a remote server issued a stub to a client for invocations, the client would then be redirected to a MG when making an invocation. The purpose of the callback was to ensure that if a mobile server relocated and reconnected to the network at a different MG, then the stubs that it would issue after the relocation would then be automatically updated for accurate representation of the current gateway.

The process by which this was achieved was by enabling a callback on the server to reswizzle the references it serves. This was achieved by invoking a method within the `SwizzledUnicastRef` class when a relocation occurred. To enable that method to be invoked then it had to be registered with the ML at runtime, this was done using the following sockets+ methods:

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

These C methods, when accessed using JNI, allowed a Java socket object identified by the `sockfd` parameter in the argument list to register a callback function (CBF) identified by the`cbf` argument in the parameter list, after a reconnect occurs. Essentially I registered a specific socket object from the upper layers when the application began, and from then on if it relocated, the ML would invoke a method of my choosing on the swizzling layer to call and update the references. This is done using the updated versions of the`ALICESocketImpl` that I supplied to the RMI runtime as part of the`SocketFactory`. This file was the actual implementation of the socket object and by changing it allowed the ML to be introduced.

In the following `bind()` method, which is called after a socket makes its initial connection to the server socket, you can see that a method has been registered in case a callback should occur at the ML level. This allows me to specify a method to call to ensure that the integration with the ML will occur on a per-socket basis. I also include the code for the method to execute when the callback has occurred and how it updates the references held by the swizzled references.

The following section is taken from ALICESocketImpl.java and it shows how the new server socket is bound to the specified address and how a callback function is specified for invocation when this server socket object relocates to a new MG.

```
protected synchronized void bind(InetAddress address, int lport)
        throws IOException
    {
        socketBind(address, lport);
        if (socket != null)
            socket.setBound();
        if (serverSocket != null)
            serverSocket.setBound();

        add_callback(fd, "ML_SERVER_CBF");
    }
```

The callback is added here in the final line and the `ML_SERVER_CBF` method is specified as the method to invoke upon relocation. This is the upcall from the ML. Once invoked, the method updates the `SwizzledUnicastRef` class and its static member, the `mg_name` variable. It also updates the previous gateway that the server was connected to. A simple socket is created to the SRMI$_{MG}$ component and the new MG address is sent to it.

5.2.4 Dealing with Callbacks

The callback functionality allows the reswizzling of references by a host and also updates any remote objects running on the host so when they serve new stubs, they are up to date. This is handled by the `SwizzledUnicastRef` class on the host in question. When the reconnection takes place, the sockets+ API invokes the registered callback function and provides three arguments to it:

1. Socket identifier -> allows identification of server object
2. New MG address -> provides up to date connectivity
3. New server port -> updated port where to find server

```
 public static void reswizzleReference(String new_mg)
  {
    mg_name = new_mg;
  }
```

This ensures that any `SwizzledUnicastRef` objects that are created after the method has been executed are now updated with the correct gateway information. Any stubs that have previously been issued to clients or are inserted into the registry will have have an out of date gateway address and will have to use the forward pointer method specified by redirection section 4.4.2.

5.3 Swizzling Layer

The purpose of the swizzling layer is to perform functions that are necessary for servers to be hosted on MH's. Because any client cannot connect directly to a server, it must connect using the mobility gateway, the host replaces its address with the gateway so any client will only ever connect to the gateway when making an invocation i.e. swizzling.

When the S/RMI$_{MG}$ component receives an invocation from a client it then looks up the local table of hosts for the endpoint and makes the invocation, receives the result and returns this to the client.

endpoint = ip_address : server_symbolic_name

5.3.1 Gateway Management of Hosts

For the swizzling layer to operate effectively, it is necessary to manage the hosts connected to the gateway. For this purpose I used a Java `ArrayList` to manage the host objects which were specified using the following class, existingMH. The use of this container was seen as the most efficient type of structure as it allows for dynamic addition or removal of host objects and there is no compile time upper limits on connectivity. This class is a template for the hosts and contains instance variables to monitor the connectivity and position of them.

```
public class existingMH
{
  boolean local;
  String mh_server_name = null;
  String mh_address = null;
  String forward_pointer = null;

  public existingMH(String name, String address)
  {
    local = true;
    mh_server_name = name;
    mh_address = address;
  }
  public synchronized void updatelocal(String newAddress)
  {
    local = false;
    forward_pointer = newAddress; }}
```

5.3.2 Client Redirection

This is performed as part of the swizzling layer component on the MG, and occurs when an invocation request is received at a gateway from a swizzled unicast reference contained in a stub.

All references held in stubs are swizzled, i.e. they refer to a gateway not the actual server object, so when an invocation occurs the gateway must be contacted to process it. If the MH is no longer hosted at this gateway then a forward pointer is fetched and returned or if this is not available then the reply contains the address of the home-agent to fall back on.

To achieve the redirection, the remote reference layer must be changed to introduce the functionality of the `SwizzledUnicastRef`. The swizzled stub is now created using this new extended `UnicastRef` which overrides the `Invoke()` method contained in the standard stubs. This new stub now effectively acts as a wrapper around the superclass `Invoke()` method and ensures the invocation is directed to a gateway to check for the location of the server at that time.

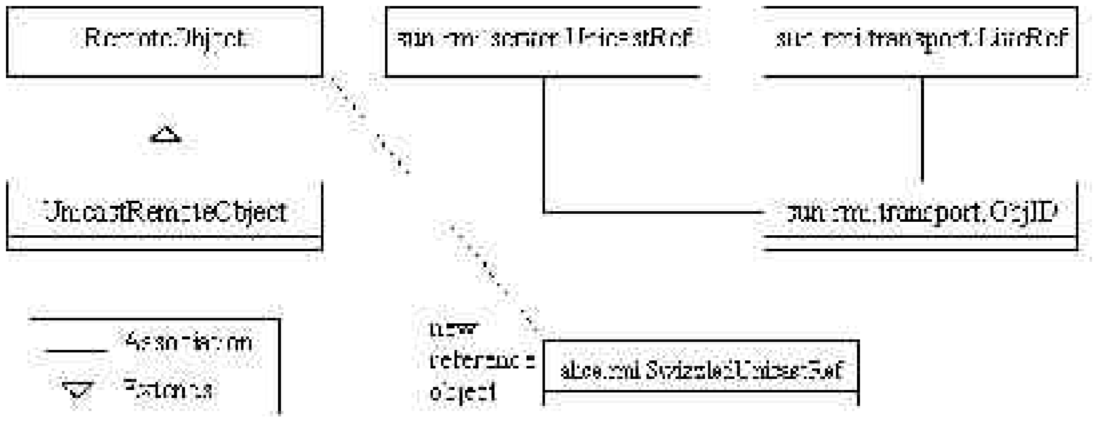


Fig 5.1 ALICE Remote Reference Layer for Redirection

This is implemented as the S/RMI$_{MG}$ component. The daemon has a server socket listening on port 9999 (arbitrary), waiting for requests from swizzled stubs. When a request message is received, the message contains the symbolic name of the server, then the S/RMI$_{MG}$ identifies whether the MH is unknown, locally hosted or has reconnected to a different gateway with a forward pointer. If the MH is local, then the client receives a reply message and proceeds to forward the invocation parameters to the gateway. Otherwise it receives a message to fall back on the home-agent or a forward pointer is given allowing it to implement the chaining algorithm.

/** Taken from SRMI_MG_Daemon.java **/

```
while(true)
    {
        sock = ss.accept();
        openStreams(sock);
        server_name = ois.readObject().toString();
        function = parse_function(server_name);

        switch (function)  {
          case 1:
            oos.writeObject("$LOCAL");
            object_id = get_object_id(ois.readObject().toString());
            process_invocation(object_id);
            break;
          case 2:
            oos.writeObject("$REDIRECT");
            object_id = get_object_id(ois.readObject().toString());
            oos.writeObject(getFP(object_id));
            break;
          case 3:
            oos.writeObject("$HOME-AGENT");
            break;
                               }
    }
```

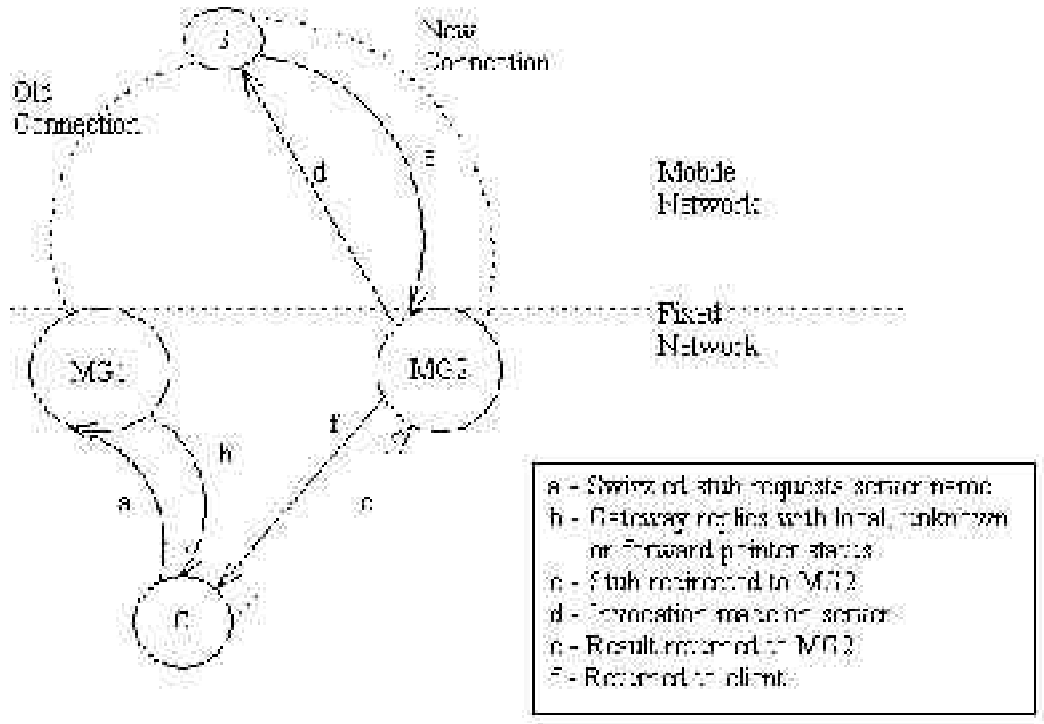

New Connection
Old Connection
Mobile Network
Fixed Network
MG1
MG2
C
a
b
c
d
e
f
a - Swizzled stub requests server name
b - Gateway replies with local, unknown or forward pointer status
c - Stub redirected to MG2
d - Invocation made on server
e - Result returned to MG2
f - Returned to client

Fig 5.2 Invocation example.

'Case 1' above manages an invocation when the host is connected to the gateway at this point. The`object_id` variable refers to the position in the table storing the address details for the actual server and is passed to the `process_invocation()` method. 'Case 2' returns the forward pointer address to the client in question, this address is then used to make a preliminary invocation on that MG. 'Case 3' informs the client that it has no record of the server and tells it to request a MG address from the home-agent.

The `process_invocation`method receives the invocation parameters from the client and makes the invocation on the actual server. This is achieved using reflection as the parameters cannot be serialized as they are of type `java.lang.reflect.Method.` The invocation is then processed by the MG using the standard stub available from the host and returned to the client.

The following section of code shows how the invocation takes place using reflection at the gateway. It is required as the actual method invoked must be passed over an object stream as a string as it is not serializable.

```
Method getFooMethod;
Class[] parameterTypesEmpty = new Class[] {};
Class[] parameterTypesString = new Class[] {String.class};
Class s = server.class;
Class d = DRMI_Server.class;
Object[] arguments = new Object[] {};

getFooMethod = c.getMethod("getFoo", parameterTypesEmpty);

if(method.equals("getFoo"))
{
 resultString = (String) getFooMethod.invoke(ref, arguments);
 oos.writeObject("$RESULT");
 oos.writeObject(resultString);
}
```

This allows the invocation to take place using the gateway as a proxy and implements the forward pointer algorithm quite easily using the simple `ArrayList` Java data structure.

5.3.3 Server Reference Management

This is performed as part of the swizzling layer component on the MH. The purpose is to ensure that after a reconnection happens, the variable containing the address of the gateway is updated to reflect the state of connectivity. The role of this component is to receive an upcall from the ML registered sockets and reswizzle the servers on the host. This ensures future swizzled unicast references issued are up to date at the time of issue. This allows the references to be “reswizzled” - made up to date before they are published.

No effort is made to change previously issued references in the architecture.

```
public static void reswizzleReference(String new_mg)
  {
    mg_name = new_mg;
  }
```

5.4 Disconnected Operation Layer

5.4.1 Cache Management

To allow an object be cached it must implement the `DRMI_Server` interface. This ensures that it implements the `Replicate()` method which marshalles the replica object into a string and then returns that string to the actual server. Using this principle, the caching of server objects can be achieved easily. When the `Replicate()` method is invoked the returned string is demarshalled and used to create a locally stored object that future invocations are made upon until the reconcile method is invoked.

To enforce this, any invocation is initially routed by the `SwizzledUnicastRef` held in the stub to the D/RMI$_{MH}$ component. A simple table is held on the D/RMI$_{MH}$ which stores the names of all locally cached objects and this ensures that when an invocation is made on a cached object, the invocation is made on the local store and not the actual object.

5.4.2 Replication

This allows the server object to return a marshalled string of its current contents, then the D/RMI$_{MH}$ uses this string to construct a replica locally. The following code shows how the replica is managed and stored.

```
// CREATE CACHED OBJECT AT THE MH + PUT STUB INTO MH REGISTRY
String marshalled_replica = ois.readObject().toString();
server_Impl replica = new server_Impl(marshalled_replica);
Registry localReg = LocateRegistry.getRegistry("127.0.0.1");
addToCachedTable(server);
localReg.bind(server, replica);
```

This code results in the cached replica being added to the local table and the stub is added to the registry to allow cached invocations occur. The marshalled_replica is received via an object stream from the S/RMI$_{MG}$ component and then uses a special constructor which unmarshalles the string and creates a new replica server object with the member values received from the actual server.

5.4.3 Reconciliation

Once the object is cached, the `Reconcile()` method allows the replica copy to be reconciled with the actual server object. This is achieved when the client invokes the `Reconcile()` method. The replica values are passed to the S/RMI$_{MG}$ component and the gateway processes the method on the actual server object. The gateway makes the invocation of `Reconcile()` with the updated values via reflection. The following section of code is taken from the S/RMI$_{MG}$ component and shows the replica values being read from the client and then used to make the invocation.

```
Object[] params = (Object[]) ois.readObject();  //READ FROM CLIENT
System.out.println("Reconciling cache with remote server ...");
ReconcileMethod.invoke(ref, params);            //INVOKE RECONCILE
oos.writeObject("$UNCACHE");                    //INFORM CLIENT
```

You can see from the above code that the real object has now been updated with the parameters from the client method call, `params`. Then the `invoke` method is called using reflection and the server reference is used, `ref`.

To finish, a confirmation message is sent to the client which will now remove its cached copy. Now, all further invocations are dispatched to the real server, not the replica which has been removed.

5.5 Problems Encountered

There were two main problems that I encountered when implementing the design of ALICE for RMI. These problems were associated with maintaining the overall objectives of the project, they were not specific to programming issues.

5.5.1 Transparency of Swizzled Stubs

The traditional architecture of RMI is that the server stubs are available to all clients using the lookup service which is the `rmiregistry`. This registry which is hosted on a well known port (1099) allows stubs to be downloaded and then invocations are made on the server using the stub. The purpose of this is to allow any invocation on that stub to communicate with the server object using the standard `UnicastRef` object held in the stub. The problem here is that when the swizzled stubs are created on the server side and added to the registry, thereafter the downloaded stubs then invoke the superclass `UnicastRef` method, which is not the desired result. The expected result is an invocation of the subclass `SwizzledUnicastRef` `invoke()` method which was extended to allow redirection.

The benefit of the swizzled stub is that the invocation will be redirected to a gateway. This breaks the transparent nature of ALICE for RMI as to make a swizzled invocation, the client must dynamically build the stub on his side. This is a fundamental problem as the registry is a core element in the RMI architecture. A possible solution may be to analyze how bytecode for the `SwizzledUnicastRef` class could be made available on the client side using an RMI code base. This would allow a client to dynamically download the bytecode for the stub from a web server over HTTP and allow the stub downloaded from the registry to be cast as a `SwizzledUnicastRef` object.

5.5.2 Non-Serializability of `java.lang.reflect.Method` Parameter

When a stub invokes a method on a remote server, the actual method called is passed over the wire using the JRMP proprietry protocol and is invoked at the server side by reflection. The problem when altering this structure to address the ALICE structure is that this parameter cannot be serialized.

The basic `java.lang.reflect.Method` class does not implement the serializable interface and without altering the standard API's it is not possible to simply pass the method over as an object like the other invocation parameters. So to send the parameter to the MG to invoke the required method I had to get the string version of the method invoked which is possible using the `getString()` method available in the `Method` class and then send this individually using an object stream.

This adds another degree of specialization as the pattern of sending parameters between the MH and the MG is changed to allow the `Method` object be changed to a string value, hence involving more processor time and extra coding on the MG side to identify the method in question.

CHAPTER 6
-
EVALUATION

6.1 Introduction

In this chapter I will look at the performance of two different configurations in a static environment. I will look at the invocation times of simple RMI methods and how they're affected by the swizzling layer and the disconnected operation layers. The performance times are seen to be different and I will attempt to explain these differences.

6.2 One Hop Invocation Times

The one hop invocation configuration is a simple setup whereby the server on the MH is currently connected to the MG that the client invokes upon. There is no need for redirection so the invocation can proceed at the very first attempt. For this experiment, I setup a simple RMI server which implements `get()` and `set()` methods on string variables. This was running on Debian Linux machines over the LAN at Trinity College.

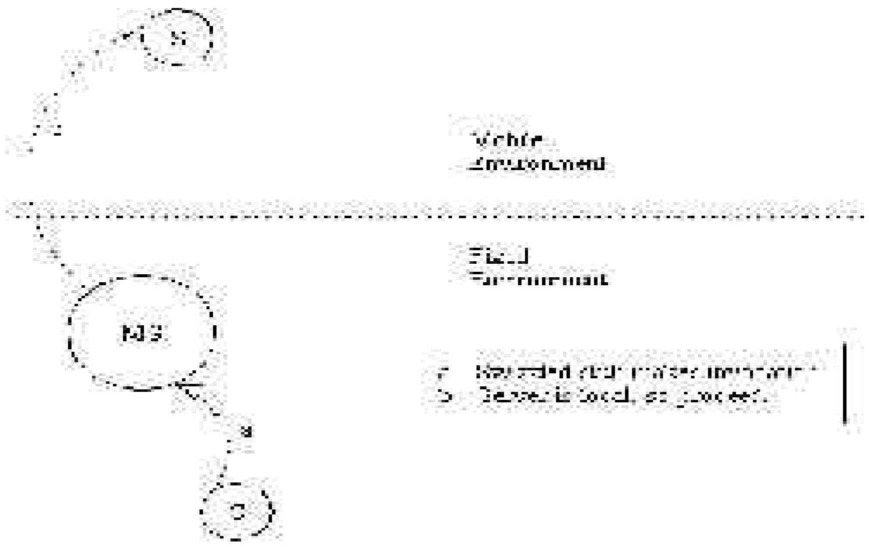


Fig 6.1 One Hop Invocation Configuration

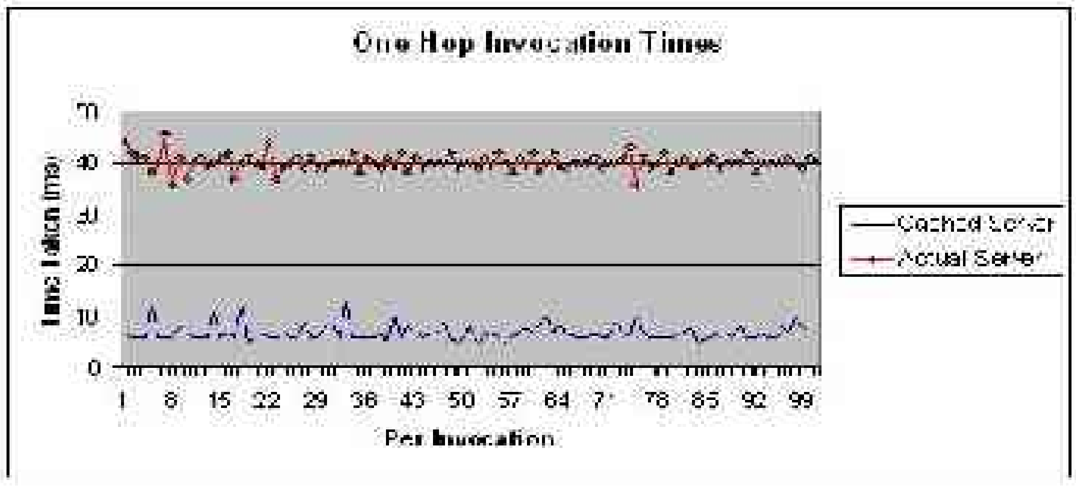


Fig 6.2 One Hop Invocation Times

The results above show the larger invocation times when the server is on the MH. In the cached scenario, the replica server is available on the client and this reduces invocation times which also having a guaranteed connection so improving reliability. The mean cached invocation time is 7ms but the mean invocation time on the actual server is 40ms.

6.3 Two Hop Invocation Times

The two hop invocation is more time consuming on the client side as redirect occurs at the first gateway. The swizzled stub receives notification that the server has relocated to MG2 and must re-invoke on that gateway.

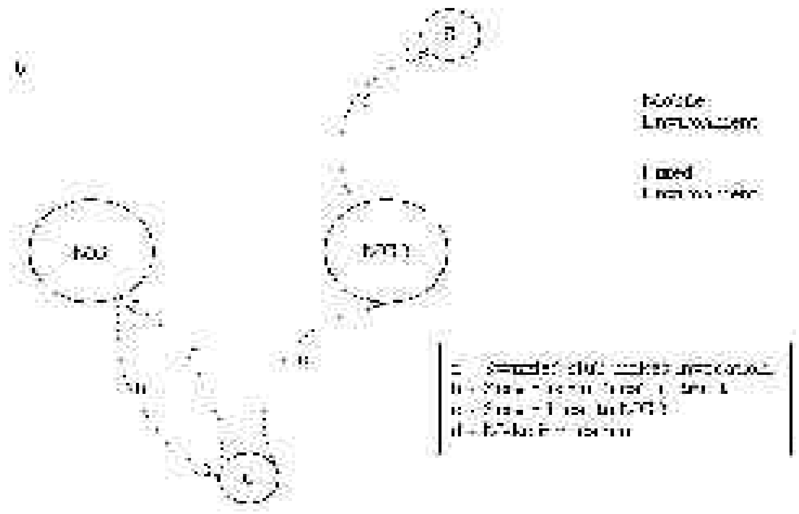


Fig 6.3 Two Hop Invocation Configuration

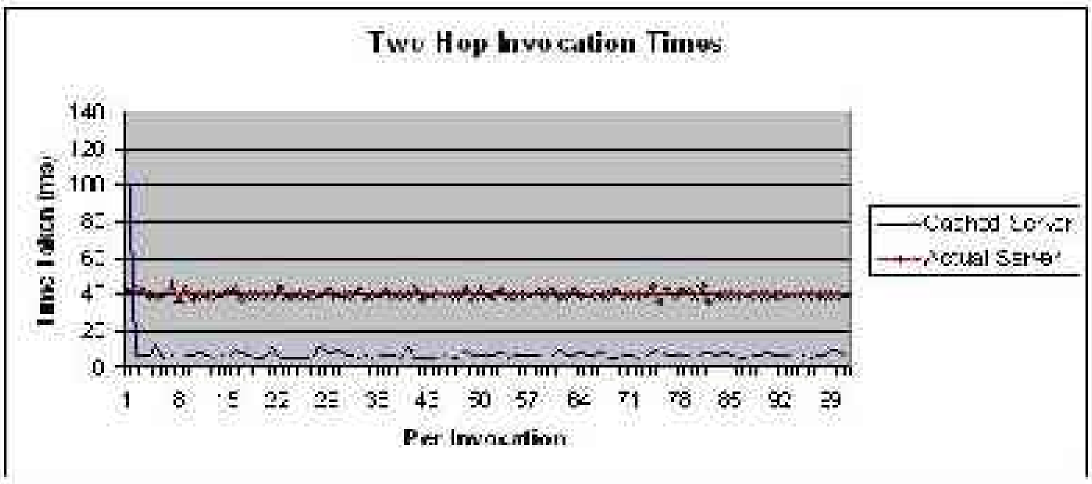


Fig 6.4 Two Hop Invocation Times

In the two hop scenario, the invocation times are similar to the one hop times telling us that the cached replica still shows a significant advantage over contacting the real server on the MH. The problem also associated with the replica is that it has alarge setup time on the client and increases the number or replicas both held in the caching table and the local registry. This is managed by the D/RMI$_{MH}$ component and requires extra time on the processor.

6.4 Summary

It can it seen from the previous two sections that the cached invocations represent a significant opportunity to provide both a relatively quick invocation while also having vastly increased reliability. This presents an improvement in the operation of RMI in a mobile environment and when combined with some time of “anticipation” of disconnection could be an interesting area to pursue for future work.

CHAPTER 7
-
CONCLUSION

7.1 Introduction

This chapter will present the conclusions from the project. It focuses on an analysis of the results and how they reflect on the implementation of RMI for ALICE.

7.2 Achievements

The implementation of both the swizzling and disconnected operation layers was demonstrated during my presentation on this thesis. I was able to demonstrate operation in both the connected and disconnected mode whilst showing complete transparency for the RMI client. Also I was able to demonstrate the redirection mechanism implemented for the swizzling layer and show this in operation with a simple RMI ALICE enabled server on a MH.

Transparency was one of the primary objectives outlined in section 1.4 and is a fundamental test of whether the architecture is successfully implemented in the RMI domain. A minimum amount of changes from the application developers code is a sign that the transparency issue has been placed at the forefront of the middleware designers mind. This was obviously successful in my demonstration as the client had no runtime changes required when making invocations that ran over both the swizzling and disconnected operation layers.

For the mobility layer, all the classes that are required for the integration of it with the RMI runtime are present and the next stage is to test and debug the errors thatoccurred when I tested the supply of ALICE sockets to my test application across a wireless environment.

7.3 Future Work

There are three main areas that I have identified for future work in reference to RMI in the ALICE environment.

- **Anticipating Disconnection:** The operation of the disconnected layer can be seen in chapter 6 to be beneficial to RMI invocation times when operating in a wireless environment. If it is possible to introduce some time of anticipation to the disconnection layer so that a replica could be created without the client having to invoke the `Replicate()` method, there are advantages to this. If a period of disconnection is identified the D/RMI$_{MH}$ component could replicate the object immediately and when the link is restored the `Reconcile()` method would also be invoked transparently to the client.
- **Synchronized Updates:** This would occur when there are many different caches of the same server object on different clients. If several caches attempt reconciliations at the same time, then a possible corruption of the actual server may occur. It is important that some mechanism exists to ensure that the most recent update is held and not a much older held cache on a client. This could possibly be addressed by storing a cache on the MG or by letting a cache remain on the client once it has been reconciled and so simply setting it as an out of use cache for invocations.
- **Stub Transparency:** The final issue is that of the swizzled stub that is downloaded from the rmiregistry. It is vital that the stub when invoked calls the subclass `invoke()` method and not the superclass. This is an area that needs more work to understand why the errors are thrown and how to cast the stub as the required `SwizzledUnicastRef` instance on the client side.

Bibliography

[1] Massachusetts Institute of Technology - IT Dictionary
http://web.mit.edu/powersac/www/glossary.html
Valid on 3/8/04.

[2] Java Remote Method Invocation -White Paper,
http://java.sun.com/products/jdk/rmi/reference/whitepapers/javarmi.html
Valid on 23/2/04.

[3] Common Object Request Broker Architecture - Specification,
http://www.omg.org/technology/documents/spec_catalog.htm
Valid on 30/4/04.

[4] Transmission Control Protocol (TCP) - Specification,
http://www.rfc-editor.org/rfc/rfc793.txt
Valid on 12/8/04.

[5] William Grosso,*Java RMI*, O'Reilly, November 2001

[6] Internet Protocol (IP) – Specification,
http://www.faqs.org/rfcs/rfc791.html.Valid on 12/8/04.

[7] R.E. Shantz et al. "*The architecture of the cronus distributed operating system.*" IEEE Procedings of the 6th International Conference on Distributed Computing Systems, p250-259, May 1986.

[8] Object Management Group (OMG) - Homepage - http://www.omg.org
Found valid on 12/5/04.

[9] Java Message Service – Specification - http://java.sun.com/products/jms
Found valid on 22/8/04.

[10] Connected Limited Device Configuration - White Paper -
http://java.sun.com/j2me/reference/whitepapers
Valid on 8/4/04.

[11] Java Community Process – Special Requests,
http://www.jcp.org/en/jsr/detail?id=66#2
Found valid on 23/7/04.

[12] IBM's Java Based Autonomous Software Agent Technology (AGLETS),
http://www.trl.ibm.com/aglets - found valid on 5/8/04.

[13] Java Based Distributed Agent Architecture, http://www.cougaar.org
Found valid on 5/8/04.

[14] Sumatra Mobile Java Code - http://www.cs.arizona.edu/sumatra/
Found valid on 13/6/04.

[15] Cajo Transparent Distributed Computing – http://cajo.dev.java.net
Found valid on 14/7/04.

[16] Ninja Project API User Guide,
http://www.eecs.harvard.edu/~mdw/proj/old/ninja/ninjarmi-docs/javadocs/packages.html - found valid on 25/7/04.

[17] KaRMI Specification - http://www.ipd.uka.de/JavaParty/api/index.html
Found valid on 3/9/04.

[18] Bluetooth Wireless Technology White Paper,
http://www.bluetooth.org/spec - found valid on 2/9/04.

[19] Cheng-Wei Chen et al. "*Support and Optimization of Java RMI over Wireless Environments*" - National Tsing Hua University, Taiwan, October 2003.

[20] - Gregory Biegel et al."*A Dynamic Proxy-Based Architecture to Support Distributed Java Objects in Mobile Environments*" in Proceedings of the International Symposium on Distributed Objects and Applications (DOA 2002), Lecture Notes in Computer Science, volume 2519.

[21] Mads Haahr, "*Supporting Mobile Computing in Object Oriented Middleware Frameworks*" PhD. Trinity College Dublin, October 2003.