Mobile RMI: Supporting Remote Access to Java Server Objects on Mobile Hosts

Tom Wall
Broadcom Eireann Research Ltd.
Dublin, Ireland
tom.wall@broadcom.ie

Vinny Cahill
Department of Computer Science
University of Dublin, Trinity College
vinny.cahill@cs.tcd.ie

Abstract

Java Remote Method Invocation (RMI) is a specification for building distributed object-oriented applications. RMI was designed primarily for use in conventional, wired computing environments and provides no mechanisms to allow objects hosted by mobile, wireless-enabled computers to interact with other RMI objects. Mobile devices regularly change their point of connection to the Internet making the task of correctly locating and invoking methods on the hosted RMI server objects difficult. The nature of wireless communication also means that the TCP/IP connections used to access these RMI objects are frequently broken, potentially resulting in data being lost and leaving the two communicating parties in inconsistent states.

This paper outlines an architecture that supports such mobile RMI objects and describes an implementation of this architecture and its performance. This architecture provides mobility support in the form of two main components. The first is a session layer component that provides the low-level support services required to maintain transport connections in a mobile environment. The second is an application layer component that uses application-level proxies to address the difficulties of invoking methods on mobile RMI servers.

1. Introduction

The market for portable computing devices continues to grow at an astonishing rate, a fact well illustrated by the huge market penetration achieved by cellular mobile phones. Such phones, when enabled to use the Wireless Access Protocol (WAP) standard, provide the user with a simple yet effective means of communicating with certain fixed network based services. Continuing advances in mobile device and wireless communications technologies are greatly improving the processing, networking, data storage, and display capabilities of affordable Personal Digital Assistants (PDAs). Soon, PDA users will have at their disposal devices capable of interacting with shared software services of types currently available only to users on wired networks. Ideally, these services will be continually and transparently available even when the PDA user is roaming across different sections of the wireless network and even in the face of degradation of the wireless network coverage.

Such services require some form of middleware to allow physically separate software components to communicate with one another and co-ordinate their actions. Commonly used forms of middleware include distributed object technologies such as those implementing the Common Object Request Broker Architecture (CORBA) standard and Java Remote Method Invocation (RMI). These systems are designed to allow a programmer to build complex distributed systems without much knowledge of the underlying low-level networking details.

However, designing distributed object-oriented software for mobile environments presents challenges not encountered while designing for traditional wired systems. The inherent characteristics of mobile computing can essentially be divided into three main considerations: wireless communication, mobility and device portability [1]. Methods of wireless communication are typically much less efficient than wired communication due to environmental interference, low bandwidth and limited range of carrier signals and the associated increases in error and loss rates. Mobility of devices introduces added complexity to the system due to the difficulty of locating a mobile host with no fixed address and communicating effectively with that device as it moves from one point of connection to the wired network to another. Portability considerations restrict the weight and size of mobile devices and hence their computing, power and display capabilities.

The mobility addressed in this paper is of the host or terminal, not the mobility of software objects or code. Although Java does allow objects to migrate between Java Virtual Machines (JVMs), the problems that are solved by the design proposed in this paper are related only to host mobility and the corresponding physical change in the network address of the hosted objects. While sections of the design do involve the movement of Java code from one JVM to another this should not be confused with the mobility of the client or server objects on the mobile host. These objects do not change the JVM in which they are executed.

1.1 RMI

Java RMI [2] is a specification from Sun Microsystems that allows Java objects to invoke methods on objects in other address spaces while preserving the semantics of local method invocations. RMI uses sockets as the communication mechanism between the two JVMs but abstracts the communication interface to the level of a method invocation, hiding the complexities of socket protocols from the programmer. This results in a much simpler and more intuitive means of building complex client-server systems.

An important feature of RMI, and one that most distinguishes it from other RPC-like implementations is its dynamic code loading ability. This allows a JVM to download the implementation of a particular class when required, for example when passed as a parameter to an RMI call or returned as a result of such a call. Another RMI feature known as serialization allows for the member data of a Java object to be to be turned into a stream of bytes to allow it to be transmitted (via whatever transport protocol is being used) to another JVM. This ability for code and objects to move between address spaces is fundamental to the design and operation of RMI.

In RMI any server that wishes to be able to export code sets a codebase property which is tagged onto serialised objects and indicates to clients where a class file for the object can be found. The codebase generally points to a directory or Java archive (.JAR) file that is serviced by a HTTP server. A client receiving a serialized object will try to reconstitute it and will load the relevant class file from the specified codebase. In addition to its dynamic code loading capabilities, RMI provides mechanisms for distributed garbage collection, server replication and activation of remote objects to service requests [3].

As with most other conventional RPC implementations, RMI assumes that both communicating parties have fixed addresses and that the transport connections that are used by RMI do not break frequently. Both of these assumptions do not hold true in a mobile computing environment. This paper presents a design to allow RMI client or server objects to reside on a mobile host which changes its point of connection to the Internet and which uses an unreliable wireless transport connection to interact with other RMI objects. This design is based on the Architecture for Location Independent CORBA Environments (ALICE) [4]. ALICE addresses similar mobility concerns in the Common Object Request Broker Architecture (CORBA) [5]. ALICE provides support for CORBA objects on mobile hosts to interact transparently with other CORBA objects without the need for a centralised location register to monitor the location of the hosts. The ALICE model takes a combined session layer and application layer approach to solve the problems of mobile CORBA objects.

1.2 Roadmap

The rest of this paper describes the technical details of our solution to RMI object mobility. Section 2 discusses related work in the field of host mobility. The various components of the ALICE architecture, their operation and functions are presented in Section 3. Section 4 outlines the differences between RMI and CORBA and the required changes to the ALICE architecture to allow for mobile RMI server objects and then a design for mobile RMI. Section 5 presents the results of performance tests carried out on the design. Finally, section 6 presents the conclusions of the paper and some proposals for future work.

2. Approaches to Host Mobility

The problems associated with mobility can be approached in many different ways and solved at different layers of the protocol stack. Perhaps the most widely known method of handling host mobility is Mobile IP. Mobile IP [6] modifies the Internet Protocol (IP) to allow applications to be mobility transparent and allows for seamless roaming of mobile hosts. This network layer approach does however require that all involved parties use the new network protocol. Mobile IP also requires every mobile host to have a single Home Agent that keeps track of the mobile host’s current location and routes packets addressed to the mobile node to that location. Although route optimisation improvements to the protocol have reduced the communication latency involved somewhat, Mobile IP still presents an element of routing indirection and, in the form of the home agent, requires a centralised location register. Mobile IP also does not address the fundamental problems of maintaining wireless connections to the mobile host as ALICE does.

A much different approach to handling host mobility is illustrated by [7], an end-to-end architecture in which Domain Name Service (DNS) servers are updated dynamically to track host location. Such a design requires no changes to be made to the underlying IP substrate as in

Mobile IP; instead it modifies the transport protocols and the applications at the end hosts themselves. When a host changes its point of attachment to the network it sends an update to a DNS server in its home domain to reflect the location change. Name-to-address mappings are uncacheable by other domains so there can be no stale bindings. The transport connections to and from the mobile host can be migrated transparently between fixed gateways to support continuous communication while the host changes location. Unlike this end-to-end approach, ALICE allows mobility to be fully transparent to applications, providing conventional CORBA applications designed for wired networks with full mobility support.

[8] is a recently proposed design for supporting wireless access and terminal mobility in CORBA. This design envisages wireless mobile terminals (each of which is its own ORB domain) connecting to other ORB domains through gateways called Wireless Access Bridges (WABs). A terminal can have a location register in its home domain to provide a means for other entities to find its current location. The mobile terminal can move between WABs, open connections being tunneled between WABs as the terminal moves to provide a means of processing unfinished object invocations. To overcome the unreliable nature of the underlying wireless transport connections a mapping of the General Inter-ORB Protocol (GIOP) onto UDP is provided. Since UDP itself is unreliable a protocol layer called the GIOP Session Protocol (GSP) is used to provide the required reliability and mobility support. In addition GSP provides fragmentation, error detection, flow and congestion control as well as protection against flooding and masquerade attacks. Overall this design resembles ALICE, especially with the use of a session layer to overcome an unreliable underlying transport layer. As mentioned before however, ALICE does not require a centralised location register to keep track of a CORBA server object's location.

A similar approach was used to implement an RPC mechanism for mobile clients in [9], known as M-RPC. RPC clients on mobile hosts access servers on the fixed network via agents located on mobility gateways called Mobility Support Routers (MSRs). The agents on the MSRs receive RPC requests from the mobile host and then forward them on to the originally intended destination. Library support is provided to RPC client applications to hide the communication details with the agents on the MSRs. Two new transport protocols for use over wireless links were developed for M-RPC, Reliable Data Protocol (RDP) and Indirect TCP (I-TCP) [10]. If an M-RPC client with open connections to a server switches MSRs then the new MSR informs the previous MSR of the movement and the M-RPC agent at the previous MSR transfers state information about the RDP or I-TCP connections to the new MSR agent. Although this approach is similar to that taken by ALICE, it does not allow RPC clients to interact with servers on the mobile host. Providing support for mobile servers is an integral component of the ALICE design.

3. CORBA and the ALICE Architecture

This section provides a brief introduction to CORBA and the ALICE architecture and the operation of its components. The original implementation of ALICE was CORBA-specific but many of the components were found to be equally applicable to other protocols, including Java RMI. A brief overview of CORBA is given first followed by a description of the physical infrastructure required by ALICE and an outline of the ALICE software architecture.

3.1 CORBA

The CORBA [5] specification from the Object Management Group (OMG) describes a means for providing interoperability between objects in a heterogeneous, distributed environment. CORBA allows programmers to access remote objects in a transparent manner. The OMG Object Model defines common object semantics for specifying the externally visible characteristics of objects in a standard and implementation-independent way. In this model, client objects can invoke methods on server objects through a well-defined interface. This interface is specified in the Interface Definition Language (IDL). A client accesses an object by issuing a request to the object. The information sent in the request includes the operation being performed, the Interoperable Object Reference (IOR) of the server object, and any parameters supplied to the method call. IORs are used in CORBA to uniquely identify and locate a server object. An IOR essentially consists of a hostname and port number at which to connect to the server object.

The central component of CORBA is the Object Request Broker (ORB). The ORB provides services to clients to enable them to identify and locate server objects, handle connection management and deliver data. The basic functionality of the ORB consists of passing the requests from clients to the object implementations on which they are invoked. In order to make a request, the client can communicate with the ORB either through the IDL stub or the Dynamic Invocation Interface (DII). The stub represents the mapping between the language of implementation of the client and the ORB. Thus the client can be written in any language as long as the ORB supports that language.

GIOP specifies a standard protocol for communication between ORBs. GIOP is specifically built for ORB to ORB interactions and is designed to work directly over any connection-oriented transport protocol that meets a minimal set of assumptions. The Internet Inter-ORB Protocol (IIOP) element specifies a GIOP mapping onto TCP/IP, the most pervasive transport layer.

3.2 ALICE Physical Environment

ALICE presumes a mobile environment such as that shown in Figure 1. A Mobile Host (MH) connects to a Foreign Host (FH) via a wireless link to a Mobility Gateway (MG) that has a wired connection to the rest of the network. The Mobile Hosts can move between MGs, thereby changing their point of connection to the fixed network. The gateways relay communications from the MH to the rest of the network and from remote hosts to the MH. Gateways also have the responsibility of carrying out CORBA specific duties such as translation of IORs to cater for the mobility of server objects.

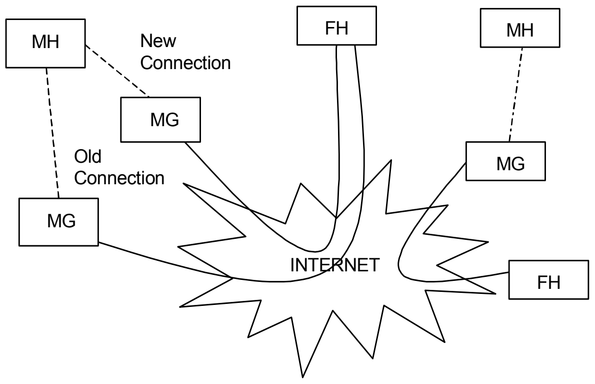

Figure 1. ALICE Environment

The software architecture of ALICE consists of several layers, with different mobility problems being solved at each layer. ORBs generally use TCP/IP at the transport level, however TCP/IP connections are unreliable in a wireless mobile environment and are subject to breakages and high error rates. This can result in data being lost and the client and server states becoming inconsistent. To address this problem ALICE introduces the Mobility Layer, which sits on top of TCP/IP and hides broken connections from the layers above it.

The IIOP layer in the ALICE architecture is mobility-unaware and implements the minimum amount of ORB functionality to allow it to send and receive inter-ORB messages. The S/IIOP (or Swizzling IIOP) Layer is a mobility-aware extension to the IIOP layer and is used to perform address translation on CORBA IORs.

3.3 ALICE Mobility Layer

The Mobility Layer (ML) provides the low-level support services required to maintain connections in a mobile environment. The ML can provide support for any transport protocol and is independent of the CORBA and IIOP specific components of the ALICE architecture. Basically, clients of the ML use it to create what they consider to be normal TCP socket connections. What is instead created is a connection to the current MG, which then connects to the clients' desired communication endpoint using a normal socket connection. Connections from the MH to the MG are multiplexed over a single transport connection in order to conserve the limited and expensive bandwidth available to a wireless device and make the tasks of handoff and connection re-establishment easier. Connection multiplexing also makes the task of error correction easier, a fact that is vitally important in a mobile environment where line quality is quite often poor. If the MH-MG connection breaks it is the task of the ML on the MH to re-establish it.

There are individual message types to indicate whether the MH wishes to establish a connection, shutdown a connection, send data, reconnect after a break, plus corresponding acknowledgements for each type. A special header identifying the type of message, payload length, an identifier for the destination, etc. prefixes all data sent. In addition to transparently re-establishing a broken connection, the ML must also cache any data sent and wait for an acknowledgement for this data. Data being sent is first cached, along with the Logical Connection Identifier (LCID), a unique identifier allocated to each virtual connection and the request identifier, which is used to identify the acknowledgement of a packet. To increase efficiency, the ML will delay opening a connection for a socket until there is actual data to be sent or received for it.

The ML provides four main services to the layers above it [4] –

- It hides broken connections by transparently restoring links when they are lost.
- It allows TCP ports on the MG to be allocated by the IIOP Layer to accept incoming connections.

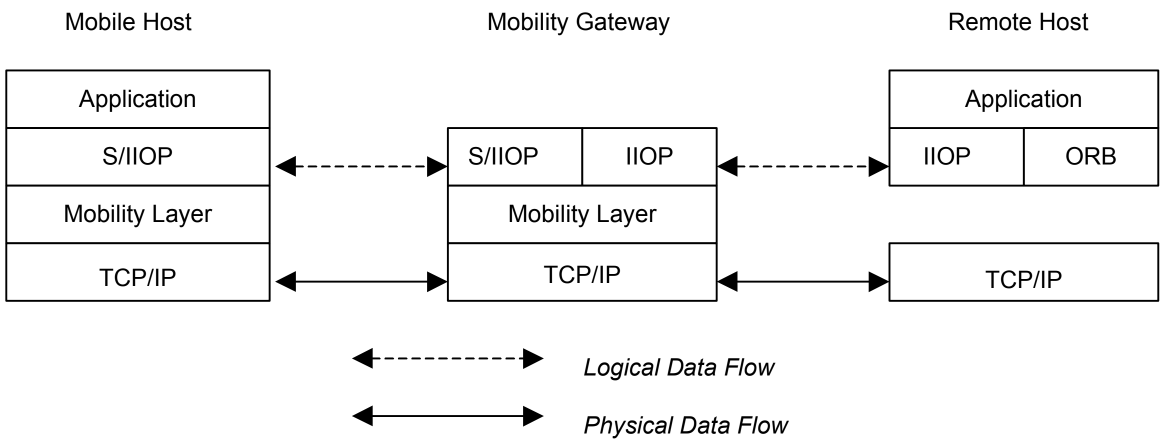

Figure 2. ALICE Software Architecture

- It provides mobility information to the S/IIOP layer so it can translate addresses and forward requests.
- It performs handoff between MGs and tunnels existing connections from the old MG to the new one.

3.4 IIOP and S/IIOP Layers

The IIOP standard specifies how inter-ORB messages should be sent using a TCP/IP transport connection. The IIOP layer developed for ALICE was designed to be as efficient as possible and to have a small memory footprint to accommodate the limitations of mobile devices. The API for the IIOP Layer also hides much of the complexity of the actual protocol from the application programmer while still allowing for manipulation of relevant parameters when required. In addition, the IIOP Layer allows the ML to be plugged in and out cleanly whenever mobility support is necessary or not.

The S/IIOP Layer performs functions that are necessary for the IIOP layer to operate correctly when server objects are hosted on the mobile host. As noted above, an IOR essentially consists of a hostname and port number at which to connect to the server object. Server objects on the MH export IORs that point to the MH. Since no remote host can directly contact the MH this is useless. To overcome this problem the S/IIOP layer on the MH replaces the hostname with that of the current MG in a process called 'swizzling'. The S/IIOP layer uses the Mobile Layer to obtain information about the MH's current MG.

When a remote host receives the swizzled IOR and contacts the MG, the S/IIOP layer on the gateway will forward the request to the MH. When the MH changes it's point of connection to the network to a different MG, it must 'reswizzle' any IORs to point to the new MG. The S/IIOP layer on the old MG will also change any IOR's it holds that pointed to the MH to point to the new MG.

3.5 Handoff

The limited range of wireless communication mechanisms means that roaming mobile hosts must change their mobility gateway at regular intervals. To do this the mobile host will cause handoff to occur between the new gateway and the old one. The host will send a Handoff Request message to the new MG stating the address of its last MG and the identifiers of any logical connections that were in use [4]. The new MG will then negotiate the handoff of each of these logical connections from the old MG. In doing this the contents of each of the caches containing unacknowledged data, acknowledgements received and any unsent data are transferred to the new MG and will be sent to the MH as soon as is appropriate. When the handoff procedure is complete the old MG sends a Finished Handoff message to the new MG, which will then send another Finished Handoff message to the ML on the mobile host.

Any transport connections that were open between the old MG and remote hosts will be tunnelled to the new MG for as long as they remain open. This leaves open the possibility of the creation of a long chain of MGs each tunnelling open connections to the next without having any knowledge of where the chain ends or any means of shortening the chain. This should only prove to be the case on rare occasions.

4. Mobile RMI Design

The ALICE architecture provides a useful model for extension to other distributed object technologies, Java RMI being an obvious candidate for this treatment. RMI and CORBA both provide the programmer with a means of transparently invoking methods on remote objects. CORBA however is language independent whereas RMI is a Java-only implementation. This makes the task of programming in RMI significantly simpler than CORBA and allows for some of the useful features mentioned in Section 3 such as dynamic code loading. The differences between RMI and CORBA however do mean that the task of creating an architecture to support mobility in RMI is not as trivial as converting the C/C++ ALICE code to Java. The significantly different approaches taken by RMI to object addressing and naming mean that a new technique of addressing remote objects is needed. Instead of altering object references at the mobility gateway as in the original CORBA implementation of ALICE, we instead introduce application level proxies (implementing the same interface as the mobile server object) to relay method invocations to the server and return data to the client.

Of the main components of the ALICE architecture, the Mobility Layer can easily be used with RMI as it is protocol independent. However, Java applications cannot directly interface with the original ALICE Mobility Layer as it is coded in C. There were two possible ways to provide Java applications with access to the ML; one was to completely recode the ML in Java, and the other was to add a Java Native Interface 'glue code' layer to the existing C code. The first option was taken once it was discovered that the size and complexity of the glue code would far outweigh the costs of a full Java rewrite of the layer.

4.1 Operation of the Mobility Layer with RMI

When a Java RMI program requires a socket to be created it calls the createSocket() method in java.rmi.RMISocketFactory which returns a socket to use. The default socket factory returns an ordinary java.net.Socket or java.net.ServerSocket if a server socket is required. However, RMI supports programmer-defined custom socket factories that return other types of socket if a custom transport protocol is to be used rather than TCP, which RMI uses by default. The Mobile RMI design uses

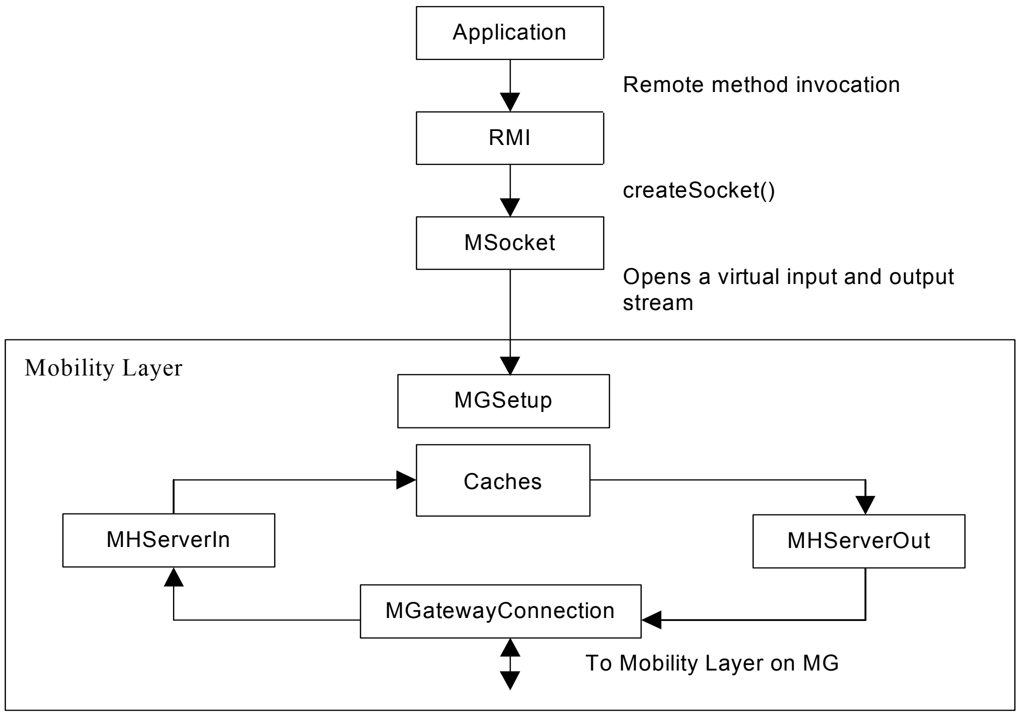

Figure 3. Normal operation of the Mobility Layer

a custom socket factory to return a custom socket (called an MSocket) to the RMI applications. MSockets implement the functionality of the Mobility Layer including multiplexing socket connections onto a single transport connection, caching of sent data and data acknowledgement.

The basic operation of the Java Mobility Layer is illustrated in Figure 3. When an RMI application requires a socket it makes a createSocket() call to the RMISocketFactory which returns a reference to an MSocket. If this is the first MSocket to be created then the MGSetup thread first initialises the ML and connects to the ML on the local gateway by creating an instance of the MGatewayConnection class. The application can then write to the MOutputStream, which places the data in the cache from where the MHServerOut thread will write it out over the connection to the gateway. Data returned from the gateway will be placed in a cache by the MHServerIn thread from where it will be read by the MInputStream and returned to the application.

4.2 Invoking Methods on Mobile RMI Server Objects

The Java version of the ML provides RMI applications with enough mobility support to function as mobile clients (see section 4.3). It does not provide enough support for mobile servers, however, for a number of reasons, the most important being that the mobile host is not directly accessible to remote hosts and can only be accessed through the gateway. Therefore even if a remote client held a stub referring to a server object on the mobile host it would not be able to successfully contact it. The CORBA version of ALICE solved this problem by introducing the S/IIOP layer.

The S/IIOP layer however is CORBA dependent and relies on the fact that IORs can be altered at runtime. This allows IORs to be changed to point to the gateway instead of the MH. When invocations arrive at the MG the IORs are simply reswizzled and then the request redirected to the MH.

The object addressing scheme in RMI is substantially different to that of CORBA. The RMI equivalent to the IOR, the RemoteRef, cannot be created independently of a remote object nor can it be accessed or manipulated at the application layer. In addition, all client-server interactions in RMI do not pass through a single entity as invocations and returns pass through an ORB in CORBA. In RMI once a client has received an object reference from the server's registry it communicates directly with the server object. Hence the swizzling approach to addressing mobile objects cannot be used with RMI and a different means must be employed.

Our approach is to use application-layer proxies to relay method invocations and returned data between the client and the server through the gateway. This allows the difficulties of address translation and request forwarding to be solved at the application layer instead of a lower level in the protocol stack. This also eliminates the requirement for an RMI version of the S/IIOP layer since there is no manipulation of RemoteRefs. When the mobile server changes gateway its proxy on the old gateway is updated to point to the new proxy on the new gateway. This behavior mimics the 'reswizzling' of IORs on the old gateway that occurs during handoff in the CORBA version of ALICE.

4.3 Mobile Host as Client

The ML provides all of the support necessary for a mobile RMI client to interact transparently with a remote server. The ML tunnels the lookup requests, method invocations and all other interactions initiated by the client with the server through the mobility gateway. Any data sent by the client over the virtual connection is sent to the mobility gateway and redirected from there to its intended target. Similarly, data returned from the server to the gateway is forwarded to the client.

If the transport connection between the mobile host and the gateway breaks at any point then the ML on the mobile host will transparently reconnect and any data lost during transmission will be resent. If the mobile host hands off to another gateway then any existing connections to the remote server will be tunneled between the old and new gateways for as long as they exist. The RMI stub for the remote server object held by the mobile client is still valid for making calls through the new gateway as the server host has not changed location.

4.4 Mobile Host as Server

When an RMI server is located on a mobile host the ML will provide the same low-level support as described in the previous section. As remote hosts cannot directly contact a mobile server, all communication must go through whichever MG the server is currently connected to. Since the client cannot hold an RMI stub that refers directly to the server on the mobile host, it is instead given a stub that refers to a proxy object on the gateway. The mobile server gives the code for the proxy to its current MG to use in forwarding incoming calls

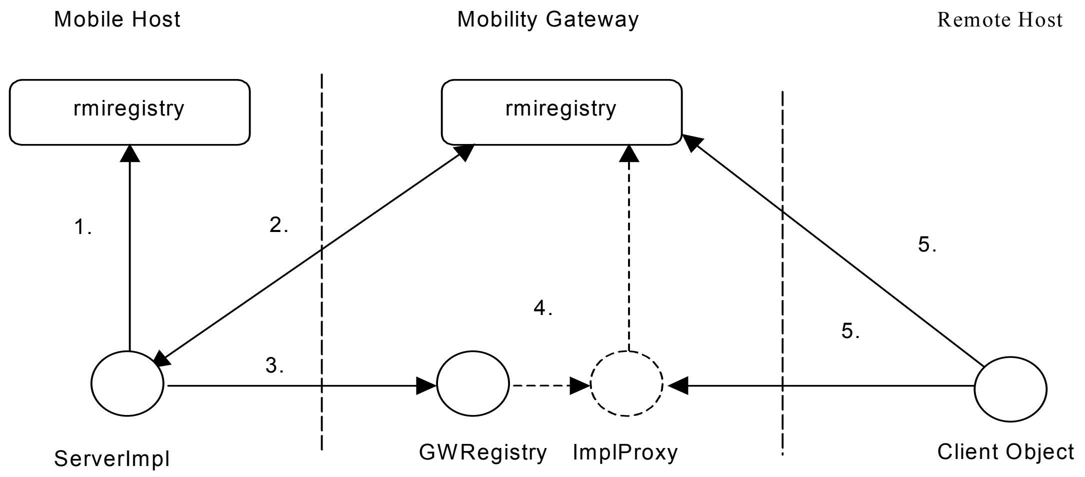

Figure 4. Procedure for passing Proxy Object

to the actual server on the mobile host. To facilitate the uploading and registration of proxies from mobile servers, a special remote object called the GatewayRegistry is registered on the gateway. The server can download the stub for this object and call various methods on it to pass the proxy class and associated parameters to the gateway. The proxy class implements a well-known interface called MobilityProxy. The GatewayRegistry creates an instance of this proxy class and calls a method on it that downloads its RMI stub from the server and then registers it with the local rmiregistry. The procedure for passing the proxy and registering it are shown in Figure 4 above.

The procedure follows the following steps:

1. The server object, ServerImpl (which implements some interface called Server) starts execution on the mobile host and registers itself with the local rmiregistry.

2. ServerImpl contacts the rmiregistry on the gateway and downloads the stub for the GatewayRegistry.

3. ServerImpl then invokes a method called register() on the GatewayRegistry object, passing as parameters the name of the Proxy class and the address of a web server where it and all associated classes can be found.

4. The GatewayRegistry object then downloads the Proxy and any implemented interface classes from the mobile host and instantiates a proxy object (here called ImplProxy) and registers it with the local rmiregistry.

5. A client object located on a remote host can then contact the gateway's rmiregistry and obtain a stub for the ImplProxy object. Invoking methods on the ImplProxy object will cause it to download the stub from the ServerImpl object and forward the calls to it. Any data returned to the ImplProxy is then returned directly to the client.

A complication arises when an invocation of a remote method on the server returns a reference to another remote object. When a reference to a remote object is returned from a remote method call, RMI returns the stub for the remote object. Returning this stub to the client via the proxy is pointless, as the client cannot contact the mobile host itself. Instead the ServerImpl gives the GatewayRegistry not only a proxy class for itself, but also proxy classes for any remote object type that it returns in a method invocation. When a method call on ServerImpl returns a reference to a remote object the ImplProxy can create a separate proxy for the returned object. The stub returned from the mobile server is passed to the newly created proxy that will then use it to forward calls to the original remote object on the mobile host. The ImplProxy will then return a stub for this proxy object to the client and the new proxy will relay any calls to the actual object in the same manner as for the ImplServer and ImplProxy objects.

4.5 Home Mobility Gateway

An inherent difficulty in distributed object systems is finding a reference to a remote object to communicate with. In a conventional RMI or CORBA application, the address of the server could be supplied to the client as a command line argument, read from a file, hardcoded into the application itself etc. In the ALICE architecture however, a mobile server does not have a fixed address and can only be contacted via its current mobility gateway. To provide prospective clients of the mobile server with a means of finding its current location, we introduce the concept of a Home Mobility Gateway (HMG). The HMG is a gateway that is permanently associated with the mobile host, perhaps one that the MH spends the most amount of time connected to, such as a gateway near the users home or office. Clients wishing to connect to the mobile server connect to the HMG, lookup the service on the rmiregistry there and are returned a stub for the proxy on the mobile host's current gateway. The stub for the server's proxy on the HMG must of course be updated every time the MH changes gateway. The MH notes the address of the HMG so that it can be passed to a new gateway after handoff. This is discussed in the following section.

4.6 Handoff in Mobile RMI

Handoff of the mobile host from one gateway to another should be transparent to both the mobile RMI server application and any remote clients of that server. To make handoff transparent to clients we must ensure that the following requirements hold true:

1. Method invocations that are being processed when handoff occurs are completed.

2. Clients that currently hold a stub for a proxy on a previous gateway can still communicate with the server through this proxy.

3. The proxy on the Home Mobility Gateway is updated whenever the server changes gateway.

4. New clients looking up the server on one of the server's previous gateways are given a stub to talk to the current gateway.

The first requirement is satisfied by the use of the ML since all currently open transport connections between the MH and the previous MG are tunneled between the two gateways until they are closed. Although this introduces some delay to the method invocation it does allow it to be completed successfully and also transparently to both communicating parties. Addressing the remaining three requirements needs the co-operation of both the ML and the RMI proxies on the gateways.

The second requirement is essential to the operation of the system. Since the server's mobility is transparent to the client, there is no way to prompt the client to update the proxy stub it holds by downloading one from the current MG. When the client looks up the rmiregisty on a gateway and receives a proxy stub, it expects to be able to use this stub for as long as it wishes to communicate with the mobile server. Therefore a proxy on a previous gateway must itself relay the clients' method invocations to the proxy on the current MG. This is accomplished by having the GatewayRegistry object on the new gateway contact the corresponding object on the previous gateway and supply it with the stub for the new proxy. The proxy on the old MG replaces the stub for the mobile server object (which of course it can no longer contact) with the stub for the proxy on the current MG. Subsequent method invocations on the previous MG's proxy will be relayed to the current MG's proxy and from there onto the server. In this way a 'chain' of proxy objects is set up between the MH and the client, which is unaware of communicating with anything other than the old MG.

Satisfying the third requirement is simply a matter of the new MG contacting the previous MG and giving it the stub for its proxy. The previous MG will register this stub in its rmiregistry in place of the old stub that pointed to the mobile server. If there are any older MGs in the chain then these will be contacted in the same way and their stubs updated. Similarly, the new gateway will contact the HMG and update its' stub as well, satisfying the fourth requirement.

To explain how handoff is implemented in the Mobile RMI model we will start by assuming that the system has been initialized so that the fixed client has downloaded the ImplProxy stub and is able to call methods on the remote server object on the mobile host via the ImplProxy. Handoff to a new mobility gateway is achieved as follows:

1. The Mobility Layer on the MH sends a Handoff Request message to the new MG and all existing server-client connections through the old gateway are tunneled through the new gateway. The MH

downloads the stub for the GatewayRegistry object from the new gateway.

2. By calling the register() method on the GatewayRegistry stub, the ImplProxy and all associated classes and interfaces are uploaded to the new MG. The ImplProxy is instantiated and registered with the rmiregistry on the new MG.

3. MH calls a method on the GatewayRegistry stub from the new gateway that causes the GatewayRegistry service on the new MG to download the stub for the GatewayRegistry service from the old MG.

4. The new MG calls a method handoff(…) on the GatewayRegistry stub from the old MG. This forwards a call to the method changeStub(..) in the ImplProxy object on the old MG, causing it to discard the stub it had previously downloaded from the MH. The ImplProxy replaces this stub with one downloaded from the new MG.

5. The old MG repeats step 4 for an older MG in the chain if one exists. This is repeated until all previous MGs have been contacted and their stubs updated.

6. MH calls a method notifyHMG(…) on the current GatewayRegistry that causes it download the stub from the HMG and call the method changeStub(…) on it. This uploads the current MGs proxy stub to the HMG where it is registered in the rmiregistry.

5. Performance

The purpose of these tests was to obtain approximate results for the performance of the Java Mobility Layer and RMI proxy scheme. The results shown in Figure 5 below compare the performance times of method invocations between ordinary RMI over a wireless link with those obtained when the calls are being made through a proxy and with Mobility Layer support enabled. In all of the tests the server host used was a laptop using the Windows 98 operating system and equipped with a WaveLAN wireless LAN card. The mobility gateway and remote client were desktop PCs using Solaris OS with wired LAN connections.

As can be seen from the table above the data caching, multiplexing and other functions carried out by the Mobility Layer introduce significant overhead to the process of invoking a remote method. However, the vast majority of the delay is caused by widespread use of Java Thread.sleep() statements used to facilitate the synchronization of different components of the code. As this is still only a prototype version of the Java Mobility Layer it is realistic to assume that the invocation time with the ML could be reduced by at least an order of magnitude when these sleep statements are removed. Further optimisation of the manner in which data is read from java.net.InputStreams would also improve the performance considerably.

6. Conclusions and Future Work

This paper has discussed how the mobility support provided to CORBA applications by the ALICE architecture can be applied to Java RMI. This project proved that a valuable feature of the original ALICE architecture was the separation of mobility related issues from CORBA-specific issues. The Mobility Layer solves mobility issues in a protocol independent manner and thus a simple recoding of the layer in Java was sufficient to provide the same support to RMI. The CORBA request forwarding techniques used in ALICE's S/IIOP layer were found to be incompatible with RMI and so application layer proxies on the mobility gateways

Figure 5. Average method invocation times

Message Size (bytes)	Invocation Time with ML (millisecs)	Invocation Time without ML (millisecs)
8	1910	17
256	1930	21
384	1940	22
512	4770	49
768	4810	49
1024	7600	51
1280	7730	52
1536	7820	56

were implemented to forward method invocations to the mobile server. The use of the Java Mobility Layer and the proxies proved to be a simple and effective means of allowing mobile RMI servers and clients to interact with remote hosts with no knowledge of their mobility.

These application layer proxies were straightforward to implement. Those sections of ALICE that were independent of CORBA and IIOP (i.e. the Mobility Layer) were coded in Java for use by RMI applications. The initial performance tests carried out on the architecture show that much work remains to be done on improving the speed of method invocations through the Mobility Layer and the proxies. The delay currently experienced should be massively reduced following such improvements. Handoff in the Java Mobility Layer also remains to be implemented.

Further work is required to make the measures to support mobility completely transparent to the programmer of a mobile server. The proxy classes exported by mobile servers have a generic format. The only difference between proxy classes exported by servers implementing different remote interfaces is the name of the proxy classes, the name of the remote interfaces that the proxies implement, and the inclusion of the method signatures of the remote interface. As a result, it is possible for the proxy classes to be automatically generated provided the remote interface and the proxy class name are supplied as arguments. The code to pass the proxy class to the MG can be hidden in the RMI source code so that a call to register an RMI object with the local registry will automatically notify the current MG. The MG will then download the server interface and the server objects stub and automatically create a proxy object for the mobile server.

Extending the design to incorporate the Jini technology [11] provides an interesting solution to some of the difficulties inherent in the system. For example, the Jini lookup service provides a client with a means of finding a service that matches its supplied preferences so that the client does not need to know the precise address of the server's current gateway or its Home Mobility Gateway in order to access a service. In addition, the Jini registrar provides a simple means of supplying the mobile server objects stub to the lookup service. This functionality is recreated in our Mobile RMI design when the server passes the GatewayRegistry the location of the HTTP server to download the stub class from. Thus aspects of Jini complement and neatly dovetail with the Mobile RMI architecture.

References

[1] George H. Forman and John Zahorjan. *The Challenges of Mobile Computing*. IEEE Computer Journal, April 1994.

[2] Sun Microsystems. *Remote Method Invocation Specification*. http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html

[3] Ann Wollrath, Roger Riggs, and Jim Waldo. *A Distributed Object Model For the Java System*. In Conference on Object Oriented Technologies, Toronto Ontario (Canada), 1996.

[4] Mads Haahr, Raymond Cunningham and Vinny Cahill. *Supporting CORBA Applications in a Mobile Environment*. In Proc. of MobiCom99, Seattle, WA, pages 36--47. ACM, August 1999.

[5] Object Management Group. The Common Object Request Broker: Architecture and Specification, V2.2. Object Management Group, February 1998.

[6] Charles E. Perkins. *Mobile IP*. IEEE Communications Magazine, vol. 35, no. 5, pp. 84-99, May 1997.

[7] Alex C. Snoeren and Hari Balakrishnan. *An End-to-End Approach to Host Mobility*. 6th ACM MOBICOM, August 2000.

[8] Joint Nokia/Vertel Response to OMG Request for Proposal. OMG Document telecomm/99-05-05. ftp://ftp.omg.org/pub/docs/telecom/99-05-05.pdf

[9] Ajay Bakre and B.R. Badrinath. *M-RPC: A Remote Procedure Call Service for Mobile Clients*. Proceedings of the 1st ACM Mobicom Conference, 1995, pp. 2-11.

[10] Ajay Bakre and B.R. Badrinath. *I-TCP: Indirect TCP for Mobile Hosts*. 15th. Int. Conference on Distributed Computing Systems (ILCDS), May, 1995.

[11] Ken Arnold, Bryan O'Sullivan, Robert W. Schiefler, Jim Waldo, Ann Wollrath. *The Jini Specification*. Prentice Hall, July 1999.