A Dynamic Proxy Based Architecture to Support Distributed Java Objects in a Mobile Environment

Gregory Biegel, Vinny Cahill and Mads Haahr

Distributed Systems Group
Department of Computer Science
University of Dublin, Trinity College
Dublin 2, Ireland
{Greg.Biegel, Vinny.Cahill, Mads.Haahr}@cs.tcd.ie

Abstract. Java Remote Method Invocation (RMI), as a distributed object technology, has poor existing support for operation in wireless mobile computing environments. The use of RMI in a mobile environment poses a number of problems related to hardware mobility and the characteristics of wireless networks. This paper describes an implementation of an architecture supporting RMI client and server applications in a wireless mobile environment. Mobility support is provided for in two major components. Connectivity management manages wireless connections and hides the inherent unreliability of wireless media from higher layers. Location management addresses the difficulty of correctly locating and invoking RMI server objects hosted by mobile devices. The implementation is evaluated in terms of transparency and the associated cost of introducing mobility support for RMI applications.

1 Introduction

Java Remote Method Invocation (RMI) [1] permits the invocation of methods on Java objects residing in remote address spaces. As well as being an important distributed object technology in its own right, other platforms such as Enterprise Java Beans (EJB) [2] and JINI [3] use RMI for communication. The widespread adoption of Java technology has led to an increasing number of small mobile computing devices with built in Java capability. RMI was designed primarily for use in static wired networks and in its present form, RMI does not support operation in mobile computing environments well.

Mobile computing environments [4] have a number of constraints not present in wired networks which need to be accounted for when providing support for applications in such environments -

Device Constraints: Mobile devices, by definition, are smaller and lighter than their static counterparts and have more limited display, processing, and power capabilities.

Network Constraints: Mobile devices rely on wireless communications which are presently characterised by low bandwidths, high latencies, and intermittent connectivity. The Architecture for Location-Independent Computing Environments (ALICE) [5] was designed to provide mobility support for a range of client/server application protocols through the introduction of connectivity management addressing the characteristics of wireless networks such as high latency and low bandwidth. ALICE is independent of application protocols and hence may be maintained across different distributed object technologies.
Location Management: Mobile devices change their point of connection to the network frequently, and this may invalidate references held to objects resident on these devices. This invalidation of references poses a problem to clients attempting to contact objects on such devices. In this paper, we present a solution to the problem of location management of RMI servers in mobile environments based on dynamic proxies. We demonstrate how dynamic proxies may be used to redirect method invocations between fixed and mobile devices in a manner that is largely transparent to the client, and does not dramatically impact on the time taken per method invocation.

The resulting implementation permits the transparent operation of both RMI clients and servers within a mobile environment.

2 Roadmap

The remainder of this paper describes our solution for supporting RMI applications in mobile environments. Section 3 provides some background. Section 4 examines the integration of the application protocol independent module of ALICE into the RMI runtime in order to support mobile RMI clients. Section 5 presents our solution, based on dynamic proxies, to the problem of location management for the operation of mobile RMI servers. Section 6 describes our implementation, whilst Sect. 7 presents a performance evaluation. Section 8 introduces related work and finally Sect. 9 presents our conclusions.

3 Background

This section provides some background on the the mobile environment and the ALICE architecture, Java RMI, and dynamic proxies.

3.1 The ALICE Architecture

ALICE is an architectural framework that provides mobility support for a range of application-level client/server protocols [5]. The ALICE architecture allows such application-level protocols to provide their own mobility support through location management, disconnected operation support, and connectivity management. ALICE permits the operation of mobile servers with no centralised location register to keep track of the whereabouts of the servers. The physical

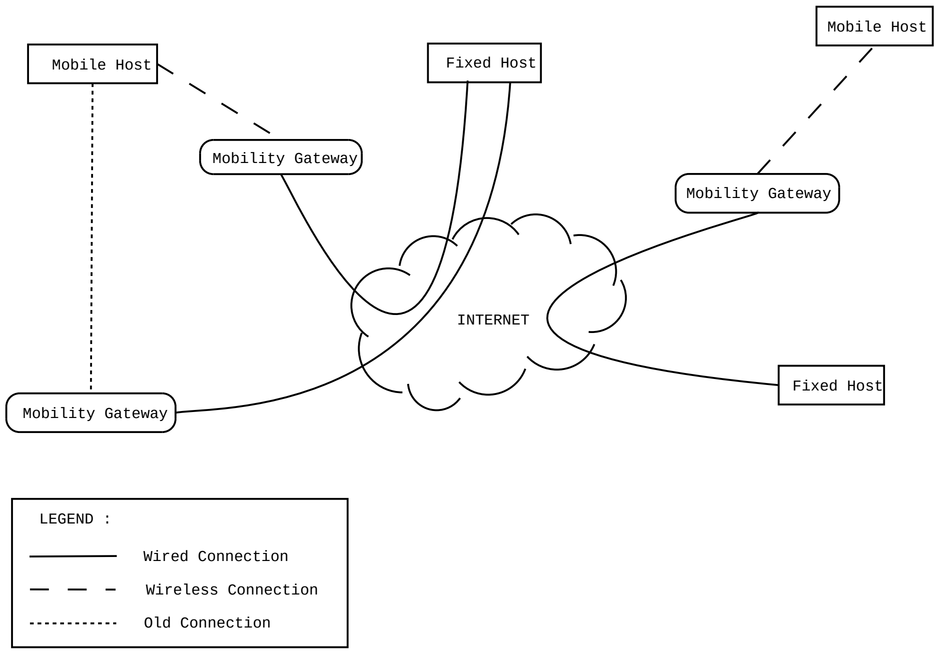

Fig. 1. The mobile environment

environment for which the ALICE framework was developed is illustrated in Fig. 1. Mobile Hosts (MH) are mobile computing devices with wireless network interfaces. These MHs are connected to stationary Mobility Gateways (MG) that maintain both wireless and wired network interfaces and act as the entry point from the wireless network into the wired network. Fixed Hosts (FH) are stationary computing devices that communicate with the MHs via the MGs. The MHs are physically mobile and may change their point of connection from one MG to another during a procedure known as handoff. ALICE provides mobility support in the form of a layered architecture, with each ALICE layer solving specific problems introduced by the mobile environment.

- The *Mobility Layer* manages connections between the MH and the MG, hiding the complexity of the wireless network characteristics from higher layers
- The *Swizzling Layer* supports server mobility by translating server references to refer to a MG rather than directly to an MH
- The *Disconnected Operation Layer* allows clients to cache server functionality during periods when an MH becomes disconnected

3.2 Java RMI

RMI is part of the Java Distributed Object Model [6] and provides the illusion of invoking a method on a local object, whilst in fact the method may be invoked on an object in a different address space. RMI was designed specifically to be language dependent and hence able to take advantage of the existing Java Object Model. Java's Garbage Collection mechanism has also been extended to encompass remote objects.

RMI makes use of a proprietary protocol on the wire known as the Java Remote Method Protocol (JRMP), which in turn operates on top of TCP/IP. An important feature of RMI is the ability to replace standard TCP sockets with programmer-defined socket types through the specification of custom socket factories.

3.3 Dynamic Proxies

Dynamic proxies are a feature of the Java 2 Platform since version 1.3. A dynamic proxy class is a class that implements a set of interfaces specified at runtime in order to provide a type-safe proxy through which an invocation of an interface method, on the proxy, is dispatched to another object. Invocations on the proxy are dispatched to a single `invoke()` method in the proxy class. This method is then free to do anything with the invocation, including dispatch it to another object, before returning the result of the invocation to the client.

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

4 Mobile Host as Client

When the MH is acting as a client, connectivity management between the MH and the MG is provided for in ALICE by the Mobility Layer (ML) which manages the wireless connection and hides the inherent unreliability of the medium from higher layers. Connectivity management takes the form of transparent reconnection of broken transport layer connections, as well as tunneling of existing connections after a MH has moved from one MG to another. The ML was designed to operate independently of any application-level protocol issues, and thus may be used across a range of distributed object technologies [5].

4.1 Mobility Layer

The ML, which is implemented in C, consists of a superset of standard BSD sockets, known as *sockets+* [5]. Replacements of the standard socket functions are provided, as well as functions to manage callbacks from the ML to the application. Such callbacks may be used to notify the application of changes in state of connectivity. Their use requires that the application be aware of its mobility, but application use of callbacks is optional.

4.2 Replacing the Java Socket Implementation

The Java Native Interface (JNI) is the native programming interface for Java, allowing access to native code from Java code. There are two ways in which the ML could be integrated into RMI using JNI in combination with custom socket implementations, and although both ways alter the underlying socket implementation used by RMI, there are differences in how the socket replacement classes are integrated into RMI.

The Java networking package provides the `java.net.Socket` and `java.net.ServerSocket` classes which provide client and server communication endpoint functionality respectively. By default JRMP uses instances of these socket classes to provide communication between remote objects, although it is possible to specify extensions of these classes to be used by RMI.

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

Given the way in which sockets are implemented in Java, the two ways in which the ML socket functions may be integrated into RMI at the native library level are as follows.

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

The major difference between the `java.net.PlainSocketImpl` class and the `alice.rmi.ALICESocketImpl` class is the additional functions for the management of callbacks to the application layer that are part of the `alice.rmi.ALICESocketImpl` class. Another difference between the implementations occurs in the loading of the shared library. The call to load the native shared library containing the platform-specific socket functions is altered to rather load the ML socket functions, thus providing these to the RMI runtime in place of standard system sockets.

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

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

4.3 Chosen Approach

The method used to integrate the ALICE ML into the RMI runtime system is the creation of a custom socket implementation. This approach is considered the most closely aligned with the goals of the Java language, where new socket implementations are expected to be developed to fulfill certain requirements. Following this approach, the custom `Socket` and `ServerSocket` classes will maintain a socket implementation attribute of the type `alice.rmi.ALICESocketImpl`.

The connectivity management support offered to the RMI runtime system by the ML is sufficient when the MH is acting as a client of a remote server object.

5 Mobile Host as Server

Addressing problems arise when an RMI object resident on a MH acts as a server. Servers traditionally export references to themselves so that clients know where they may be contacted. RMI servers export references of type `java.rmi.server.-RemoteRef`. These references contain an IP address and port number combination referring to the machine on which the server object is hosted. In the mobile environment assumed by ALICE, a MH is not directly contactable and all communication to and from it must pass through a MG. Thus, if a server is hosted on a MH and exports a reference based on the MH's address, the reference is invalid (since the host is not directly contactable at this address) and any attempts to invoke the server using this reference will fail. ALICE introduced the Swizzling Layer to overcome the problem of location management for CORBA, through the changing of the endpoint of a server reference to point to the MG rather than the MH. Swizzling object references was not considered appropriate for RMI due to differences in the way RMI implements remote object referencing, and we rather adopt an invocation redirection mechanism for location management which relays invocations between the MG and MH.

5.1 Remote Object Referencing in Java

The way in which an RMI server object (`ServerImpl`) is related to the address space in which it is resident is illustrated in Fig. 2. The `rmiregistry` is a simple, non-persistent, bootstrap name server from which a reference to a remote object on a given host may be obtained by a client.

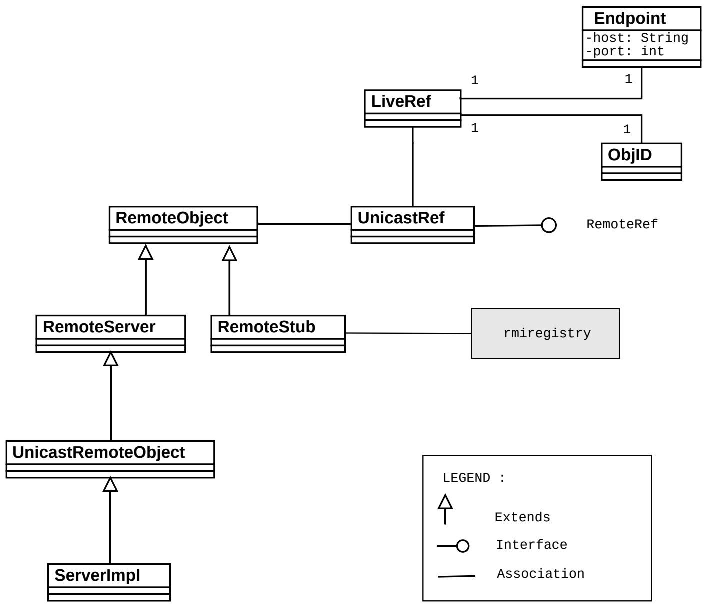

Fig. 2. Class hierarchy for remote objects in RMI

The reference that a client receives from the registry is in most cases actually to a stub and not the object itself [1]. As illustrated in Fig. 2, the stub contains a `sun.rmi.server.UnicastRef` attribute, which in turn contains a `sun.rmi.transport.LiveRef` attribute, that contains a `sun.rmi.transport.Endpoint`, giving the hostname and port number at which the object resides, and a unique identifier in the form of an object ID.

In addition to the retrieval of a remote object reference by way of lookup on an `rmiregistry`, there are two further ways by which an RMI client may obtain a reference to a remote object :

- **As the return value from a method invocation**
  A method invoked by a client on a remote object may return, as a result of the method invocation, a reference to a remote object. The client may then

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

invoke methods on this object as if it resided locally. The registry lookup essentially obtains a reference as a result of the invocation of the `lookup()` method.

– **As an argument to a method called on the client**
The third way in which a client may obtain a reference to a remote object, is as an argument to a method invoked on the client, by the server. This case arises when the client is also an RMI server object. In such a scenario, an RMI server may make a callback to the client and may potentially pass it a reference to a further remote object.

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

5.2 Using Dynamic Proxy Classes for Redirection

Dynamic proxies provide the basis of the invocation redirection mechanism developed. It is possible using dynamic proxies to create a proxy representation of a class, which implements a set of interfaces, at runtime. Method invocations are then handled by an implementation of the `java.lang.reflect.InvocationHandler` interface, which if present on the MG, could forward the invocation onto the MH attached to it. It is possible to remove all knowledge of creating the proxy from the client code which enables client-side transparency in the process.

Following the dynamic proxy approach, a proxy class, say a `ServerProxy`, is developed which implements the `java.lang.reflect.InvocationHandler` interface. This class takes a generic `Object` as an argument to its constructor and then uses reflection to determine what interfaces the object implements. Each object hosted on a MH needs to bind to the registry running on the MG that it is currently connected to. What is actually bound to the registry, however, is a dynamic proxy object transparently implementing the same remote interface as the remote object. This `ServerProxy` object is bound to the registry on the MG and is what is actually returned to clients performing a lookup against the remote object name. The `ServerProxy` also maintains a reference to the MG from which it originated and any invocations made on the proxy are propagated firstly to the MG and then on to the actual remote object, the location of which is known to the MG. Invocation responses are likewise propagated back through the MG. Figure 3 presents the operation of the scheme in more detail. In the scenario shown, a client obtains a reference to a remote object by way of a registry lookup.

The dynamic proxy based scheme operates as follows :

1. The MG process binds to the Mobility Registry daemon on the MG, using a well-known name.
2. A remote object resident on a MH performs a lookup, against the well-known name, on the MG to which it is attached.
3. A reference to the MG process is returned to the remote object.

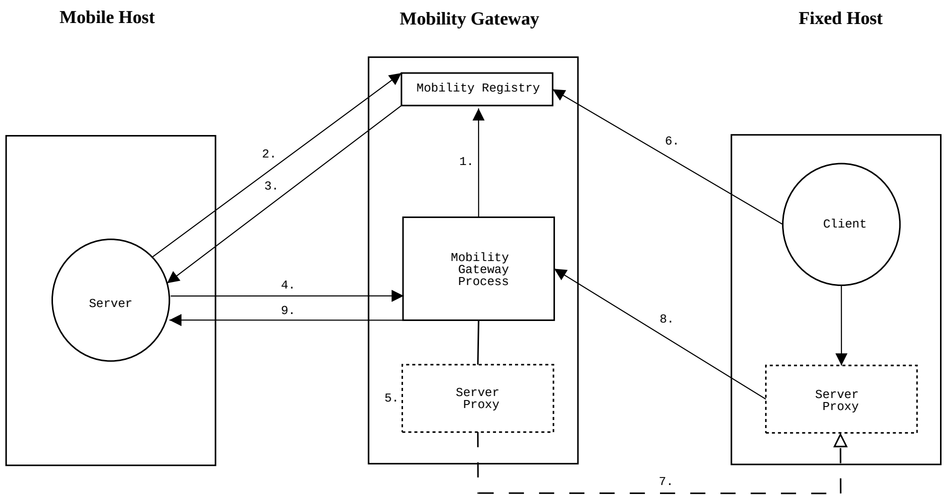

Fig. 3. Dynamic proxy architecture

4. The remote object invokes the `register()` method on the Mobility Gateway process, passing itself as a parameter.
5. The registration process creates a `ServerProxy` object on the MG which implements the same set of interfaces as the remote object. This completes the set up process.
6. A client performs a lookup against the server name, on the MG.
7. A reference to the `ServerProxy` object, rather than to the actual remote object is returned to the client.
8. Any invocations made against this reference are forwarded to a single method within the `ServerProxy` object and then onto the MG process.
9. The MG process then forwards the invocation onto the actual remote object resident on the MH, and returns the result along the same path back to the client.

This approach is achievable with no changes to the existing RMI architecture and therefore is one of the most attractive solutions to the location management problem.

Reference obtained as return value of invocation. The case where a FH receives a reference to a remote object by way of the return value of a method invocation, is illustrated in Fig. 4 and operates in a similar manner except that the value returned by the invocation in step 3 of Fig. 4 is inspected at the MG. If it is a reference to a remote object, a dynamic proxy representing the object is created and registered with the MG and a reference to the proxy object is returned to the client in step 4. The `rmiregistry` is not necessarily involved in the process, although it may have been involved in obtaining the reference to s1.

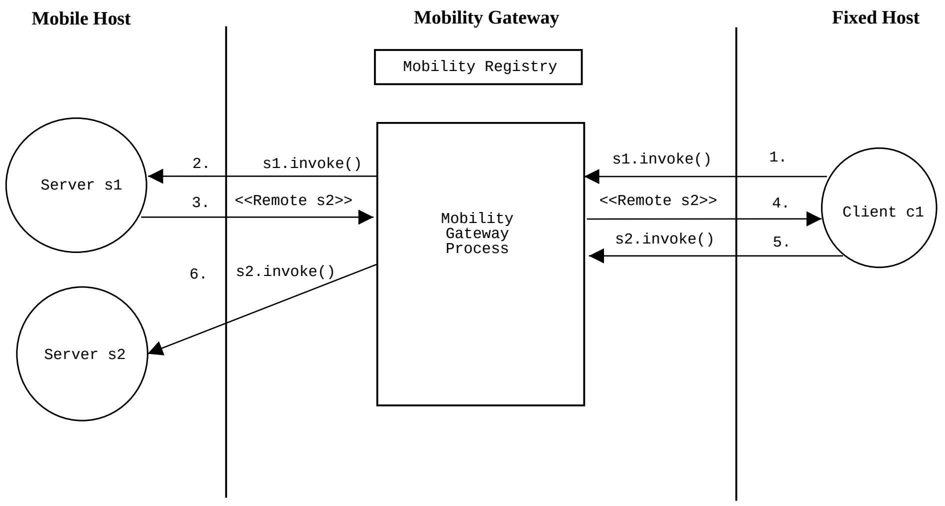

Fig. 4. Reference obtained from invocation

Reference obtained by argument. A similar scheme is employed to detect whether a reference to a remote object is passed as an argument in a callback from a MH to a FH. Each argument is inspected at the MG and if a remote reference is being passed, a proxy representation is created at the MG, and a reference to this proxy replaces the original argument.

6 Implementation

This section describes the implementation of our solution to connectivity management and location management for mobile RMI. We use the same layer notation and terminology as in [5]. When referring to an entire layer, either the full name (the *Mobility Layer*) or its abbreviated form (the *ML*) will be used. When referring to a single component of a layer, the location of the component will be subscripted. For instance, the component of the ML residing on the MG will be referred to as ML_{MG}. The position of each of the components implemented is illustrated in Fig. 5.

6.1 Connectivity Management

Connectivity management is achieved through integration of the ML into RMI by the specification of alternative socket factories to RMI as discussed in Sect. 4.2.

ML$_{MH}$ Component. The ML$_{MH}$ component is the component of the ML present on the MH. It is this component that provides the ALICE socket replacement functions and the sockets+ API (for mobile-aware applications) to layers above it, and which replaces the TCP transport layer in mobile-enabled applications. The ML_{MH} component also contains a daemon, the `mlmhd`, which multiplexes applications' connections onto a single transport connection to the MG.

In terms of RMI, socket functions are not accessed explicitly by the application programmer, but rather by the RMI runtime system which creates sockets through the standard API as they are needed for communication. Consequently, it is for this API that an interface to Java needs to be constructed.

ML_{MG} Component. The ALICE ML_{MG} component consists of a daemon, the `mlmgd`, which executes on the MG. The `mlmgd` daemon is connected to the `mlmhd` daemon running on a MH and is responsible for relaying connections between the MG and the MHs. Since this part of the ML is a daemon process and does not require any interaction with the application programmer, there is no need to provide an interface to it from Java.

Integration into RMI. The integration of the ML into the RMI runtime entails a number of steps, discussed below.

- **The creation of a custom Shared Object Library**
  We created a file named `ALICESocketImpl.c`, which was functionally equivalent to the `PlainSocketImpl.c` file, but contained additional methods for the sockets+ API, and linked this file against the ALICE Mobility Layer socket replacement functions, to produce a shared object file called `libALICEnet.so`.

- **The creation of a custom Socket Implementation**
  The next step was to create a custom Java socket implementation class, ex-

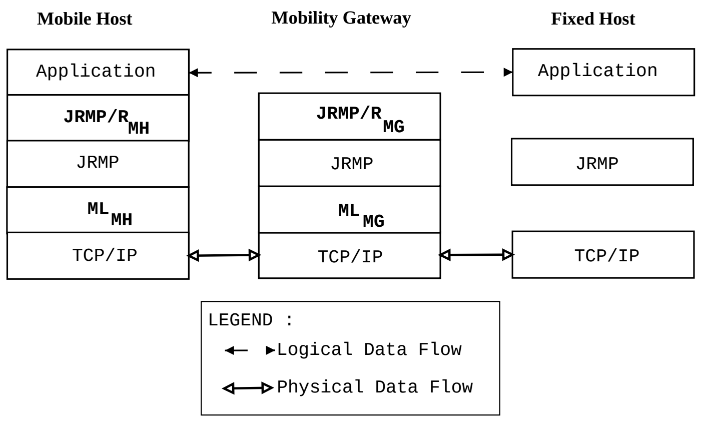

Fig. 5. Software architecture

tending from the `java.net.SocketImpl` class and making calls to `libALICEnet`.

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

- **The creation of a custom Socket Factory**
  The final step is the creation of client socket factory and server socket factory objects which RMI uses to supply instances of `alice.rmi.ALICESocket` and `alice.rmi.ALICEServerSocket` respectively for communication.

6.2 Location Management

A new layer, named the Java Remote Method Protocol Redirect (JRMP/R) Layer, was developed using dynamic proxy objects to provide location management for mobile RMI servers.

The JRMP/R layer consists of two components, that component resident on the MH, JRMP/R$_{MH}$, and that resident on the MG, JRMP/R$_{MG}$.

JRMP/R$_{MH}$ Component. The JRMP/R$_{MH}$ component consists of a modified version of the `java.rmi.Naming` class. The `alice.rmi.Naming` class maintains a reference to the current MG to which the MH is connected. When mobility support is required, RMI server objects use the `alice.rmi.Naming` class to register themselves with the RMI runtime. This class presents the same API to the application programmer as the `java.rmi.Naming` class.

The `alice.rmi.Naming` class overrides a subset of the methods in the `java.rmi.Naming` class and introduces some additional methods. The most important change introduced by the `alice.rmi.Naming` class is the overriding of the `bind()` and `rebind()` methods. The overridden methods still take the same arguments as the methods in `java.rmi.Naming`, that is a `String` name for the object, and the remote reference to the object. The `java.rmi.Naming` class causes the binding of the remote reference and the name of the object in a table within the rmiregistry subject to the condition that a remote object may only register with an rmiregistry running in the same address space as itself.

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

JRMP/R$_{MG}$ Component. The JRMP/R$_{MG}$ component of the JRMP/R layer consists of the following objects that collectively work together to transparently intercept invocations on server objects as described in Sect. 5.2.

- **Mobility Gateway Process**
  The MG process executes on the MG itself and provides a set of methods to the JRMP/R$_{MH}$ component to allow the registration and deregistration of remote objects resident on a MH that is connected to the gateway. The MG process is involved with server handoff.

- **Mobility Registry**
  The Mobility Registry is an rmiregistry running on the MG and providing a lookup service for clients wishing to obtain a reference to a server hosted by a MH. The Mobility Registry contains the name of the server object bound to a proxy representation of the object created upon registration.

- **Proxy objects**
  Each remote object that registers with the MG has a dynamic proxy object, implementing the same remote interface, created on the gateway. This proxy object is an instantiation of the `alice.rmi.ServerProxy` class, which is part of the JRMP/R$_{MG}$ component.

7 Evaluation

This section evaluates our dynamic proxy based location management scheme for mobile RMI applications in terms of performance and transparency of the solution.

7.1 Performance

The performance evaluation first compared the cost of standard one-hop RMI with that of standard one-hop RMI with the introduction of a dynamic proxy at the server. This indicated the overhead introduced by dynamic proxies for standard one-hop RMI. Since mobile RMI introduces an extra hop per invocation, we then compared the cost of standard two-hop RMI via an MG, with that of standard two-hop RMI via an MG using dynamic proxies to determine the cost of introducing dynamic proxies in this scenario. Finally, this was compared with the cost of introducing full JRMP/R support for mobile RMI to determine the cost of introducing full JRMP/R functionality.

Parameterless Invocation (Type 1). For one-hop RMI between a client and a server, the introduction of a dynamic proxy that simply forwarded the invocation to the real remote object, resulted in a marginal increase of 3.6%

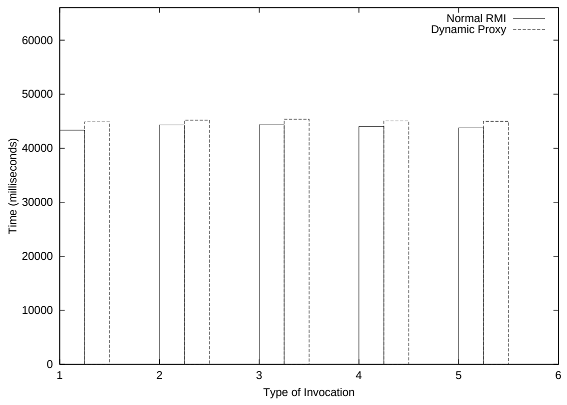

Fig. 6. Invocation times for 10 000 remote method invocations in one-hop RMI for 5 types of invocation

in the time taken to perform a remote invocation as illustrated in Fig. 6. This increase is due to the use of reflection by the dynamic proxy.

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

The operation of full JRMP/R support for this type of invocation led to an increase of 5.1% in the time taken over that of using dynamic proxies, and an increase of 20.5% in the time taken over standard two-hop RMI. This is illustrated in Fig. 7. The additional time taken to perform an invocation using full JRMP/R support is introduced by the need to use additional reflection at the MG in order to determine the method to be invoked at the server due to the non-serialisability of the `Method` type. Since the method being invoked is both `void` and parameterless, there is no need for the replacement of parameters or return types with proxy representations at the gateway.

Primitive Parameter (Type 2). Once again, the introduction of dynamic proxies between client and server in standard, one-hop RMI led to a slight increase of 2% in the time taken to perform an invocation, as illustrated in Fig. 6.

For RMI in a mobile environment, the invocation of a `void` method with a primitive parameter (an integer in this case) was once again least expensive at the level of standard RMI. The introduction of dynamic proxies led to an increase in the time taken to perform an invocation of 16.6% which is comparable to the increase observed in the `void` parameterless invocation and has the same explanation.

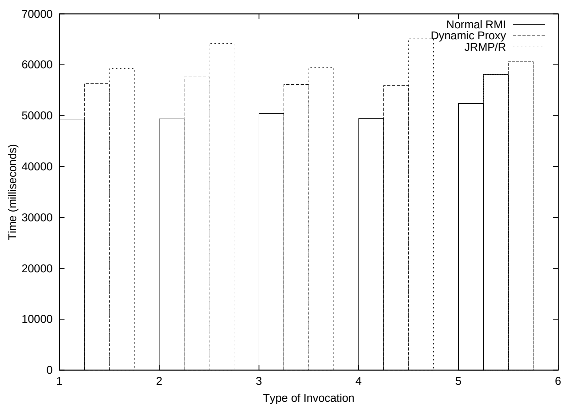

Fig. 7. Invocation times for 10 000 remote method invocations in two-hop RMI for 5 types of invocation

Full JRMP/R operation in this type of invocation led to an increase of 11.4% in the time taken over that using only dynamic proxies, and an increase of 30% over that using standard RMI. The increase is illustrated in Fig. 7.

The significant increase in the time taken over that using standard RMI is due in part to the need to perform reflection at the MG in order to determine the method type, and due in part to the need to inspect the method parameter at the MG. The inspection of the parameter is needed in order to determine whether a reference to a remote object is being passed as a parameter, in which case a proxy representation is required (see Sect. 5.2). The process of inspection utilises reflection which accounts for the additional time introduced into the invocation.

Object Parameter (Type 3). A marginal increase of 2.3% in the time taken to perform a remote method invocation is observed with the introduction of a dynamic proxy in one-hop RMI as illustrated in Fig. 6.

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

Additional overhead is introduced by reflection at the MG in order to determine both the method type and in order to inspect the parameter type to see whether a reference to a remote object is being passed. In this case, the method type determination and parameter inspection together resulted in an increase of 5.9% in the time taken to perform an invocation, over that of just using dynamic proxies, and an increase of 17.8% over standard RMI (see Fig. 7). This is significantly less than the overhead introduced by the inspection of primitive parameters at the MG. This may be explained by the fact that object creation is an expensive operation in Java. For the primitive parameter scenario (Type

2), an object representation of the primitive data type needs to be constructed so that it may be serialised for the RMI call. For example, if the parameter is an `int`, then an object of type `Integer` must be created to represent this integer.

Primitive Return (Type 4). For one-hop RMI an increase of 2.3% in the time taken to invoke a remote method is observed with the introduction of dynamic proxies (Fig. 6), due to the reflection performed by the dynamic proxy.

For mobile RMI, the invocation of a method with no parameters, but a primitive return type (an integer in this case) was least expensive under standard RMI, with an additional 13.1% being introduced into the time taken to perform an invocation with the introduction of dynamic proxies.

The operation of the full JRMP/R Layer led to an increase of 16.4% in the time taken over that of RMI using dynamic proxies, and an increase of 31.6% over standard RMI. This is illustrated in Fig. 7.

Once again,the additional cost of making an RMI call may be accounted for by the need to determine the method type and inspect the return value of the invocation.

Object Return (Type 5). The introduction of a dynamic proxy to one-hop RMI for this type of invocation increased the time taken to invoke a remote method by 2.7% as shown in Fig. 7.

For mobile RMI, the invocation of a method with no parameters, but which returns a Java object is the most costly type of invocation in terms of standard RMI. This is due to the need to serialise the return value of the method.

The use of dynamic proxies for this type of invocation increases the time taken to invoke such a method by 10.8% due to the cost of reflection. Full JRMP/R Layer support increases the time taken to invoke a method of this type by 4.3% over the use of dynamic proxies and an overall increase of 15.6% over standard RMI (see Fig. 7).

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

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

8 Related Work

The problems of host mobility addressed by our architecture are also addressed by Mobile IP [7] at the network layer. However, the Mobile IP solution to host mobility requires all hosts to use a modified network protocol and requires the maintenance of a centralised location register. An element of routing indirection is also introduced in Mobile IP. Our architecture does not require the replacement of the existing IP protocol, nor the maintenance of a centralised location register. Previous work has been carried out on the extension of the ALICE architecture to RMI [8], resulting in a Java version of the ML. Our architecture improves upon this approach through re-use of the existing ML component, dealing with all possible ways that a client may obtain a remote reference and by making mobility support transparent to the client by removing the need to hand code proxy classes.

Software mobility of RMI remote objects in a network has been addressed by [9] resulting in enhanced remote objects which are able to migrate between different address spaces.

A number of projects have examined the operation of RMI over a wireless link from the perspective of the efficiency of the communication mechanisms employed by RMI [10, 11]. These projects deal with aspects of connectivity management of a wireless connection with specific reference to the operation of RMI over such a connection. Location management of mobile clients and servers is not dealt with in these projects. [12] addresses the operation of Remote Procedure Call in a mobile environment.

Work has been carried out on interceptors for RMI utilising dynamic proxies, custom socket implementations and replacement of shared libraries [13] to intercept RMI method invocations. The RMI Proxy [14] is a commercial application protocol which makes use of dynamic proxies to provide an approach to allow controlled penetration of firewalls by RMI clients and servers.

9 Conclusion

This paper discussed the provision of support to RMI applications in a mobile environment including connectivity management, in the form of management of the wireless connection and insulation of RMI applications from the inherent unreliability of the medium, and location management using dynamic proxies to support RMI servers on mobile hosts.

Connectivity management was provided for RMI applications through the re-use of the application protocol independent Mobility Layer module of ALICE and provided for the full operation of mobile RMI clients.

A location management scheme for RMI based on dynamic proxies was developed to provide invocation redirection via a gateway between the wireless and wired networks, and permitted the operation of mobile RMI servers.

The mobility support provided by our architecture enabled the full operation of both RMI clients and server objects within a mobile environment. Mobility support was provided on top of the existing network (IP) protocols, with a high degree of transparency and a low degree of overhead and without the need for a centralised location register.

Acknowledgements

Gregory Biegel was in receipt of a Beit Fellowship at the time of this research and is very grateful to the Beit Trust for their generous support.

References

1. Sun Microsystems, Java Remote Method Invocation Specification Revision 1.7, `http://java.sun.com/products/jdk/rmi`, December 1999.
2. Sun Microsystems, Enterprise JavaBeans 2.0 Specification, `http://java.sun.com/products/ejb`, August 2001.
3. Sun Microsystems, JINI v1.1 Specification, `http://java.sun.com/jini`, October 2000.
4. George H. Forman and John Zahorjan The Challenges of Mobile Computing IEEE Computer Journal, April 1994
5. Mads Haahr, Raymond Cunningham and Vinny Cahill, Towards a Generic Architecture for Mobile Object-Oriented Applications, In *SerP 2000: Workshop on Service Portability*, December 2000.
6. Roger Biggs, Ann Wollrath and Jim Waldo, A Distributed Object Model for the Java System, In *USENIX 1996 Conference on Object Oriented Technologies (COOTS)*, pp. 219-231.
7. Charles E. Perkins, Mobile IP IEEE Communications Magazine, Vol. 35, No. 5, pp. 84-99, May 1997
8. Tom Wall, Mobile RMI : Supporting Remote Access to Java Server Objects on Mobile Hosts, In *Proceedings, International Symposium on Distributed Objects and Applications*, pp.41-51, September 2001
9. Avvenuti et al., MobileRMI: a ToolKit to Enhance Java RMI with Mobility, In *6th ECOOP Workshop On Mobile Object Systems: Operating System Support, Security and Programming Languages*, June 2000.
10. Stefano Campadello, Oskari Koskimies and Kimmo Raatikainen, Wireless Java RMI, In *4th International Enterprise Distributed Object Computing Conference*, pp. 114-123, September 2000.
11. Vijaykumar Krishnaswamy and Dan Walther and Sumeer Bhola and Ethendranath Bommaiah and George Riley and Brad Topol and Mustaque Ahamad, Efficient Implementations of Java Remote Method Invocation (RMI), In *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '98)* pp. 19-35 April 1998.
12. Ajay Bakre and B.R. Badrinath, M-RPC: A Remote Procedure Call Service for Mobile Clients, In *Proceedings of the 1st ACM Mobicom Conference* pp.2-11, 1995.
13. N. Narasimhan, L.E Moser and P.M Melliar-Smith, Interceptors for Java Remote Method Invocation, *Java Grande - Concurrency : Practice and Experience* 2000.
14. Esmond Pitt and Neil Belford, The RMI Proxy White Paper, `http://www.rmiproxy.com` March 2001.