Mobility and Java RMI

Thomas Wall

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

September 2000

Declaration

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

Thomas Wall
14th September 2000

Permission to lend and/or copy

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

Thomas Wall
14th September 2000

Abstract

Recent advances in computing device and wireless communication technologies are enabling the widespread use of mobile computing devices. Mobile computing presents many problems not encountered in a static computing environment. The limited storage and processing power on the mobile device, the limited bandwidth available on wireless networks and the difficulties of locating a mobile device mean that mobile environments are more difficult to design distributed systems for than fixed networks.

The Common Object Request Broker Architecture (CORBA) is a standard for building distributed object-oriented applications. However the standard was designed primarily for static environments and therefore does not address the problems encountered by objects residing on mobile devices. The Architecture for Location Independent CORBA Environments (ALICE) adds support for such mobile objects to the CORBA standard. This project examines how the mobility support provided by ALICE can be applied to distributed applications constructed using Java RMI. Sections of the ALICE architecture that were independent of CORBA were reused in the design.

This dissertation outlines the design of the RMI specific components of the architecture to replace the CORBA specific components, and the completion of a Java implementation of the ALICE component that provides session layer mobility support. The completed set of components provides support for mobile RMI client and server objects that can interact transparently with other RMI objects. All of the implementation was done in Java using Java sockets and RMI to communicate across the network.

Acknowledgements

I would like to thank my supervisors Vinny Cahill and Mads Haahr for their endless patience and guidance throughout the course of this project. I would also like to thank all my friends, family and classmates for their friendship, support and help during the year.

Contents

Table Of Figures

Chapter 1

Introduction

The astonishing rise of cellular phone ownership worldwide and the corresponding advances in mobile voice communication technology has been mirrored by the advent of wireless mobile computing devices and the integration of wireless networks with the Internet. However, device mobility introduces problems that are not encountered in a static computing environment and for this reason many distributed applications designed primarily for fixed networks are not as effective or indeed cannot function normally at all when used in a mobile setting. The opportunity therefore exists to provide mobility support for many distributed applications to enable them to be applied to a mobile environment. With this opportunity comes the challenge of designing for an environment that is much more complex and demanding than that provided by conventional wired networks. This thesis describes how a distributed object technology, Sun Microsystem's Java Remote Method Invocation (RMI) [RMI Spec '00], can be applied to a mobile environment based on the Architecture for Location Independent CORBA Environments (ALICE) [Haahr '99] developed by Trinity's Distributed Systems Group.

This chapter provides an introduction to the area of mobile computing, followed by an overview of the main components of the design, ALICE and Java RMI. The main goals and achievements of the project are outlined as well as a brief guide to the overall structure and format of the remainder of this dissertation.

1.1 Mobile Computing

Mobile computing devices are typically characterised by a number of factors. They are constrained by portability requirements to be small in terms of size and weight; they are capable of connecting to a data network using some form of wireless communication and they have limited computing resources compared to traditional non-mobile computing devices such as desktop Personal Computers (PC's). Typical examples of mobile computing devices are laptops, Personal Digital Assistants

(PDA's) equipped with some form of infra-red or GSM transceiver, and Wireless Application Protocol (WAP) enabled mobile phones.

The birth of mobile computing has brought with it many advantages. The ability to access and share information such as personal files or web pages is a major benefit in the modern work environment. There is a growing trend amongst organisations to offer their employees greater freedom in terms of when and where they do their work. No longer are people required to be sitting at their desks in their offices in order to accomplish anything worthwhile. With a full mobile computing infrastructure, work could be done when and where the worker wishes. This flexibility is being enabled by the emergence of mobile computing as a robust and viable means of doing work. Many organisations are now being transformed by the new business opportunities presented by the new technology and many more will be created. It is not difficult to see that mobile computing will change not only the way we do business but ultimately, the way we live.

The continuing downward trend in the size and cost of microprocessors and memory chips is enabling more powerful mobile devices but the fact remains that they are still and perhaps always will be at a significant disadvantage when it comes to display and input capabilities. The emergence of WAP technology, which has brought Internet access to digital mobile phone users, is naturally hampered by the limitations on interactions due to the small size of the user interfaces, which were primarily designed for inputting short strings of digits, and the limited bandwidth offered by the network itself.

Ideally, the roaming user of a mobile computing device would be capable of performing exactly the same computing operations as a user working at a desktop PC while at the same time availing of the benefits of not being tethered to a particular location. While a cursory glance at the device and communications technology currently available seems to suggest that this is quite feasible, the fact remains that much of the software and hardware infrastructure that is used in a fixed network is completely unsuited to supporting mobile devices.

For example, the transport protocol used to route data packets around the Internet, IP, cannot direct network traffic to take account of the change in location of a mobile device and needs significant alterations in order to be able to cope with this (see Section 2.2). In fact, the majority of distributed application technologies were designed for fully wired networks with completely stationary hosts and make no attempt to address mobility. Wireless communication technologies themselves are prone to interference, range and bandwidth limitations and data has to be reformatted in order to optimise use of the link e.g. Wireless Markup Language (WML) is basically the HyperText Markup Language (HTML) reformatted for wireless transmission to mobile phones.

For these reasons, a mobile user is still far behind the wired network user in terms of computing capabilities and considerable work remains to be done to bridge the gap. The ALICE project aims to narrow this gap by providing mobility support for CORBA objects. Such support is not addressed by current implementations of CORBA, which were designed for static, fixed-network environments.

1.2 CORBA and ALICE

CORBA (Common Object Request Broker Architecture) is a distributed object technology standard for creating client – server applications. CORBA is object oriented so that a client can call a method on a remote server object with much the same semantics as a call to a method on a local object. The actual implementation of the server is hidden from the client and all interactions between the two are specified by an interface defined in a language neutral format. An entity called an Object Request Broker (ORB) acts as a middleman between the two communicating parties and provides services such as forwarding requests and responses between objects and providing clients with references to remote servers. CORBA relies on a specified protocol known as GIOP (General Inter-ORB Protocol) as a means of transferring method invocation requests and responses between clients and servers. IIOP (Internet Inter-ORB Protocol) is a mapping of the GIOP specification onto TCP/IP.

The CORBA standard does not cater well for mobile client or server objects. CORBA assumes that servers do not change their location and that the transport connections that are used by IIOP do not break frequently. Both of these assumptions do not hold true in a mobile computing environment. The ALICE [Haahr ‘99] architecture was designed to address the limitations in OMG’s CORBA standard that constrains mobile CORBA objects from operating as effectively as non-mobile objects. In the environment envisaged in ALICE, mobile devices hosting CORBA objects (mobile hosts) connect via wireless links to mobility gateways, which have wired connections to a fixed network. These mobility gateways act as bridges and perform the tasks of forwarding CORBA requests to the mobile host from remote hosts and returning replies and similarly forwarding requests from CORBA clients on the mobile host to remote servers. Due to the range limitations of the wireless links, mobile hosts can change their point of connection to the fixed network by changing which gateway they are connected to.

To address the problems posed by mobility, ALICE introduces extensive session layer mobility support and some support at the application layer. ALICE uses a Mobility Layer that sits on top of the TCP/IP layer and monitors the transport connection between the mobile host and the gateway. If the connection is lost the Mobility Layer on the mobile host hides the broken connection from the higher layers and transparently reconnects to the gateway. The Mobile Layer also performs other functions such as multiplexing all socket connections over a single transport connection to make more efficient use of the limited bandwidth, caching of all data sent pending acknowledgement and providing mobility information to higher layers.

ALICE also uses the Swizzling IIOP Layer (S/IIOP) to perform address translation and request forwarding on IIOP requests at the mobility gateway. The S/IIOP layer on the mobile host alters the object references used by clients to locate the server so that the clients will connect to the current gateway instead. When the gateway receives an IIOP message from a client it checks the object reference and redirects the message to the correct server object on the mobile host. The S/IIOP layer also allows clients to be redirected to the new location of the mobile host after it has changed mobility gateway.

ALICE therefore brings mobility support to CORBA objects while at the same time keeping the task of programming mobility capable applications as transparent as possible. In addition mobile CORBA objects using ALICE are fully capable of interacting with existing ORB implementations.

1.3 Java RMI

The Java Remote Method Invocation (RMI) [RMI Spec ‘00] technology is, like CORBA, designed to allow programmers to write distributed object-oriented applications. Whereas CORBA is a language independent specification with many varying implementations from different vendors, RMI is a 100% Java technology and any interaction with non-Java code must take place using the Java Native Interface (JNI). RMI integrates a distributed object model with the local Java object model in a natural way with a few exceptions to the semantics to make the difference between remote and local object method invocation obvious.

RMI makes extensive use of many of the capabilities that make Java an attractive choice of programming language in the first place. For example, with an entirely Java based system, Java objects and classes can be moved from machine to machine to distribute where the actual work is performed. RMI allows for the transfer of entire Java objects not just simple data types between virtual machines as parameters or results of remote method invocations. Programming solely in one language also greatly simplifies the task of the programmer. RMI also provides distributed garbage collection and object activation mechanisms.

RMI provides a simple naming service, the rmiregistry, which allows a client to obtain a server reference by specifying a URL. RMI clients use proxies called stubs, downloaded from the server host via a web server, to communicate with the remote server object. This stub is responsible for the marshalling of data from the client into a format suitable for on-the-wire transmission. At the server side there is a corresponding skeleton, which unmarshalls parameters passed to it from the client and marshals data returned from the server. These stubs/skeletons effectively provide an interface between the application and the rest of the RMI system.

1.4 Project Goal

The aim of this project was to design and build a system to allow RMI server and client objects to operate on mobile hosts and to interact with normal RMI objects without them being aware of the others mobility. The system should reuse as much of the CORBA independent parts of the original ALICE architecture as possible. The design should make use of the protocol independent ALICE Mobility Layer to provide mobility support at a low level and then some means of address translation and request forwarding such as that done in the ALICE S/IIOP layer must be found. The task of programming such mobile capable applications should be made as easy and transparent to the programmer as possible.

The project can be divided into two sections. The first part was concerned with completing the implementation of a Java version of the original Mobility Layer. The second section dealt with RMI specific mobility support, effectively replacing the S/IIOP layer with a means of providing the same functionality to RMI applications.

1.5 Overview of Design

Initially it was hoped to be able to mimic the operation of the S/IIOP layer with some form of manipulation of RMI object references. After much research and testing it was decided that although this worked well for IIOP and CORBA objects, it was not an appropriate approach for RMI. Instead it was decided to perform the tasks of the S/IIOP at the application layer using a form of object delegation. Effectively all mobile server objects provide their current mobility gateway with code required to perform the task of operating as a proxy for the mobile server. Remote hosts can contact the gateway and communicate with the proxy, which will relay the client method invocations back to the mobile server. The handoff performed in part by the S/IIOP layer in the CORBA ALICE model is replaced by an application level handoff scheme in the RMI implementation.

The design required the construction of RMI objects to perform the role of proxy, objects to allow code to be exported to the gateways and for allowing communication between gateways when handoff of the mobile host occurs. The design and construction of this system required much experience to be gained in using RMI, especially its mobile code facility, methods of locating objects and parameter passing routines.

1.6 Project Achievements

By the conclusion of this project most of the major aims outlined in the project goals had been achieved. The Java Mobile Layer implementation was completed and extensively tested using RMI. The proxy scheme to allow for mobile RMI servers was designed and constructed and successfully tested on top of the Mobile Layer. Sections of the design concerned with making the system more transparent to a programmer writing Mobile RMI applications, such as an automatic proxy code generator, remain to be implemented.

1.7 Roadmap

The following is a brief introduction to the material covered in the rest of the chapters of this dissertation:

In Chapter 2 (State Of The Art) various projects dealing with aspects of mobile computing and other topics of relevance to this project are examined. In Chapter 3 (Background) a more detailed discussion of the ALICE model and the Java RMI distributed object model is presented. Chapter 4 (Design) outlines the design for the proxy object scheme and explains how handoff and remote reference passing are handled.

Chapter 5 (Implementation) explains how the objects specified in the design were constructed along with various other implementation specific details. It also details the continuation of the implementation of the Java Mobile Layer. Chapter 6 (Evaluation) provides a brief examination of the performance of the design in comparison with non-mobile RMI and an analysis of the size of the code used in the design. Finally in Chapter 7 (Conclusions) the conclusions arrived at by the completion of the project are discussed.

Chapter 2

State Of The Art

This chapter provides an introduction to recent mobile computing projects and other topics of relevance to this project. An introduction to mobile networking and the broad range of approaches that can be used to build mobile systems and applications is given first. Following this a number of current mobile computing designs are discussed varying from network protocols that take account of device mobility (Mobile IP and Monarch), to client –server distributed object systems for developing mobility capable applications (Rover), to systems for supporting data sharing amongst mobile users (Bayou). Next a number of projects addressing a key issue in mobile computing, disconnected operation, are introduced. Ideally, mobile devices that have become disconnected from the network would be able to provide the user with the same computing capabilities as a fully connected device. In reality this is seldom possible but the projects discussed here look at ways to allow a user to continue working without a permanent connection to the network. Finally the Jini model is examined. Jini [Venners '00] provides a runtime infrastructure that allows service providers to offer their services to clients and for clients to locate and use these services without prior knowledge of the services existence or location.

2.1 Mobile Networking

As technological advances continue apace, the capability and availability of mobile computing is growing and is enabling a new shift away from the traditional desktop personal computer (PC) to a more portable, flexible and ultimately more useful computing resource. The combination of portable computing with wireless communications is changing the way people think about computing and indeed about how they work. It is now being realized that information processing does not have to be limited to the time spent in front of a PC in the office but can be done at home, at any office or even in transit. In order to enable this 'nomadicity' the infrastructure must be put in place to support mobile computing devices and mobile information access. The ability to automatically adjust all aspects of the user's computing, communication and storage functionality in a transparent and integrated fashion is the essence of a nomadic environment [Kleinrock '95]. For example it is desirable for a mobile device with several communication mechanisms to dynamically and transparently choose the best one to use depending on available bandwidth, error rates, cost etc.

Mobile computing suffers from constraints that are not experienced in a fixed desktop computing environment. These constraints, which are intrinsic to mobile computing, are gradually being alleviated as technology advances but the fact remains that mobile devices will never have the same performance and resource capabilities as fixed devices. Due to the need to conserve weight, size and power, mobile devices are resource poor compared to static devices hence characteristics such as processor speed, main memory capacity, screen and disk size are limited. Unlike fixed network connections in general, mobile devices' network connections may be variable in terms of availability, reliability, performance and capacity. Wireless technologies in particular suffer from interference and coverage restrictions. In addition, despite advances in battery technology, the limited power sources used by mobile devices must be taken into account when devising new systems.

There are several important factors that have to be addressed when designing new mobile computing systems [Duchamp '92]. These include the nature of the mobile device including its computing resources, size, input and display types; the nature and type of available network connections including reliability, capacity, quality etc. and the movement and data access patterns of the people the systems are being designed for.

One of the most vital questions is whether or not applications should be made aware of their environment or whether they should be insulated from any environmental details. The latter case implies that ordinary desktop applications should be 'mobile-transparent', that is, they should be useable in a mobile environment without modification and that the systems underlying the application (such as distributed object middleware) should take care of all adaptation required to account for changes in location, connection bandwidth etc. In the former case we assume that there is no system support and that all mobility adaptation is undertaken by the applications themselves. Such applications are termed 'mobile-aware'. In between these two poles is what is known as application aware adaptation [Satya '96], which is characterised by collaboration between the system and the application (see Figure 2.1 below).

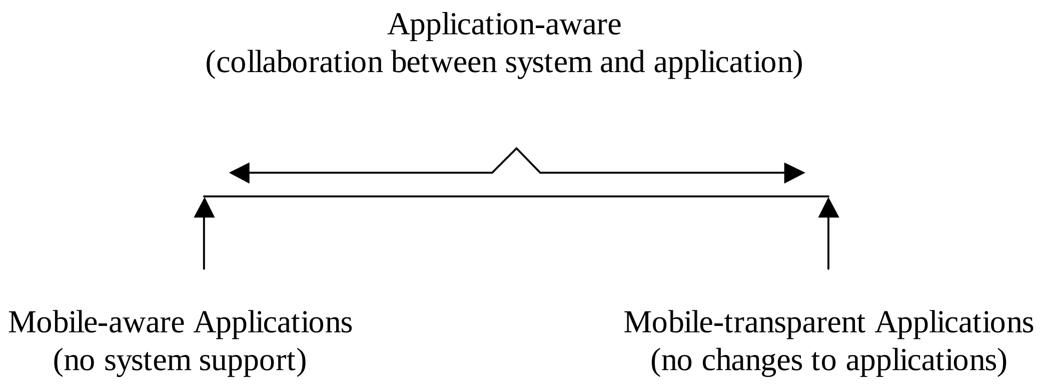

Application-aware
(collaboration between system and application)
Mobile-aware Applications
(no system support)
Mobile-transparent Applications
(no changes to applications)

Fig 2.1 Range of adaptation strategies

Systems can embody adaptation at many levels. Coda [Satya '96], which will be discussed in more detail later, implements application-transparent adaptation in the context of a distributed file system. Since the Posix interface is preserved, legacy applications can run on Coda without the need for any modifications. In contrast, the Rover toolkit, which will also be examined, supports application-aware adaptation that is better suited to multimedia data such as speech and video. The resolution of such data can be modified to make the best use of the limited available bandwidth on a wireless connection while still providing information to the mobile user e.g. a colour video signal might be downgraded to black and white if the bandwidth is reduced due to a coverage problem induced by the movement of the user.

2.2 Mobile Networking Architectures

2.2.1 Bayou

The Xerox PARC Bayou project [Demers '94] is a system for supporting data sharing among mobile users. The system was designed from the outset to run in a mobile computing environment where the availability of connections between machines and the quality of those connections can never be taken for granted. The designers identified the main characteristic of such environments as being that machines may become disconnected from other machines with which they wish to share data, perhaps involuntarily, for indeterminable periods of time.

The Bayou architecture is based on a logical division between servers and clients. However, unlike Coda where servers and clients are physically distinct devices, a Bayou server is any machine that can hold a copy of a database (which in Bayou denotes any collection of data items). Any machine holding such a database can service read and write requests from any client machines that are able to communicate with it. A Bayou server can act as a server for some machines and can also be a client for other servers. The architecture chosen for the system reflects the fact that many mobile devices such as PDA's have insufficient storage capacity to hold all the data that their users wish to access.

Bayou uses a read-any/write-any replication scheme, which means that any user can read or write any data object in any copy of a database. The system attempts to reach consistency by means of a 'reconciliation' process. Updates are propagated to another database copy as soon as a communication link becomes available. Periodically a server will select another server with which to exchange writes so that their two databases reach consistency. This results in a weakly consistent replication scheme that maximises data availability for the user but cannot guarantee correctness [Terry '94].

If conflicts arise when updating a data object that has been concurrently written to by two different clients (write-write conflict) or which has been updated based on out-of-date data (stale update), they will be detected in an application specific manner. Bayou also provides a mechanism for application-specific resolution of conflicts by including a procedure called a 'mergeproc' with all write operations. This mergeproc is called in the event of a write-write or stale update conflict arising and will then read the data copy resident on the server where the conflict occurred and decide how the conflict should be resolved. Mergeprocs provide an application specific, flexible means of conflict resolution and may be customised to suit the semantics of the application and the intended effect of the write operation.

2.2.2 Mobile IP

Conventional hierarchical Internet protocols such as IP [Tannenbaum ‘96] do not cater for host mobility. Hierarchical routing, in which the address is split into a network number and a host number, only allows packets sent to a mobile host to be sent to its home network, even if the mobile host is not currently located there. To cater for mobile hosts new protocols were designed which allowed packets to reach these hosts regardless of their location.

Mobile IP from the IETF is one such protocol [Perkins ‘94]. Using Mobile IP each mobile host has a fixed home agent which receives all its packets while it is not connected to the home network and forwards them on to the mobile hosts current location, given by its care-of-address. This care of address is the address of a foreign agent on a different network that has agreed to provide such a service to the mobile host. The foreign agent then relays registration requests and replies between the home agent and the mobile host and decapsulates traffic from the home agent and forwards it on to the mobile node. A procedure called agent discovery is used to find a willing foreign agent. This can be achieved either by foreign agents advertising their presence or by a mobile host connecting directly to a known foreign agent. When one has been discovered the address of the agent must be made known to the home agent so that traffic can be redirected to the new location. The Mobile IP standard also addresses the issues of authentication of mobile host registrations with the home agent and IP packet encapsulation methods.

2.2.3 Monarch

The Monarch project at CMU has also dealt with routing protocols for mobile hosts, specifically with optimising the IETF Mobile IP protocol in order to reduce latency and congestion [Johnson ‘96]. In the original Mobile IP design, all packets destined for the mobile node had to be routed through the home network and then tunnelled to the foreign agent by the host home agent. This causes a lot of overhead on the home network and adds latency to the delivery process that could be avoided.

To counter this the Monarch project adds a number of extensions called ‘Route Optimisation’ to Mobile IP. These extensions provide a means for the sender of a packet to learn the mobile hosts’ foreign address so that it may send packets directly to that location, bypassing the home network. The first packet is sent via the home agent as per usual, but the home agent replies to the sender with the mobile hosts current binding, which is then cached so that packets may be sent directly in the future.

In order to allow packets to reach the mobile host after it has changed foreign agent, the mobile host sends the previous foreign agent a binding update, which allows for packets arriving at the old agent to be rerouted to the new one. If the old foreign agent receives any packets for the mobile host, as well as forwarding them on to the new location, it also sends a binding warning message to the home agent requesting it to inform the correspondent node of the new binding.

Another area being investigated by the Monarch project is ad-hoc networking. This is wireless networking in areas where no normal network infrastructure is available such as wilderness areas or where the infrastructure has been incapacitated such as disaster sites or war zones. In such situations it is desirable to be able to set up temporary ad-hoc networks without any central planning or administration. Due to the short range of wireless links, ad-hoc networks usually rely on participating hosts forwarding packets on behalf of other hosts.

Monarch uses a different type of routing protocol to the distance vector or link state routing implemented in other ad-hoc designs. This new routing protocol is called dynamic source routing. In conventional source routing a host sending a packet determines the sequence of nodes that the packet must pass through in order to reach the destination node. The sender lists the addresses of these nodes in the header of the packet. When each node in the list receives the packet it simply sends the packet on to the next node listed. In dynamic source routing each node maintains a route cache where it stores all routes it knows about. When it receives a packet to send to another node it first checks the cache for a route to that node and uses this route if it is found. Otherwise a procedure called route discovery is initiated by means of which an appropriate route can be found. Protocols are also provided to allow for failure of nodes along pre-existing routes.

The dynamic source routing protocol has now been implemented. Detailed simulations have shown that the protocol is able to provide routes that are on average within a factor of 1.01 of the optimal, showing that is should be able to track the movement of mobile hosts to a high degree.

In a further Monarch project an API and a set of extensions to the Mobile IP protocol have been designed to provide notification to mobile-aware applications on a mobile host about the conditions of the wireless connection when it changes foreign agents. The notification provides information, supplied by the foreign agent, about connection bandwidth, cost, error rate or latency in its local network. This notification is integrated with the mobile host-foreign agent registration and can be performed at little extra cost.

2.2.4 Rover

The Rover project at MIT [Joseph '97] aims to provide a toolkit to support the development of both mobile-transparent and mobile-aware applications. In contrast to Coda which attempts to hide environmental considerations from the applications, Rover was designed to make environmental information available to applications and involve them in the decision making process. In this sense Rover can be seen as a result of applying the end-to-end argument [Saltzer '84] to mobile applications. Rover provides an application-programming interface to allow mobile aware applications to be developed using common features and techniques.

In essence Rover is a client-server distributed object system in which servers run on stationary hosts and clients can run on either stationary or mobile hosts. Communication is supported by means of relocatable dynamic objects (RDOs) and queued remote procedure call (QRPC). An RDO is an object that can be transferred from client to server and vice-versa in order to minimise network communication in the form of RPC calls between the two. QRPC allows hosts to make non-blocking remote procedure calls during disconnection. These calls are queued and then resolved when reconnection occurs.

When building a mobile-aware application with Rover the main task is to define RDOs for the data manipulated by the application and the data units passed between the clients and servers. The application must then be divided into client and server sections and methods to operate on the RDOs must be implemented. The finished application should be able to import objects onto the local host, invoke methods on these objects, export logs of method invocations on those methods to the server and then reconcile these copies with the copies resident on the server. Rover

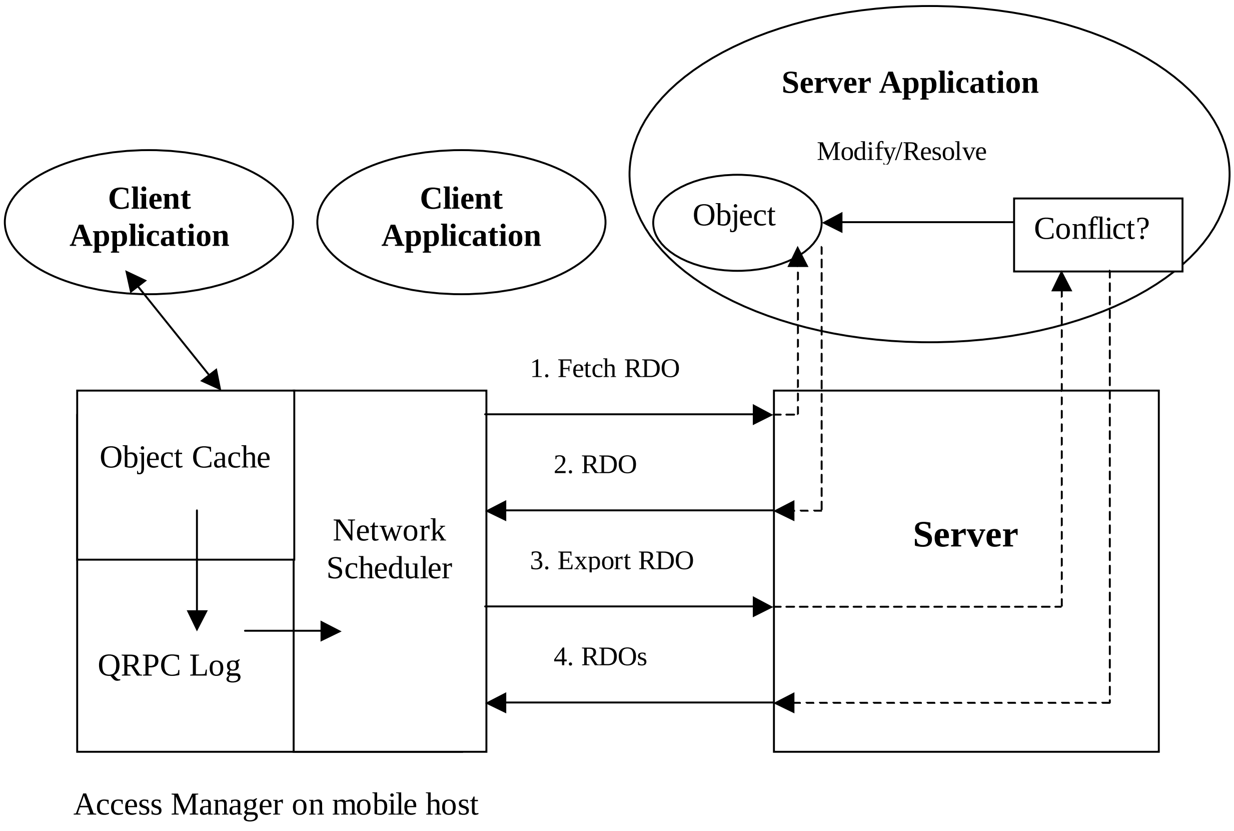


Fig. 2.2 Rover Distributed Object Model

also allows applications to use polling or callback methods to determine the current state of the environment for its own decision-making or to display to a user.

By moving RDOs across the network as required the application can control exactly where computation will occur and so limit communication costs. For example, a server could migrate a GUI RDO to a client. The actual code required to implement the GUI is small compared to the resulting data it produces, so this data will not have to cross the network. The locally cached GUI RDO will also be able to respond to user interaction without generating network traffic.

Rover clients use QRPC to fetch RDOs from servers. When a QRPC is issued it is stored in a local log and control is immediately returned to the issuing application. Callbacks can be used to notify the application of the arrival of the RDO or alternatively the application can block until it receives the RDO. If a locally held RDO is updated it is marked as tentatively committed and the updates are propagated to the server as soon as possible using QRPC.

QRPCs can be delivered out of order depending on priorities and costs associated with the sending action. Rover allows split-phase QRPC operation, that is, if a mobile host sends a QRPC and subsequently becomes disconnected before receiving a reply, the replier will periodically try to contact the sender and deliver its reply. This also allows request and response pairs to be sent over different communication channels, which is a significant advantage in a mobile environment by allowing communication to be directed over the most efficient, most available or cheapest channel.

Rover applications can provide prioritised prefetch lists that the user can select to download RDOs prior to disconnection for applications that are to be used while disconnected. These lists are based on heuristic data compiled from knowledge of the users previous actions. The choice of replicated data consistency scheme is left to the application, since the requirements will vary dramatically depending on the given application. Rover does however provide support for primary-copy, tentative-update optimistic consistency, which is considered the most appropriate for mobile computing applications.

Extensions have been made to Rover to further increase the reliability of mobile-aware applications built using the toolkit [Joseph ‘96]. The original Rover failure model provided client-server message delivery guarantees and support for client or communication failures but did not address server failures. The extensions allow for recoverable server faults such as power glitches but not repeatable or non-recoverable failures. Recoverable failures are handled by such means as stable message logging by the server, automatic server process restart and programmer supplied failure recovery procedures.

2.3 Disconnected Operation

2.3.1 Coda

The Coda file system [Kistler ‘93] developed at Carnegie Mellon University is a direct descendant of the Andrew File System (AFS) [Mullender ‘93]. Coda attempts to provide a much more reliable and available file service than that provided by AFS. Coda supports the integration of mobile computers with fixed file servers by means of disconnected operation. This allows users to move between zones of connection with the home network and zones where connection to the home network is not possible and still continue accessing files that were read from the file servers.

Coda uses the same Venus – Vice (client – server) architecture as AFS. File availability is increased in Coda through server replication and disconnected client operation. AFS clients can hold local copies of an object in a cache to save access time and network traffic. AFS uses pessimistic cache consistency that assumes that objects cached by a client will be modified by another client as well and so a mechanism is provided that allows the server to contact a client in the event of the object being updated elsewhere so that the client can 'revalidate' its cached copy by downloading the modified version from the server. This mechanism is called a callback. Coda implements the client caching and server callbacks that were the core of AFS. As long as the client remains in contact with at least one server it operates in the connected state, much the same as normal operation in AFS. When this connection is lost the client operates in the disconnected state.

Venus operates in one of three states – hoarding, emulation and reintegration. In the hoarding state Venus collects data that it anticipates will be required if disconnection occurs. This hoarded information is stored in a priority based hoard database (HDB). The HDB is periodically walked to determine which objects should be retained in the cache and which should be discarded. An object that has not recently been read or written would be a good candidate for discarding.

In the emulation state Venus effectively mimics the server by allowing access to cached objects. All updates to cached objects are recorded in the replay log to be replayed on reconnection to the server. In the event of a cache miss the default behaviour is for Venus to return an error code. When the connection is re-established the replay log is forwarded to the server where it is executed. Venus also updates its cache contents to be consistent with the servers.

In the event of a conflict (caused by separate writes to the same object by different holders of the object) occurring during reintegration an application specific resolver (ASR) is used to resolve the conflict. An ASR provides the detailed application specific knowledge required to distinguish between resolvable differences and genuine inconsistencies. If the ASR is unable to reconcile the differences then the user is presented with a manual repair tool.

In more recent work on Coda [Satya '96] the concept of connection has been extended to include weak connectivity. A weak connection can take the form of a low-bandwidth connection, an expensive network service (which can only be used sparingly) or an intermittent connection, which only lasts for a short time. A number of modifications have been made to Coda to exploit weak connectivity.

One of the drawbacks of the original implementation of Coda was the large amount of time required to revalidate a clients' cached callbacks on reconnection when the network was slow. To reduce this time, volume version stamps were introduced, which are analogous to the object version stamps already used (avolume being a collection of objects). Whenever an object in a volume is updated the volume version stamp is updated in addition to the object version stamp being incremented. As part of the hoarding process Venus caches any required volume version stamps. On reconnection to the server Venus checks its cached volume version stamps with those on the server. If the stamps are the same then all cached objects from that volume are also unchanged. If not then each cached object from that volume must be revalidated. Experiments have shown that this approach greatly reduces the average time required for cache validation.

Trickle reintegration is another means of exploiting weak connectivity. Instead of reintegrating all updates on full reconnection to the server, updates are periodically sent over the weak connection while still not interfering with user activities in the foreground. Implementing trickle reintegration did however require significant alterations to be made to the structure of Venus.

Weak connectivity also provides the opportunity to implement user-assisted cache miss handling. At low bandwidths the user is given the option of fetching the file, which would incur a long wait, or of continuing without it. At high bandwidths the file is fetched without prompting the user. In developing this approach the designers incorporated a user patience model to balance between the two factors, file delay and user patience.

Coda appears to work well in academic and research environments [Satya '93] although it remains untested in other deployments. The hoarding strategy used allows most disconnected sessions to complete without any cache misses, although this is due in part to the voluntary nature of the disconnections i.e. the common case is for the user to shut down the machine or disconnect it after the hoarding process has cached enough objects for disconnected use. Cache misses in this case are also rarely fatal to the disconnected session and work can be continued. An involuntary disconnection caused for example by a network failure may mean that the required objects have not been cached and a cache miss may be more disruptive. The performance of the system in the event of a reintegration storm (a large number of clients trying to forward their replay logs to the server at the same time) following a server or network failure has yet to be established in the context of large client populations although several strategies have been proposed. One would be to have each client wait a random amount of time before trying to contact the server again so that the reconnection of clients is more spread out.

2.3.2 Disconnected AFS

Whereas AFS works well in a normal desktop environment with fixed connections, when the network becomes partitioned clients that have been isolated from the server cannot propagate the changes they have made to cached objects to the server and hence make these changes known to the rest of the system. To overcome this a group in the University of Michigan has modified the structure of the client part of AFS (Venus) to allow for disconnected operation [Huston '93]. Unlike in Coda where extensive changes were made to both the AFS client and server structure, the developers of Disconnected AFS decided to limit any modifications to the client structure so that existing AFS cells could still be accessed. Unlike Coda and Ficus, the developers considered only disconnected operation in their design, ignoring server replication. The rationale behind this was that server replication is of no advantage in nomadic computing due to the fact that a disconnected client cannot contact any replica of a server.

The modifications to Venus were in essence (and much like Coda) altering the cache manager to provide optimistic cache consistency rather that the pessimistic approach taken by conventional AFS. Optimistic cache consistency works on the premise that usually only a single client updates a cached object at any given time and so the system is optimised to cater for this common case. When a Disconnected AFS client is disconnected the user can still access objects stored in the cache and all mutable operations are logged and replayed at the server on reconnection. On a cache miss an error code is returned to the calling program. Unlike Coda, the user must issue `disconnect` and `reconnect` commands manually. If on reconnection it is discovered that a connected client is modifying the file concurrently then both versions are stored on the file server and the user is notified that a conflict has occurred. The user must then resolve the conflict manually although, as of yet, no tools have been provided to allow the user to perform this resolution.

2.3.3 Disconnected NFS

A group working at the University of Washington took concepts from Coda and extended them to produce a methodology for structuring client software in a disconnectable file system [Fiuczynski '94]. This methodology was applied to a disconnectable version of Sun's Network File System (NFS). The notion of a client – agent – server (CAS) model for mobile devices was defined. The agent, resident on the mobile device, operates in the connected or disconnected state and functions are provided to switch context from one state to the other. In the connected state the agent forwards all client requests to the server and gathers data to store in the cache to be used in event of disconnection. When disconnection occurs the agent calls a transition function to switch to the disconnected state and cached data is used to satisfy client requests as far as possible.

Although the Disconnected NFS implementation contained no new concepts, this work provided a well-structured approach to developing applications for disconnected operation. Code for operation in the connected and disconnected states could be written separately and compartmentalised, allowing for easy replacement and easier coding, since each component only has to deal with one given state. The notion of transition functions also provided a means of isolating and localising actions concerned with moving between states. This model presents a flexible and elegant technique for designing disconnectable mobile applications.

2.3.4 Laputa

The Laputa project at Columbia University took a slightly different approach to file caching than Coda's hoarding mechanism [Skopp '93] and applied it to a software development environment. This was based on the concept of process-centred environments in which complex rule sets are used to relate and order the different files and tools used in a development task such as writing a program in C. Laputa exploited these rules to automate the caching of objects prior to disconnection.

Basically the user specifies which tasks are to be undertaken while disconnected and the system will then chain through the relevant rules and select any objects to be cached. Although Laputa does not deal with such issues as involuntary disconnection or weak connectivity, it does provide an interesting take on intelligent file prefetching and caching.

2.3.5 Ficus

Ficus [Page '98] is a modular addition to the Unix kernel that provides a peer-to-peer replicated file system, designed to be highly scalable and reliable. The basic concepts behind Ficus, providing a network–transparent file system that supports partitioned updates, are logically descended from the Locus [Walker '83] operating system. Ficus shares many features with Coda such as optimistic replication, the primary difference being that Ficus has a peer-to-peer structure unlike Coda's client-server arrangement. Ficus allows updates to be made to a data object provided that at least a single copy of it is available. A process called reconciliation, analogous to Coda's reintegration state, ensures that updates are propagated to any other replicas of the object.

Although not primarily designed for disconnected operation, Ficus does happen to work quite well in a voluntary disconnected mode [Heidemann '92]. As any machine running Ficus is able to provide a full file service, the disconnected machine can operate as normal in a disconnected mode and indeed Ficus makes no distinction between disconnected and connected states. With periodic connections to other Ficus machines on the network, the disconnected node can propagate updates and reconcile differences. This type of operation has been shown to work well, however the time and network traffic required for reconciliation works out to be quite expensive when performed over a normal telephone line, as would be the case in a 'home use' scenario.

2.4 The Jini Model

Jini [Edwards '99] from Sun Microsystems is a model for building distributed systems and is comprised of a set of protocols and facilities to do this. Jini was designed to provide an infrastructure that would support spontaneously created and self-healing communities of services. These services can consist of anything from actual hardware devices such as scanners or printers (and their attendant software) to purely software services. Jini uses Java as the common language of these communities and Java RMI provides the default communication mechanism. 'Spontaneously created' means that services can appear and disappear from the community fluidly and without any user involvement. The term 'self-healing' means that the communities will be resilient to changes in services, network topology and machine crashes.

Jini can be thought of as a thin layer of services built on top of, and using Java, especially Java RMI. Jini is perhaps best explained in terms of five concepts that together form the basis of the whole Jini idea of spontaneously created, administration-free networks of services. These five concepts are –

- Discovery
- Lookup
- Leasing
- Remote Events
- Transactions

Discovery

Jini services are grouped into 'communities'. Communities usually consist of all Jini services currently available on a given network subnet. In order to join this community, when a Jini service starts it must contact a lookup service, which keeps track of all currently available services. Note that the lookup service is itself a Jini service. When a lookup service has been detected, the prospective community member can step through a procedure called the join protocol to publish its own service to the community. Jini supports several methods of discovering the lookup service such as the Multicast Request Protocol, which is used by services when they first become active to announce their presence to any lookup services in the vicinity and the Unicast Discovery Protocol (UDP). UDP is used when there is a need to create static connections between two Jini services when one of the services in question knows the name of the other service. This form of discovery allows a service to connect to a lookup service on another network to publish its service there.

When discovery has been completed the service doing the discovery will receive an RMI stub (see Chapter 3.1) that can be used to talk to the lookup service for the local (default) community. The discovering service can then communicate with the lookup service and negotiate the publishing of its service.

Lookup

The Jini lookup services provide functionality not unlike that of a name server except that they can be searched for more than just string names [Jini Spec ‘00]. Each lookup service in all practicality contains a list of service names and corresponding objects that other members of the community can download in order to allow them to use that service. Lookup services can understand Java type semantics and therefore they can be searched for objects of different types of classes, superclasses, superinterfaces etc. Clients downloading these ‘proxy’ objects need not know anything about the actual implementation of the service, only the interface that they know the service implements. This separation of the definition of the means of interaction with the service and the actual implementation of that service is another major concept in the Java RMI and Jini model.

When a community member wishes to publish a service it calls a method on the proxy supplied by the lookup service upon discovery and provides its own proxy object as an argument. The publisher can supply attribute objects that describe the service such as name, location, comments etc. The proxy object supplied is a serializable Java object that provides clients with a downloadable front end that allows them to access the service. This is one of the key ideas in Jini, no specialised device drivers or software needs to be installed by clients as they can simply download code to do this as and when they need to use the service. The proxy object itself can be an RMI stub that communicates with the remote service, it may be an object that actually provides all of the service functionality itself without any need for a remote ‘back end’ or it could use some sort of private communication protocol to talk to the service such as when hardware devices are involved. The proxy contains a reference to the remote service that specifies where it can be contacted i.e. a hostname and port number pairing.

Leasing

Leasing is the mechanism used by Jini to allow communities to be resilient and self-healing in the event of sudden loss of services, network failures, machine crashes etc.

This is a very important aspect for any distributed system where small network and other failures can very often cripple the entire system. As a countermeasure against these problems Jini uses time-based resource reservation in which services are granted space in a lookup service for a limited period of time. These services must update their leases if they wish to remain listed as active services beyond the initial lease period. Any service that is unable to renew its lease will eventually be removed from the lookup service and hence from use by the other members of the community.

Remote Events

Jini (in common with most of Java) uses the concept of events to allow for asynchronous notification. However, in distributed systems like Jini different types of events, remote events, have to be designed for. Remote events are different from local events in many ways including difficulty of delivery, difficulty of determining correct delivery order and determining whether or not the event has been received.

Jini provides a single interface, `RemoteEventListener` (it has a single method, `notify()`) for objects that wish to listen for remote events and there is only one type of event object, `RemoteEvent`. Any Jini component that wishes to be able to send events specifies the conditions under which it will send them and provides its own means for listeners to express an interest in these events. As all events are simply variants of `RemoteEvent`, Jini allows third parties to use, forward or store these events without actually knowing what exactly they mean.

Jini takes the approach that distributed systems designers must decide themselves what level of certainty they need in their applications that events that are sent will reach their destinations. By using third party delegates, an application can store its messages for later referral or use if it fails simply by sending events to the delegate, which then stores them persistently. These events can then be resent if their target does not receive them. Designers can also plug a guaranteed delivery component into their systems that listens for events and then continually resends them to their targets until they are acknowledged.

Transactions

Partial failure is a problem that occurs when only part of a distributed computation succeeds and the rest fails. This means that the distributed system has not progressed to the new state that it should be in but yet it cannot return to its the original state. To address partial failures, distributed systems use a concept known as transactions. Transactions are a means of grouping and managing the execution of related operations so that either all or none of them are carried out. This ensures the system is always in a known and stable state. If one of the operations cannot complete then the whole transaction is aborted and the system is returned to its original state.

Two-phase commit is a protocol commonly used to manage transactions. In this protocol an entity called a Transaction Manager controls all operations that are participants in the transaction. The manager tells the participants to enter a precommit phase in which they each calculate their own bit of the overall result and then store this temporarily. Each operation then informs the manager whether or not they have been successful in doing this. If all the participants reply that their results have been computed then the manager will send a commit message that tells each participant to make their changes permanent. If any participant tells the manager that its task has failed then the manager will tell every participant to abort. They will then erase the results they have just calculated and return to the state they were in before the operation started.

Jini, in fact, does not fully implement the two-phase commit protocol outlined above. Jini provides a `TransactionParticipant` interface that each participant in a Jini transaction must implement and this interface has the obvious sounding methods `prepare()`, `commit()` and `abort()`. However, this is just an interface and the actual implementation of these methods is left to whoever is building the system. Basically what Jini does is act as the transaction manager and will call these methods on the participants as and when appropriate, rolling the participants back or forward depending on the replies it gets and calling `commit()` and `abort()` when it decides it is the right time. What actually happens when these methods are actually called on the participants is left up to the programmer. This allows the Jini designer to add secure transactions when he/she sees fit and to leave them out altogether if circumstances do not require them.

Chapter 3

Background

This chapter introduces the two main building blocks of this project, Sun Microsystems Java Remote Method Invocation (RMI) technology and the Architecture for Location Independent CORBA Environments (ALICE). Relevant points in each will be discussed including architectures, design features and capabilities, and comparisons to related technologies.

3.1 RMI

3.1.1 Introduction to RMI

Java's platform independence and security model combine to provide a powerful tool for developing distributed applications. Platform independence dispenses with the need for different versions of code for different machine types or operating systems as any platform that can provide a JVM can run Java code and Java's security model enables code to be loaded remotely and be trusted not to act maliciously. Together these two capabilities allow for network-mobility of Java code [Venners '00]. RMI and object serialization were developed to allow Java objects to become network mobile and to create a distributed object model that in essence allows objects in different address spaces to communicate. By granting objects the ability to hold references to objects in other JVMs (these will henceforth be called remote objects), to call methods on these remote objects and to pass parameters between themselves, what is effectively created is a distributed object-oriented programming framework.

The designers of RMI identified several important goals for supporting distributed objects in Java [Wollrath '96] –

- To support remote invocation of Java object methods in different virtual machines

- To retain as much of the original non-distributed Java semantics as possible while integrating the system into Java in a natural way
- To preserve the security provided by the Java model
- To make remoteness as seen by the RMI clients and servers as simple as possible

3.1.2 Semantics of RMI

RMI makes extensive use of one of the fundamentals of Java object-oriented programming, the separation of interface and implementation. In RMI, an object that will be called remotely must implement the Remote interface. The syntax of a call to a remote object by a client is exactly the same as that of a call to a local object. The client is programmed to the remote interface not to the implementation by the server object of that interface. Therefore apart from the knowing that the interface implements Remote and that a RemoteException may be raised during a remote method call, the whole RMI scheme is mostly transparent to the client.

RMI has many similarities with the Java object model including casting a remote object to any interface supported by the implementation and references to a remote object can be passed as arguments or returned as results in remote method calls. There are some notable differences between the two models however, the client objects can only ever interact with the interfaces to a server object never with the implementation itself, the semantics of parameter passing are slightly different and some of the methods provided by the Java object class have to be changed to take remoteness into account.

An RMI server object can either inherit from the RemoteServer class or the UnicastRemoteObject class. Each of these takes care of the process of exporting an object and readying it for calls when their constructors are called. Obviously, the server object must implement the same interface as provided by the Remote interface. Before invoking a method on a remote object a client must first obtain a reference to that object, this is usually returned as a result of a call to the rmiregistry, a simple bootstrap naming service provided by RMI (the stub which provides the means to make the remote call will be discussed later). References to remote objects are stored and can be retrieved by using the URL-based interface java.rmi.Naming. This interface allows servers to register references to themselves and clients to obtain these references by contacting the registry on the host specified by a URL supplied to the Naming.lookup(URL) method.

Calls by a client are then performed in exactly the same manner and with the same syntax as calls on a local object. Parameters passed and arguments returned in RMI calls (apart from remote objects) are passed by copy. This means the content of the non-remote object is copied before being passed on. Objects other than Java primitives and remote objects must implement the Serializable interface if they are to be passed. Remote objects on the other hand are passed by reference meaning that the stub for the object is passed rather than the object itself. Therefore the semantics of RMI parameter passing is pass by value in the common case, and pass by reference in the remote case. In the latter what actually happens is that RMI passes the calling object a stub with which to communicate with the remote object.

3.1.3 RMI Architecture

There are three layers in the RMI architecture – the stubs/skeletons layer, the remote reference layer and the transport layer.

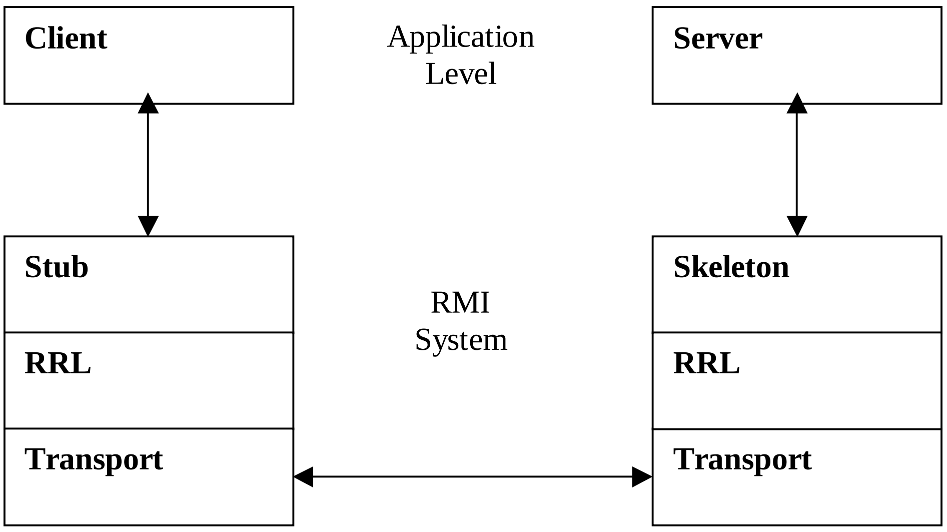


Figure 3.1 RMI Architecture

Stub/Skeleton Layer

The Stub/Skeleton layer provides the interface between the application layer that the client and server reside in to talk to the rest of the RMI system. The client uses the stub and the server uses the skeleton. The stub implements the same interface as the server object and is responsible for making calls to the remote object by calling the remote reference layer, marshalling parameters passed to it from the application into a format suited to on-the-wire transmission (a process known as pickling), and unmarshalling any returned data.

Likewise the skeleton is responsible for unmarshalling any parameters passed by the client, passing these parameters back to the application layer code and marshalling any return values. Both stubs and skeletons are automatically produced by running the rmic command.

Remote Reference Layer

The Remote Reference Layer deals with the specific invocation protocol chosen by the remote object and transfers data between the Transport and Stub/Skeleton layers. The invocation protocol can be unicast (in the case of the server object being a UnicastRemoteObject), muliticast (which allows for replication of remote server objects) or some other replication pattern. The RRL consists of client and server sections, both of which use a stream-oriented connection abstraction to convey data to the Transport layer. The Transport layer actually implements the connections and presents an interface to the RRL.

Transport Layer

The transport layer deals with the mechanics of transferring data between address spaces - it makes connections to other JVMs, monitors the liveness of these connections, maintains a list of all remote objects in the local address space, listens for incoming calls and locates the dispatcher for the target of a remote call and passes the connection to this dispatcher.

The Transport layer uses the information contained in the remote reference passed to it by the RRL to set up a connection to the relevant remote address space. The client passes the server the object ID from the remote reference so the servers transport layer can tell which object the client wishes to connect to. Although the default transport medium is TCP the RMISocketFatory interface allows for custom protocols to be used, creating a custom socket whenever RMI makes a call to the factory to obtain a new socket.

3.1.4 Basic RMI Operation

The standard format for client-server RMI applications is shown in the diagram below [RMI Spec '99].

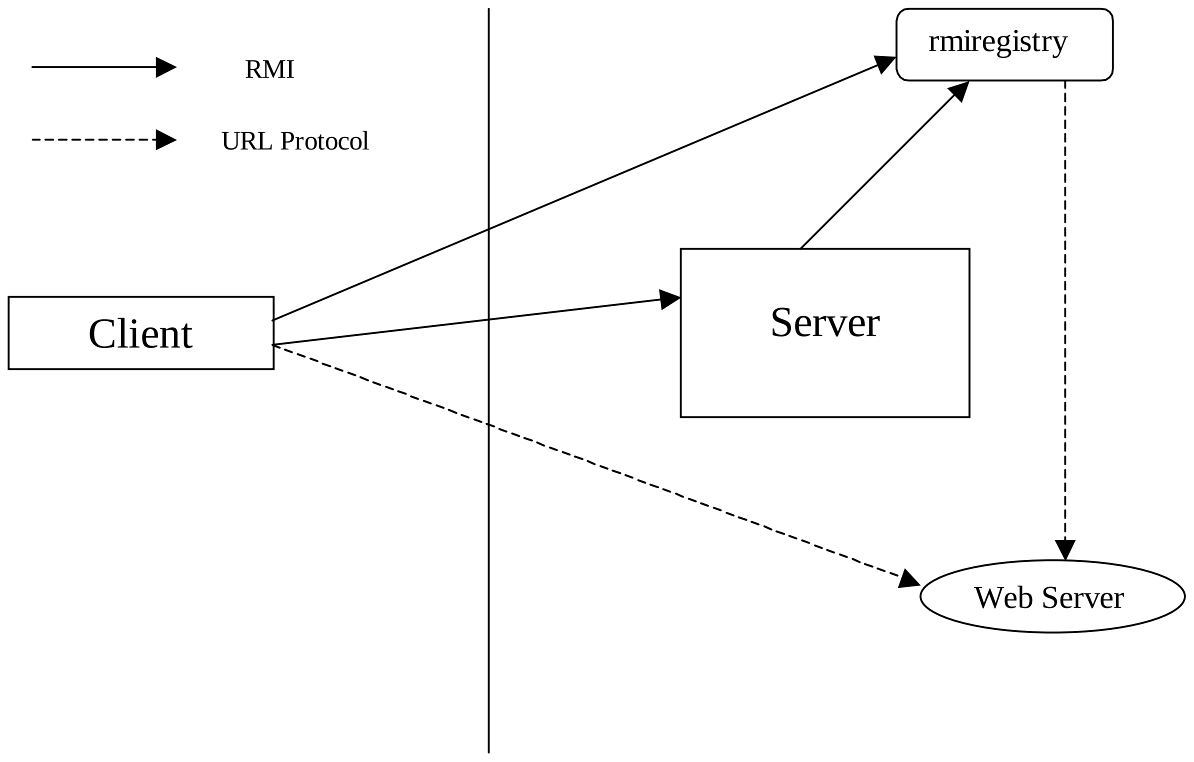


Figure 3.2 RMI Distributed Application

The remote method call proceeds as follows:

- The remote server object starts and registers itself the rmiregistry on the local machine. The rmiregistry checks that the Stub for the object is available from the web server pointed to by a codebase property supplied by the server.

- The client contacts the rmiregistry on the servers machine and is returned a reference to the object and the location of the Stub with which to contact it with. The client then downloads the Stub from the server.

- The client then makes the method call on the server object using the Stub, passes any required parameters and the corresponding result data, if any, is returned to the client. The remote method invocation is then complete.

3.1.5 Serialization and Dynamic Code Loading

As mentioned previously, the ability for code and objects to move between address spaces is fundamental to the design and operation of RMI. Serialization allows for the member data of a Java object to be turned into a steam of bytes to allow it to be transmitted via TCP or whatever transport protocol is being used to another JVM. Any Java object can be serialised as long as it has a public, no-arguments constructor, it implements the Serializable interface and it contains no references to any objects that aren't themselves serializable.

What serialization provides is a means to transmit the member data of an object but this is of little use without the accompanying class code. The code for the object cannot be transmitted by the same means so some other way must be found to give callers the implementation of a class. This is known as Dynamic Code Loading and marks RMI out from other remote procedure call implementations.

Usually a Java application looks in its classpath to find the implementations of any classes it requires. 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. Once a client gets a serialized object it will try to reconstitute it and when it can't find the class in its own classpath it will download it from the codebase location.

A MarshalledObject can represent a serialised object by passing the object to be serialised to its constructor. MarshalledObject provides a convenient means of storing a serialised object without having to reconstitute it. MarshalledObjects are automatically annotated with the codebase of the relevant implementation classes thereby providing a representation of everything a client needs to know to be able to utilise the object at a later stage.

Naturally the ability to dynamically download code also presents a security hazard, as the client needs to be able to trust the downloaded code not to do anything malicious. In an applet environment, only allowing the applet to connect to the machine it was downloaded from enforces a crude form of security. In the Java 1.2 platform the security capabilities have been extensively expanded. Any piece of RMI code that wishes to download code must first set a security manager. This manager will control the behaviour of downloaded code according to a set of security permissions set in a special policy file and passed at runtime as a property to the program. The policy file can give the manager exact instructions as to what the code can and can’t do, what directories it can access, what remote machines it can connect to, etc. This provides the programmer with a very flexible and powerful means of controlling the security of networked applications.

3.1.6 Garbage Collection

In order to clean up remote objects that are of no more use RMI employs a reference counting mechanism. Put simply, the RMI runtime keeps count of all live references to remote objects within each JVM, every time a live reference is encountered the count is incremented and a message is sent to the server informing it that the object has been referenced. Any time that a reference is discarded another message is sent to the server informing it of the discard. There are complex issues that must also be dealt with regarding the timing and order of these messages to ensure that the objects are not prematurely garbage collected.

When an remote object is no longer referred it is marked as being a candidate for collection by the RMI runtime. As long as a remote or local reference to an object exists the object is safe to use. The possibility of network partition occurring between a client and server means that a reference to a remote object may not always refer to a currently existing object and an attempt to use such an object will raise a RemoteException.

3.1.7 Activation

A distributed object system can contain many thousands of remote objects. If all of these objects were to remain persistently active then it is obvious that valuable system resources would be in continual use, even if no calls were being made on the vast majority of the objects [RMI Spec ‘99]. RMI gives the distributed application programmer the capability to deactivate objects that may not be in use for long periods of time and then activate them when they are required.

Activation in RMI is taken care of by three main components; an Activatable object, a wrapper class around this object and the rmid activation daemon. The Activatable object is much the same as a normal RMI remote server object except that it extends the java.rmi.Activatable class and has a special two-argument constructor that takes a MarshalledObject and an ActivationID as arguments. The wrapper class should be set up to inform the rmid what type of environment to run the object in when it is reactivated. The rmid will create a new JVM as a child process to run the object in when it detects a call for it. The wrapper will at least declare an RMISecurityManager to govern the security properties of the JVM the activated object is running in.

Note that although the use of activatable objects can increase the performance of a large distributed object system, care must be taken in its use to prevent system resources being wasted by objects being activated and deactivated between closely occurring method calls [Edwards ‘99]. Objects that have been activated should remain active for a certain period of time (perhaps calculated from the average time between object method invocations) before being deactivated again to prevent the needless creation of new JVMs. Activatable objects can also inform the rmid that they wish to be automatically restarted whenever the rmid itself restarts. This allows for server objects to be brought back on line quickly after a system reboot.

3.1.8 RMI Protocol

RMI uses a simple on-the-wire protocol format; a client sees an Input Stream and an Output Stream, both being paired to a socket connection to the server. Since the stream connections are paired there is little need for much header information in the PDUs. When the first PDU is sent over a connection, a header indicating the RMI

Version and the specific protocol being used prefixes the message. The protocol can either be the default Stream Protocol, the SingleOp Protocol or the Multiplex Protocol. When the Stream Protocol is being used all messages after the first are sent with no headers.

The SingleOp Protocol is used if the messages are being wrapped in HTTP requests and interactions more complicated than a single request and response are impossible. This is commonly used to allow remote object methods to be invoked by clients residing behind a firewall. The Multiplex Protocol which is used when only one of the endpoints can open a bi-directional TCP connection and allows the two endpoints to each open multiple full-duplex connections to each other. An instance of such a situation would be an applet in which the security manager prevents the downloaded applet from creating a server socket to the host it was downloaded from. If the applet was permitted to open a normal socket connection to the host then connections could be multiplexed over this socket and the originating host could invoke methods on any RMI objects exported by the applet [RMI Spec '99].

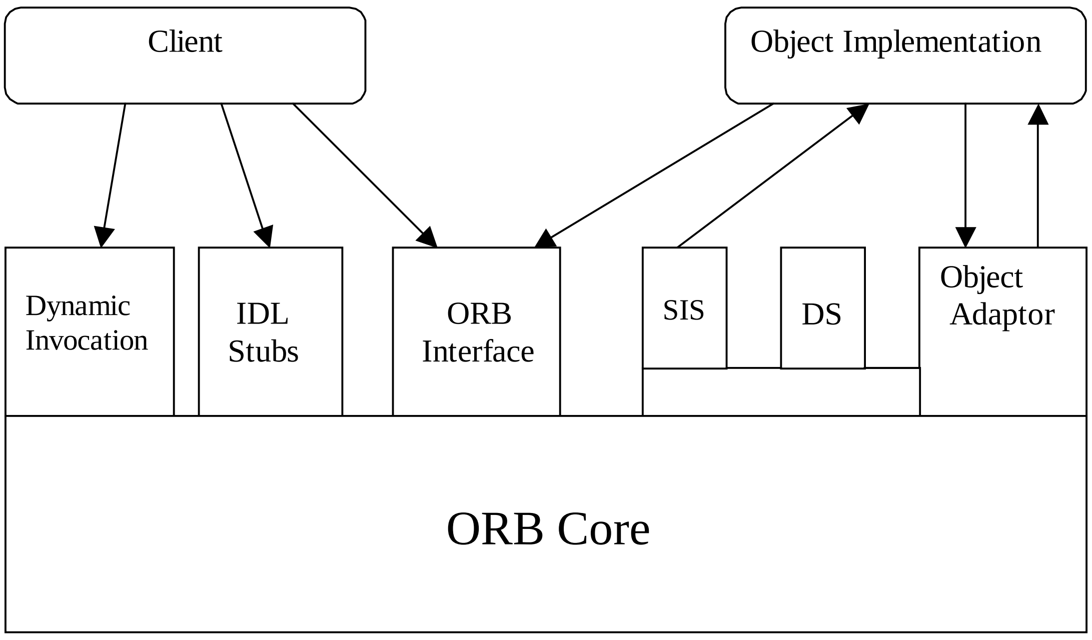
There are three types of messages that can be sent from an Output Stream: an RMI method call, a Ping to test a connection, and a Distributed Garbage Collection ACK that tells the JVM in the server machine that remote objects have been received by the client as a return value. There are also three types of Input messages: Call Return, which is the return from a method invocation; HttpReturn, the same as Call Return except it is wrapped in HTTP, and PingACK, the response to the Ping message.

3.1.9 Comparison with CORBA

RMI is just one of many distributed object paradigms available, one of the most popular rival technologies in use today is the Common Object Request Broker Architecture (CORBA) standard from the OMG. CORBA, like RMI, provides a means for programmer to construct object-oriented distributed applications, freeing them from the responsibility of having to deal with low-level communication issues. In contrast to the simple structure of RMI, the myriad of entities and services available in CORBA makes it appear comparatively complicated.

Central to the design of CORBA is the Object Request Broker (ORB), which acts as an object bus through which clients can interact with other remote or local

CORBA objects. The ORB “is responsible for all the mechanisms required to find the object implementation for a request, to prepare an object implementation to receive a request, and to communicate the data making up the request” [OMG ‘95]. The client uses the ORB to obtain a reference to a CORBA object and then a client can call methods on that object using either the Dynamic Invocation interface or an IDL stub. The ORB conveys the request to the object and returns the reply to the caller. The references to server objects in CORBA are called Interoperable Object References (IORs). An IOR consists of a hostname and a port number where the client can locate the server object, and also a special identifier called an object key that uniquely identifies the server object at the given location.



SIS = Static IDL Skeleton

DS = Dynamic Skeleton

Figure 3.3 Structure of ORB Interfaces

As in RMI an (IDL) interface is completely independent of the actual implementation of an object. IDL allows objects written in different languages to interact with each other, so the client can server can be written in different languages. When the IDL interface is compiled, information about it is stored in an Information Repository (IR). This allows a client to obtain runtime information about the interface by querying the IR and use this information to dynamically invoke a method on a remote object by using the Dynamic Invocation Interface. The Dynamic Skeleton Interface on the server side allows a client to invoke methods on a CORBA server object that has no knowledge about the object it is implementing.

An entity called an Object Adaptor sits on top of the ORB and connects the server object to the ORB, providing services like method invocation, mapping object references to implementations etc. As RMI uses the RMI Protocol as its underlying remote protocol, CORBA uses the Internet Inter-ORB Protocol that runs over TCP/IP. IIOP uses the Common Data Representation (CDR) scheme to transfer data types across the wire.

3.2 ALICE

This section will look at the ALICE (Architecture for Location Independent CORBA Environments) framework for providing mobility support for CORBA objects on mobile devices. After a brief introduction to the topic a more detailed discussion of relevant details such as the ALICE software architecture and the various layers in this architecture will be presented.

3.2.1 Introduction to ALICE

ALICE is a design to allow mobile computing devices to carry CORBA server objects that can be invoked by both non-mobile and other mobile hosts with no knowledge of the mobility of the server. Equally, client objects on the mobile host can interact with objects on other hosts. One of the most important aspects of ALICE is that no centralised register is used to track the current location of mobile servers. ALICE tackles the problems associated with the mobility of CORBA objects by using a session layer approach with some application level support [Haahr ‘99].

ALICE presumes a mobile environment such as that shown in Figure 3.4. Mobile hosts (MH) connect via wireless links to Mobility Gateways (MG) which are wired to the rest of the network. The mobile hosts can move between MGs, thereby changing their point of connection to the fixed network in a process known as handoff. The gateways perform the role of proxies by relaying communications from the MH to the rest of the network and from remote hosts to the MH. The gateways also have the responsibility of carrying out CORBA specific duties such as address translation of IORs to account for the mobility of server objects.

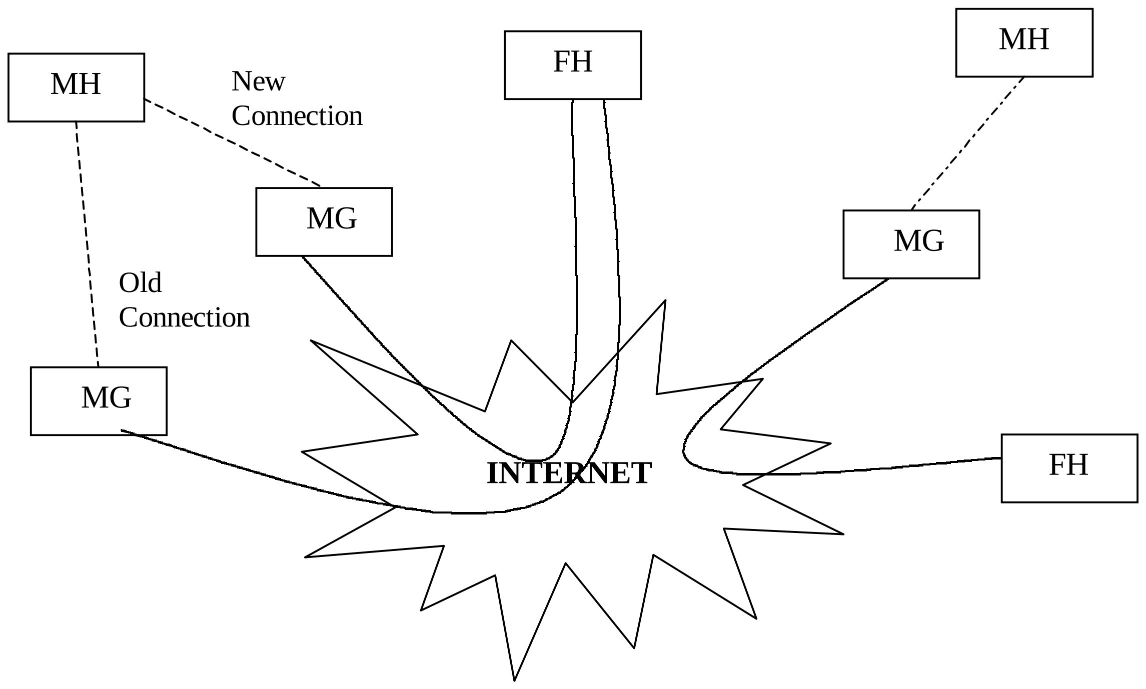


Figure 3.4 ALICE Environment

The general software architecture of ALICE follows a layered approach with different mobility problems being solved at different layers. As mentioned before, ORBs generally use TCP/IP at the transport level, however TCP/IP connections are frequently unstable in a mobile environment and are subject to being broken. 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 is a mobility-unaware part of the architecture that implements the minimum amount of ORB functionality to allow it to send and receive inter-ORB messages. The footprint of the IIOP is designed to be as small as possible to cater for the limited memory capacity on a mobile device. The S/IIOP (or Swizzling IIOP) Layer is the mobility-aware component of the IIOP layer and is used to perform address translation on CORBA IORs.

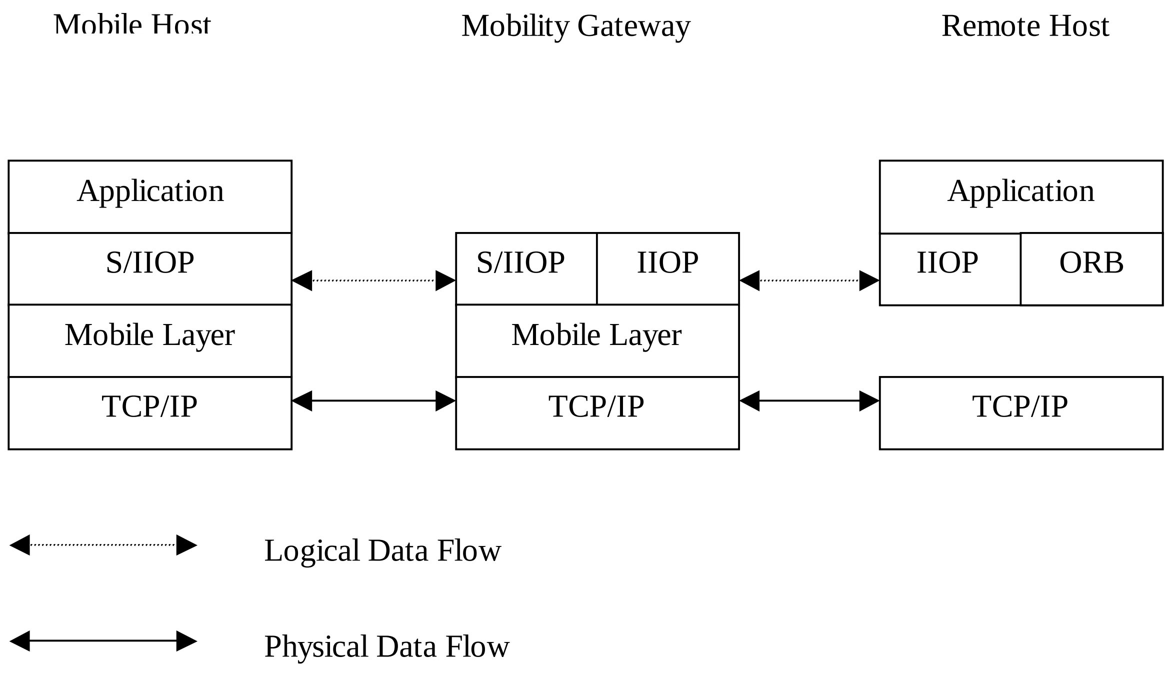


Figure 3.5 ALICE Software Architecture

3.2.2 ALICE Mobile Layer

The Mobile Layer provides the low-level support services required to maintain connections in a mobile environment. Basically clients of the ML use it to create what they think to be normal TCP socket connections. What is instead produced is a connection to the current Mobility Gateway, 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. When the MH-MG connection is broken 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 Mobile Layer provides four main services to the layers above it [Haahr '99] –

- 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
- The S/IIOP layer uses it to obtain mobility information so that it can perform address translation and request forwarding
- It performs handoff between MGs and tunnels existing connections from the old MG to the new one

3.2.3 IIOP and S/IIOP Layers

IIOP Layer is an OMG defined standard that 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 a lot of the complexity of the actual protocol from the application programmer while still allowing for manipulation of relevant parameters when required. The IIOP Layer also allows the Mobile Layer to be plugged in and out cleanly whenever mobility support is necessary or not.

The IIOP Layer performs functions that are necessary for the IIOP layer to operate correctly when server objects are being used on the Mobile Host. As explained in Chapter 3.1.9, Interoperable Object References (IORs) are used in CORBA to uniquely identify and locate a server object. An IOR essentially consists of a hostname and port number to connect to the server object at. Server objects on the MH will export an IOR that points 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 an operation 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 there will forward the request to the MH. When the MH changes it 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 IORs it holds that pointed to the MH to now point to the new MG.

3.2.4 Handoff

The limited range of wireless communication methods 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 [Haahr ‘99]. 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. Hopefully, this should only prove to be the case on rare occasions.

Chapter 4

Design

4.1 Overview

This chapter discusses a design for supporting mobile RMI clients and servers. The design assumes that mobile hosts will communicate with remote hosts on a wired network via mobility gateways in the same manner as discussed for the ALICE environment and use as much of the ALICE software architecture as is appropriate. This chapter looks at the differences between the addressing and naming schemes in CORBA and RMI and examines how this affects locating mobile server objects in Mobile RMI and the changes required in the overall ALICE architecture. An application level solution to these problems is introduced along with methods for dealing with passing references to remote objects and performing handoff between mobility gateways.

4.2 Comparison with Previous Implementation

As discussed in Chapter 3, RMI and CORBA have dissimilar means of locating remote objects. One of the major differences is that in CORBA the programmer can easily manipulate the IORs used to locate remote server objects to point to the mobility gateway and can then have the S/IIOP layer there forward requests to the mobile host. In RMI the equivalent to the IOR, the RemoteRef, cannot be created independently of a remote object, accessed or manipulated. This effectively means that the ALICE method of redirecting method invocations from the gateway to the mobile host cannot be used in the same way for RMI.

A number of approaches were considered to overcome the differences between the RMI and CORBA addressing models. One was to use downloadable RMI socket factories which the client would download from the gateway and which would create a socket connection to the gateway on a known port. The gateway would then forward the call to the mobile host by opening another socket across the wireless link. There are a number of difficulties inherent in this approach.

Firstly, the rmiregistry on the gateway would still need to have a remote object registered in with it that implemented the server interface so some form of code downloading as described in the next section would still have to occur. Secondly, the gateway listening on the well-known port for client connections would have to intercept the first RMI call and change endpoint identifiers. In the reply to the first RMI call by a client the server specifies an endpoint identifier (a hostname, port number pairing) that it can see the client is using to connect to it on. The client can use this information to determine its own hostname if it is unable to do so, perhaps due to security restrictions. The client then responds with an endpoint identifier at which it will accept connections from the server. If calls were tunnelled through the gateway the first endpoint identifier would have to be changed to point to the mobile server instead of the gateway and the second would have to be changed so that the returned endpoint pointed to the gateway instead of the remote host. These changes could only be affected by direct manipulation of the incoming and outgoing byte streams, as there is no API to change RMI calls.

Overall this design soon became overly complex and inelegant (with both high-level application layer and very low-level transport layer components) and so it was decided to try a different approach. In the end it was decided to deal with all of the difficulties of address translation and request forwarding at the application layer instead of a lower level in the protocol stack. This eliminates the requirement for an RMI version of the S/IIOP layer, as there will be no manipulation of RemoteRefs. So instead of having the S/IIOP layer on the gateway alter the RMI calls and then forward them to the server, proxy objects on the gateway are used to effectively relay the calls to the server and relay any responses back to the client. The updating of the RMI proxy stubs at the gateway to point to the new proxy on the new gateway to which the server has moved mimics the ‘reswizzling’ of IORs on the old gateway that occurs during handoff in CORBA ALICE.

4.3 Mobile Host as Client

Consider an RMI client object located on a mobile host interacting with a remote server via a mobility gateway and with the support of the Mobility Layer (see Figure 4.1 below).

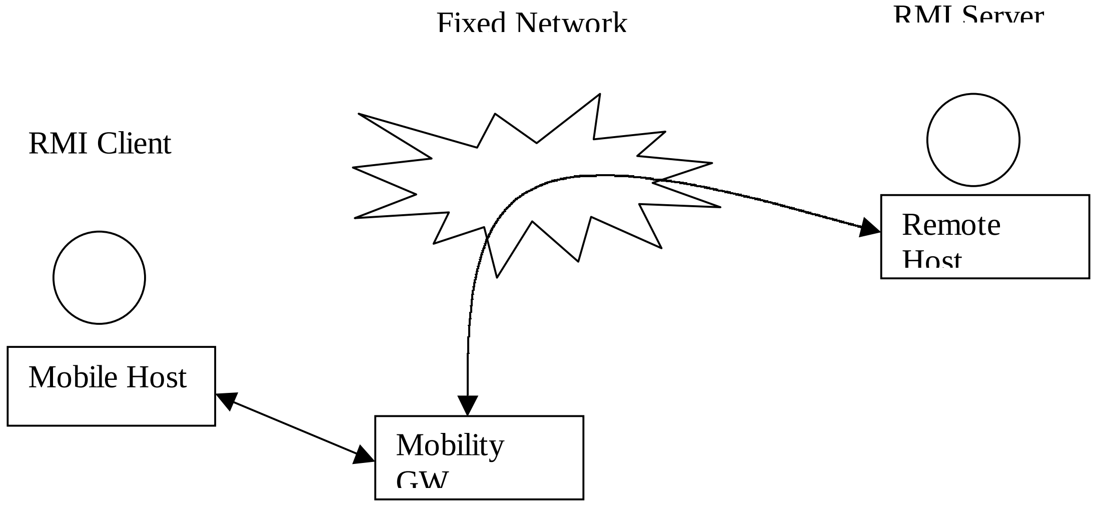


Figure 4.1 Mobile RMI client connects to remote RMI server via Mobility GW

In this case the Mobility Layer will provide all the support necessary for the mobile client to interact transparently with the remote server. The ML will tunnel the lookup requests, method invocations and all other interactions initiated by the client with the server through the mobility gateway. Whenever the client calls the RMISocketFactory to produce a socket connection to the server it will instead return a reference to a virtual socket to the client. The client has no idea that this virtual socket is connected to anything but the address it specified to the RMISocketFactory. When the client tries to send or receive any data on the socket the ML will send a message to the mobility gateway directing it to open a connection to the original address specified by the client. Any data sent by the client over the virtual connection will be sent to the mobility gateway and redirected from there to its intended target. Similarly, data returned from the server to the gateway will be 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 tunnelled 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 position. Therefore the ML provides all the mobility support required for mobile RMI clients. One proviso is that the client does not pass references to remote objects as parameters to methods invoked on the remote server. If the server then tries to invoke methods on these objects then the mobile host is effectively acting as a server itself. This introduces another set of difficulties that will be discussed in the next section.

4.4 Mobile Host as Server

When an RMI server is located on a mobile host the Mobility Layer will provide the same low-level support as described earlier. However whereas the movement of a mobile client did not necessitate any mobility support apart from that supplied by the ML, the ML on its own is not enough to support mobile RMI servers. In CORBA ALICE the S/IIOP Layer gave the required additional support. The differences between the addressing schemes in CORBA and RMI mean that a similar approach is not appropriate. As remote hosts cannot directly contact a mobile server, all communication must go through whichever mobility gateway the server is currently connected to. Since the client cannot hold an RMI stub that refers to the mobile host so it must instead hold a stub that refers to an object on the gateway.

It is the use of such a ‘proxy’ object that forms major part of the solution. The mobile server gives the code for the proxy to the current mobility gateway to use in forwarding incoming calls to the actual server on the mobile host. To provide the services to the server object to allow it to give the gateway the proxy and register it there, a special remote object called the GatewayRegistry is used. 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 is not actually passed directly to the gateway instead a special class called a Carrier class which implements a well-known interface is passed. The GatewayRegistry creates an instance of this

Carrier class and calls a method on it that downloads the Proxy class, the interface it implements and its RMI stub from the server and then instantiates the Proxy class and registers it with the local rmiregistry. The basic operation of the proxy scheme is shown in Figure 4.2 below.

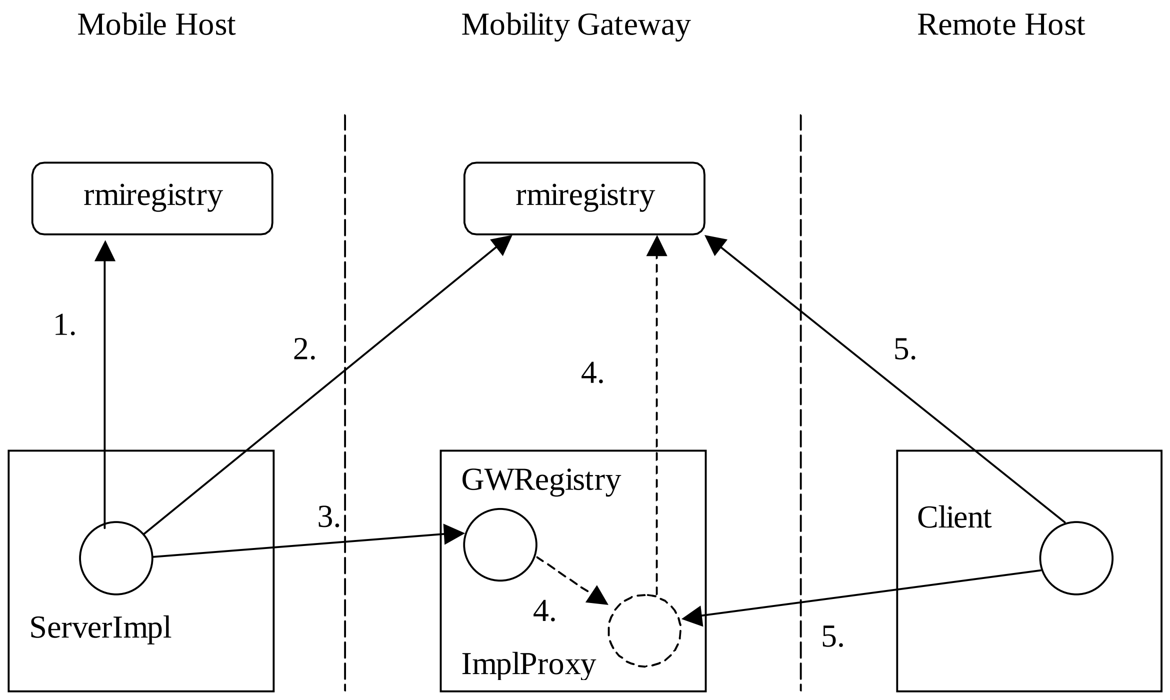


Figure 4.2 Procedure for passing Proxy Objects

The procedure follows the following steps:

1. The server object, ServerImpl, which implements the Server interface, 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 calls a method called register() on the GatewayRegistry object, passing as parameters the name of the Carrier class and the address of a web server where it and all associated classes can be found.

4. The GatewayRegistry object then downloads the Carrier, Proxy and Server interface classes from the mobile host and instantiates a Carrier object. It then calls a well-known method on the Carrier object, which creates a new ImplProxy object 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 number of other difficulties must be considered for full operation of the proxies. Firstly, the stub for the ImplProxy object (which was compiled on the mobile host) is needed both to allow the ImplProxy to be registered on the gateway (the rmiregistry checks for the presence of stubs before it allows any bind(…) or rebind(…) operations) and it is obviously needed so that clients can download it to talk to the proxy. This stub is not implicitly downloaded with the class file for the ImplProxy as it would be when a client makes a lookup call to an rmiregistry. Instead the stub must be explicitly downloaded by the Carrier object from the same web server running on the mobile host that the ImplProxy and Carrier classes were downloaded from.

Another problem occurs when an invocation of a remote method on the server returns a reference to another remote object. As discussed in Section 3.1.2, RMI returns the stub for any remote object returned from a remote method call. Returning this stub to the client via the proxy is pointless, as the client cannot contact the mobile host itself. What must happen instead is that the ServerImpl must give 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 proxy for the returned object and pass the stub returned from the mobile server to the newly created proxy that it will then use 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 Handoff

Much of the mechanics of the handoff process are taken care of by the Mobile Layer with the actual server application being unaware of any change in location. When the mobile host moves to a new gateway any existing connections between a client and the server will be tunnelled to the new gateway as discussed in Section 3.2.4. Therefore any currently open connections at the time of handoff will still be valid even after the movement of the server. However since the old gateway can no longer contact the mobile host, the ServerImpl stub held by the ImplProxy object on the old gateway is no longer of any use and any new clients connecting to it will not have their calls forwarded. To overcome this a means of communication between the two gateways has to be introduced. This is achieved by extending the functionality of the RMI GatewayRegistry objects held by the gateways.

To explain how handoff occurs in the Mobile RMI model we will start by assuming that the system (i.e. the mobile host, the mobility gateway and the remote host) has been initialised 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 affected as follows:

1. MH sends a Handoff Request message to the new MG and all existing server-client connections through the old gateway are tunnelled 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 Carrier and ImplProxy objects and all associated classes and interfaces are uploaded to the new MG. The Carrier object is run and the ImplProxy object is registered with the rmiregistry on the new MG.
3. MH calls the method callOldMG(…) on the GatewayRegistry stub from the new gateway. This 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 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 (i.e. before the handoff process had begun) downloaded from the MH. The ImplProxy replaces this stub with one that it downloads from the new MG.

In this way a ‘chain’ of proxy objects is set up between the MH and the RH. The client application on the RH, which is unaware of communicating with anything other than the old ‘home’ MG, calls the ImplProxy on the old MG. This in turn forwards the call to the ImplProxy object on the new ‘current’ MG. Lastly the new MG calls the actual Impl object on the MH, which actually performs the service. In the same way data is returned through the proxies from the RH to the FC.

The call from the new MG to the old MG also causes the old MG to replace the stub registered with the local rmiregistry with that of the ImplProxy on the new MG. In this way any new clients accessing the old MGs rmiregistry will receive a Stub to talk directly to the new location of the ImplProxy instead of communicating via the old gateway. Also, whenever the mobile host changes gateway, all gateways that it had previously registered its server object with must be notified of the change of location. Since all the gateways in the chain all the way back to the original gateway have the required GatewayRegistry stubs this is just a matter of each Gateway making a call to the previous one and telling it the location of the new stub.

Another factor to consider is the race condition that occurs when the handoff call from the new gateway reaches the old gateway and a client connected to the old gateway makes a call on the ImplProxy object. The call on the proxy will have to be suspended using a lock on the ImplProxy object until the stubs have been swapped.

4.6 Semantics of Proxy and Carrier Classes

The proxy and carrier classes exported by the mobile server object have a generic format. The only difference between proxy and carrier objects exported by servers implementing different remote interfaces is the name of the proxy and carrier classes, the name of the remote interface that the proxies implement, and the inclusion of whatever methods the remote interface specifies. All of the rest of the actual code including the packages imported, methods to download the required classes, register the ImplProxy etc. are entirely generic. As a result of this it is possible for both the carrier and proxy classes to be automatically generated (e.g. with a Perl script) provided the remote interface and the carrier and proxy class names are supplied as arguments.

4.7 Summary

In this chapter a design for allowing similar mobility for RMI clients and servers as is currently afforded to CORBA servers using the ALICE architecture was outlined. The differences between object addressing in RMI and CORBA that necessitate a radical change in the ALICE architecture were outlined and an application level solution based on the use of proxy objects was given along with schemes for allowing the return of remote references from remote method invocations, again based on proxy objects. A means of allowing handoff to occur between mobility gateways was then described which allowed not only current clients of the server to continue making calls but allowed new clients contacting the old gateway to be redirected by use of RMI stubs, to the new gateway. Finally, the semantics of the carrier and proxy classes needed for the design were examined and it was concluded that they were very much generic and as such were candidates for automatic generation.

Chapter 5

Implementation

This chapter discusses how the Java Mobility Layer was completed and how the proxy object design discussed in the previous chapter was implemented. Both of these were completed and successfully tested together. Additional work on making the provision of mobility support more transparent to the programmer was not completed although the issues involved were examined in detail. The chapter starts by discussing the structure of the Mobility Layer and the work that remained to make it fully functional. Following this the RMI objects used to implement the proxy scheme are closely examined.

5.1 Implementation Goals

The main aim of the implementation was to construct an RMI server capable of residing on a mobile host and can interact with remote clients that have no knowledge of its mobility. The server should be able to perform handoff between mobility gateways allowing it to change point of contact with the fixed network while at the same time not disturbing any existing transport connections to clients and remaining contactable to new clients. The Java Mobility Layer should provide the session layer mobility support, re-establishing broken transport connections, caching data, connection multiplexing etc. The application layer proxies on the gateways should forward RMI calls to the mobile server and the GatewayRegistry objects should provide the mechanism for the gateways to communicate with each other when handoff occurs. The task of writing mobility capable RMI servers should be made as easy for the programmer as writing a normal RMI application. To achieve this the tasks of producing proxy and carrier objects for a mobile server should be automated and the code required to register the proxy with the gateway should be hidden behind the code that normally performs the RMI task of binding to the local rmiregistry.

5.2 Java Mobility Layer

Java applications cannot interface with the original ALICE Mobility Layer as it is coded in C. There were two possible ways to provide Java applications with the same mobility support that was afforded to C applications; one was to completely recode the Mobility Layer 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 after it was realised that the size and complexity of the glue code would far outweigh the costs of a full Java rewrite of the layer [Corbett '00]. At the start of this project much of the Java Mobility Layer had been coded but was not fully functional and still required extensive debugging.

5.2.1 Previous Work on the Java Mobility Layer

The following is a brief description of the main classes used in the implementation of the Java Mobility Layer.

MSocket

The MSocket is the virtual socket connection that is returned to the RMI client when the createSocket() method is invoked on the RMISocketFactory. The MSocket has an associated MInputStream and MOutputStream, which are used to copy data to and from the relevant caches when data is sent or received by the RMI application.

MHServerIn

The MHServerIn thread is responsible for listening on the transport connection to the mobility gateway for any new messages, interpreting the message and then writing any contained data to a cache from where the data can be read by the MInputStream.

MHServerOut

The MHServerOut thread constantly checks the output caches written to by the MOutputStream for new data and if it finds any it writes it out to the gateway using the MGatewayConnection.

MGatewayConnection

MGatewayConnection represents the actual transport connection to the gateway and its input and output streams are used by the MHServerOut and MHServerIn threads to send and receive messages. The MGatewayConnection class also monitors the state of the transport connection and re-establishes it if it breaks.

MGSetup

This thread initialises the Mobility Layer when the first call to create a socket occurs. It sets up the connection to the gateway by instantiating the MGatewayConnection class, sets up the SocketArray class and starts the MHServerIn and MHServerOut threads. After this it creates a dummy socket that MSocket objects connect to in order to be able to get their MInputStream and MOutputStream objects.

SocketArray

The SocketArray class contains an array of all MSocket objects that have been created.

Semaphore

The concurrent operation of the numerous threads used in the Mobility Layer is controlled by extensive use of locking mechanisms. Since the synchronization mechanisms used in Java are different to those available in C/C++ a Java model of the semaphore used in the original C Mobility Layer was implemented so the same form of locking could be used. The resulting Semaphore class provided an object to act as the lock and synchronised acquire() and release() methods to obtain and release the lock object.

CachedData

All data that is placed in any of the caches is placed in a container class called CachedData. The class contains the actual data along with the length of the data and a unique identifier for the packet. This identifier is used to locate the data and remove it from the cache when an acknowledgement of its reception by the communicating party is received.

Basic Operation of the Mobility Layer

The basic operation of the Mobility layer is illustrated in Figure 5.1 below. When an RMI application requires a socket it makes a createSocket() call to the RMISocketFactory which will return a predetermined type of socket. In this case it will return a reference to an MSocket. If this is the first MSocket to be created then the MGSetup thread will first initialise the Mobility Layer and make the connection to the gateway by creating an instance of the MGatewayConnection class. The MSocket will connect to the dummy socket provided by MGSetup. 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.

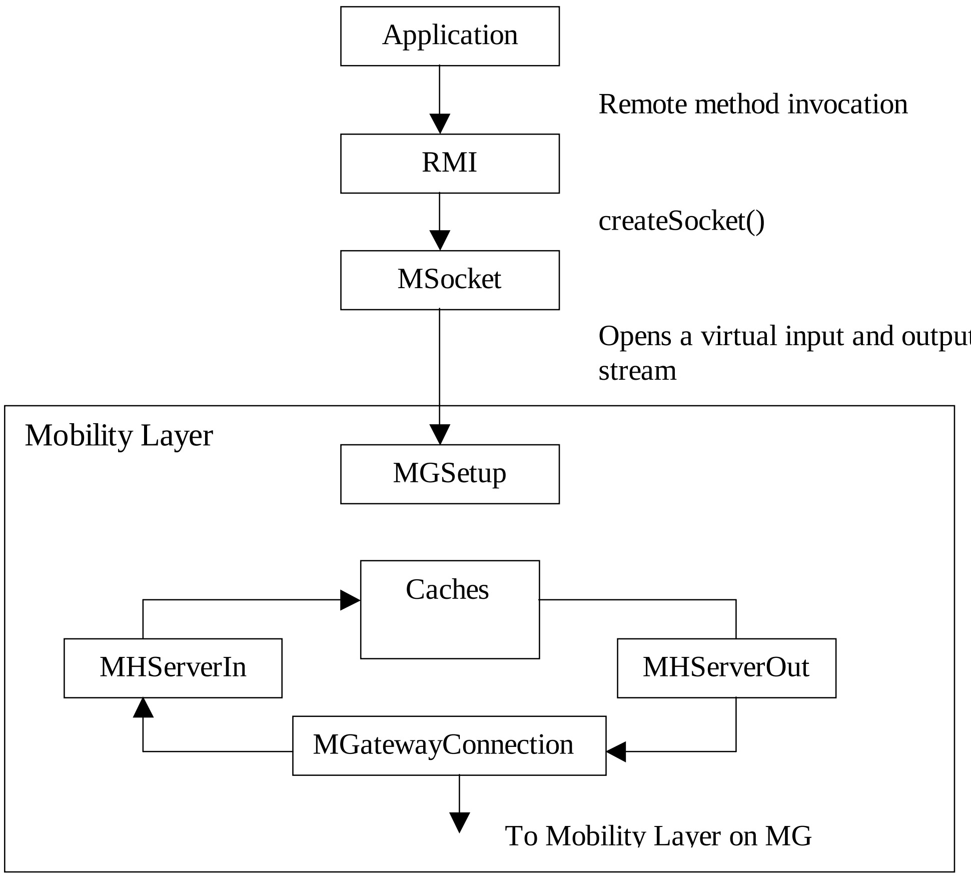


Figure 5.1 Normal operation of the Mobility Layer

5.2.2 Completion of Java Mobility Layer

The components described in the previous section had already been implemented but had not been fully tested and debugged. Continuing work on the layer in this project successfully completed the implementation and tested it using RMI applications. Much of the debugging concerned fixing small synchronisation problems with the various caches used in the layer and their interaction with the input and output threads. Other work involved the addition of a means of properly closing sockets, an RMISocketFactory to return the correct type of socket depending on whether the socket target is local or remote and a new scheme to assign logical connection identifiers to MSockets to replace the incorrectly functioning original.

Also developed during the course of the project was a skeleton Java mobility gateway that mimicked the operation of the C gateway, accepting socket connections from the mobile host, responding to messages with the correct acknowledgement, relaying data to the remote host etc. The Java gateway was an invaluable tool for debugging the Mobility Layer. However the Java gateway did not perform all the tasks of the C mobility gateway; all sockets created on the gateway to connect to the mobile host were ordinary Java sockets not ALICE MSockets. Further work is required to integrate the Java Mobility Layer with the existing C mobility gateway and to implement handoff.

5.3 RMI Mobility Proxy Objects

With the completion of the Java Mobility Layer the session layer section of the required mobility support was in place. Next to be implemented were the RMI objects that would provide the application layer support for mobile RMI server objects. These objects include the Proxy objects that forward RMI calls to the mobile server from their position on the mobility gateway; the GatewayRegistry objects that the server uses to pass the Carrier and Proxy objects to the gateway and which provide the means for gateways to communicate during handoff and the Carrier objects which allow the GatewayRegistry to interact with Proxy objects which implement interfaces unknown to it.

5.3.1 Overview of Implementation

The design requires having RMI objects on the mobile host and the gateway. The RMI server object resides on the mobile host, which also holds the code for the Carrier and Proxy classes in file directories serviced by web servers so that they can be downloaded by other machines. The Carrier class is basically a wrapper class for the Proxy that is used to pass the Proxy class to the GatewayRegistry RMI object on the mobility gateway. Every mobility gateway contains one of these GatewayRegistry RMI objects that implement an interface (called MobilityRegistry) that is known to the mobile host and hence it can invoke methods on it after downloading its stub. The remote host holds a client object for the service implemented by the remote server. The mobile host and each gateway must have rmiregistries operational at all times.

5.3.2 Classes and Interfaces

The following are brief descriptions of the classes and interfaces used in the implementation of the proxy scheme.

Server Interface

This is the interface that the server on the mobile host implements and is known to both the remote server and the client wishing to invoke methods of that interface on the remote server object. RMI requires that interfaces used by remote servers extend the java.rmi.Remote class and that each method of a remote interface declares a java.rmi.RemoteException [RMI Spec '99]. For the purposes of this discussion a server interface (see Figure 5.2 below) called Arith is examined that defines a single method to add together two arrays supplied by the client as parameters and returns the resulting array.

```
public interface Arith extends java.rmi.Remote {
     int [] add(int a[], int b[]) throws RemoteException;
}
```

Figure 5.2 Arith interface

Carrier Interface

The carrier class that is exported to the mobility gateway by the server object implements the Carrier interface. In this discussion the carrier class is called ArithCarrier. This Carrier interface is known to the GatewayRegistry and is used to forward calls to the Proxy object, which implements interfaces unknown to the GatewayRegistry. The interface only defines two methods – getHostURL(), which returns the server location used as a unique identifier and passNewAddress() that the GatewayRegistry uses to pass the location of the Proxy stub on the new gateway to the Proxy object on the old gateway so it can change its stub.

```
public interface Carrier
{
     public String getHostURL();

     public void passNewAddress(String newMGAddress);
}
```

Figure 5.3 Carrier interface

MobilityRegistry

The MobilityRegistry interface is a remote interface that defines the interactions that occur between the mobile server object and the GatewayRegistry object on the gateway that implements the MobilityRegistry interface. Three methods are defined by the interface – register(), callOldMG() and handoff(). The mobile server calls register() when it first makes contact with a mobility gateway that it wishes to use and passes as parameters to register() the location of the carrier class code to be downloaded by the gateway, an rmi-formatted address pointing to the mobile servers RMI stub in the rmiregistry on the mobile host and the name of the carrier class. The mobile server calls callOldMG() on the Stub of the new GatewayRegistry object when it connects to a new mobility gateway after leaving another. The server passes the new GatewayRegistry object the address of the last gateway the mobile host was connected to and the address from which the mobile server objects RMI stub can be downloaded. The new GatewayRegistry object will then contact the gateway at the specified address and download the stub for its GatewayRegistry object and invoke the handoff() method on it. Passed as parameters to handoff() are the address where the new proxy objects RMI stub can be found (i.e. the new gateways' rmiregistry) and the String serverURL that was passed to the new GatewayRegistry object by the mobile server when it called callOldMG(). The old GatewayRegistry object uses this String to identify the correct Carrier object from an array of stored Carrier objects. The passNewAddress(serverURL) method will then be called on the Carrier object and this will in turn call the changeStub(serverURL) method on the Proxy object. Each of these methods returns a Boolean value as a signal of success or failure.

```
public interface MobilityRegistry extends java.rmi.Remote
{
     boolean register(String codeLocation,
          String serverImplLocation, String className)
          throws java.rmi.RemoteException;

     boolean callOldMG(String oldMGAddress, String serverURL)
          throws java.rmi.RemoteException;

     boolean handoff(String serverURL, String newMGAddress)
          throws java.rmi.RemoteException;
}
```

Figure 5.4 MobilityRegistry interface

Proxy Class

The proxy class, here called ArithImplProxy, implements the above interface and also extends the UnicastRemoteObject class since it must register itself with the rmiregistry on the gateway. The ArithImplProxy class has one field, a RemoteStub object that represents the RMI stub that the Proxy uses to communicate with either the server object on the mobile host or another Proxy on a different gateway if the mobile host has changed gateway since registering the Proxy with the gateway it is located on. The ArithImplProxy class defines the method getInitialStub(..) in which the stub from the mobile server object is downloaded using the URL supplied to the Proxy from the Carrier object and the method changeStub(..) which discards the old stub and replaces it with the new one. changeStub(..) also replaces the stub registered in the local gateways' rmiregistry with the Stub for the ArithImplProxy on the new gateway.

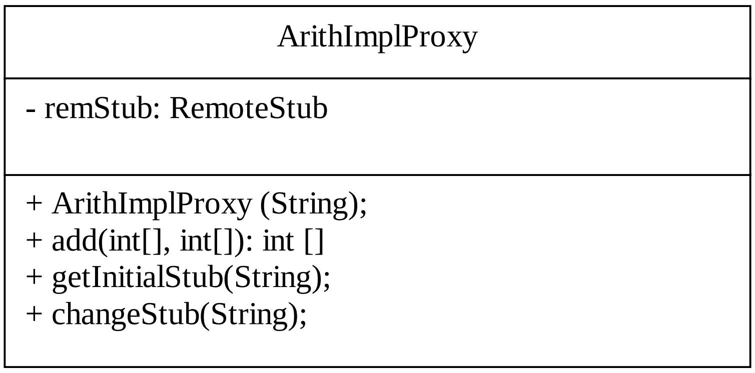

ArithImplProxy
- remStub: RemoteStub
+ ArithImplProxy (String);
+ add(int[], int[]): int []
+ getInitialStub(String);
+ changeStub(String);

Figure 5.5 Class diagram for ArithImplProxy.

GatewayRegistry

The GatewayRegistry object, resident on every mobility gateway, implements the MobilityRegistry interface. GatewayRegistry must also extend the UnicastRemoteObject class to allow remote method invocations to be made on it. GatewayRegistry maintains an array of all Carrier objects registered at that gateway. When a gateway invokes the handoff() method on the previous GatewayRegistry it supplies a String representation of the actual location of the servers' stub in the rmiregistry on the mobile host (e.g. rmi://foo/ArithServer if the mobile host is called foo and the mobile server object has been bound in the rmiregistry with the name 'ArithServer'). Since this address can only refer to a single server object it can be used to uniquely identify the corresponding Carrier object for the server. The GatewayRegistry uses it to locate the required Carrier object from those stored in the array.

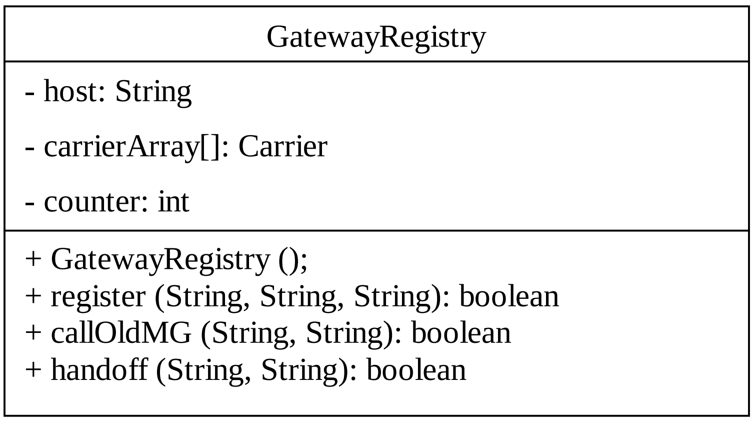

GatewayRegistry
- host: String
- carrierArray[]: Carrier
- counter: int
+ GatewayRegistry ();
+ register (String, String, String): boolean
+ callOldMG (String, String): boolean
+ handoff (String, String): boolean

Figure 5.6 GatewayRegistry class diagram

Carrier Class

The class that implements the Carrier interface in this example is called ArithCarrier. It also implements the Runnable interface so that when the class is downloaded by the GatewayRegistry object from the mobile host and instantiated, the run () method can be called and the work of creating the ArithImplProxy object and registering it with the local rmiregistry will be done inside the run () method. Also performed inside the run () method is the downloading of the Stub class for the ArithImplProxy from the mobile host. This Stub is required for the registration of the ArithImplProxy object and is also required for handing to clients when they contact the rmiregistry on the gateway. The server interface (Arith in this case) is also required and so is downloaded as well.

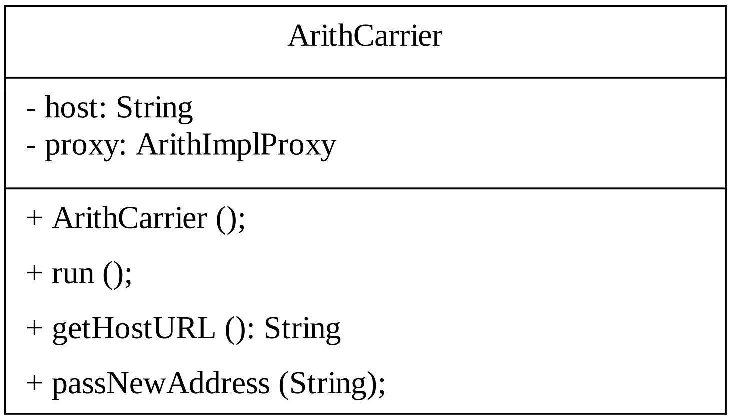


Figure 5.7 ArithCarrier class diagram

5.3.3 Handoff

The interaction diagram in Figure 5.8 below illustrates the ordering of the method calls that are made when the mobile server initiates the handoff procedure by registering with the new gateway and calling callOldMG() on its GatewayRegistry object. In the diagram ArithImpl is the server object on the mobile host, GR2 is the GatewayRegistry object on the new gateway, GR1 is the corresponding object on the old gateway, ArithCarrier is the relevant carrier object on the old gateway, and ArithImplProxy is the relevant proxy object on the old gateway.

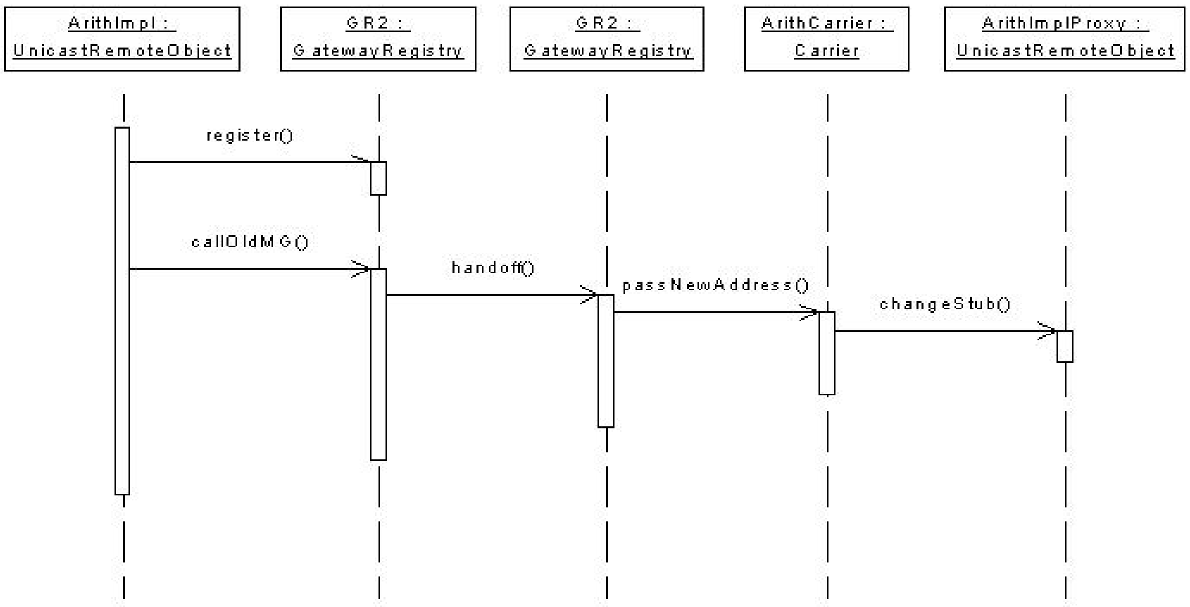


Figure 5.8 Methods and objects invoked during handoff

As handoff has not been implemented in the Mobility Layer yet the callOldMG() method invocation has to be artificially triggered. When handoff is implemented at the lower levels then an upcall to the application level should make this call instead.

Chapter 6

Evaluation

This chapter briefly examines the performance of the Java Mobility Layer and RMI Proxy scheme implemented in this project and compares this to the performance of normal RMI with no mobility support. The tests involved provided only provisional and rough indications of the performance of the system as the lack of fully functional, properly integrated mobility gateway software limited the validity of the results.

6.1 Code Size

Since the amount of memory on a mobile device is limited it is desirable that the Java Mobility classes take up as little space as possible. Figure 6.1 below details the size of the code for each group of classes used in a mobile RMI server. The code size given for the second and third rows in the table refer to the size of the class files for a server implementing the Arith interface specified in the previous chapter and its associated carrier and proxy classes.

Software Component	**Size of Code (kB)**
Java Mobility Layer Classes	60
Java RMI Server & Stub/Skeleton	11
Proxy, Carrier & Stub/Skeleton	16
Total	**87 kB**

Figure 6.1 Java Code Size for Required Classes on Mobile Host

The total of 87 kB for all classes required to act as a mobile RMI server is quite small compared to the memory available on most PDAs available today. For example, the Palm Pilot from Palm Inc. has 8 MB of memory while the basic model of Microsofts' H/PC comes with 16 MB.

6.2 Invocation Times

The purpose of these tests was to obtain approximate results for the performance of the Java Mobility Layer and RMI Proxy scheme. In all of the tests the mobile host used was laptop using the Windows 98 operating system and equipped with a WaveLAN wireless LAN card. The mobility gateway was a desktop PC using Solaris OS with a wired LAN connection.

Mobile Server Registration Time

The purpose of this test was to determine the average time taken for an RMI server running on a mobile host to register its Proxy object with a GatewayRegistry object on a mobility gateway. Essentially what happens during the registration process is that the server invokes a method on the GatewayRegistry object which then downloads the carrier and proxy classes from the mobile host, instantiates them and registers the proxy object with the rmiregistry on the gateway.

The gateway software used was the skeleton Java gateway implementation that allowed the mobile host to connect to the gateway using Mobility Layer sockets. Any sockets created on the gateway to connect to the mobile host (such as when the class files for the carrier and proxy classes are downloaded by the GatewayRegistry object) were ordinary sockets that did not have Mobility Layer support. This was due to the limited capabilities of the skeleton gateway. The GatewayRegistry downloaded a total of 15 KB of class files during the test, which was carried out a total of 10 times. The average time taken for the entire registration process was found to be 15 seconds.

Remote Method Invocation Using Mobility Layer

In this test the performance of the Java Mobility Layer is examined by having a client on the mobile host invoke a method on an RMI server object resident on a remote host. RMI is run on top of the ML connecting to the same skeleton gateway as before. The server and gateway were located on the same machine. The method was invoked 100 times and the average time for 1 method invocation was calculated. The test was conducted for different message sizes (i.e. the parameters passed in the method invocation were changed to give the desired message size). The results are shown in Figure 6.2 below along with the corresponding times for invocations made without using the ML.

Message Size (bytes)	Invocation Time with ML (millisecs)	Invocation Time without ML (millisecs)
8	1910	17
256	1930	21
384	1940	22
448	4730	42
512	4770	49
768	4810	49
1024	7600	51
1280	7730	52
1536	7820	56
1792	7850	60

Figure 6.2 Average method invocation times

As can be seen from the table above the data caching, multiplexing and other functions carried out by the Mobility Layer introduces much overhead to the process of invoking a remote method

Chapter 7

Conclusions

This chapter provides a brief description of the work completed during the course of the project. Remaining work on the implementation of the design is listed as well as possible future continuations of work started in this project.

7.1 Work Completed

The first major section of work completed was the completion of the implementation of the Java Mobility Layer. This required an in-depth knowledge of the ALICE architecture and the correct operation of the different software components of the Mobility Layer. A good understanding of the issues and techniques involved in distributed socket programming and multithreading in Java was reached as well as an appreciation for the issues involved in designing distributed systems capable of supporting object mobility.

In the second major part of the project a system of proxy objects to enable the mobility of RMI server objects was designed and implemented. This system was then tested and evaluated using the Java Mobile Layer and a basic Java implementation of the original C/C++ mobility gateway. The end result of the implementation showed that RMI server objects on remote hosts could change mobility gateway and still have methods invoked by remote hosts by relaying the invocation through a proxy object on the mobility gateway.

Although the design worked, the significant differences between the approaches taken to remote object locating and naming by RMI and CORBA meant that major changes had to be made to the ALICE architecture to allow the same mobility for RMI objects. The work previously done by the S/IIOP layer was instead performed by RMI objects. The work provided a good introduction to the intricacies of programming with a distributed object technology like RMI. Sample RMI applications were constructed for testing with the Mobility Layer and the proxy system. The benefits and disadvantages of using RMI as opposed to CORBA were also understood.

7.2 Remaining Work

The work remaining on this project can be summarised as follows –

- The Java Mobile Layer must be integrated with the C/C++ ALICE mobility gateway code. It is expected that this will not involve significant code changes.

- An automatic code generator, perhaps written as a Perl script, to produce the Proxy and Carrier classes given the specification of the server interface and the desired names of the two classes. This should allow the programmer to produce these classes by a simple command such as the rmic command that produces the RMI stub and skeleton classes.

- The code to register the Proxy object with the mobility gateway should be hidden behind the Naming.bind(…) method in the Java source code on the mobile host to make producing mobile RMI server objects easier for the programmer.

- A race condition occurs during the handoff procedure when the handoff() method is invoked on the GatewayRegistry object on the old gateway and a client connected to that gateway tries to invoke a method on the Proxy object. The Proxy should be locked as soon as the handoff() call is received so that the Stub can be changed and the call can be redirected to the new Proxy object on the new mobility gateway.

- Handoff remains to be implemented both in the Java Mobility Layer and in the C/C++ ALICE Mobility Layer.

7.3 Future Work

An interesting possibility for future work in this area is the integration of the RMI mobile proxy model with the Jini distributed computing infrastructure. Initial work on integrating the two carried out during this project showed that they worked well together and no major changes have to be made to the proxy system architecture. The biggest change is that the rmiregistry is replaced by a Jini lookup service, and the discovery and lookup processes are introduced. The Jini lookup service does provide a more powerful means for locating RMI objects than that currently provided in the proxy system by the rmiregistry. Further investigation of how Jini’s facilities for remote events, leasing of services and distributed transactions affect the system would also need to be carried out.

Bibliography

[Corbett ‘00]	Feasibility Study of the Implementation of ALICE in a Jini - based Environment, Mark Corbett, TCD Computer Science Final Year Project Report, May 2000
[Duchamp ‘92]	Issues in Wireless Mobile Computing, Dan Duchamp, Proceedings of Third Workshop on Workstation Operating Systems, IEEE, Key Biscayne Florida, April 1992
[Demers ‘94]	Dealing with Tentative Data Values in Disconnected Work Groups, M. M. Theimer, A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, and B. B. Welch, Proceedings of the Workshop on Mobile Computing Systems and Applications, Santa Cruz, California, December, pages 192-195, December 1994
[Edwards ‘99]	Core Jini, W. Keith Edwards, Prentice Hall, Upper Saddle River, NJ, USA, 1999
[Fiuczynski ‘94]	Programming Methodology for Disconnected Operation, Marc Fiuczynski, Univ. of Washington, ECOOP ’95, 1995
[Haahr ‘99]	Supporting CORBA Applications in a Mobile Environment, Mads Haahr, Raymond Cunningham and Vinny Cahill, MobiCom '99: 5th International Conference on Mobile Computing and Networking. Seattle, August 1999
[Heideman ‘92]	Primarily Disconnected Operation: Experiences With Ficus Heideman, Page, Guy, Popek Second Workshop On Management Of Replicated Data, IEEE November 1992.

[Huston ‘93] Disconnected Operation for AFS (Andrew File System), Huston, Honeyman, CITI, University of Michigan, CITI Technical Report 93-3, June 18, 1993

[Jini Spec ‘00] The Jini Specification, Ken Arnold, Bryan O’Sullivan, Robert W. Scheifler, Jim Waldo, Ann Wollrath, Prentice Hall, July 1999

[Johnson ‘96] Protocols For Adaptive Wireless and Mobile Networking, David B. Johnson and David A. Maltz, IEEE Personal Communications, 3(1):34-42, February 1996

[Joseph ‘97] Mobile Computing with the Rover Toolkit Anthony D. Joseph, Joshua A.Tauber, M. Frans Kaashoek. IEEE Trans. on Computers: Special issue on Mobile Computing, 46(3), March 1997

[Joseph ‘96] Building Reliable Mobile-Aware Applications with the Rover Toolkit, Anthony D. Joseph, M. Frans Kaashoek ACM Wireless Networks, 1996

[Kistler ‘93] Disconnected Operation in a Distributed File System, J.J. Kistler, PhD. Thesis, Dept. Of Computer Science, CMU May 1993

[Kleinrock ‘95] Nomadic Computing - An Opportunity, Leonard Kleinrock, ACM SIGCOM Computer Communication Review, pp. 36-40

[Mullender ‘93] Distributed File Systems (in "Distributed Systems"), Satyanarayanan M., Distributed Systems, 2nd Ed., S. Mullender ed., New York, NY, ACM Press, pp. 353-383 1993

[Page ‘98]	Primarily Disconnected Operation: Experiences With Ficus Heideman, Page, Guy, Popek, Second Workshop On Management Of Replicated Data, IEEE, November 1992
[Perkins ‘98]	Mobile Networking Through Mobile IP, Charles E. Perkins, IEEE Internet Computing, January 1998
[RMI Spec ‘99]	Java Remote Method Invocation Specification, Revision 1.7, Java 2 SDK, Standard Edition, v1.3.0, December 1999
[Saltzer ‘84]	End-to-End Arguments In System Design, J. H. Saltzer, D. P. Reed, and D. D. Clark, ACM Transactions on Computer Systems, pages 277-288, 1984.
[Satya ‘96]	Application-Aware Adaptation for Mobile Computing, M. Satyanarayanan et al, Proceedings of the 6th ACM SIGOPS September 1994
[Satya ‘96]	Mobile Information Access, Mahadev Satyanarayanan, IEEE Personal Communications, February 1996
[Satya ‘93]	Disconnected Operation in a Distributed File System, J.J. Kistler, PhD. Thesis, Dept. Of Computer Science, CMU May, 1993
[Skopp ‘93]	Disconnected Operation in a Multi-User Software Development Environment, Skopp, Kaiser (Columbia Univ.)
[Tannenbaum ‘96]	Computer Networks (3rd Edition), Andrew Tannenbaum, Prentice Hall, pp. 412-448

[Terry ‘94] The Bayou Architecture: Support for Data Sharing among Mobile Users, Demers, Petersen, Spreitzer, Terry, Theimer, Welch (PARC Xerox)

[Walker ‘83] The LOCUS Distributed Operating System, Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. Proceedings of the Ninth ACM Symposium on Operating Systems Principles, pages 49-70, Bretton Woods, New Hampshire, October 1983.

[Wollrath ‘96] A Distributed Object Model for the Java System, Ann Wollrath, Roger Riggs and Jim Waldo. Proceedings of the USENIX 1996 Conference on Object-Oriented Technologies (COOTS), Toronto, Canada, June 1996

[Venners ‘99] Inside the Java 2 Virtual Machine, Bill Venners, McGraw-Hill Professional Publishing, December 1999