Architecture for Location Independent CORBA Environments

Raymond Cunningham

B.A.

September 1998

A Dissertation submitted in partial fulfilment of the requirements for the Degree of MSc in Computer Science

University of Dublin
Trinity College Dublin

Declaration

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

_____

Raymond Cunningham

September 1998

Permission to Lend and/or Copy

I hereby declare that Trinity College may lend or copy this dissertation upon request.

_____

Raymond Cunningham

September 1998

Abstract

The construction of distributed applications is a complex and time consuming task, which has been addressed by the Object Management Group's Common Object Request Broker Architecture. However implementations of this standard are typically designed for stationary hosts connected to a fixed network and do not take into account the problems associated with mobile computing. These problems include limited processing resources on the mobile host and the use of unreliable and low-bandwidth wireless networks.

A full CORBA implementation is unsuitable for use on mobile hosts, such as laptops and personal digital assistants, since it is too resource intensive. It is however desirable for mobile hosts to be able to interopate with existing CORBA applications while simultaneously taking advantage of mobility.

This dissertation describes the design and implementation of a collection of components suitable for building applications, which are capable of interoperating with CORBA implementations and which are suitable for mobile hosts. The collection of components allows an application to act as a client or as a server in a CORBA context.

The implementation was carried out on Windows NT and Solaris using C and C++, Windows and Unix Sockets are used for network communication.

Acknowledgements

Many thanks to my supervisor, Dr. Vinny Cahill, for his guidance and advice throughout the course of this project.

Thanks to Mads Haahr, who was ever willing to help with any problems, however trivial. Thanks also to other members of the Distributed System Group; Tilman Schaefer, Jim Dowling, Frank Siquera, Stefan Weber and Tim Walsh.

Thanks to other members of the MSc course for their assistance and friendship throughout the year.

Special thanks to my family. To my three brothers for their support through thick and thin. To my Mother and Grandmother, without whose love and support, this project would never have been completed.

Chapter 1

Introduction

In recent years, two noticeable areas of growth in the computer industry have been the number of distributed applications being built using Object Request Brokers (ORBs) and the rapid growth in the area of mobile computing. These ORBs are implementations of the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [OMG'98]. The OMG has specified a protocol to allow ORBs to interwork called the Internet Inter-ORB Protocol (IIOP). This thesis describes an implementation of the IIOP protocol, which is suitable for mobile devices.

This chapter introduces the area of mobile computing and the OMG's recent standard to facilitate ORB interoperability. An analysis of the requirements to bring ORB interoperability into the mobile computing arena is, then, presented. Following this, the specific goals of this project are covered. An overview of the project is then presented as well as some of the project's achievements. Finally, a roadmap of the rest of the dissertation is presented.

1.1 Mobile Computing

Over the past decades, many transitions in the computer industry are clearly discernible. The transition from mainframes to minicomputers during the 1970's, then the subsequent move from minicomputers to personal computers during the 1980's and early 1990's are clearly discernible. A similar transition has recently begun and that is from the widespread use of personal computers to the widespread use of mobile computers. Mobile users typically employ a laptop computer or a personal digital assistant (PDA), perhaps with a mobile phone being used for network communication. Other equipment that is used to provide the communications mechanism includes Wireless LAN and Wireless ATM technologies.

The obvious advantage provided by mobile computing, is freedom of movement. Computer users will no longer be restricted to their office or their home in order to use applications that are essential to their work. Sales representatives will no longer need to return to their office to place an order for a company's product or make a phone call to place the order. Instead the sales representative will place orders electronically using his/her mobile device. This will also allow greater flexibility in people's work practices and working hours as they increasingly utilise the emerging mobile computing devices.

Along with the obvious advantages brought by mobile computing, there are obstacles that need to be overcome. These problems are faced by both the users of mobile devices and the developers, of applications, for these devices. To make the mobile device portable, the processing resources available on the device are usually somewhat limited in comparison to what can be found on a typical desktop machine today. This restricts the user of the mobile device in that only a limited number of applications are available and a smaller subset of functionality of these applications is possible. This restriction also applies to the application developer as the onus is upon him/her to maximise the use of the available resources.

Other disadvantages include the limited bandwidth available over the wireless communications media in comparison to typical Local Area Networks (LANs) (9600bps for wireless media versus 10/100Mbps for LANs). As well as the limited bandwidth, there is a significant difference in the error rates between, for example, a GSM phone and a LAN. Another difference to be aware of is the expense of using a GSM phone compared to the minimal cost of using a LAN. All of these problems impact on a developer as they begin development of new mobile computing applications.

As the solutions to these problems are more fully understood, more applications will quickly follow. Obvious applications are those that are currently common on today's desktop machines. Web browsers and E-mail reading programs are typical of the type of applications that are required for the mobile environment.

However, limiting the applications on mobile devices to those that are common on fixed networks does not capitalise on the advantages of mobile computing. Applications that are aware of their mobility are only beginning to be thought of. An example of such a mobile-aware application would be an in-car navigation system.

The applications, whether they are mobile-aware or not, will require a complex distributed system to be in place to enable mobile devices to be located, receive and send information as they move. One of the most widely used architectures for building distributed applications is the OMG's CORBA standard.

1.2 CORBA

The Common Object Request Broker Architecture (CORBA) from the Object Management Group (OMG) defines a framework for developing object oriented distributed applications. This framework makes network programming easier by allowing a distributed application to be built as though it were being implemented for a single computer. CORBA also brings object-oriented techniques to a distributed environment.

CORBA defines a standard architecture for Object Request Brokers (ORBs). An ORB allows the creation of software objects whose member functions can be invoked by client programs located anywhere in the network.

The CORBA standard initially made no provision for interoperability between ORBs, leaving it to ORB developers to create their own proprietary interoperability protocols for their own products. This was addressed by the OMG with the CORBA 2.0 standard. This defined a standard protocol, which allowed 2.0 compliant ORBs from different vendors to interoperate. This standard protocol is called the General Inter-ORB Protocol (GIOP) and a mapping of this onto TCP/IP is called the Internet Inter-ORB Protocol (IIOP).

The IIOP protocol makes a distinction between clients and servers using request/reply interactions. A client creates an IIOP request message to invoke the method of an object, which is stored at a particular server (typically supported by an ORB). IIOP also defines the format of reply messages that can be sent by the server in response to the request message. Other facilities that are possible with IIOP include querying for the location of an object and the cancellation of a previous request message.

With IIOP, applications using different ORB implementations now have the capacity to interoperate. These applications are typically developed for stationary hosts as the CORBA standard was not developed with mobile computing in mind. A naive solution to developing distributed applications in the mobile environment, would be to

port an ORB implementation to a mobile device. However, this is infeasible due to the complexity of the CORBA standard and the limited processing resources typically available on the mobile device.

Thus support for building mobile applications, which are able to interoperate with ORB implementations is required. This support would obviously have to be based around IIOP and would also require some form of mobility support. The mobility support is necessary since the wireless link in a mobile environment is not as reliable as its LAN counterpart. Therefore TCP/IP connections will typically be lost with greater frequency in a mobile environment.

1.3 Project Goal

The goal of this project was to produce a collection of software objects that allow computer programmers to build applications that are suitable for mobile devices and have the capacity to interoperate with existing ORB implementations.

These objects should be simple and intuitive to program with. They should hide the complexity of the IIOP protocol, which it supports to provide ORB interoperability. In addition it should attempt to hide broken TCP connections from the applications using it. This becomes important when the characteristics of the wireless communications medium are considered. Finally, the collection of objects should be useable on different operating systems architectures, in particular, it should be useable on both the Windows NT and Solaris operating systems. Although these objects do not provide any multithreading facilities, their implementations should be re-entrant and allow multithreaded applications to be built using them.

1.4 Overview of Design

The basis of the software objects identified above is to aid application developers when building applications, which need to interoperate with other CORBA objects. These software objects model a particular aspect of IIOP, request messages or the marshalling of data into and out of communication buffers for example.

To produce the required software objects, a sequence of research, testing, design and implementation was carried out. The research involved investigating approaches

taken in other mobile computing related research projects. In addition to this, a proprietary IIOP implementation, IONA Technologies' IIOP Engine, was used to gain experience of programming with IIOP. Other programming experience was gained in the area of inter-process communication and multithreading.

Following an analysis of the IIOP protocol and other issues relating to characteristics of mobile devices, a design to allow mobile devices to send and receive CORBA object invocations was produced. The design features three distinct components: the Mobile layer, the IIOP layer and the Swizzling IIOP (S/IIOP) layer.

The Mobile layer allows broken or lost TCP/IP connections to be hidden. The IIOP layer allows IIOP messages to be sent to and received from CORBA objects. Finally, the S/IIOP layer allows a mobile device to store CORBA objects and receive method invocations for these objects.

1.5 Project Achievements

A design for the three components introduced above was produced. The IIOP layer was implemented using C++ [Stroustrup'97] and C [Kernighan'88] on the Windows NT 4.0 operating system. Subsequent work was done to port the IIOP layer to the Solaris operating System. The two implementations were then tested with the proprietary implementation mentioned above. One part of the two part Mobile layer was implemented which allows clients on mobile devices to create connections on a LAN. This has yet to be tested.

1.6 Roadmap

The remaining chapters in this document detail the various phases of the project introduced above. The following is an outline of the particular chapters:

Chapter 2 State of the Art

This chapter presents the problems encountered in a typical mobile environment and introduces general techniques used to overcome them. Related research projects, which utilise these various techniques are then presented.

Chapter 3 CORBA and Interoperability

An introduction to the OMG's CORBA standard is presented, followed by a detailed discussion of how interoperability is achieved using the GIOP and IIOP protocol.

Chapter 4 Design

An analysis of the interoperability problem, as it relates to mobile environments, is given along with the design of the Mobile layer, IIOP layer and Swizzling layer.

Chapter 5 Implementation

This chapter describes the implementation of the various layers.

Chapter 6 Evaluation

An evaluation of the work carried out is given in this chapter as well as a comparison of the performance of the IIOP layer with another IIOP implementation.

Chapter 7 Conclusion

Details of further possible work are given along with some concluding remarks.

Appendix A IDL Definitions for GIOP and IIOP

The OMG IDL definitions of the GIOP and IIOP protocols are given.

Chapter 2

State of the Art

This chapter provides an overview of the main technical problems that are encountered in mobile computing environments, followed by a description of the most important techniques that have been used to address these problems.

This chapter includes a review of a number of influential research projects in which these techniques have been employed including: Rover, Bayou, Dolmen, Mobile IP and, finally OnTheMove.

2.1 Introduction

Figure 2.1 is a typical architecture allowing a mobile host to access services available on a fixed network and, in the opposite direction, clients on the fixed network to access services on a mobile host.

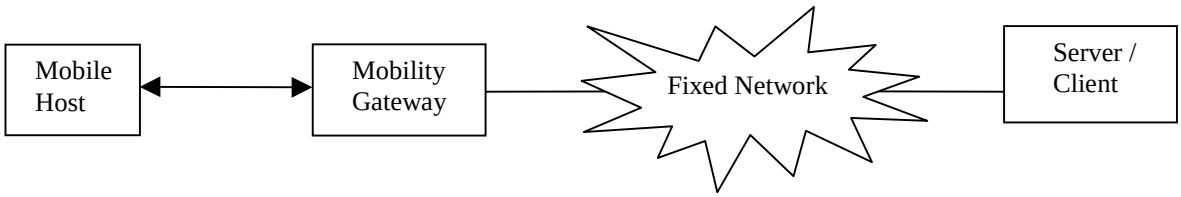


Figure 2.1 Mobile Host accessing and providing services

The mobile host (MH) is a laptop or personal digital assistant with one or more interfaces for communication across a wireless link (e.g. WaveLAN and/or GSM phone). The MH may also have an interface that allows it to connect directly to the fixed network. Supporting multiple interfaces on the MH introduces the problem of support for ***connection transparency***. For example, a user with a MH connected directly to the fixed network goes to a meeting within the same building. The MH might then switch to the Wireless LAN interface and later, as the user moves out of the building, to the GSM phone interface. This transition should ideally be done transparently to the user of the MH.

The Mobility Gateway (MG) is similar to a Base Station in mobile telephony and provides a routing facility for the mobile hosts that it is currently serving. The MG routes data from a MH to nodes on the fixed network and relays data from nodes on the fixed network destined for the MH. There can be many MGs attached to the fixed network and as a MH moves, it can be "handed over" from one MG to another. The cause of this "hand over" could be the MH going out of range of one MG or a superior quality wireless link to a different MG being established. This again introduces the problem of connection transparency as a user of the MH should not necessarily need to know that the MG has changed (unless, for example, a different tariff scheme is being used at that MG).

One problem in the mobile environment described above is the low bandwidth wireless link between the MH and the MG. Added to this is the fact that such a wireless link is typically more unreliable than a typical local area network and also usually more expensive to use. Moreover, the MH could be disconnected from the MG for intermittent periods. This again raises the issue of connection transparency, as the MH and MG should be able to continue operation despite a connection being lost and regained. All these problems suggest that transmission over this wireless link should be minimised as much as possible. Another issue is provision of support for disconnected operation to allow applications on the MH's to continue operation even when the MH is disconnected from the fixed network for a prolonged period of time.

To overcome the problems described above, three main techniques are used in object-support systems designed for use in mobile environments [Chen'97]. These are:

- Object Replication
- Object Migration
- Object Delegation

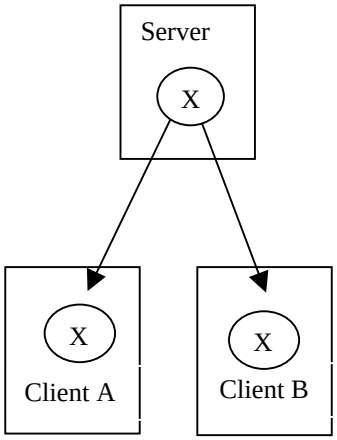
2.1.1 Object Replication

To increase the availability of various server resources on the fixed network, objects held by the server can be replicated and stored at multiple nodes on the fixed network or possibly at MHs. This technique is particularly useful in the event of network partitions (e.g. when an MH becomes disconnected from the fixed network) but it introduces the problem of consistency management. In essence, the server object is copied across the

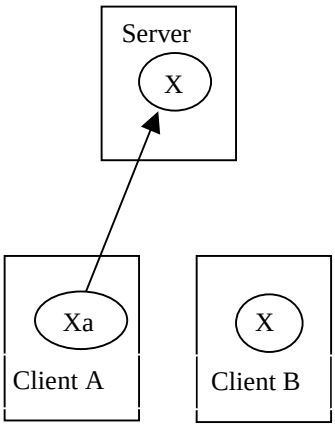
network to a secondary host. The secondary host must then create a context for the newly created object (known as a replica) so that it can be accessed in a similar way to the original object.

A strategy for the management of these multiple server objects must be chosen. Two possible choices are pessimistic replica control and optimistic replica control. In a pessimistic replica control scheme, such as read any/write one, a write lock has to be obtained by the object's user before the object can be modified. Such a scheme leads to low availability and is impractical in the mobile environment since a MH may obtain the write lock and then become disconnected from the fixed network preventing other users from modifying the object for prolonged periods.

In optimistic replica control schemes a read any/write any policy is used. Any object's user can read or modify any replica of the shared object at any time. It is clear that preserving absolute consistency with this strategy is not possible but it allows applications to be minimally interrupted when there are network partitions or temporary disruptions in the network connection. In this case, there is a possibility of read/write and write/write conflicts arising. These are illustrated in the figures below.

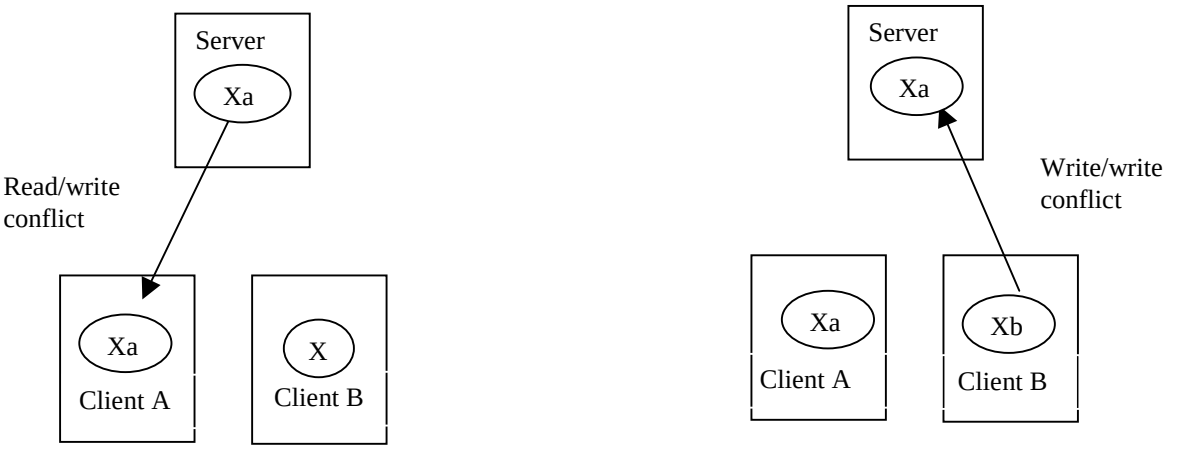


(a) Client A and client B both connected, make local replicas of object X



(b) Client A remains connected and client B becomes disconnected. Client A modifies X, yielding Xa at both client A and server, while X continues to be stored at client B.

Figure 2.2(a)



(a) Client B reads its local replica of X resulting in a read/write conflict.

(b) Client B modifies its local replica of X to produce Xb, reconnects, and attempts to write Xb to the server, producing a write/write conflict.

Figure 2.2(b)

Read/write conflicts are usually not detected immediately when they happen. This is because the client reading the object may not know that the version of the object that it is reading is not the most up to date. Read/write conflicts can be detected by a server, which maintains a list of all replica objects. If an object on the list is modified by some application while another application, on a MH, reads the object while it is disconnected, the application must be notified of the changes when the MH becomes connected again.

Write/write conflicts may easily be detected when multiple replicas are reconciled or merged back into one primary copy by the use of version vectors [Colouris'94]. The time taken for this clean-up stage may be short if consistency among the replicas is maintained during replication.

In summary, object replication may incur a high set-up cost, maintenance during usage to ensure consistency, and clean-up overhead.

2.1.2 Object Migration

Like object replication, object migration suffers from the same high set-up costs when the server object migrates from its original host to a secondary host. The uniqueness of the

server object eliminates the need to propagate modifications to multiple replicas. This technique is particularly useful for a MH when it is subject to disconnection from a MG and consequently the fixed network. This allows an application on a MH to continue operation during disconnection since the server objects it requires will have been pre-fetched by the MH and cached. This is also possible with object replication but object migration doesn't require replicas to be merged back into a single copy during a clean-up phase, as there is only ever one server object at any time.

2.1.3 Object Delegation

As already stated, object replication and object migration are expensive in set-up costs and thus may not be the ideal solution to provide access to a fixed network server's resources. Object delegation is when a request, from a client, is forwarded by an intermediate host to a remote host. The intermediate host contains a proxy object, which appears to the client to be the actual remote object. Object delegation introduces an extra level of indirection, which results in poorer performance. However, it may be used to allow a MH to act as a server to clients on the fixed network or other MHs. The proxy object in this instance would be placed at the MG to forward requests to the MH and return replies.

2.1.4 Connection Transparency and Disconnected Operation

In a mobile environment, a connection to a server may not be maintained through the lifetime of an application session or even of a single operation. MHs may move, causing a switch to another MG, to fulfill an application's requests. To the end-user, the application should appear to run uninterrupted, oblivious to the MH's connection status. To achieve such connection transparency, a session abstraction must be provided, which is capable of spanning several connections and disconnections in sequence. The particular details of reconnecting should remain hidden from the user as much as possible. In effect, the layers of software underlying the application must emulate the server during disconnected operation.

Pre-fetching:
Pre-fetching uses object migration to cache server objects at the MH in order to support disconnected operation. This requires that the application and/or underlying layers can predict which objects will be needed, depending on what tasks the user will perform during disconnection.

This introduces the problem of what level of knowledge the user will need to ensure that the correct server objects are cached. The user could explicitly pick the objects that will be needed. A more transparent approach would be to allow the user to merely specify which applications will be used during disconnection. The burden is then placed on the MH to automatically determine which objects a particular application will require.

Server Emulation and Reintegration
During disconnection, the goal of the system is to emulate the server as much as possible in order to permit continued execution of applications that invoke operations on objects held at servers. Operations originally performed on objects at the server are instead performed on replicated or migrated objects in the cache on the MH.

If an application needs an object but does not find it in the cache, the operation may be recorded in an operations log to be performed later upon reconnection. This may cause an application on the MH to become blocked waiting for the requested operation to complete, thus defeating the usefulness of disconnected operation.

When the MH becomes reconnected, any cached objects, which were modified, must be reconciled with the object held at the server.

2.2 Influential Research Projects

All of the above techniques have been employed in various research projects and in formulating proposals for the various Internet standards that address support for mobile computing. Several research projects, of which Rover and Bayou are among the most influential, have provided object replication and migration as well as supporting connection transparency. Mobile IP addresses the issue of routing data to mobile hosts within the context of a proposal for an Internet standard. The Dolmen project dealt

specifically with supporting mobility in a CORBA environment. This latter project built an Environment Specific Inter-ORB Protocol, which was very similar to the OMG's Internet Inter-ORB Protocol (IIOP). Each of these research projects will now be discussed.

2.2.1 Rover: A Toolkit for Mobile Information Access

The Rover [Joseph'97] toolkit provides mobile applications with a set of tools designed to isolate them from the limitations of mobile communications systems. The Rover Toolkit provides mobile communications support based on two ideas: Relocatable Dynamic Objects (RDOs) and Queued Remote Procedure Call (QRPC).

An RDO is an object with a well-defined interface that can be dynamically loaded into a client computer from a server computer (or vice versa) to reduce client-server communication requirements. QRPC is a communications abstraction that permits applications to continue to make non-blocking Remote Procedure Calls (RPCs) even when a host is disconnected, with client requests and server responses being exchanged upon network reconnection.

Rover client applications import RDOs into their address spaces from servers, invoke methods provided by the RDOs and export the RDOs back to the servers. Complex RDOs may create threads of control when they are imported. This is possible since RDOs are executed in a controlled environment. Rover objects are named by unique object identifiers, which are called Universal Resource Names (URNs).

Rover permits disconnected hosts to update shared objects. Object consistency is provided by application level locking or by using application-specific algorithms to resolve uncoordinated updates to a single object. In Rover, every object has a home server and update conflicts are detected at this server when two or more replicas are reconciled.

The toolkit supports several transport protocols including TCP/IP, HTTP, and SMTP. Using these protocols, an E-mail reader, a calendar program and a WWW browser were developed, as these were thought to represent the applications that mobile users are likely to require.

Rover Operation

Rover applications employ a check-in/check-out model of data sharing, they import objects into a local object cache, invoke methods provided by the objects and export the objects back to the servers as shown in Figure 2.3 on the following page.

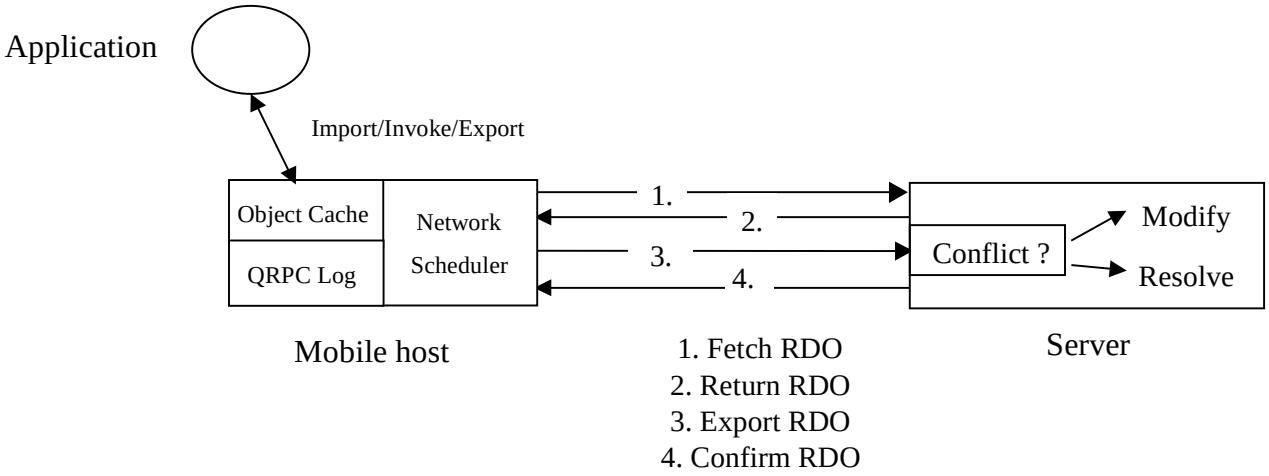


Fig 2.3 Rover Operation

Rover applications can choose the degree of consistency used for replicating objects. Rover caches objects on mobile hosts in a cache that is shared by all applications running on that host. Cached objects are secondary copies of objects, the exporting server maintains the primary copy.

When Rover invokes a method on an object, it first checks the object cache. If the object is resident in the cache, Rover modifies the object without contacting the server. The updates to the cached copy are marked as being tentatively committed.
If a required object is not present, Rover lazily fetches it from the server using QRPC. Rover stores a QRPC in a stable log at the mobile host and returns control to the application. The application can register a call back routine, which will be called by Rover to notify the application when the object has arrived.

Upon receipt of a fetch request, the server retrieves the requested object and sends it back to the mobile host. If the mobile host becomes disconnected between sending a request and receiving a reply, Rover will replay the request from its stable log upon

reconnection. Upon receiving a fetch reply Rover inserts the object returned into the cache and deletes the QRPC from the log. In addition if a callback routine is registered, Rover will perform the callback. The application can then invoke methods on the local copy.

When a method modifies a cached object, Rover lazily updates the primary copy at the server by sending the method call in a QRPC to the server and returns control to the application. When the QRPC arrives at the server, the server checks whether the object has changed since it was last imported by a mobile host. Rover maintains version vectors for each object so that methods can easily detect such changes. Upon arrival of a reply, the Rover toolkit on the MH changes the object from tentatively committed to committed. If a method call at the server detects a write/write conflict then the conflict is resolved in the application-specific manner. The Rover toolkit itself provides a mechanism for detecting conflicts but leaves it up to applications to reconcile them.

2.2.2 Xerox Parc Bayou Project

Bayou [Petersen'97] is a replicated, weakly-consistent storage system designed for a mobile computing environment. To maximise availability, users can read and write any accessible replica. Bayou's design has focused on supporting application-specific mechanisms to detect and resolve the update conflicts that naturally arise in such a system ensuring that replicas move towards eventual consistency. It introduces techniques for conflict detection called dependency checks and pre-write conflict resolution based on client provided merge procedures.

This requires that Bayou servers must be able to rollback the effects of previously executed writes and re-do them according to a global serialised order. Furthermore, Bayou permits clients to observe the results of all writes received by a server including tentative writes whose conflicts have not been ultimately resolved.

The Bayou design requires only occasional pair-wise communication between computers. This takes into account the characteristics of mobile computing such as expensive connection time, frequent or occasional disconnections, and the fact that collaborating computers may never be all connected simultaneously. Groups of computers may be partitioned from the rest of the system, yet remain connected to each

other. Supporting disconnected workgroups is a central goal of the Bayou system. By relying only on pair-wise communications in the normal mode of operation the Bayou design copes with arbitrary network connectivity.

The goal of Bayou was not to provide transparent replicated data support for existing file system and data base applications. Applications built using Bayou must be aware that they may read weakly consistent data and also that their write operations may conflict with other users and applications. Moreover, applications must be involved in the detection and resolution of conflicts since these naturally depend on the semantics of the application. Bayou allows applications to achieve automatic conflict detection and resolution using merge procedures.

Automatic conflict resolution is highly desirable because it enables a Bayou replica to remain available. Conflicts may be detected arbitrarily far from the users who introduced the conflicts. Moreover, conflicts may be detected when no user is present. Bayou allows clients to read data whose conflicts have not been fully resolved either because human intervention is needed or because other conflicting updates may be propagating through the system.

Bayou System Operation

In the Bayou system, each data collection is replicated in full at a number of servers. Applications running as clients interact with the servers using the Bayou Application Programming Interface (API), which is implemented as a client stub bound with the application. This API, as well as the underlying RPC protocol, supports two basic operations: read and write. Read operations permit queries over a data collection while write operations can insert, modify, and/or delete a number of data items in the collection. Figure 2.4, on the following page, illustrates the Bayou Architecture.

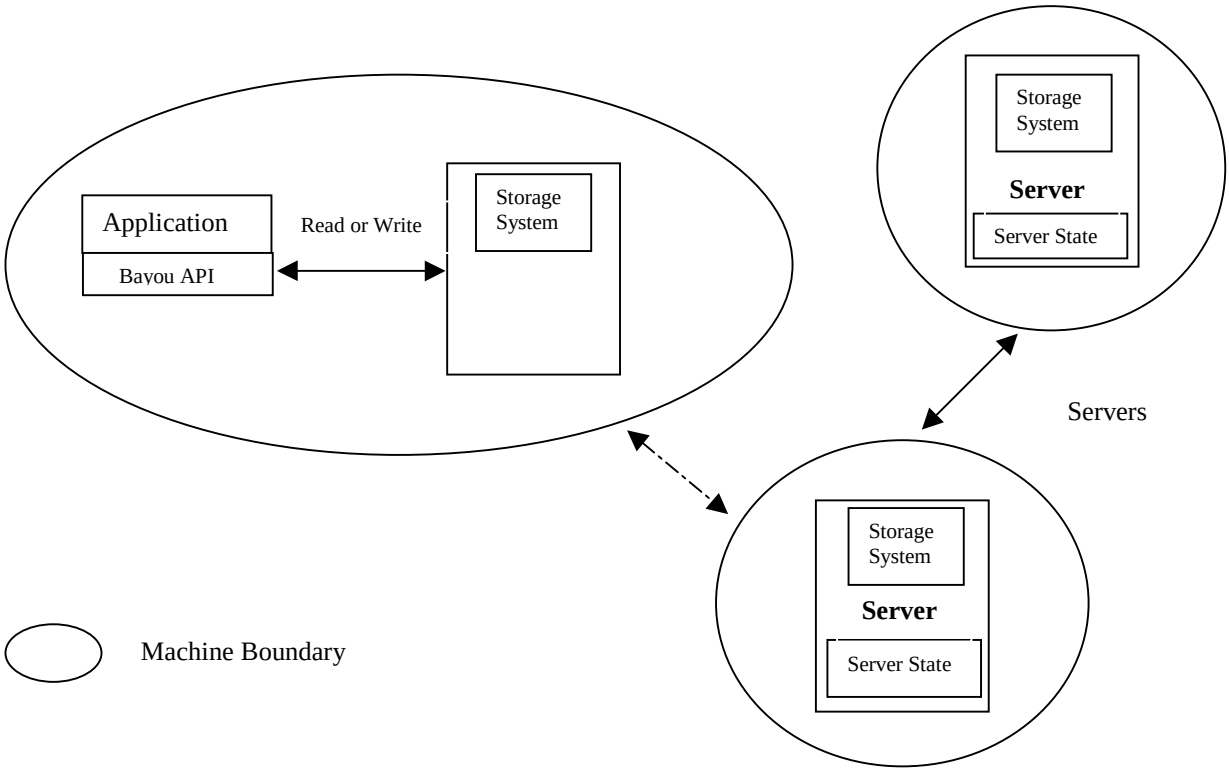


Figure 2.4 Bayou Architecture

Note that a client and a server may be co-resident on the same machine as would be typical of a laptop or PDA during disconnected operation. Access to one server is sufficient to perform useful work. The client can read the data held by that server and submit writes to the server. Having done so, the client has no further responsibility for that write. In particular, the client does not wait for the write to propagate to other servers.

To support application-specific conflict detection and resolution, Bayou writes contain more than a typical file system write or database update. Along with the desired updates a Bayou write carries information that lets each server receiving the write decide if there is a conflict and, if so, how to fix it. In particular, each Bayou write contains a globally unique write identifier assigned by the server that first accepted the write.

The storage system at each Bayou server conceptually consists of an ordered log of these writes plus the data resulting from the execution of the writes. Each server performs each write locally with conflicts detected and resolved as they are encountered during execution.

Unlike Rover which makes the effects of all known writes available for reading, Bayou servers propagate writes among themselves during pair-wise contacts called "anti-

entropy" sessions. The two servers involved in a session exchange write operations so that when they are finished they agree on the set of Bayou writes they have seen and the order in which to perform them.

Conflict Detection and Resolution

Supporting application-specific conflict detection and resolution is a major emphasis in the Bayou design. The Bayou system implements the mechanisms for reliably detecting conflicts while the application must specify its notion of a conflict along with its policy for resolving these conflicts. This design goal follows from the observation that different applications have different notions of what it means for two updates to conflict and that such conflicts cannot always be identified by simply observing reads and writes submitted by the applications. The Bayou system includes two mechanisms for automatic conflict detection and resolution that are intended to support arbitrary applications: dependency checks and merge procedures

Each write operation includes a dependency check, which consists of an application supplied query and its expected results. A conflict is detected if the query when run at a server against the current copy of the data does not return the expected result. If the check fails then the requested update is not performed and the server invokes a merge procedure to solve the detected conflict.

Once a conflict is detected, the server executes a merge procedure in an attempt to resolve the conflict. Bayou merge procedures are general programs written in SQL. They can contain data such as application-specific knowledge related to the update that was being attempted and can perform arbitrary reads on the current state of the server's replica.

The merge procedure associated with a write is responsible for resolving any conflicts detected by its dependency check and for producing a revised update to apply. The complete process of detecting a conflict running a merged procedure and applying the revised update is performed atomically at each server as part of executing a write. The potential drawback of this approach is that newly issued writes may depend on data that is in conflict and may lead to cascaded conflict resolution.

Replica Consistency

A fundamental property of the Bayou design is that all servers move toward eventual consistency. The Bayou system guarantees that all servers eventually receive all writes and that two servers holding the same set of writes will have the same data contents. However, it cannot enforce strict bounds on write propagation delays since these depend on network connectivity.

To ensure eventual consistency writes are performed in the same well defined order at all servers and dependency checks and merge procedures are deterministic.

2.2.3 Dolmen

The Dolmen [Raatikainen'97] project examined CORBA-based object communication in the context of wireless networks and host mobility. Dolmen used the interoperability mechanisms introduced in the CORBA 2.0 specification to support host mobility. The mechanism is completely transparent to client and server objects. Dolmen uses the concept of interoperability bridges, described in the CORBA 2.0 architecture. This is illustrated in the Figure 2.5 below.

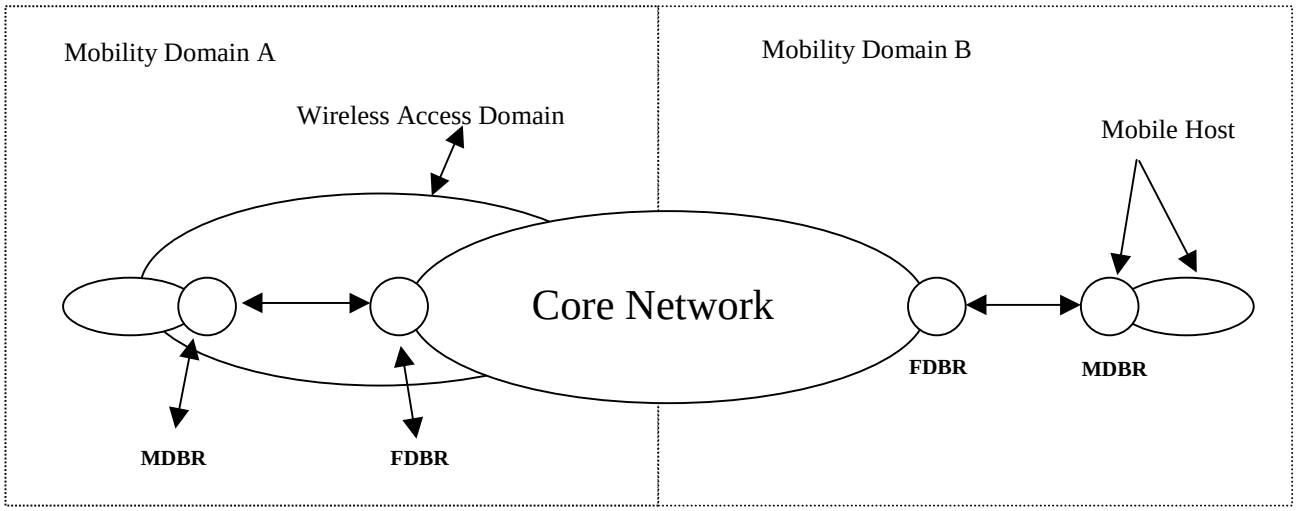


Fig 2.5 Dolmen Architecture

The Fixed Distributed Processing Environment Bridge (FDBR) serves as an access point for mobile hosts. A Mobile Distributed Processing Environment Bridge (MDBR) connects the local ORB of a mobile host to the fixed network by interacting with the

FDBR over the wireless link. Together the MDBR and FDBR perform location management functions and handover enabling host mobility. A Light Weight Inter-ORB Protocol (LW-IOP) between the FDBR and MDBR was defined to enhance reliability and performance of object communication in the mobile environment. The LW-IOP protocol defines efficient message formats and a compressed data representation for object communication.

The wireless access domain and part of the fixed network domain are divided into mobility domains. The fixed network part of each mobility domain instantiates a set of mobile specific support services including one or more FDBRs.

Each mobile host has its own ORB that provides object services to the applications running on the mobile host. Invocations of objects outside the local ORB are directed to the MDBR on the mobile host. The MDBR forwards the invocation to the FDBR, which then invokes the desired object. The FDBR acts as an endpoint for the mobile terminal within the fixed network. The FDBR also accepts invocation requests for objects located on the mobile terminal from objects within the fixed network. The FDBR forwards the request to the MDBR, which then invokes the actual object and returns the response through the FDBR.

Bridge Associations

A Dolmen bridge association models the logical relationship between a particular MDBR and FDBR. The bridge association is created whenever the mobile host appears within a new bridging domain and continues to exist for as long as there are objects using it for intercommunication.

As the mobile host moves across bridging domains, connectivity to the fixed network is maintained by a sequence of bridge associations. The most recent bridge association is called the primary association and it controls the physical signalling channel to the FDBR currently serving as the mobile terminal's access point to the fixed network. A trail of bridge associations called secondary associations are left as a mobile host moves. The secondary associations are required for processing object invocations that were in an unfinished state when the mobile host switched bridging domains. A

secondary association continues to exist for as long as objects within the fixed network store and use object references pointing to that particular FDBR.

Bridge Handover

Bridge handover in Dolmen occurs when a mobile host moves from one bridging domain to another. Bridge handover deals with maintaining and updating the bridge association. In Dolmen there are two types of handover, "backward bridge" handover and "forward bridge" handover. Backward bridge handover is the normal case when the mobile host is still able to communicate with the old FDBR. A forward bridge handover occurs when the mobile host unexpectedly looses its connection to the old FDBR and must re-establish communication with the fixed network. The old FDBR and MDBR synchronise their messages through the new FDBR.

A connection between the new FDBR and the old FDBR, called a tunnel, is set-up for all remaining unfinished invocations. The old FDBR forwards new object invocation requests to the new FDBR via the CORBA location forward mechanism.
Typically, tunnel connections are needed only for a short time. Moreover, tunnel connection chains are usually short. If a mobile host returns to an old FDBR to which it has a tunnelled association, the FDBR notices this and bypasses the tunnel loop.

Object Address Resolution

Each mobile host is assigned a Global Terminal Identifier (GTID), which uniquely identifies it. The basic problem in mobility is that, in order to enable objects in the fixed network to invoke objects in the mobile terminal, the location of the mobile terminal must be known at all times.

In particular, the bridge through which the mobile host can currently be accessed must always be known. The location register (LR) is a database that stores this knowledge within the fixed network. The address of the current primary bridge of the mobile host can always be retrieved from the LR if the GTID of the mobile host is known. The LR stores copies of the GTID and FDBR address and, during bridge handover, the new FDBR replaces the old FDBR's address for the particular GTID.
There are two cases of object address resolution that need to be taken into account. The simplest case is an invocation that originates from an object on a mobile host destined for

an object on the fixed network. The invocation request is intercepted by the MDBR and forwarded to the FDBR. The FDBR then invokes the object method operation directly. An object invocation that terminates at a mobile host is more difficult. The Dolmen bridging mechanism translates the reference of an object into a relocatable object reference (ROR). The ROR contains the current address of the FDBR serving the mobile host and the GTID of the mobile host. The FDBR address identifies the FDBR, which is currently serving the mobile host and the GTID identifies the mobile host to the FDBR. As the FDBR address in a reference may become out of date after a handover, FDBRs can receive invocations for objects located on mobile hosts that they are no longer bridging. When such an invocation arrives at a FDBR, the FDBR uses the GTID to query the LR for the current address of the primary FDBR serving the mobile host.

The secondary FDBR returns an up to date object reference to the client proxy on the fixed network using the CORBA location forward mechanism. The client proxy then proceeds to invoke the required operation at the primary FDBR using the returned object reference from the secondary FDBR.

2.2.4 Mobile IP

Mobile IP [Perkins'96] is a proposed standard protocol that builds on the Internet Protocol (IP) with the goal of making mobility transparent to applications and high level protocols such as TCP and UDP. The most fundamental obstacle that mobile IP is designed to address is the way that IP routes IP packets to their destinations based on IP addresses. These addresses are associated with a fixed network location. As a mobile host moves, it attaches at new points to the fixed network. This means that each new point of attachment is associated with a new network number and hence a new IP address. This makes mobile transparency impossible at the IP level.

However Mobile IP solves this problem by allowing the mobile host to use two IP addresses: a fixed home address and a transient "care-of" address that changes with each point of attachment to the fixed network.

Mobile IP Operation

IP routes packets from a source to a destination by allowing routers to forward packets from incoming network interfaces to outbound network interfaces according to routing tables. The routing tables typically maintain the next hop (outbound interface) information for each destination address according to the number of networks to which that IP address is connected. To maintain existing transport-layer connections the mobile host must keep the same IP address as it moves from place to place. On the other hand correct delivery of packets to a mobile host's current point of attachment depends on the IP address at its current point of attachment. In Mobile IP, two addresses are used, the home address is used to identify TCP or UDP connections. The care-of address changes at each new point of attachment. The home address makes it appear that the mobile host is continually able to receive data on its home network, at which Mobile IP requires the existence of a network node called the home agent. A foreign agent also exists for each care of address, its purpose is to route data to the mobile host using the care-of-address. Whenever the mobile host is not attached to its home network, the home agent gets all the packets destined for the mobile host and arranges to deliver them to the mobile host's current point of attachment.

Whenever the mobile host moves, it registers its new care-of address with its home agent. To get a packet to a mobile host from its home network, the home agent delivers the packet from the home network to the care of address. This requires a redirection where the IP packet is placed in another IP packet with the destination address of the outer IP packet set to the care-of address. This allows intermediate routers to correctly route the IP packet to the care-of address. The care-of address extracts the inner IP packet and then sends it to the mobile host. This encapsulation of the original IP packet within another IP packet is called tunnelling, since the original packet bypasses the usual effects of IP routing.

The operation of Mobile IP is based on three separate mechanisms:

- Discovering the care-of address
- Registering the care-of address
- Tunnelling to the care-of address.

Discovering the care-of address

Mobile IP extends the standard protocol for Router Advertisement as specified in RFC 1256. It simply extends the original fields to provide mobility functions. A router advertisement typically carries information about default routers. It can also carry further information about one or more care-of addresses. When a router advertisement contains the additional care-of address, it is known as an agent advertisement. Home agents and foreign agents typically broadcast agent advertisements at regular intervals.

If a mobile host needs to get a care-of address and does not wish to wait, the mobile host can broadcast or multicast a solicitation that will be answered by any foreign agent or home agent that receives it. Home agents use agent advertisements to make themselves known, even if they don't offer any care-of addresses. Once a mobile host has a care-of address it must register it with its home agent. If advertisements are no longer detectable from a foreign agent, that had previously offered a care-of address, the mobile host presumes that the foreign agent has gone out of range and it begins to "hunt" for a new care-of address.

Registering the Care-of-address

Once a mobile host has a care-of address from an agent advertisement, it notifies its home agent about it. The mobile host, possibly with the assistance of a foreign agent, sends a registration request with the care-of address information.

When the home agent receives this request, it typically adds the necessary information to its routing table and sends a registration reply back to the mobile host. Registration requests contain parameters and flags that characterise the tunnel through which the home agent will deliver packets to the care-of address. When a home agent accepts a request it begins to associate the home address of the mobile host with the care-of address until the registration lifetime expires.

The home agent must be certain that the registration request originated from the mobile host and not some other malicious host pretending to be the mobile host. A malicious host could cause the home agent to alter its routing table with erroneous care-of

address information, which would cause the mobile host to be unreachable to all incoming communications from the Internet. To overcome this problem, Mobile IP uses the Message Digest Algorithm, which provides unforgeable digital signatures.

When a mobile host cannot contact its home agent, it can use automatic home agent discovery. This method works by using a broadcast IP address instead of the home agent's IP address as the target for the registration request. Other home agents on the home network send a rejection in reply to this but include their own IP address for the mobile host to use in a freshly attempted registration message. The broadcast in this case is a directed broadcast that only reaches hosts on the home network.

Tunnelling to the Care-of Address

Using IP within IP, the home agent inserts a new IP header, or tunnel header, in front of the IP header of any datagram addressed to the mobile host's home address. The new tunnel header uses the mobile hosts care-of address as the destination IP address.
To deliver the original packet, the foreign agent must eliminate the tunnel header and deliver the rest to the mobile host.

2.2.5 OnTheMove

The objective of the OnTheMove project was to design a Mobile Applications Support Environment (MASE) [Meyer'96]. The MASE was designed to support both mobile-aware and non-mobile-aware applications running on mobile hosts. Non-mobile-aware applications are applications, which typically run on stationary computers and don't require knowledge of the host's mobility to operate. Typical non-mobile-aware applications include Web browsers and E-mail reading programs. Mobile aware applications utilise the host's mobility in some fashion to execute. An example would be navigation systems, which are currently being used in some luxury car models.

The MASE provides a layered architecture and is illustrated in the figure below.

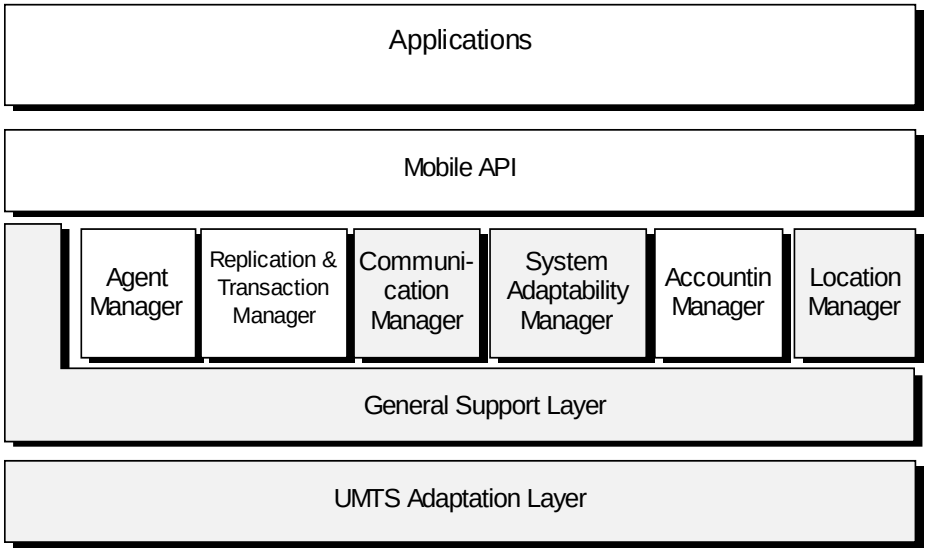


Fig 2.6. Layered Architecture of the MASE

As can be seen from the above figure, the MASE provides a large set of facilities that can be accessed through the Mobile API. These components will now be discussed.

The UMTS Adaption Layer. (UAL)

This layer provides a uniform interface to network specific details. The UAL is responsible for selecting the appropriate transport protocol and to configure it for efficient use. The UAL chooses the appropriate transport medium depending on various Quality of Service parameters if more than one transport medium is available. The UAL also monitors the protocol stack for changes in the Quality of Service provided and the network state.

The General Support Layer (GSL)

As its name suggests, the GSL provides common mobility support functionality for both applications as well as other MASE managers. The GSL is broken into four distinct components: the Mobile Object Manager, the Event Manager, the Security Manager and Directory Services.

The Mobile Object Manager (MOM) provides a storage facility for MASE mobile objects. These mobile object differ from the typical notion of objects (methods + data) since the mobile object comes from a set which include amongst others MIME objects,

Files and HTML links. The MOM provides pre-fetching, caching and accounting of these mobile objects.

The Event Manager (EM) API provides functions to monitor all activities in the MASE. Clients of the EM can register interest in certain events or states of the MASE. The outcome of the event can be specified as notification or log-file entry. Immediate response can be provided by notifications while log-file entry is used for post processing responses. Typical events of interest would be battery life or strength of signal from the transport medium.

The Security Manager exports an API, which provides typical security facilities such as authentication, non-repudiation and confidentiality. The Directory Services provides a uniform logical address representing each mobile device as well as providing a "Yellow pages" service.

Communication Manager (CM)

The CM component provides applications and other MASE managers with the facilities to communicate. This includes establishing, maintaining and terminating application communication. More importantly, it provides support for disconnected operation and connection transparency by protecting applications against disruption if a transport failure occurs.

Applications can specify various Quality of Service parameters using the CM's API depending on the information being sent or received (e.g. voice, text, video). The type of communication can also be specified whether it is synchronous or asynchronous. Communication exchanges can also be assigned a higher priority to ensure delivery as soon as communication has been restored. Other features include assured delivery of information.

The CM can be thought of as three components: Session Control, Disconnected Operation and Application independent protocols. The Session Control handles requests for communication from local and remote applications. The Disconnected operation component deals with problems related to unplanned disconnections such as radio signal loss. This requires pre-fetching of required data in conjunction with the MOM. Application independent protocols provide various types of communication such as

Messaging, E-mail, HTTP and Alerting. Each of these is provided as part of the CM's API. Messaging support is based on the IBM MQSeries and has the following functionality.

- Messages are delivered in order
- Messages can be linked together to form a single atomic transaction (using roll-back or commit operations)
- Message can be prioritised, ensuring delivery of higher priority messages first

The electronic mail API follows the X.400 standard closely. The HTTP API provides operations to build Web based applications such as web page retrieval. Finally, the Alerting API builds on the Messaging API to allow users and/or applications to be notified when an event occurs, such as reception of an e-mail message.

System Adaptability Manager (SAM)

The SAM maintains information about the current configuration and state of the mobile device. Examples of such configuration information would be a set of user preferences such as applications or files to be pre-fetched by the MOM. State information examples would be available bandwidth, battery life and strength of radio signal. The SAM also attempts to optimise bandwidth usage according to user preferences. This may involve changing the format of a mobile object to suit the characteristics of the mobile device (e.g. changing a colour image to black and white if a mobile host does not have a colour display).

MASE V2.0

The components, described above, were part of the initial version (1.0) of the MASE. The other components in Fig 9 are specified as part of the second version of the MASE [Kemp'96]. The Agent Manager provides an execution environment for mobile agents, which have the capability to migrate to or from a mobile device as necessary. The Replication and Transaction Manager (RTM) is in charge of replicating, managing and synchronising shared data objects. Local copies of the objects that are needed by the mobile host are stored in the mobile host's MOM, however it is the responsibility of the RTM to ensure consistency of the local copies of the objects with the primary copies on

the fixed network. The Location Manager (LM) was part of the first version of the MASE and was extended in the second version. The LM simply provides functions, which allow applications to retrieve information about a mobile host's current position. The Accounting Manager allows the building of electronic commerce applications on mobile hosts. This manager utilises the Secure Electronic Transaction (SET) standard to provide the payment mechanism.

Chapter 3

Background

This chapter provides a brief introduction to the Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA). In particular, this chapter describes the CORBA Interoperability standard, the General Inter ORB Protocol (GIOP) [GIOP'98]. A brief introduction to the CORBA standard is given first. This is followed by description of the goals of GIOP. The methods used to marshall GIOP messages are then described. Finally, the GIOP messages are described.

3.1 CORBA and Interoperability

The heterogeneity of modern data communications networks and computer systems make the task of network programming very difficult. Distributed applications often consist of several communicating programs, possibly running on different operating systems and written in different programming languages. Network programmers must consider all of these factors when developing applications.

CORBA defines a framework for developing object-oriented distributed applications. This framework makes network programming easier by allowing a distributed application to be built as though it were implemented in a single programming language to run on a single computer. CORBA also allows object-oriented techniques to be employed in distributed environments. This allows distributed applications to be designed as collections of co-operating objects as well as the re-use of existing objects.

3.1.1 Object Request Brokers

The central component of the CORBA architecture is the Object Request Broker or "ORB". An ORB allows the creation of software objects whose member functions can be invoked by client programs located anywhere in the network. In particular, using an ORB, the complexity of network communications is hidden from the applications developer(s).

When a client invokes a CORBA object's methods, the ORB intercepts the method call. It then redirects the method call across the network to the target object and eventually returns results of the method call (if any) to the client.

3.1.2 Interface Definition Language

Although CORBA objects are implemented using standard programming languages, each CORBA object has a clearly defined interface. This interface is specified in the CORBA Interface Definition Language (IDL) and primarily describes the methods provided by an object that are available to its clients.

Note that the interface definition makes no assumption about the implementation of the object. This allows the object implementation to be changed without needing to change the clients access to the object.

To use an object, a client need only know the IDL definition. It does not need to know details such as the programming language used to implement the object, the operating system on which it runs, or the host at which the object is located.

3.1.3 Interoperability

Several ORB implementations conforming to the CORBA standard are currently available from different vendors. The existence of multiple ORB implementations gave rise to the requirement for the OMG to define a protocol for communication between distinct ORBs in order to allow objects hosted by one vendor's ORB to interwork with objects hosted by an ORB supplied by a different vendor.

This requirement has been addressed by the OMG with the General Inter-ORB Protocol (GIOP). GIOP can be mapped onto any connection-oriented transport protocol. A mapping of GIOP that runs directly over TCP/IP, called the Internet Inter-ORB Protocol (IIOP), has also been specified by the OMG. IIOP must be supported by any ORB that claims to conform to the CORBA standard, regardless of other aspects of its implementation.

3.2 GIOP and IIOP

3.2.1 Goals of GIOP

GIOP and, in particular IIOP, are based on the most widely used and flexible communications transport mechanisms available (TCP/IP in the case of IIOP) and define the minimum additional protocol necessary to transfer invocations between ORBs. This should lead to the widest possible availability of GIOP and IIOP.

Moreover, GIOP is intended to be as simple as possible in order to ensure a variety of independent and compatible implementations. GIOP and IIOP should also be scalable and support ORBs to the size of today's Internet and beyond. Adding support for GIOP/IIOP to an existing ORB should also require a small engineering investment as the GIOP specification makes minimal assumptions about the architecture of the systems that support it.

3.2.2 Common Data Representation

The Common Data Representation (CDR) is a transfer syntax mapping OMG IDL data types into a low level representation for "on-the-wire" transfer between ORBs. CDR has the following features:

- Variable byte ordering - Machines with a common byte order may exchange messages without byte swapping. When communicating machines use different byte orders, the message originator determines the message byte order and the receiver is responsible for swapping bytes to match its native ordering. Each GIOP message contains a flag that indicates the appropriate byte order.
- Aligned primitive types - Primitive OMG IDL data types are aligned on their natural boundaries within GIOP messages, permitting data to be handled efficiently by architectures that enforce data alignment in memory.
- Complete OMG IDL Mapping - CDR describes representations for all OMG IDL data types.

3.2.3 CDR Transfer Syntax

The CDR transfer syntax is the format in which GIOP represents OMG IDL data types in the octet stream. An octet stream corresponds to a memory buffer that is to be sent to another process over some inter-process communication mechanism. It can be an arbitrarily long (but finite) sequence of eight-bit values with a well-defined beginning. Each octet can be logically indexed from 0 to n-1 where n is the length of the octet stream. The position of the octet in the stream is called its index. These indices are used to align OMG IDL data types in the octet stream.

GIOP defines two distinct kinds of octet streams, messages and encapsulations. Messages are the Protocol Data Units in GIOP. OMG IDL data types may be independently marshalled into encapsulation octet streams. In this case, the first octet contains a boolean value indicating the byte ordering of the encapsulated data.

3.2.4 Primitive Data Types

Primitive data types are specified for both big-endian and little-endian machines. To allow primitive data to be moved into and out of octet streams with instructions specifically designed for those data types, CDR requires that all primitive data types must be aligned on their natural boundaries. The alignment of a primitive data type is equal to the size of the data type in octets. A primitive data type of size n must start at an octet stream index that is a multiple of n. In CDR, n can be either 1, 2, 4, or 8.

When necessary, an alignment gap precedes the representation of a primitive data type. The values of octets in these alignment gaps are undefined. The following table gives the alignment boundaries for the OMG IDL primitive data types. Alignment is relative to the beginning of the octet stream.

Table 3.1 Alignment requirements for primitive data types

TYPE	OCTET ALIGNMENT
Char	1
Octet	1
Short	2
unsigned short	2
Long	4

unsigned long	4
long long	8
unsigned long long	8
Float	4
Double	8
long double	8
Boolean	1
Enum	4

The size and bit ordering in big-endian and little-endian encodings of primitive data types is well-defined. These primitive data types include short, long, long long, float, double and long double.

3.2.5 Constructed Data Types

Constructed data types are built from OMG IDL primitive data types using facilities defined by the OMG IDL language.

3.2.5.1 Struct

The elements of a Struct data structure are encoded in the order of their declaration. Each element is marshalled into an octet stream according to its data type.

3.2.5.2 Union

The discriminant tag of the Union is marshalled into an octet stream first followed by the selected member according to its data type.

3.2.5.3 Array

As the length of the array is known, each element of the array is marshalled into an octet stream as defined for their type. Multidimensional arrays are ordered so the index of the first dimension varies most slowly.

3.2.5.4 Sequence

An unsigned long value containing the number of elements in the sequence is marshalled in an octet stream followed by each element of the sequence according to its data type.

3.2.5.5 Enum

Enum values are encoded as unsigned long values. The first enum identifier has the numeric value zero. Successive identifiers take increasing numeric values.

3.2.5.6 String

A String is encoded as an unsigned long value specifying the length of the string in octets. Each element of the String is then marshalled into an octet stream as a char. Both the String length and contents include a terminating null character.

3.2.6 Other IDL Types

3.2.6.1 TypeCode

Type Codes are marshalled into an octet stream as a TCKind Enum value (unsigned long), potentially followed by values that represent the TypeCode parameters.

3.2.6.2 Principal

Principal types are marshalled into an octet stream as a Sequence of octets. They are used to identify the potential caller of an object's methods.

3.2.6.3 Exception

Exceptions are encoded as a String followed by Exception members. The String contains the Interface Repository Identifier for the Exception.

3.2.7 Interoperable Object References

An Interoperable Object Reference is encoded as an IDL Struct with two components: a String called the type identifier and a Sequence of Tagged Profiles. The type identifier is marshalled as a String into an octet stream first. The type identifier identifies the most derived type of the object at the time that the reference was published.

The Sequence of Tagged Profiles are then marshalled into the octet stream. A Tagged Profile is an IDL Struct consisting of an unsigned long value, which identifies the type of the Tagged Profile and a Sequence of octets containing the data in the Tagged Profile.

There are currently two types of Tagged Profile specified by the OMG: An IIOP Tagged Profile and a Multiple Component Tagged Profile. The IIOP Tagged Profile contains a hostname and port number to identify the process at which the object is stored and an object key to identify to the process the particular object that is being invoked.

A Multiple Component Tagged Profile is used to indicate ORB services that are being used A typical ORB service that uses this feature of IORs is the Transaction Service. Each Tagged Profile can also contain one or more Tagged Components, which are typically used to provide security as part of the CORBA Security Service.

3.2.8 Stringified IORs

To allow IORs to be passed between different ORB implementations, a stringified representation of an IOR is specified. A stringified IOR consists of a prefix followed by a sequence of hexadecimal digits. The prefix is the string "IOR:" The hexadecimal digits are obtained by converting the octet stream, into which an IOR was marshalled, using the following procedure. Each octet in the stream beginning at index zero is divided into two 4 bit values. This 4 bit value is then used to give the ASCII representation of the hexadecimal digit. The most significant four bits are converted first then the second 4 bits.

3.3 GIOP Messages

GIOP supports full CORBA functionality between ORBs with only seven GIOP messages. These seven messages allow an object implementation to be activated at different locations during its lifetime and, also, allow object migration. ORBs are not required to implement these mechanisms, but should of course implement the full IIOP protocol.

GIOP is designed to operate over any connection-oriented transport protocol. (Recall that IIOP is a mapping of GIOP onto TCP/IP). It is important to make a distinction with respect to the usage of connections for GIOP messages. The client side opens the connection to the object server and sends object invocations over the connection. The server side receives requests and returns replies. The server side may not

send object invocations over the connection to the client. This restriction allows certain race conditions to be avoided.

Multiple clients within an ORB may share a connection to send object invocations to another ORB or server. Multiple independent invocations for different objects or the same object may be sent on the same connection. GIOP also defines messages for cancellation of object invocations and for connection shutdown. These features allow ORBs to reuse or reclaim unused connections.

3.3.1 GIOP Message Header

All GIOP messages begin with the following header.

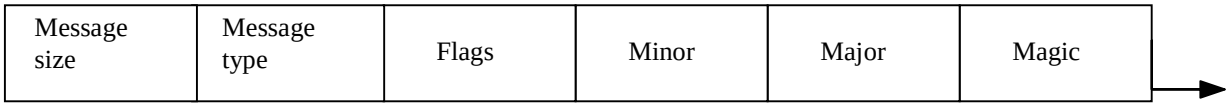


Fig 3.1 A GIOP Header

The GIOP Header identifies GIOP messages and their byte ordering (Big-Endian or Little-Endian). The Header is independent of byte ordering except for the field encoding the message size.

The magic field is four octets and always contains the upper case characters "GIOP". The major and minor octets identify the version of GIOP that is being used. For the current versions of GIOP, the major version number is 1 while the minor version number can be 0 or 1.

The flags field is an octet whose least significant bit indicates the byte ordering. The second least significant bit indicates whether, or not, more GIOP fragments follow. The most significant 6 bits are reserved for future use. The flags field is specific to GIOP version 1.1 and has replaced the byte order octet in GIOP version 1.0, which indicated the "endianness" of the remaining elements of the GIOP message.

The message type field is an octet and indicates the type of the GIOP message. It can be one of the following eight message types.

- Request (Version 1.0 & 1.1)
- Reply (Version 1.0 & 1.1)

- Cancel Request (Version 1.0 & 1.1)
- Locate Request (Version 1.0 & 1.1)
- Locate Reply (Version 1.0 & 1.1)
- Close Connection (Version 1.0 & 1.1)
- Message Error (Version 1.0 & 1.1)
- Fragment (Version 1.1)

The message size is an unsigned long encoded according to the preceding byte order indicator. It contains the number of octets in the message following the message header.

3.3.2 Request Message

Request messages encode CORBA object method invocations and are sent by a client to a server. They consist of three parts in the following order:

1. A GIOP Message Header
2. A Request Header
3. A Request Body

The Request Header has the following format

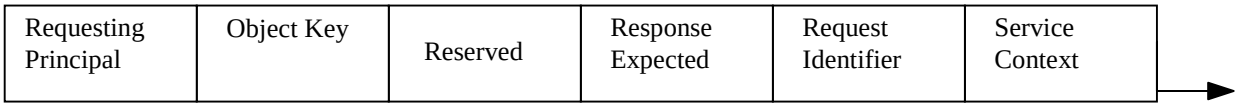


Fig 3.2 A GIOP Request Header

The Service Context field is an IDL defined Struct used to implicitly pass OMG Common Object Services specific information with a Request or Reply message.

The Request Identifier field is an unsigned long value used to associate Reply messages with Request messages. The client is responsible for generating values so that ambiguity is eliminated. The Response Expected field is set to true if a Reply message is expected for this Request. The value is false for a "oneway" invocation. The Reserved field consists of three octets that are always set to zero. It is specific to GIOP version 1.1 and was not part of version 1.0.

The object key is a sequence of octets and is used to identify the target object of the object invocation. This value is only meaningful to the server and is left unaltered by the client. Operation is a string containing the name of the object's method that is being invoked. The name identifies the method only within the scope of the object's IDL defined interface. Finally, the Requesting Principal is encoded as a string and identifies the client making the invocation.

3.3.3 Request Body

The Request Body includes all in and inout parameters, in the order in which they are specified in the operation's IDL definition from left to right. An optional Context pseudo object can be marshalled into the Request body after the in and inout parameters if the IDL interface definition includes a context expression.

For example, the Request body for the following IDL operation

```
double operation(in long m, out string str, inout short);
```

is equivalent to the following IDL Struct:

```
struct { long m ; short n ; };
```

3.3.4 Reply Message

Reply messages are sent in response to Request message if and only if the Response Expected flag in the Request is set to true. Replies are sent in three parts from server to client in the following order

1. GIOP Header
2. Reply Header
3. Reply Body

The Reply Header has the following format

Fig 3.3 A GIOP Reply Header

The Service Context field is similar to the Service Context field in the Request Header. The Request Identifier contains the same value as the Request Identifier in the

corresponding Request. The Reply Status field is an octet that indicates the completion status of the associated Request, it also determines the contents of the Reply Body.

If the Reply Status is No Exception, the invocation completed successfully and the Reply body contains return values. These are encoded as if they were an IDL Struct holding first any return value, then any inout and out parameters in the order in which they appear in the operation's IDL definition.

For example, the Request body for the following IDL operation

```
double operation(in long m, out string str, inout short n);
```

is equivalent to the following IDL Struct:

struct { double return_value; String str; short n; };

If the Reply Status is User or System Exception, then the Reply body contains an exception. If the Reply Status is Location Forward, then the Reply body contains a stringified IOR. The client must then resend the original Request to the new object location.

3.3.5 Cancel Request Message

Cancel Request Messages are sent from clients to servers to indicate to the server that the client is no longer expecting a Reply or Locate Reply for a Request or a Locate Request respectively.

The Cancel Request has two elements

1. A GIOP Header
2. A Cancel Request Header

The Cancel Request has the following format

Fig 3.4 A GIOP Cancel Request Header

The Request Identifier identifies the Request or Locate Request to which the Cancel Request applies. This value is the same as the Request Identifier in the original Request

or Locate Request message. The server is not required to acknowledge the cancellation and may subsequently send a Reply or Locate Reply.

3.3.6 Locate Request

Locate Request messages can be sent from a client to a server to determine if an IOR is valid, whether the server is capable of directly receiving Requests for the IOR and if not, to what location Request messages should be sent.

Locate Request messages are sent in two parts

1. A GIOP Header
2. A Locate Request Header

The format of the Locate Request Header is as follows:

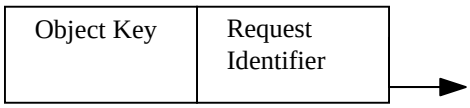


Fig 3.5 A GIOP Locate Request Header

The Request Identifier is used to associate Locate Reply messages with the corresponding Locate Request messages. The object key is a string and identifies the object that is being located.

3.3.7 Locate Reply Message

Locate Reply messages are sent from servers to clients in response to Locate Request message. A Locate Reply message has three parts encoded in the following order.

1. A GIOP Header
2. A Locate Reply Header
3. A Locate Reply Body

The Locate Reply header has the following format:

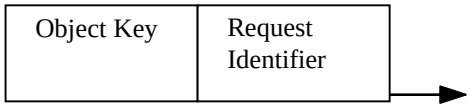


Fig 3.6 A GIOP Locate Reply Header

The Request Identifier associates Locate Replies with Locate Requests. The value is the same as the Request Identifier in the corresponding Locate Request. The Locate Status is an octet and determines whether a Locate Reply body exists. It can have one of three values

- Unknown Object - The object is unknown at this server (no body exists)
- Object Here - This server can directly receive Requests for the specified object (no body exists)
- Object Forward - A Locate Reply body exists

If the Locate Status is Object Forward, the Locate Reply body contains a stringified IOR that may be used for future Requests.

3.3.8 Close Connection Message

Close Connection messages are sent only by the server. Further Replies over this connection must not be expected after this message is received by clients. Clients can re-send any Requests, which had no corresponding Replies, on a new connection. The Close Connection message consists only of the GIOP header.

3.3.9 Message Error Message

This message is sent in response to any GIOP message whose magic or version number is incorrect or whose message type is unknown. This message consists only of the GIOP header.

3.3.10 Fragment Message

This message is added in GIOP 1.1 and is sent following a previous Request or Reply message that has the more fragments bit set in the flags field. The body of the Fragment message contains marshalled data.

Chapter 4

Design

This chapter begins with an overview of the design, introduced in Chapter 1, paying particular attention to the various layers of the design, namely the Mobile layer, IIOP layer and Swizzling layer, and their interaction. A brief comparison of the design with the related research covered in Chapter 2 is then made. This is followed by detailed descriptions of the Mobile layer, the IIOP layer and the S/IIOP layer.

4.1 Overview

In a mobile environment, there are many problems that need to be addressed. These include the use of unreliable and low bandwidth wireless link, the availability of limited resources on the Mobile Host (MH) and, possibly, the need to support multiple network interfaces on the MH (e.g., WaveLAN and GSM phone). To overcome the above problems, the design is divided between two machines, the Mobile Host and the Mobility Gateway, as illustrated in Figure 4.1.

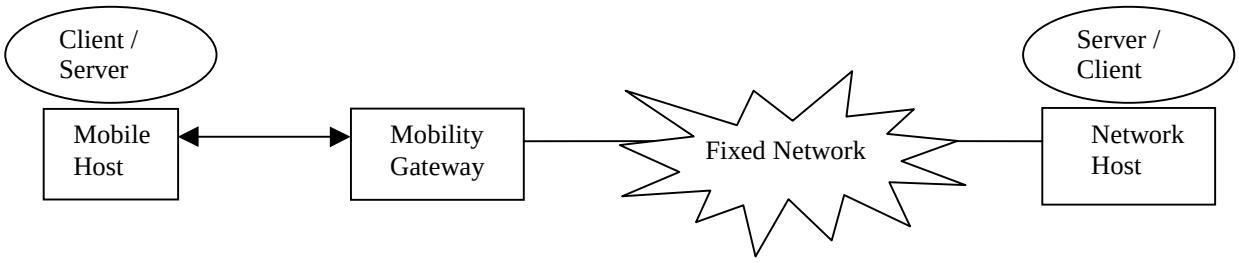


Figure 4.1

This division between the MH and the MG allows applications on the MH to communicate with applications on the fixed network. This is achieved as the MG acts as a bridge between the wireless network and the fixed wired network. A specific protocol between the MH and the MG can thus be used to overcome the problems of the unreliable and low bandwidth wireless link. This protocol can be used to optimise the use of multiple network interfaces, if present, on the MH and the MG.

The MH is typically a laptop or personal digital assistant, possibly with multiple interfaces. The MG may also have multiple network interfaces and is connected to a fixed network such as an Intranet or, more generally, the Internet. Typically, the server is located either on the fixed network, possibly at the MG or on the MH. The client application on the MH sends requests to the MG. The MG forwards these requests to the server. Replies from the server are returned through the MG to the client.

If the server is located on the MH, then requests originating from clients on the fixed network are routed to the MG. The MG then forwards these requests to the particular MH. Replies from the server are returned through the MG to the client.

Following from the description of the division in the design between the MH and the MG, the design takes a layered approach to provide IIOP functionality and to overcome the problems associated with the wireless link. This layered approach is illustrated in Figure 4.2 below.

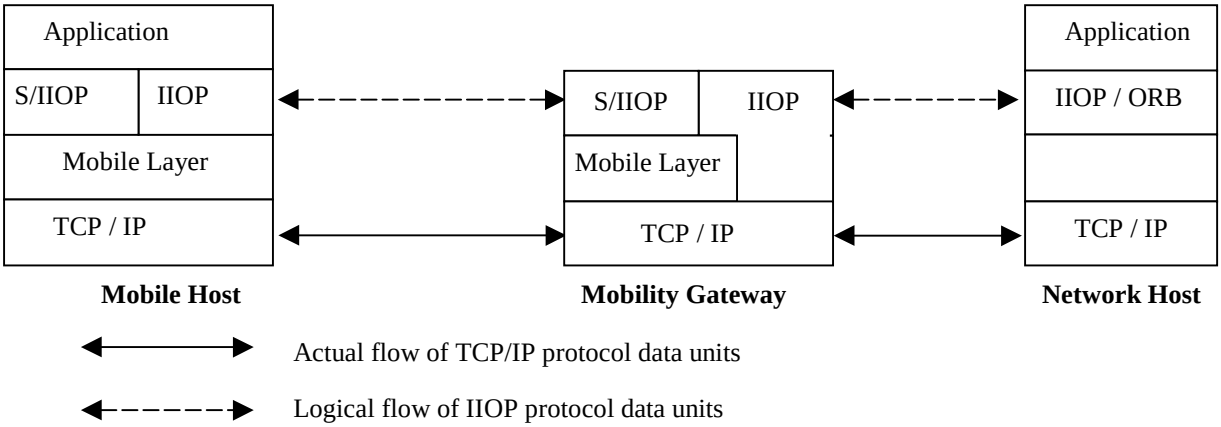


Figure 4.2 Layered Architecture

Typical ORBs are built on top of TCP/IP. However, TCP/IP connections are broken with a higher frequency in a mobile environment than in a fixed network environment. The Mobile layer hides lost TCP/IP connections from the layers above it by providing a ***logical connection*** abstraction. The Mobile layer provides mobility support, which can be plugged in and out as required. The IIOP layer allows the methods of an object located in a CORBA 2.0 compliant ORB to be invoked.

The Application on the MH uses an IIOP implementation to invoke a remote object's methods since a full implementation of a CORBA 2.0 compliant ORB would be

infeasible due to the limited processing resources of the MH. The Application on the fixed network could also use the IIOP layer but would more often have enough processing resources to run a complete ORB.

The S/IIOP layer works in tandem with the IIOP layer to allow a MH to act as a server to clients on the fixed network. The IIOP layer is unaware of the existence of the MG. When an IOR is created by the IIOP layer, the {address of MH,port #} pair is placed within the IOR. A client on the fixed network would be unable to use this IOR as it does not have direct access to the MH (i.e. the client typically does not have a wireless network interface).

It is the job of the S/IIOP layer to replace this {address of MH, port #} pair, substituting the pair {address of MG, port #} in its place. This allows a client on the fixed network to contact the MG and send data to the MG. The MG then forwards the data to the server on the MH.

It is conceivable that other swizzling layers could be used, for example with HTTP. In the HTTP case, there can be no guarantee that the port requested by the server on the MH, for example port 80, will be available on the MG since another application on this MH or another MH may have already requested this port.

The Mobile Layer on the MH and the Mobile Layer on the MG should provide the following capabilities:

- A sockets like API on the MH to allow network programs to be ported to the mobile device while still providing mobility support.
- The ability to hide broken TCP/IP connections from the layer above. This requires that any lost data is retransmitted across the wireless link and that data is received at most once by the layer above the Mobile layer on the MH and by the server on the fixed network.

4.2 Comparison to Related Research

This solution is similar to Dolmen's Fixed DPE Bridges (FDBRs) and Mobile DPE Bridges (MDBRs) with the MH as the MDBR and the Mobility Gateway (MG) as the FDBR. However, Dolmen relies on the underlying TCP/IP protocol stack to be configured for the wireless link being used. The layered approach taken in this design does provide connection transparency in contrast to Dolmen.

Dolmen does provide an Environment Specific Inter-ORB protocol called the Light Weight Inter-ORB Protocol (LW-IOP). This design provides an IIOP layer to enable interoperability with existing CORBA objects. A transformation is needed at the Dolmen FDBR to convert a LW-IOP message to the corresponding IIOP message and vice versa. No such transformation is necessary with the layered design given above.

The layered design also contrasts with Mobile IP, which provides mobility support at the IP layer. The MG acts as a bridge joining heterogeneous networks, in particular a wireless network and a fixed wired network. This bridging is done by routers at the network layer in Mobile IP, transparent to higher layers. However, this requires upgrading of router software in Mobile IP while no such router upgrades are necessary in this solution.

The Rover Toolkit provided Queued Remote Procedure Call, which required the application to make asynchronous calls to the Rover API. The Mobile layer provides a sockets like API, which most network programmers would be familiar with. As already stated, the Rover Toolkit requires asynchronous calls to the Toolkit to be made. This contrasts with this design since object invocations using IIOP are synchronous.

The Rover Toolkit also provides Relocatable Dynamic Objects (RDOs) that can migrate to and from a mobile device. No such support for object migration is provided in this design with the exception that the IIOP location forward mechanism can be used.

Bayou provides two mechanisms to help detect and resolve read/write and write/write conflicts in a mobile environment. These are dependency checks and merge procedures and have to be implemented by the application developer. This design provides at most once delivery of data to its destination, attempting to prevent conflicts from arising by using a logical connection abstraction to hide TCP/IP connections.

The Mobile layer of this design is similar to the Communication Manager in the on OnTheMove project. Similar to the CM, the Mobile layer provides facilities to establish, maintain and terminate application communication as well as providing for connection transparency.

4.3 Mobile Layer Operation

The Mobile layer provides a logical connection abstraction to allow lost or broken TCP/IP connections to be hidden from the layers above it. This abstraction requires data, originating at the MH, to be cached by the MH before sending. When the MH receives an acknowledgement of the data from the MG, the data can be removed from the cache. In a similar fashion any data sent from a network host to the MH is cached at the MG until acknowledged.

There are two cases to consider for the operation of the Mobile layer outlined above. The first and more simple case is when a client application, using the IIOP layer, is located on the mobile host. It sends requests to a server application on the fixed network and receives replies from the server application.

The second case is when a server application, using the IIOP layer, is located on the mobile host. Client applications on the fixed network or other mobile hosts send requests to the server application and receive replies. Each of these two cases will now be discussed.

4.3.1 Mobile Host as a client

When an application on the MH is a client of some service on the fixed network, there are four distinct stages to consider. These are

1. Connection establishment – These are the steps taken to create a new logical connection.
2. Data Transmission – The steps needed to transmit data associated with a logical connection
3. Connection Re-establishment – Re-establishing an existing logical connection after a TCP/IP connection has been lost.

4. Connection Shutdown – The steps involved in closing down a logical connection

Connection Establishment

During connection establishment, the IIOP layer calls the Mobile layer `socket()` function to create a socket. This call causes the Mobile layer to create a unique socket identifier and pass it back to the caller.

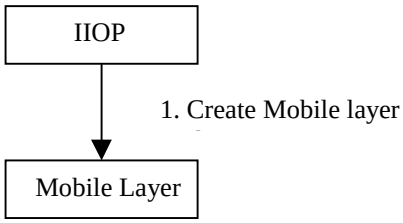


Figure 4.3

The IIOP Layer next calls the Mobile layer `connect()` function to establish a TCP/IP connection with the fixed network server (Step 2). This call causes the Mobile Layer to extract and cache the server address and port number, associating it with the socket parameter of the `connect()` function. The Mobile Layer delays opening an underlying TCP/IP connection to the MG until there is data to be sent to the server. This is done in an attempt to minimise the use of the wireless link since it is both unreliable and expensive in comparison to a fixed network. The `connect()` call returns indicating that the connection has been established.

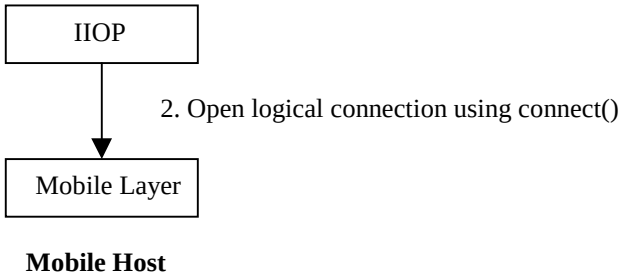


Figure 4.4

When the IIOP Layer sends or attempts to receive data for the first time using the `send()` or `recv()` functions over what appears to it to be a TCP/IP connection, the Mobile Layer on the MH sets up a *logical connection* to the Mobile Layer on the MG (Step 3). This is a two step process.

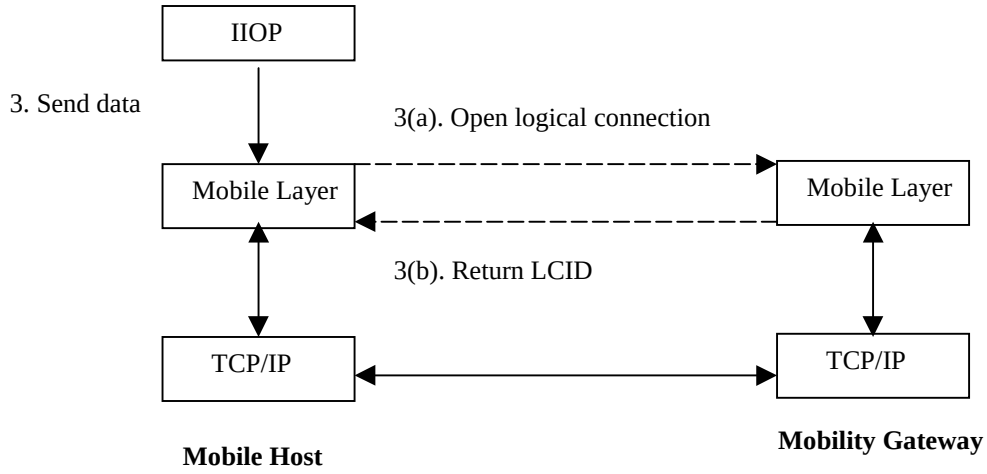


Figure 4.5

Note that the Mobile Layer on the MH has two options at this point. It could open a new TCP/IP connection to the MG for this *logical connection* or the Mobile layers on the MH and MG could multiplex any data, associated with the logical connection, onto an already established TCP/IP connection. If the Mobile Layer on the MH is using multiplexing and there are no TCP/IP connections already established, then the Mobile Layer would have to establish a new connection.

The second option was chosen since it would utilise the wireless link more efficiently, although making the implementation slightly more complicated. The first option would be inefficient when open TCP/IP connections are not being used.

The Mobile Layer on the MH sets up the *logical connection* to the Mobile Layer on the MG by passing the server address, port number and other relevant information (Step 3(a)). The server's port number and address were already cached (see Step 2). The {server address and port number} pair allow the MG to open a TCP/IP connection to the server on the fixed network at this or some future time.

The Mobile Layer on the MG assigns the server address, port number and other information a unique *logical connection identifier* (LCID) which it passes back to the

Mobile Layer on the MH (Step 3(b)). This LCID allows the MH to identify to the MG the TCP/IP connection between MG and Server when data is being transmitted.

Since the MG assigns the LCID, an unauthorised MH would have to guess this value to impersonate the MH and invoke a particular object's methods on a server. Once this logical connection has been established, data can then be transmitted.

Data Transmission

The Mobile Layer in the MH will assign a unique identifier to the data passed to it for transmission by the layer above it. The Mobile layer caches the data, identifier and LCID. The data and identifier are then sent to the Mobile Layer of the MG (Step 4). The LCID is included with the identifier and the data.

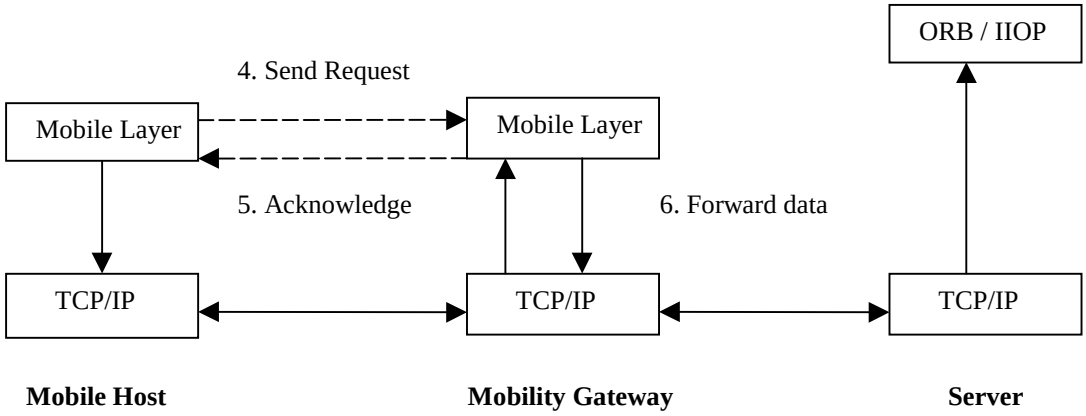


Figure 4.6

The Mobile Layer on the MG acknowledges the sent items (Step 5) and transmits the data on the TCP/IP connection to the server (Steps 6). This TCP/IP connection is established when the Mobile Layer on the MH sends data, associated with the LCID, to the MG for the first time. The Mobile Layer on the MG then retrieves the server address and port number associated with the LCID (see Step 3 above) to establish the TCP/IP connection. This data is sent using one of the socket functions `send(), sendto()` and `sendmsg()` to the server on the fixed network. When the sent items are acknowledged, the Mobile Layer in the MH removes the items from the cache.

In a similar fashion, when the Server sends data (Step 7), the Mobile Layer in the MG will assign it a unique identifier. The Mobile Layer on the MG caches the data along with the LCID.

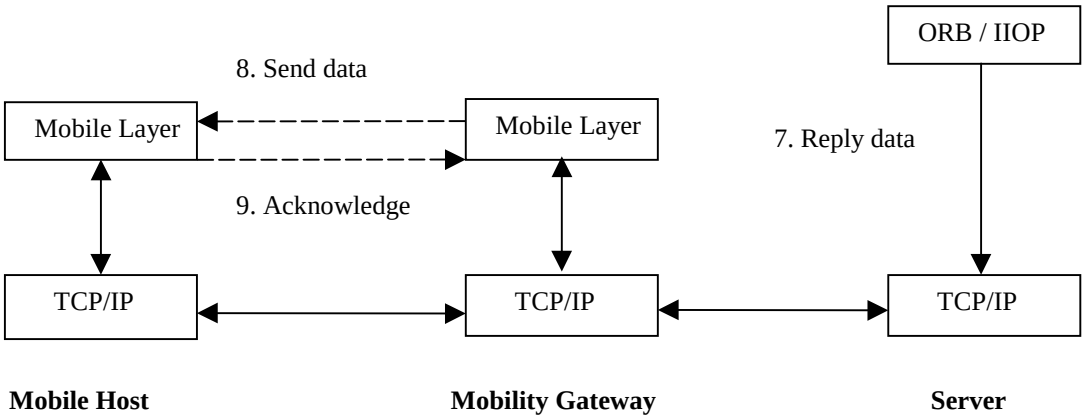


Figure 4.7

The Mobile Layer on the MG then sends the identifier and data to the Mobile Layer of the MH (Step 8). The Mobile Layer on the MH acknowledges the sent items (Step 9) and this causes the Mobile Layer on the MG to remove the previously cached items.

Connection Reestablishment

The Mobile layer, which has data to send, is responsible for re-establishing any lost TCP/IP connections between the MH and the MG. In this case, the Mobile layer sends a re-open logical connection request to its peer Mobile layer. Since multiplexing is being used, the LCID will be included only when data is being sent and is not needed when re-establishing the connection.

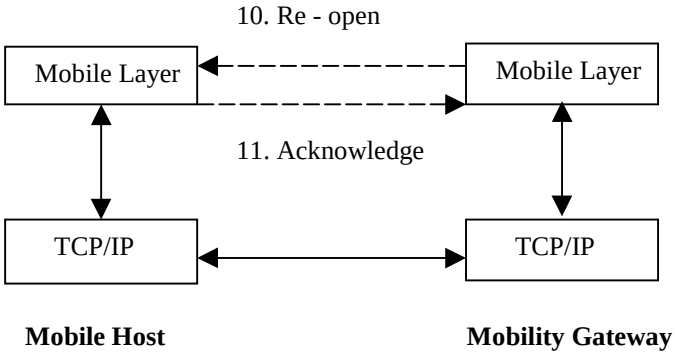


Figure 4.8

Note that both Mobile layers could conceivably have data to send after a TCP/IP connection has been lost. This race condition can easily be solved, by agreeing that the Mobile layer on the MH's Re-open logical connection is used with the Mobile layer on the MG's Re-open logical connection being ignored.

Any unacknowledged data that was sent over the lost connection is retransmitted over the new TCP/IP connection when more data arrives at either Mobile layer. If no more data arrives after a specific time, then the data can be retransmitted over the new TCP/IP connection between the Mobile layers. This helps to maximise throughput on the wireless link and minimise use of the expensive wireless link.

Connection Shutdown

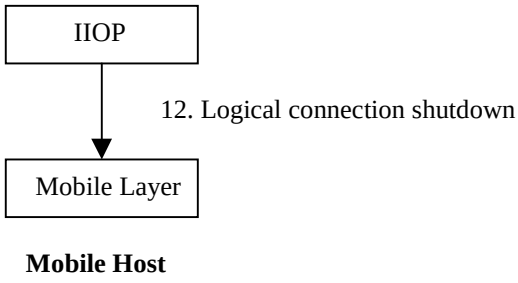


Figure 4.9

The IIOP layer calls the Mobile layer `shutdown()` function to close down a logical connection. The Mobile layer on the MH retransmits any unacknowledged data to the MG, until all the data is acknowledged by the MG. The Mobile layer on the MH then

sends a shutdown logical connection message to the MG. The MG removes all data associated with the logical connection and acknowledges the shutdown message. On receipt of the shutdown acknowledgement, the Mobile layer on the MH removes all data associated with the logical connection.

If the server on the fixed network closes down its connection to the MG, then the MG ensures that all data sent from the MG to the MH is acknowledged before sending a shutdown message. On receipt of a shutdown message, the Mobile layer on the MG removes all data associated with the logical connection and then sends a shutdown acknowledgement.

4.3.2 Mobile Host as a server

When a MH is a server to clients on the fixed network, there are two distinct stages to consider

- Accepting Connections – This allows clients connection attempts to be accepted by the server on the MH.
- Data Reception – These are the steps involved when receiving data from the client

Note that Connection Re-establishment and Connection shutdown are similar to the case where a MH is a client of services located on the fixed network.

Accepting Connections

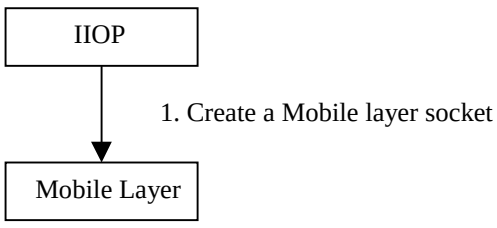


Figure 4.10

The IIOP layer calls the Mobile layer `socket()` function to establish an unbound stream socket. This call causes the Mobile layer to create a unique socket identifier and pass it back to the caller.

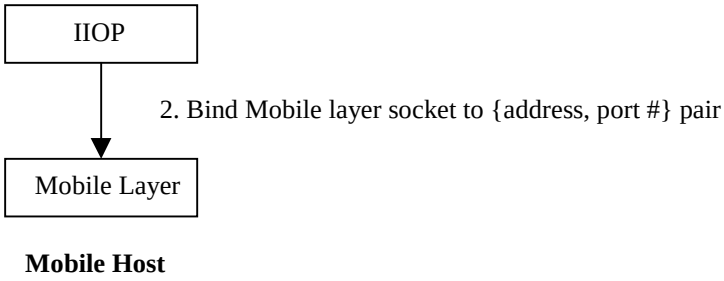


Figure 4.11

The IIOP layer calls the Mobile layer `bind()` function specifying a port number and an address to bind to a particular socket. The Mobile layer ensures that the {address, port #} pair is not already being used. The {address, port #} pair is cached by the Mobile layer and it is associated with the socket parameter passed to the bind function.

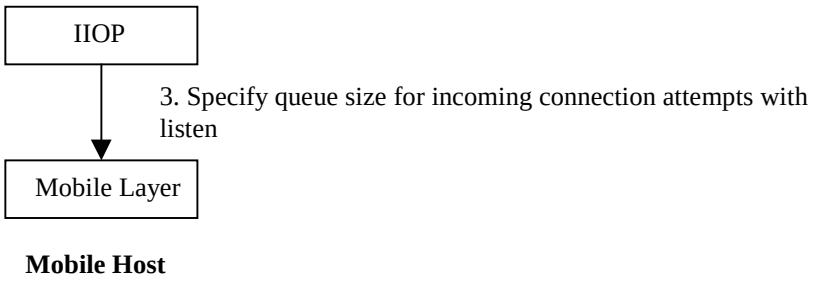


Figure 4.12

The IIOP layer calls the `listen()` socket function specifying a queue size for the number of incoming connection attempts. The Mobile layer caches the queue size parameter, associating it with the socket parameter passed to the listen function.

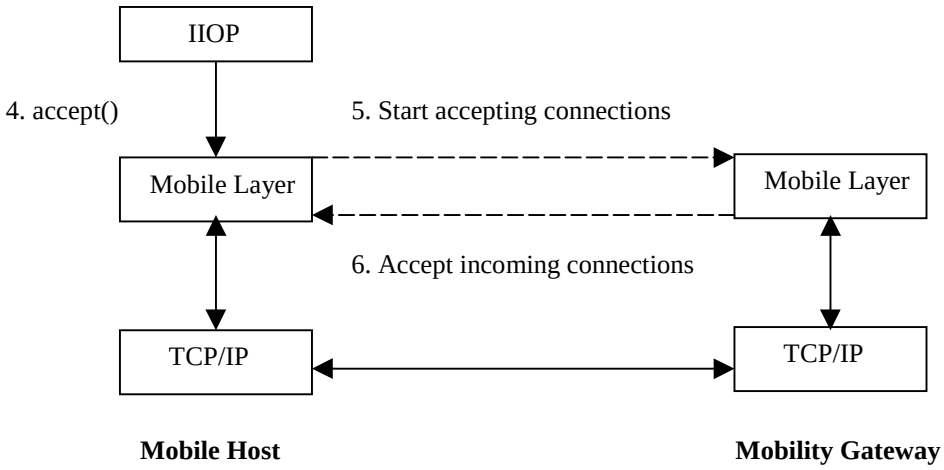


Figure 4.13

When the IIOP layer invokes the Mobile layer `accept()` function on the MH (Step 4), the Mobile layer on the Mobile host must start the Mobile layer on the Mobility Gateway accepting connections from clients on the fixed network (Step 5). This request to start accepting connections could specify the address and the port number to the Mobile layer on the MG. However, the Mobile layer on the MG returns an LCID, which identifies this {address, port #} pair. This allows the Mobile layer on the MG control over allocation of port numbers as well as solving the following problem.

Having the MG act as a proxy for server applications on mobile hosts introduces the problem of two or more MHs sharing the same MG and wishing to use the same port number. For example, if two server applications on two MHs wanted to operate as HTTP servers using the same default port, port 80, they would be unable to as there is only one such port on the MG. This is overcome in a CORBA context by using the S/IIOP layer to swizzle any IORs produced by the server application. The Swizzling layer is discussed later.

As multiplexing is being used, the Mobile layer on the MH must block the caller until a multiplexed connect request comes from the Mobile layer on the MG.

When a client on the fixed network attempts to set-up a connection with the server application on the MH (Step 7), it must possess the {address of MG, port #} pair. The Mobile layer on the MG relays the connection attempt to the Mobile layer on the MH

(Step 8). The Mobile layer on the MH acknowledges the connection attempt and un-blocks the `accept()` call (assuming there was one). If multiplexing is being used, the connection attempt between MG and MH includes the LCID already allocated (Step 6) along with a new LCID for the new connection between the MG and the client on the fixed network

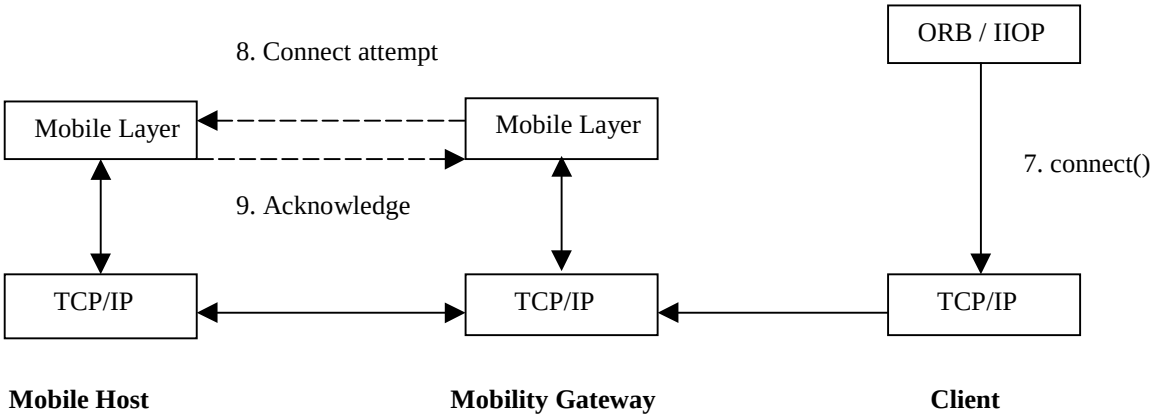


Figure 4.14

Data Reception

The IIOP layer calls the Mobile layer `recv()` function to receive data over the connection, which has previously been accepted.

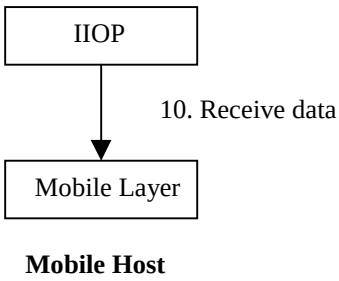


Figure 4.15

When a client on the fixed network sends data, it is cached by the Mobile layer on the MG and then sent to the Mobile layer on the MH (Step 8). The Mobile layer on the MH acknowledges the sent data (Step 9). The Mobile layer on the MG then removes the data from its cache.

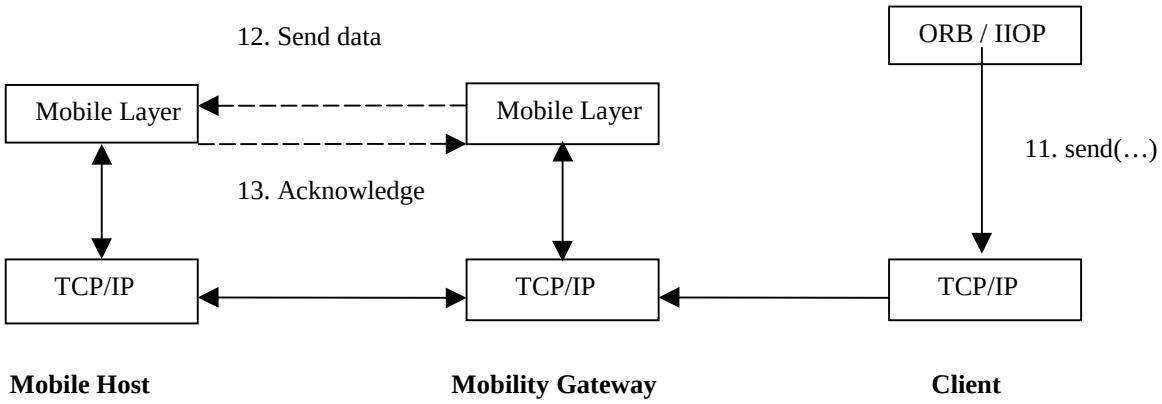


Figure 4.16

The Mobile layer on the MH un-blocks the previous `recv()` call. If there is no corresponding `recv()`, the Mobile Layer on the MH caches this data for a future `recv()` call. When the Mobile layer on the MH has data to send, it proceeds in a similar fashion to data transmission when the MH is a client as described earlier.

4.4 IIOP Layer

The IIOP layer provides software components that allow software developers to build applications, which can communicate using IIOP. This section will discuss the IIOP layer design goals and then describe the major software components used to achieve these design goals.

4.4.1 Design Goals

The first design consideration to be taken into account is that the IIOP layer should be as efficient as possible and have as small a footprint as possible. This is necessary since the implementation will be used to develop applications for mobile hosts, which have limited CPU speed and memory size.

Although the IIOP protocol only defines eight different message types, it must cater for the complexity of the CORBA Common Object Services. Another goal of the IIOP layer design was to hide the complexity of IIOP as much as possibly from the software developer. The software developer, using the IIOP layer should not need to

know every detail of the IIOP protocol. However, the IIOP layer should allow the software developer change various parameters to the IIOP protocol if necessary.

Finally, the IIOP layer should allow dynamic switching between the Mobile layer and the TCP/IP layer whenever needed. This is necessary so that mobility support can be plugged in and out whenever it is required.

4.4.2 Components of the IIOP layer

The IIOP layer implements the IIOP protocol providing an easy to use collection of objects to send and receive IIOP messages and to create IORs. The most important aspects of the IIOP layer design will now be discussed and how they address the issues raised above. For a more in depth description of the IIOP layer see Chapter 5.

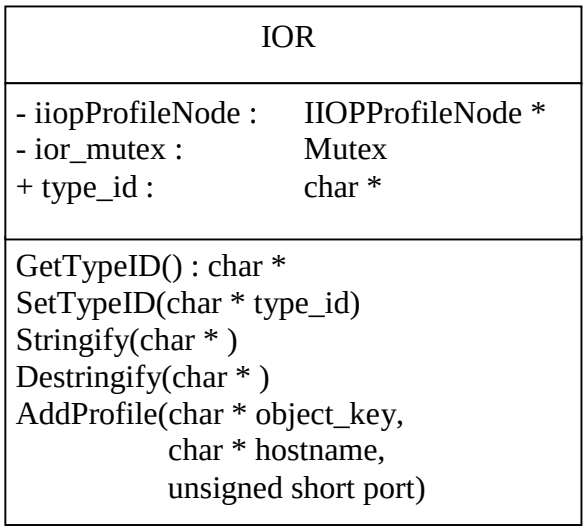


Figure 4.17 Class Diagram for IOR class

The IOR class, shown above represents an IIOP Interoperable Object Reference. The class definition allows for an IIOP IOR to be converted from its well-defined hexadecimal representation (see Chapter 3) into a more useable format using the Destringify(…) method and allows a new IIOP IOR to be constructed using AddProfile(…) and to be converted to its hexadecimal representation using the Stringify(…) method.

To allow an IOR to be swizzled, the Stringify(…) function would need to be changed to do the swizzling. How this swizzling is achieved is discussed in greater detail in the description of the Swizzling layer in the next section.

4.5 Swizzling Layer (S / IIOP)

This layer will work with the IIOP layer and the Mobile layer to allow a MH to act as a Server in a CORBA context. A Server produces Interoperable Object References (IORs) for objects stored at that Server. An IOR contains enough information (hostname, port number and object key) to allow clients to locate the object and consequently invoke its methods.

Clients on the fixed network are unable to directly contact a Server on the MH, they must go through the MG. Thus, the address information consisting of the {address of MH, port #} pair, in an IOR needs to be changed to allow clients, on the fixed network to invoke methods of an object stored in a MH. This change is done in the Swizzling layer.

There are two possible places that this change could occur. It could occur when the IOR is created or when it is stringified. This second option is better as an IOR can only be exported from the MH when it is in stringified form and not all IORs will be exported. This, possibly, reduces the number of IORs that the Swizzling layer will have to change.

The MG has a well-known proxy port on which the S/IIOP layer is listening. The S/IIOP layer on the MH knows the fixed network address and proxy port of the MG. It can then swizzle any IORs produced. This involves changing the {address of MH, port #} pair in an IOR to the {fixed network address of the MG, well-known proxy port} pair. After switching the address information contained in an IOR, the S/IIOP layer needs to prepend the {address of MH, port #} pair to the object key in the IOR. This allows the S/IIOP layer on the MG to identify the particular server application that is running on a particular MH. This is illustrated in table 4.1 on the following page.

	IOR CONTENTS
IOR before change	Address : address of MH Port : 1234 Object key : foobar
IOR after change	Address : Address of MG Port : proxy port number Object key : {Address of MH, 1234} : foobar

Table 4.1 IOR contents before and after swizzling

When a Server application is willing to accept connection attempts, the S/IIOP layer must override the Mobile layer accept(…) functionality since a Mobile layer accept call dynamically allocates ports on the MG (see section 4.3.2).

The S/IIOP layer on the MH will override this by setting up a connection to the S/IIOP layer on the MG to allow connection attempts from clients on the fixed network to be forwarded to the S/IIOP layer on the MH. The S/IIOP layer on the MH will then block until a connection attempt is received from the S/IIOP layer on the MG.

When a client receives a "swizzled" IOR and wishes to invoke the IOR's methods, it opens a TCP/IP connection to the MG, using the {fixed network interface address, allocated port number} pair described above. It sends an IIOP Request to the S/IIOP layer on the MG. The S/IIOP layer receives this Request.

The object key within the Request identifies the address of the MH and port number on which the server application is listening. The S/IIOP layer removes the {address of MH, port #} pair that was prepended onto the object key in the IOR when the IOR was stringified. The S/IIOP layer then replaces the {fixed network address of the MG, well-known proxy port} pair with the {address of MH, port #}.

Note that both the client on the fixed network and the server application on the MH are unaware that the S/IIOP layer on the MG is acting as a proxy when it forwards IIOP Requests and Replies. The client on the fixed network perceives that the object it is invoking is at the MG. The S/IIOP layer on the MG also has the capability to use the OBJECT_FORWARD capability in IIOP (see section 3.3.3) to allow object invocations

to be forwarded to other Mobility Gateways. This would be necessary when a MH is handed over from one Mobility Gateway to another and clients on the fixed network still maintain old swizzled IORs.

4.6 Summary

This chapter began with an overview of the design followed by a comparison of the design with the research projects covered in Chapter 2. The design is similar in certain aspects to these projects but also possesses features, which are unique to it. A detailed description of the Mobile layer and the S/IIOP layer was then given. The IIOP layer is described in detail in Chapter 5.

Chapter 5

Implementation

This chapter describes in detail the design and implementation of the IIOP layer. Recall from Chapter 4 that the Mobile layer implementation is near completion but has yet to be tested while the Swizzling layer has yet to be implemented. Both of these layers will not be discussed further.

The goals of the implementation are presented first. This is followed by a description of the implementation decisions, followed by a description of the various classes within the IIOP layer and their interaction.

5.1 Implementation Goals

The goal of the implementation was to produce a set of classes that allow applications to be built, which can interoperate with other applications using the OMG's IIOP protocol for communication. The set of classes should also be suitable for use on a mobile host that has limited CPU speed and a small amount of memory. This requires that the implementation be as efficient as possible and, in particular, avoids unnecessary copying of data.

In addition, the IIOP layer should be useable on both Windows NT and on Solaris. This allows IIOP enabled applications to be built for the Windows CE and/or the Palm Pilot operating systems. An application written using the IIOP layer on Windows NT should recompile on Solaris without the need to make changes to the application (assuming that the application does not use other operating system dependent functionality, such as Win32 API calls on Windows NT).

Although the IIOP layer does not make use of multiple threads, the implementation of the IIOP layer should be reentrant. This allows multithreaded applications to be built on either Windows NT or Solaris using the IIOP layer without causing synchronisation problems.

Finally, the IIOP layer should hide as much of the complexity of the IIOP protocol as possible, while still allowing the application developer access to the various parameters of the IIOP protocol.

5.2 Implementation Decisions

The IIOP layer was implemented initially on Windows NT using Visual C++. It was ported to Solaris using SparcWorks C++. The IIOP layer uses a `Mutex` class abstraction, which is implemented on Windows NT using the CriticalSection API [Pham'96] and on Solaris using the Solaris threads mutex API [Chan'97].

Network communication is achieved in the IIOP layer using Winsock sockets on Windows NT [Hall'93] and BSD sockets on Solaris [Stevens'90]. The `MobileEndpoint` class or `tcpEndpoint` class implements the network communication depending on whether mobility support is required or not. Both of these classes inherit from the Endpoint abstract base class. This makes it possible to dynamically switch between the Mobile layer and the TCP/IP layer.

5.3 IIOP Layer Classes

The IIOP layer classes encapsulate the IIOP protocol functionality. Each class implements a particular function in the IIOP protocol. This allows the application developer to instantiate various classes to implement all or part of the IIOP protocol without having to have an in-depth knowledge of the IIOP protocol. The rest of the chapter describes the various parts of the IIOP layer under the following headings:

- Overview of Class Hierarchy
- Representation of IORs
- Marshalling
- GIOP Message Representation
- Transport Classes
- Communication Endpoints

5.3.1 Overview of Class Hierarchy

As can be seen from the UML diagrams [Fowler'97] in figures 5.1a, 5.1b and 5.1c, the IIOP layer classes can be divided into the following subsets:

- Representation of IORs
- Marshalling
- GIOP Message Representation
- Transport Classes
- Communication Endpoints

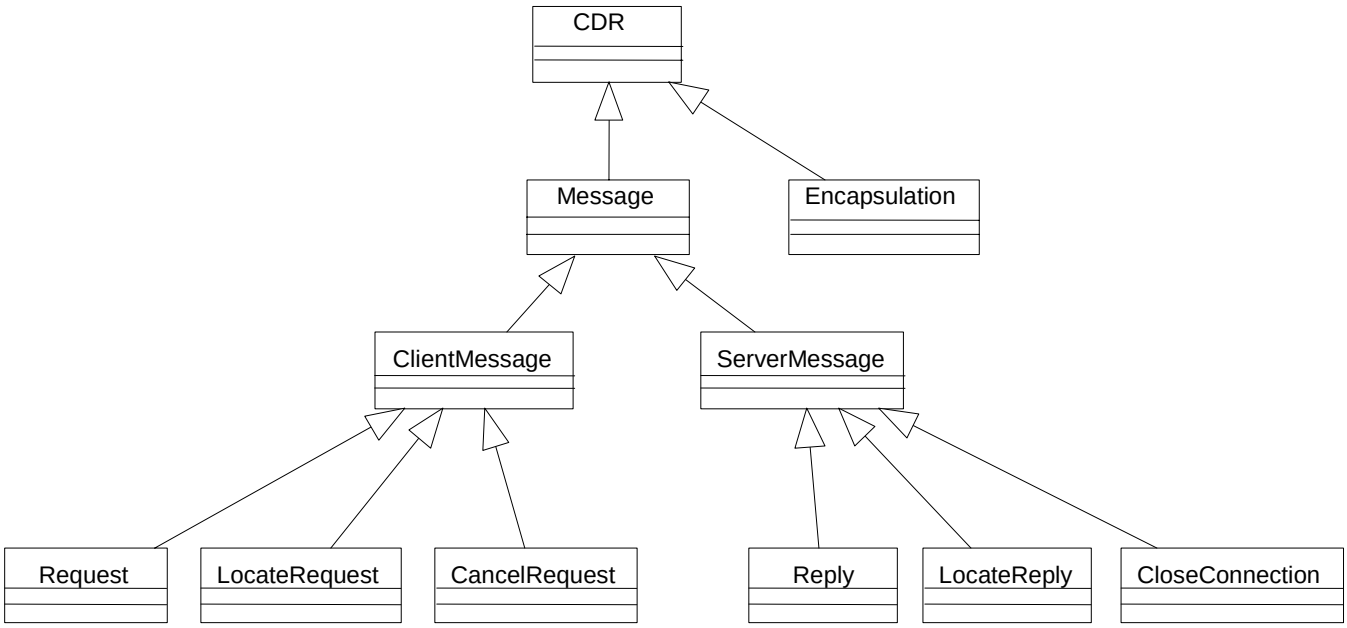


Figure 5.1a GIOP Message Representation classes and Marshalling classes

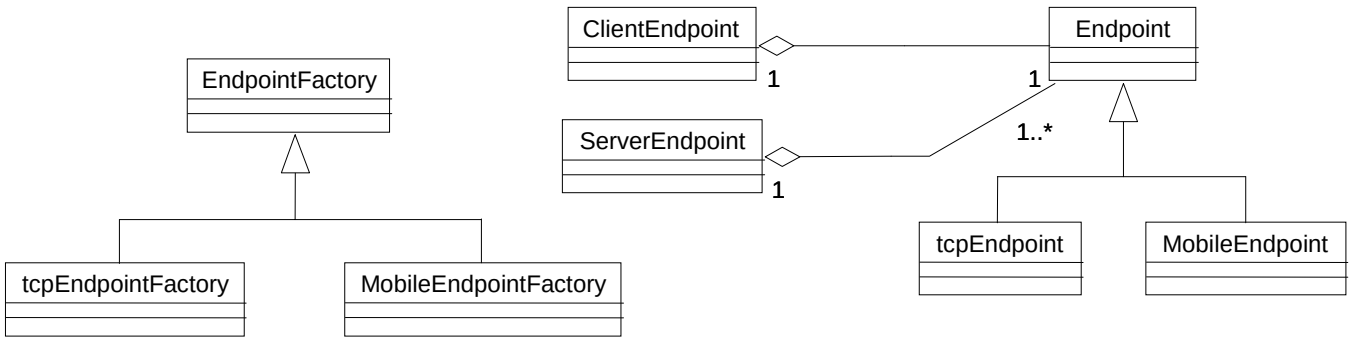


Figure 5.1b Communication Endpoints and Transport Classes

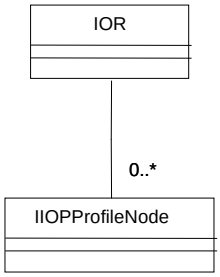


Figure 5.1c Representation of IORs

The class used to represent an IIOP IOR is the `IOR` class. This class uses the `IIOPProfileNode` class to represent an IIOP profile within the IIOP IOR. These classes are used to allow easy manipulation of the various elements within an IIOP IOR. Methods are provided to convert an IIOP IOR in stringified form (see section 3.2.8 Stringified IORs) to an IIOP layer `IOR` and vice versa. The `IOR` class is available as part of the IIOP layer API while the `IIOPProfileNode` is not since it is only used internally by the IOR class.

The Marshalling classes, `CDR` and `Encapsulation` are used to push data into and pop data out of communication buffers ensuring adherence to the alignment requirements of the IIOP protocol, described in section 3.2.4 of Chapter 3. Both these classes are available to the application developer as part of the IIOP layer API.

The GIOP Message Representation classes are used to represent GIOP messages in an easy to use form by application developers. The classes `Request`, `Reply`, `CancelRequest`, `LocateRequest`, `LocateReply`, `CloseConnection`, `ClientMessage`, `ServerMessage` and `CDR` are available for use by application developers while the other classes are not.

The Transport classes are used to allow the creation and simplify the use of TCP/IP connections and Mobile layer logical connections, using the `tcpEndpoint` and `MobileEndpoint` classes. An instance of the corresponding factory class is used to create instances of each class. Each of these classes is part of the IIOP layer API and is available to an application developer.

The Communication Endpoint classes, `ClientEndpoint` and `ServerEndpoint` are used to send and receive IIOP messages. The IIOP messages are represented as instances of Request, Reply, etc, to the application developer. Both of these classes are part of the IIOP layer API available to an application developer.

5.3.2 Representation of IORs

In the IIOP protocol, each IOR has one or more IIOP profiles, which specify where the object represented by the IOR is stored and how it is identified. In the IIOP layer, each IIOP profile is represented as a node in a linked list. Each node is an instance of the `IIOPProfileNode` class, which is illustrated in the UML diagram below. A linked list

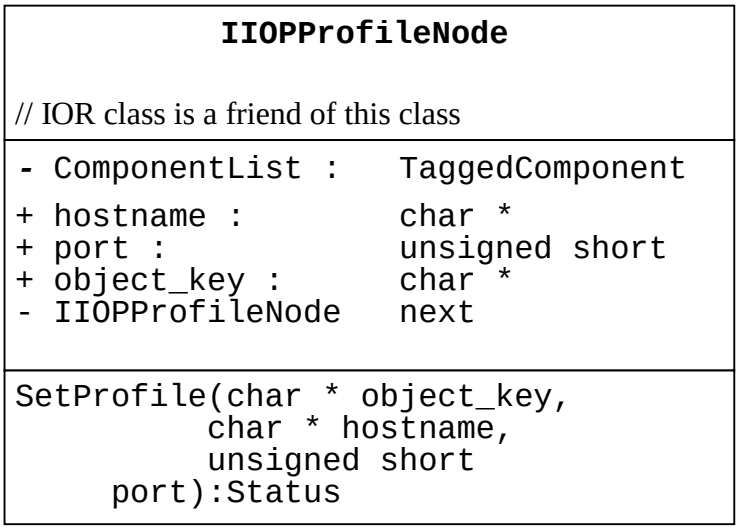


Figure 5.2 Class Diagram for IIOPProfileNode class

storage mechanism was chosen ahead of a simple array as the data within the `IIOPProfileNodes` are accessed only once by instances of the ClientEndpoint and ServerEndpoint classes in the `Connect()`and `Listen()` methods respectively. Therefore a linked list representation is more efficient in terms of adding a new IIOPProfileNode instance than a dynamically growable array.

The `componentList` instance variable points to the head of a linked list of data structures that represent the sequence of Tagged Components within the IIOP IOR. The `hostname` and `port` instance variables identify the process where the object is stored in the network, while the `object_key` identifies to the process the particular object. The `next` instance variable points to the next member of the linked list. The `SetProfile()` member function allows the location information of an `IIOPProfileNode` to be set.

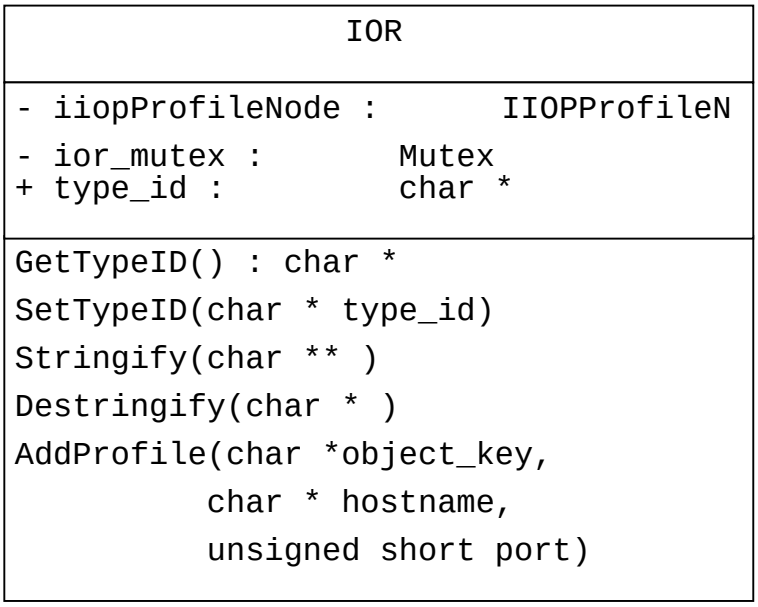


Figure 5.3 Class Diagram for IOR class

The IOR class represents an IIOP IOR. The iiopProfileNode instance variable points to the top of a linked list of IIOP profiles. The type_id instance variable corresponds to the type identifier in an IIOP IOR. The ior_mutex instance variable is used to prevent a multithreaded program corrupting the internal representation of an instance of the IOR class. The type_id instance variable can be retrieved and modified by using the member functions GetTypeID(…) and SetTypeID(…).

The member functions Stringify(…) and Destringify(…) convert an IOR object into an IIOP IOR in stringified or hexadecimal form and vice versa. Finally, the AddProfile(…) member function allows an IIOPProfileNode, which corresponds to an IIOP profile to be added to an IOR object.

5.3.3 Marshalling

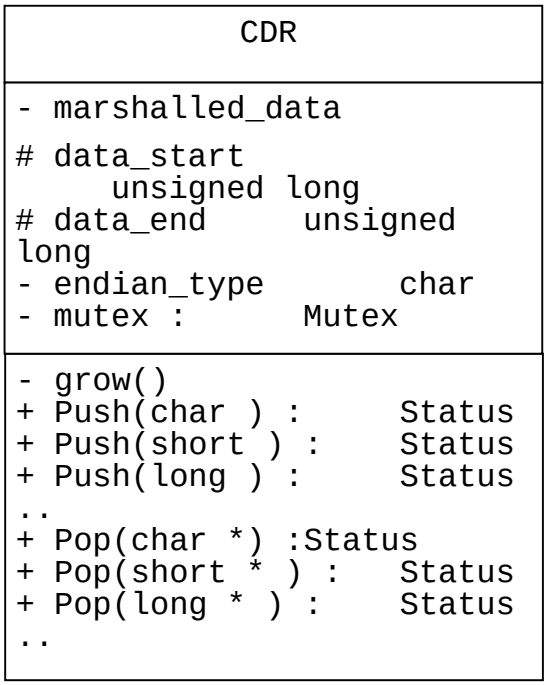


Figure 5.4 Class Diagram for CDR class

The CDR class marshalls data into and out of a marshalling buffer. The marshalled_data instance variable points to the start of the marshalling buffer and the two instance variables data_start and data_end indicate where in the marshalled buffer data will be popped from or pushed into. This is illustrated in Figure 5.5 below, where the string "HELLO" has been pushed into the marshalled buffer.

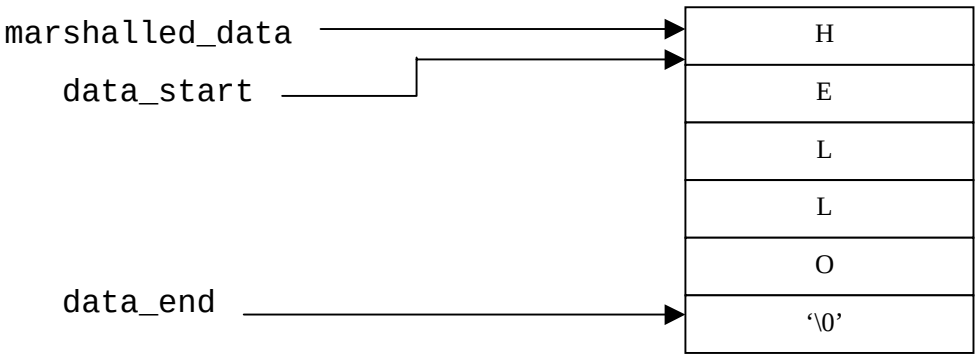


Figure 5.5 Representation of marshalling buffer

The endian_type instance variable specifies whether the data in the marshalling buffer is encoded according to the Big Endian or Little Endian machine architectures. The mutex instance variable ensures that the marshalling buffer does not get corrupted when two or more threads access it.

The grow(…) member function allows the marshalling buffer to expand and contract dynamically as data is pushed into or popped from the buffer. The Push(…) member functions push data into the marshalling buffer, ensuring that the correct IIOP alignment is maintained. It is important to note that the Push(…) member functions do not use the endian_type instance variable. Instead data is pushed into the buffer using the current machine architecture.

The Pop(…) member functions retrieve data from the marshalling buffer after the data_start instance variable. The Pop(…) member functions retrieve the data according to the endian_type instance variable and each function transforms the data from being Big Endian specific to Little Endian specific or vice versa. If the endian_type instance variable matches the current machine architecture then no transformation is necessary.

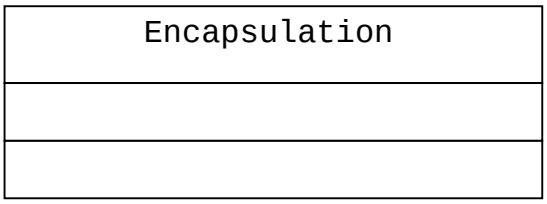


Figure 5.6 Class Diagram for Encapsulation class

The `Encapsulation` class inherits from the `CDR` class. The `Encapsulation` class allows the creation of an encapsulated sequence of data, complying with the CORBA Specification. The encapsulated data has the `endian_type` instance variable as the first in the marshalling buffer. The various methods to push data into and pop data from the marshalling buffer are inherited from the `CDR` class.

5.3.4 GIOP Message Representation

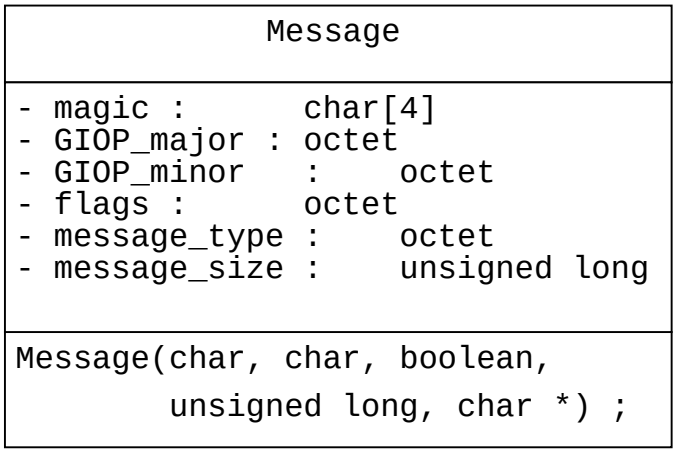


Figure 5.7 Class Diagram for Message class

The `Message` class represents a GIOP message header, as defined in Chapter 13 of the CORBA Specification. The `Message` class constructor allows various aspects of the GIOP header to be manipulated including what version of the GIOP is being used and whether the GIOP message is one fragment of a larger message.

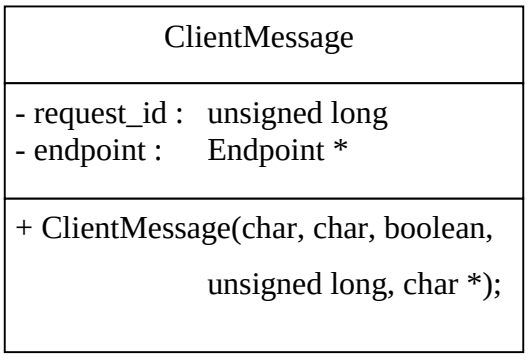


Figure 5.8 Class Diagram for ClientMessage class

The `ClientMessage` class represents any GIOP message that can be sent by a client. The class inherits the GIOP header representation from the `Message` class. The `request_id` instance variable contains the GIOP request identifier, which unambiguously identifies a GIOP message. The `request_id` is assigned a value by the `ClientMessage` class by using a static member variable, which is incremented with every IIOP message sent. This relieves the application developer from having to assign an unambiguous value to the request identifier in an IIOP message. The `endpoint`

instance variable is a pointer to an underlying transport connection, whether it is TCP/IP connection or a Mobile layer ***logical connection***.

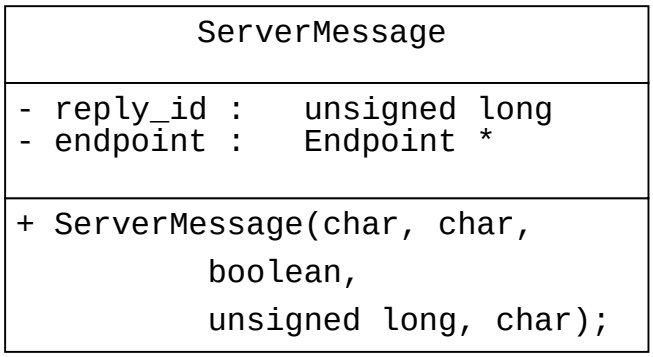


Figure 5.9 Class Diagram for ServerMessage class

Similar to the `ClientMessage` class, the `ServerMessage` class inherits from the Message class and represents any GIOP message that can be sent by a server. The `reply_id` instance variable contains the GIOP message request identifier. The `endpoint` instance variable is a pointer to an underlying transport connection, whether it is TCP/IP connection or a Mobile layer ***logical connection***.

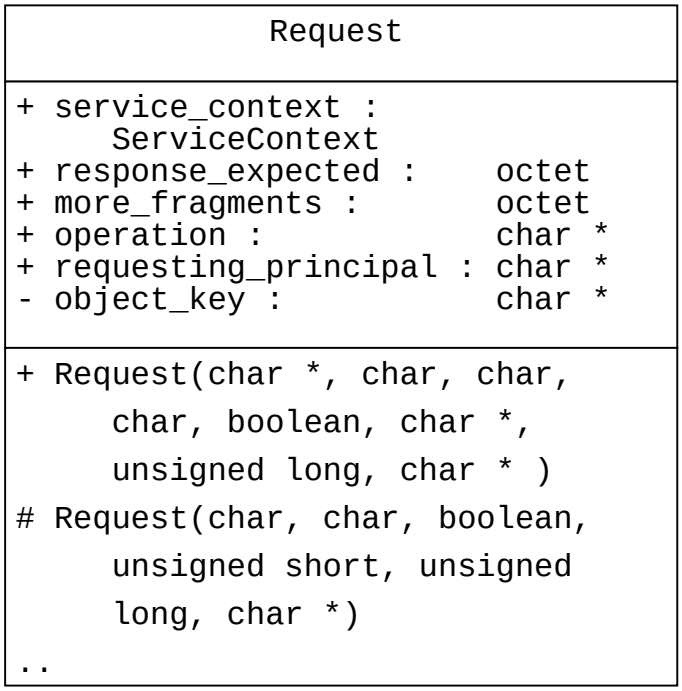


Figure 5.10 Class Diagram for Request class

The `Request` class represents a GIOP Request message and it inherits from the `ClientMessage` class. The various instance variables correspond directly to the GIOP Request header fields with the exception of the `more_fragments` instance variable, which indicates whether or not there are more fragments of this message to follow.

There are two constructors used to create a `Request` object. The first constructor, which is public, and is used by a client, to create a GIOP Request message, which is then sent to a server. It operates by pushing the GIOP header and the GIOP Request header into the inherited marshalling buffer. The GIOP header information is inherited from the inherited `Message` object, while the GIOP Request header information is obtained from the `Request` object. The various parameters to the constructor are fields in the GIOP Request header. Most of these parameters are given default values with the exception of the first, which identifies the method name being invoked. These default values relieve the application developer from having to know each field in the GIOP request message.

The construction of a Request by a client application is illustrated the interaction diagram in Figure 5.11 on the following page. The application must instantiate a Request

object passing the operation name to the constructor. This constructor causes the creation of the inherited constructors, including the CDR constructor. The parameters to the Request are then pushed into the CDR marshalling buffer using calls to the various Push() methods.

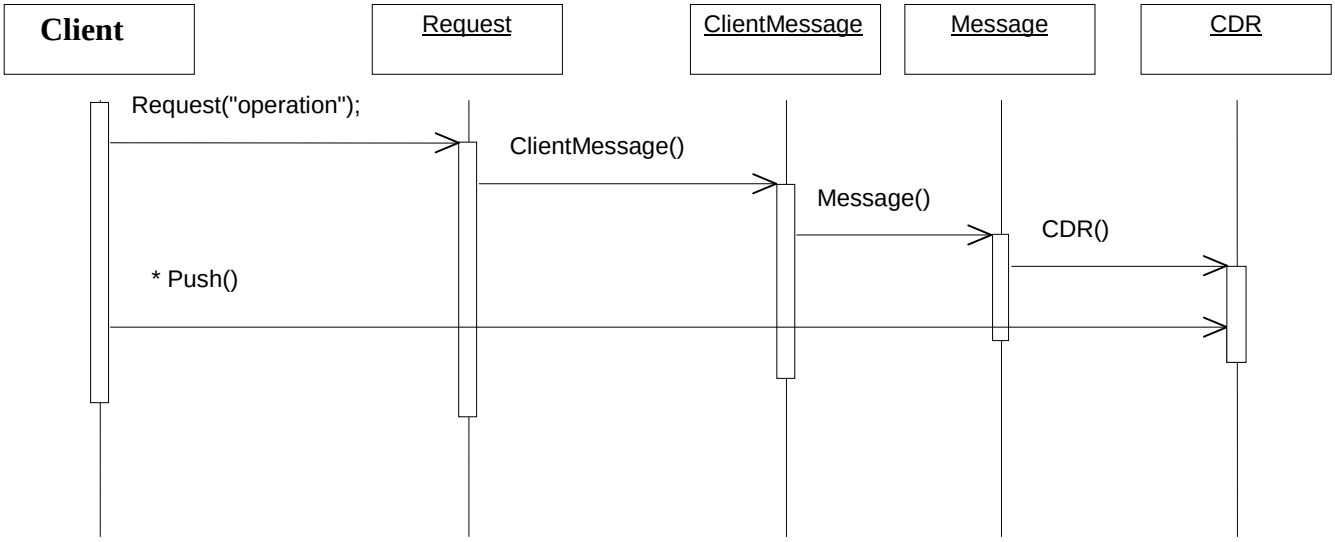


Figure 5.11 Creation of a Request object by a client

The second constructor is used to construct a `Request` object by a server application from a GIOP request message, which is stored in a buffer. The various fields of the GIOP Request header are filled in from the buffer using the `Pop()` member functions inherited from the `CDR` class.

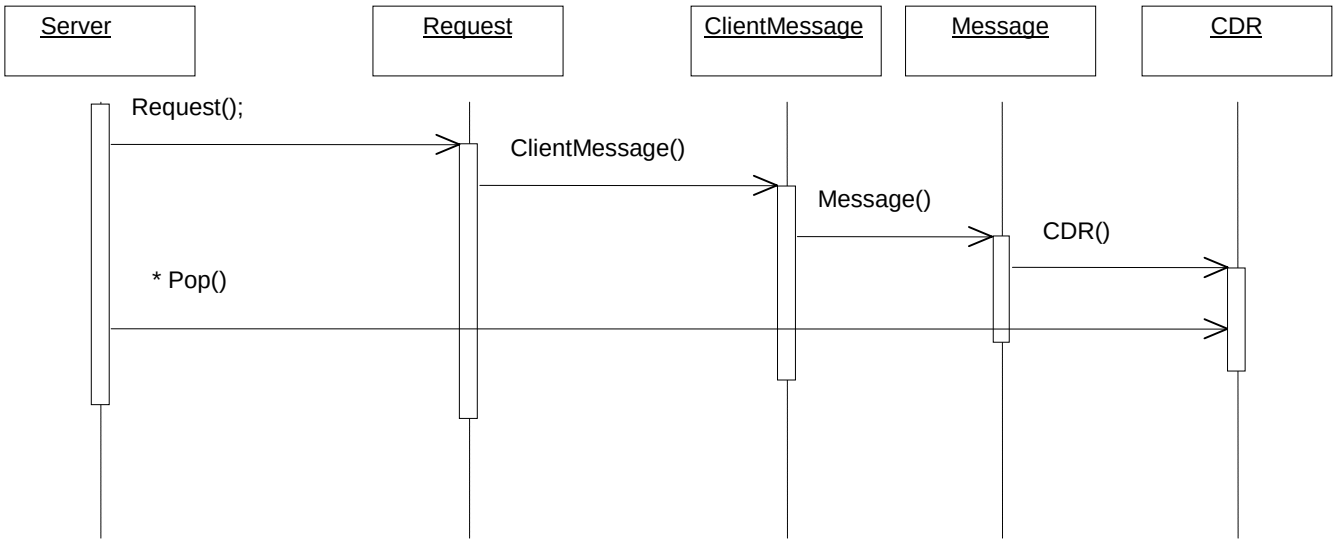


Figure 5.12 Creation of a Request object by a server

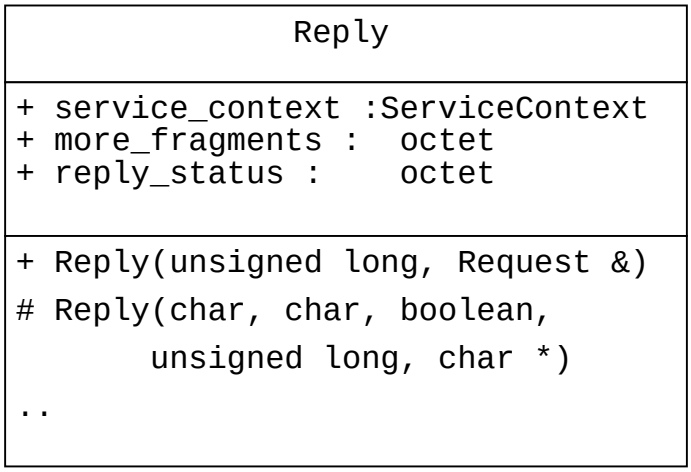


Figure 5.13 Class Diagram for Reply class

The `Reply` class represents a GIOP Reply message and it inherits from the `ServerMessage` class. The various instance variables correspond directly to the GIOP Reply header fields with the exception of `more_fragments`, which plays a similar role to more_fragments in the `Request` class.

There are two constructors used to create `Reply` objects. The first is used by a server when sending a GIOP Reply message. The constructor marshalls the GIOP header and the GIOP Reply header into a buffer for sending. The constructor takes an unsigned long as its first parameter, which corresponds to the locate status field of the GIOP reply message (see section 3.3.4). The constructor also takes a `Request` object as a parameter

to link the Reply object with the original Request. Thus the application developer does not need to know the request identifier that is part of the GIOP reply message. Once the Reply object has been created, the return values for the original method invocation can be appended to the buffer using the Push() functions inherited from the inherited CDR class.

The second constructor is, used by a client to reconstruct the Reply object from a GIOP Reply message, when it is stored in a buffer. The various Reply header fields are filled in from the buffer using the Pop() member functions from the CDR class.

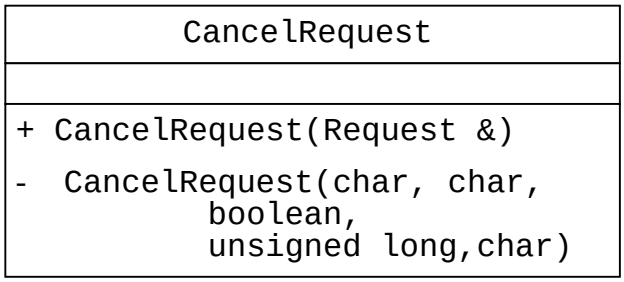


Figure 5.14 Class Diagram for CancelRequest class

Like the Request class, the CancelRequest class simplifies the construction of a GIOP CancelRequest message before sending by a client and upon reception by a server. The request_id of the Request parameter to the constructor is used as the request identifier part of the GIOP CancelRequest header. Again this eliminates the application developer from having to keep track of which request identifier belongs to which Request object.

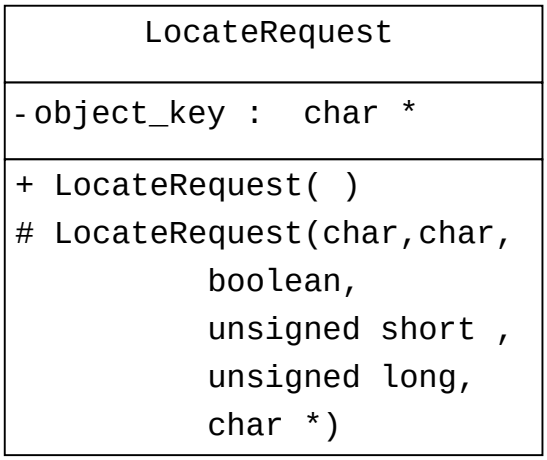


Figure 5.15 Class Diagram for LocateRequest class

Like the Request class, the `LocateRequest` class simplifies the construction of a GIOP LocateRequest message before sending by a client and upon reception by a server. Again the public `LocateRequest` constructor does not require the application developer to assign the `object_key` and `request_id`. The object key is assigned in the `ClientEndpoint` class and the `request_id` is assigned in the inherited `ClientMessage` class.

The second constructor allows the creation of LocateRequest object from a GIOP LocateRequest message. The various parameters to this constructor are used to initialise various instance variables of the LocateRequest object.

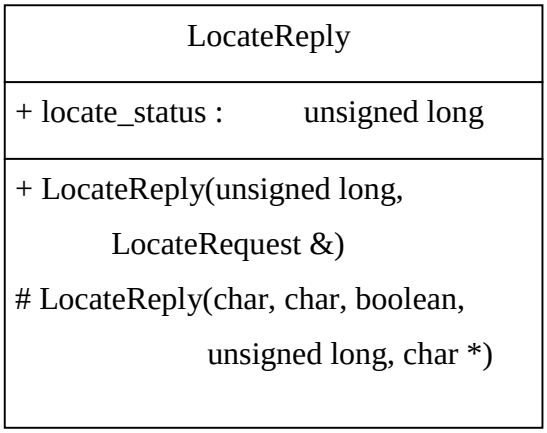


Figure 5.16 Class Diagram for LocateReply class

Like the Reply class, the `LocateReply` class simplifies the construction of a GIOP LocateReply message before sending by a server and upon reception by a client. The `LocateRequest` parameter to the public constructor allows the GIOP LocateReply to be linked with the corresponding GIOP LocateRequest message. Again this relieves the application developer from having to keep account of which request identifier value belongs to which instance of the `LocateRequest` class.

The second constructor is used at the client side. The various parameters are used to initialise various fields of the `LocateReply` object. This constructor also uses the inherited `CDR Pop()` methods to create an instance of the `LocateReply` class from a buffer, passed as a parameter.

Finally, the `CloseConnection`, `Fragment` and `MessageError` are simple classes, which allow the construction of a GIOP header with the correct message type, either `CloseConnection`, `Fragment` or `MessageError`. These classes will not be discussed further.

5.3.5 Transport Classes

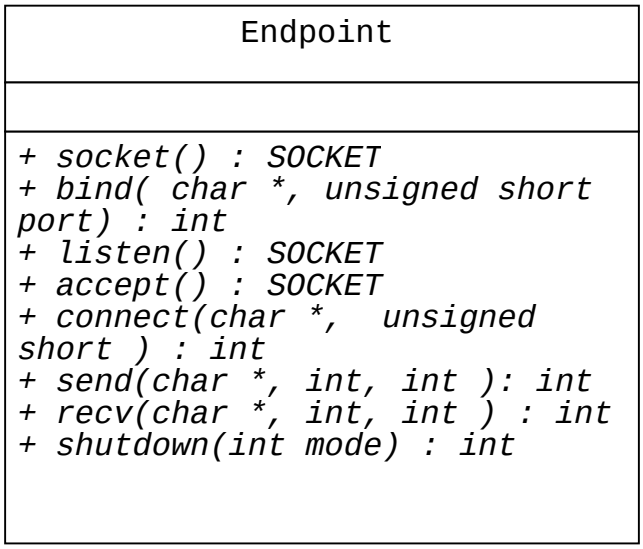


Figure 5.17 Class Diagram for Endpoint class

The `Endpoint` class is an abstract class that represents the functionality available for creating and manipulating an underlying connection. This allows dynamic switching between the `tcpEndpoint` and `MobileEndpoint` classes that inherit from this class. The `tcpEndpoint` and `MobileEndpoint` classes implement each function using TCP/IP connections or Mobile layer ***logical connections***, as appropriate.

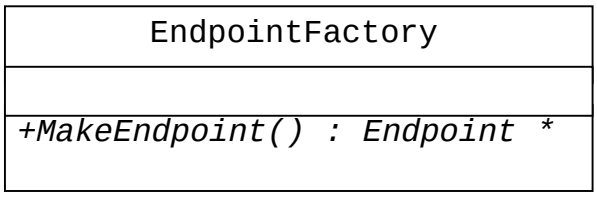


Figure 5.18 Class Diagram for EndpointFactory class

The `EndpointFactory` class is an abstract class, which is based on the Abstract Factory Design Pattern [Gamma et al]. The `tcpEndpointFactory` and `MobileEndpointFactory` classes inherit from this class, each implementing the `MakeEndpoint()` method to create `tcpEndpoint` objects or `MobileEndpoint`

objects. These classes allow the creation of instances of `tcpEndpoint` and `MobileEndpoint` classes and the Abstract Factory Design Pattern allows other factories to be added subsequently.

5.3.6 Communication Endpoints

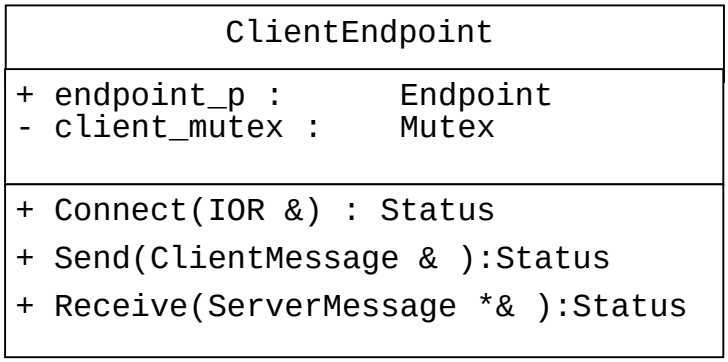


Figure 5.19 Class Diagram for ClientEndpoint class

The `ClientEndpoint` class represents a communication endpoint over which GIOP messages can be sent and received by a client. The instance variable `endpoint_p` has static type `Endpoint` but whose dynamic type can be either `tcpEndpoint` or `MobileEndpoint`. The onus is on the application developer to switch between an instance of the `tcpEndpoint` class and an instance of the `MobileEndpoint` class. Some form of end-to-end agreement would also be necessary before performing this switch. The `client_mutex` variable ensures that a multithreaded client does not corrupt the internal state of the `ClientEndpoint` object.

The `Connect()` method sets up an underlying connection, either TCP/IP or Mobile layer ***logical connection*** to the server address as specified in the `IOR`. If an `IOR` contains more than one IIOP profile, the `Connect()` method attempts to set-up a connection using each profile in turn.The `Send()` method sends a GIOP message represented by a `ClientMessage` over the previously established connection. If the underlying connection is broken or if the parameter passed is not a valid `ClientMessage`, then the `Send()` method returns an error. The `Receive()` method receives a GIOP message from the underlying connection and returns the GIOP message to the caller as a `ServerMessage` object. Note that if there is no GIOP message buffered to be received, then the `Receive()` method will block.

The creation and sending of a `Request` message using a `ClientEndpoint` is illustrated in Figure 5.20 below.

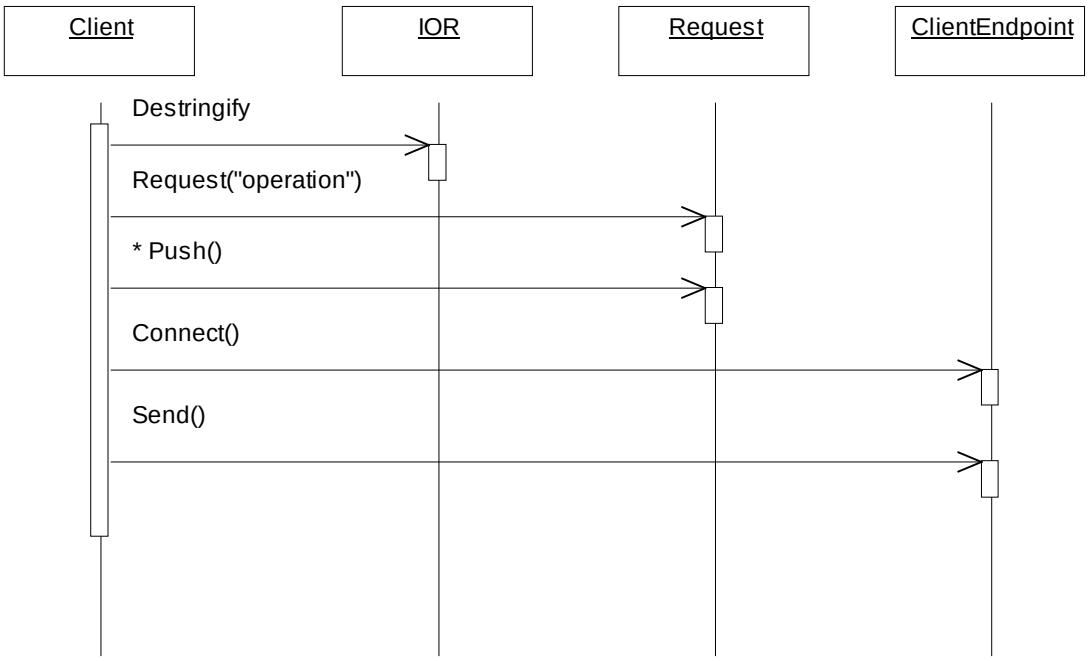


Figure 5.20 Sending an IIOP Request message

In figure 5.20, the client application creates an `IOR` object by calling the `Destringify()` method passing an IIOP IOR in hexadecimal form to it. It then creates a `Request` object, specifying the operation name it wishes to invoke ("operation" in this instance) and pushes the parameters of the method invocation. The client application then connects to the address specified in the IOR by using the `Connect()` method. The client application can then send and receive IIOP messages using the `Send()` and `Receive()` methods. In this instance it proceeds to send an IIOP Request message.

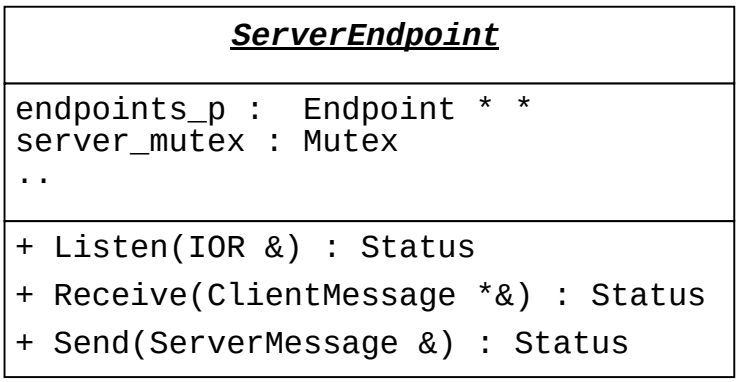


Figure 5.21 Class Diagram for ServerEndpoint class

The `ServerEndpoint` class represents a communication endpoint over which GIOP messages can be sent and received by a server. The instance variable `endpoints_p` is an array of pointers whose static type is Endpoint but whose dynamic type could be either `tcpEndpoint` or `MobileEndpoint`. An array was chosen over a linked list data structure to allow fast access to the various pointers to Endpoint objects. The `server_mutex` variable ensures that a multithreaded server does not corrupt the internal state of the `ServerEndpoint` object.

The `Listen()` method waits for connection attempts and GIOP messages to be received on the endpoints_p array. The elements of the `endpoints_p` array are initialised using the IIOP profiles in the `IOR`.

Th `Receive()` method receives a GIOP message from one of the elements of the `endpoints_p` array and returns a `ClientMessage` object to the caller. Note that if there is no GIOP message buffered to be received, then the `Receive()` method will block. The `Send()` method sends the GIOP message represented by the `ServerMessage` object on an element of the `endpoint_p` array, the element is identified from the `ServerMessage` object.

The receiving of an IIOP Request message is illustrated in Figure 5.22 below.

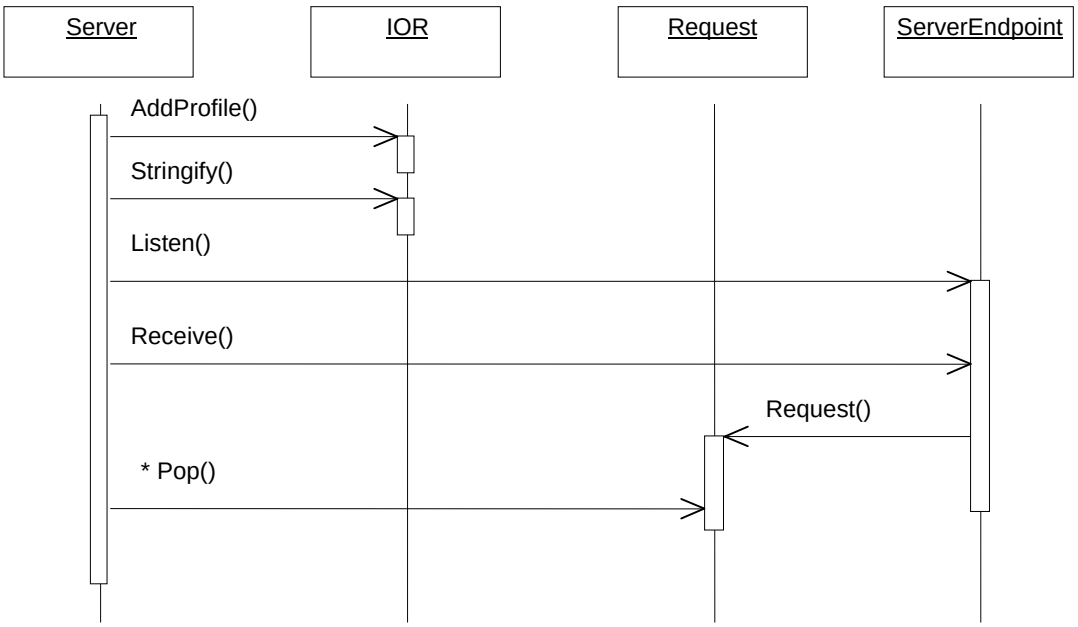


Figure 5.22 Receiving an IIOP Request message

In figure 5.22, the server application creates an `IOR` object and adds a number of profiles using the `AddProfile()` method. The server application then stringifies this IOR converting it into an IIOP IOR and passes it to the client possibly using a Name Service or by sharing a common file. The server application then waits for IIOP messages from clients using the `Listen()` method. When an IIOP Request message is sent by a client, it is received by calling the `Receive()` method. This method in turn uses the Request constructor to create a Request object, which is passed back to the server application. The server application can then pop off the parameters to the method invocation using the `Pop()` methods inherited from the `CDR` class.

Chapter 6

Evaluation

This chapter compares the implementation of the IIOP layer with that of IONA Technologies IIOP Engine. This comparison is carried out to ensure that the IIOP layer conforms to the IIOP protocol standard and to test the performance of the IIOP layer against a highly optimised IIOP implementation, namely IONA Technologies' IIOP Engine. The evaluation criteria include:

- A comparison of the footprint sizes of both the IIOP layer implementation and the IIOP Engine implementation.
- A comparison of the footprint size of a client and server application built using both the IIOP layer and the IIOP Engine.
- The number of lines of code needed to develop the client and server applications mentioned above.
- The time taken to send an IIOP request message and receive the corresponding IIOP reply message.

Conformance to the IIOP protocol was repeatedly tested both during and after development of the IIOP layer. The testing involved ensuring interoperability with the IIOP Engine on both Windows NT and Solaris and ensuring interoperability between the IIOP layer implementation on Windows and Solaris.

The comparison in the rest of this chapter will be made in two parts. Firstly the footprint size of the static and dynamic link libraries of the IIOP layer and the IIOP Engine are compared. In addition, the footprint size and code size of a distributed whiteboard application written using both the IIOP layer and the IIOP Engine are compared. Secondly, the average time taken to send an IIOP request message and receive the corresponding IIOP reply message will be compared for various IDL data types of differing sizes.

6.1 Footprint and Code Size Comparison

A distributed whiteboard application was developed to compare both the footprint size and code size of an application using the IIOP layer and the IIOP Engine. A brief description of this application will now be given.

6.1.1 Distributed Whiteboard Application

The Distributed Whiteboard Application (DWA) consists of a single server and one or more clients. The server application was initially developed on Windows NT and the code was ported to Solaris using firstly the IIOP layer and then the IIOP Engine. The client program was developed only for the Windows NT platform using firstly the IIOP layer and then the IIOP Engine. The client makes extensive use of Microsoft Foundation Classes (MFC) to provide the graphical user interface. The client application is multithreaded while the server is single threaded.

At startup, clients register with the server by sending an IIOP request, with the operation name set to “Register”. A stringified IOR is included in the IIOP request message, which the server uses to “callback” the client and update its screen when another client draws a line. The server sends an IIOP reply message back to the client, allocating a unique identifier to the client. This identifier is used when the client begins to draw.

The user of the client program is then presented with a drawing window, similar to the one shown in Fig 6.1.

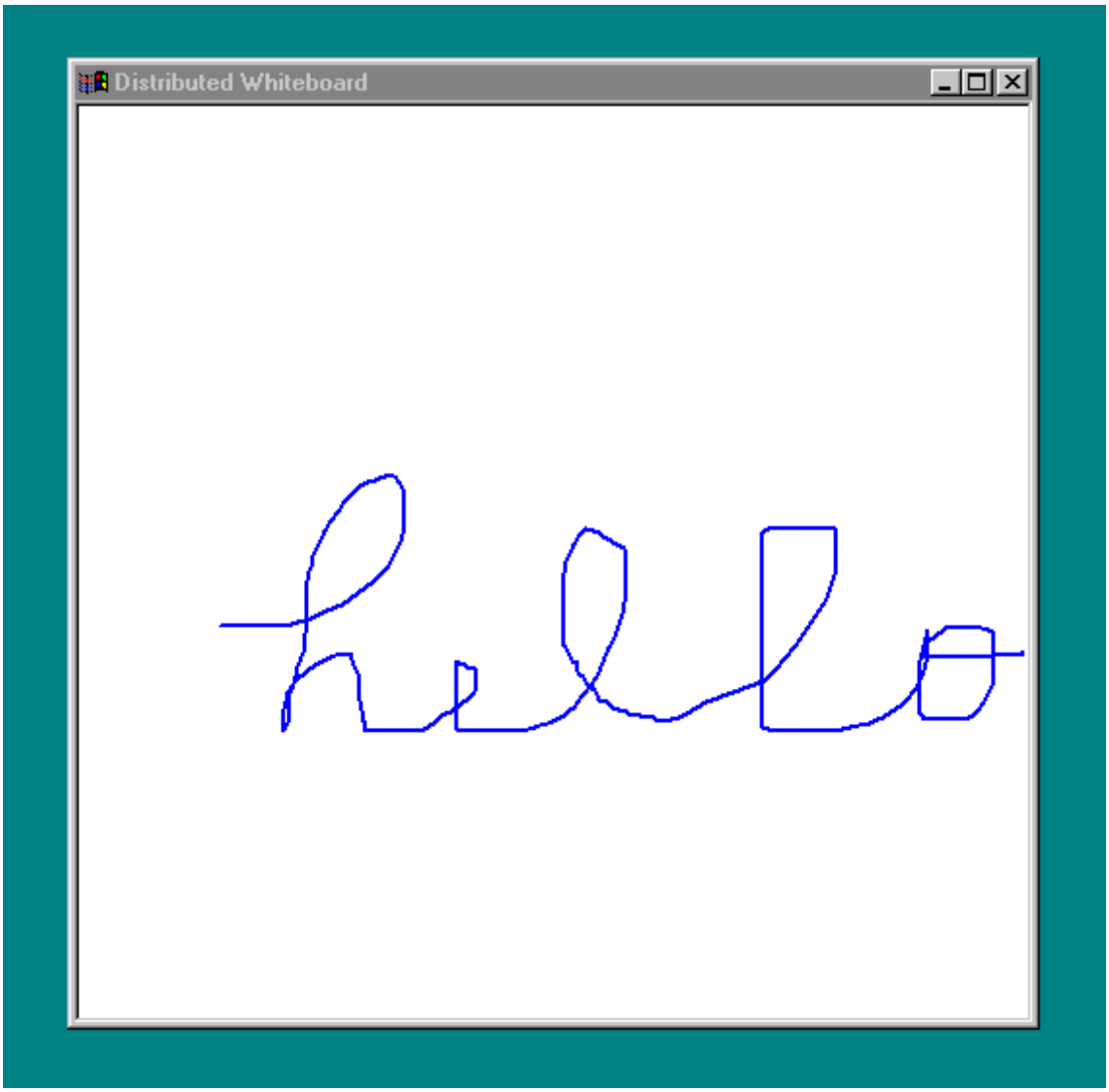


Figure 6.1

As the user moves the mouse over the client window, the client program creates an IIOP request message, with the operation name set to “DrawLine”. The IIOP request body consists of the identifier mentioned above and two (X, Y) points, indicating the old mouse position and the current mouse position. This IIOP request message is then sent to the server, which relays this IIOP request message to all other registered clients, causing the clients to draw a line at between the two points mentioned above.

6.1.2 Footprint Size

It is important to note that the IIOP Engine was designed to use the C programming language while the IIOP layer was designed to use the C++ programming language. This

difference is reflected in the sizes of the static library and dynamic link libraries for both the IIOP layer and IIOP Engine. This difference is also reflected in the footprint sizes of the client and server applications since they were only linked with the static versions of the libraries. A program developed using C++ has usually a larger footprint than a similar application developed using C. One possible reason for this, is the overhead associated with object creation and object deletion.

The footprint sizes for the various libraries on Windows NT are given in Table 6.1 below. A shared library for IIOP Engine on the Solaris platform was unavailable for this comparison.

	IIOP LAYER	IIOP ENGINE
Static library on Windows NT	96.1 KB	38 KB
Dynamic Link library on Windows NT	64 KB	16 KB
Static library on Solaris	120 KB	38.3 KB

Table 6.1 Library size comparison

The footprint sizes for the client and server programs are given below in Table 6.2. The client was developed on a 266MHz Intel Pentium Pro with 64 MB of RAM, using Microsoft Visual C++ 5.0 and running Windows NT 4.0. The server used a similar machine as its NT platform and used a 143 MHz Sun Ultra Sparc with 64 MB of RAM running Solaris 2.5. The server on Solaris was developed using GNUs gcc version 2.8.1.

	IIOP LAYER	IIOP ENGINE
Client	60.5 KB	43.5 KB
Server on Windows NT	63.5 KB	52.5 KB
Server on Solaris	90 KB	56 KB

Table 6.2 Client and Server footprint size comparison

As can be seen from table 6.2, all programs developed using the IIOP layer are larger than their counterparts developed using the IIOP Engine. This was expected due to the reasons outlined above.

The server on Windows NT using the IIOP layer is approximately 20% larger than its IIOP Engine counterpart. The client on is approximately 39% larger when using

the IIOP layer than when using the IIOP Engine. One possible reason for this larger percentage difference could be the fact that the client is multithreaded opposed to the single threaded server, with two threads having a larger footprint due to twice as many IIOP layer objects being created and deleted.

6.1.3 Code Size

One of the goals of the IIOP layer was to hide the complexity of the IIOP protocol as much as possible. This contrasts with the IIOP Engine where knowledge of the structure of IIOP requests and replies is needed in their creation. The amount of code needed to create the applications described above is given in Table 6.3. Note that all blank lines were removed from the particular files before the comparison was carried out.

	IIOP LAYER	IIOP ENGINE
Client	213 lines	295
Server	108	236

Table 6.3 Number of lines of code comparison

The server when written with IIOP Engine requires 118% more code to be written than when the IIOP layer is used. This extra code is needed to marshall and unmarshall the IIOP messages into communication buffers, the IIOP layer carries out these low level tasks.

The client when written with IIOP Engine requires 38% more code to be written than when using the IIOP layer. This however ignores the amount of common code written using the MFC to handle mouse events and painting on the screen. The amount of code to handle the GUI specific function comes to 169 lines. This implies that 44 lines were specific to the IIOP layer and 127 lines to the IIOP Engine. This works out at a 188% difference. This larger difference, when compared with the server code, could in part be due again to the fact that the client is multithreaded while the server is not. With a multithreaded application, more lines of code are necessary to create IIOP messages and send and receive them than a single threaded application. This would imply from the server comparison above that this is the reason for the percentage being higher in the client application.

6.2 Comparison of Average Invocation Time

The average time taken to send an IIOP request and receive a corresponding IIOP reply was timed on both the Windows NT and Solaris platforms. The results for each of these will now be presented. It is important to note that although the average time required to send an IIOP request and receive a corresponding IIOP reply will be similar on Windows NT and Solaris, a comparison of the IIOP layers performance on Windows NT versus Solaris would be difficult due to various hardware differences. As well as these differences, there is no uniform method of calculating millisecond timing differences on both Windows NT and Solaris.

Windows NT

The average invocation time for the IIOP layer and the IIOP Engine was calculated using two computers. One machine acted as a client sending an IIOP request message and receiving the corresponding IIOP reply message while the other machine acted as a server receiving an IIOP request message and returning an IIOP reply message.

The client machine had a 200MHz Intel Pentium Pro microprocessor and 96 MB of RAM while the server had a 133MHz Intel Pentium microprocessor with 48 MB of RAM. Each machine had a 10/100Mbps dual speed Ethernet cards connected to a 3Com Super Stack II Ethernet Switch.

The average invocation time was calculated for the IDL primitive data types of different sizes and for the String data type. The test program used to carry out the calculation consisted of creating and sending an IIOP request and receiving the corresponding IIOP reply one thousand times. The average time for sending the request and receiving the reply was calculated. This program was then executed ten times to eliminate possible timing differences due to process scheduling on the Windows NT operating system and the average of these ten executions was calculated.

The average invocation time for the primitive IDL data types and the String data type are given in Table 6.3 on the following page.

	IIOP LAYER	IIOP ENGINE
Char	(1.6337 , 1.8183)	(0.6353 , 0.6647)
Short	(1.5217 , 1.5263)	(0.637 , 0.6513)
Long	(1.5217 , 1.5263)	(0.6397 , 0.6513)
Float	(1.4787 , 1.5227)	(0.6314 , 0.6546)
Double	(1.5046 , 1.5514)	(0.6353 , 0.6568)
String	(1.6644 , 1.7916)	(0.6523 , 0.6777)

Table 6.3 Average Invocation times for Windows NT

From the above table it is obvious that the IIOP Engine outperforms the IIOP layer on average for each IDL data type tested. This presumably is due the IIOP layer client and server constructing two IIOP layer objects during execution of the test program. The client creates a Request object and sends it to the server as an IIOP request message. The server receives the IIOP request message and creates a corresponding Request object. To send an IIOP reply message to the client, the server must create a Reply object. On receiving the IIOP reply message, the client creates a Reply object.

Solaris

The client and server programs described in the calculation of the average invocation time for Windows NT were ported to the Solaris platform. The client machine and server machines each had a 143MHz Ultra Sparc microprocessor with 64 MB of RAM. The machines were connected by a 10Mbps broadcast Ethernet link.

The average invocation time for the various IDL data types is given in Table 6.4 below.

	IIOP LAYER	IIOP ENGINE
Char	(0.967 , 1.042)	(0.695 , 0.708)
Short	(0.974 , 0.980)	(694.06 , 0.698)
Long	(0.974, 0.981)	(0.696 , 0.709)
Float	(0.975 , 0.981)	(0.696 , 0.70)
Double	(0.974 , 1.002)	(0.707 , 0.710)
String	(1.038 , 1.047)	(0.836 , 0.916)

Table 6.4 Average Invocation times for Solaris

Again the IIOP Engine outperforms the IIOP layer in terms of average invocation time but not as significantly as on Windows NT. One possible factor underlying this improvement could be the increased power of the server from 133MHz microprocessor and 48 MB of RAM to 143MHz and 64 MB of RAM. Another less obvious reason could be the method used to calculate the timing difference. This method differs on Windows NT from Solaris

Chapter 7

Conclusions

This chapter gives a summary of the work completed during the course of this project and the remaining work that needs to be done. Finally, possible future work will be discussed in the context of the overall design described in Chapter 4.

7.1 Work Completed

The main achievement of this project is the design of a layered architecture to enable mobile devices to interoperate with CORBA objects. An understanding of the limited processing resources and bandwidth available in a typical mobile environment was attained during the course of the project. As well as this, significant experience was gained working with the CORBA standard. In particular, how object invocations are transferred from a client to a server that hosts CORBA objects.

An easy to use implementation of the IIOP protocol was completed. During the course of the implementation, network programming experience was gained on the Windows NT and Solaris operating systems using Winsock and BSD sockets respectively. As well as this, knowledge of multithreaded programming and in particular writing reentrant software was gained. Other experience gained includes debugging network and multithreaded programs as well as writing Windows GUI programs using Microsoft Foundation Classes in the contrast of the Distributed Whiteboard Application described in Chapter 6.

Significant testing of the implementation of the IIOP protocol was carried out both on Windows NT and Solaris. As well as this interoperability was tested using IONA Technologies IIOP Engine, again on Windows NT and Solaris. In addition to this testing, an evaluation of the IIOP layer was carried out by comparing it to the IIOP Engine. This comparison took place in three parts:

- Comparison of footprint sizes of the IIOP layer and the IIOP Engine

- Calculation of the amount of code needed to be written by both the IIOP layer and the IIOP Engine
- Comparison of the average time taken to send an IIOP request message and receive the corresponding IIOP reply message

The design for the Mobile layer and Swizzling layer, described in Chapter 4, has been completed. A significant part of the implementation of the Mobile layer is completed but has yet to be tested and implementation of the Swizzling layer has yet to be started.

7.2 Remaining Work

The implementation of the IIOP layer, described in Chapter 5, will need to be ported to a PDA, for example Windows CE and tested. In addition, the Swizzling layer and the Mobile layer need to be implemented and then ported to a PDA. This will allow the completed implementation of the design, described in Chapter 4, to be tested in a typical mobile environment.

7.2 Future Work

Other possible avenues for future work include dynamic switching between TCP/IP connections and Mobile layer connections when the available bandwidth becomes low or intermittent. This dynamic switching would include how to close down and existing connection (TCP/IP or Mobile layer) and open a new connection (Mobile layer or TCP/IP) in an application without the need for intervention from the application user. This switching would require the transfer of state information about one connection to enable the creation of another connection.

Another possibility for future work would involve the handover from one Mobility Gateway (MG) to another when a mobile device changes its location. If the mobile device acts as a server, storing CORBA objects, then object invocations directed to the old MG would need to be directed to the new MG and then forwarded to the mobile device.

Finally, the implementation of the design when completed could be used as the IIOP specific part, in the construction of an ORB. This ORB may be a minimal

implementation of the CORBA standard suitable for a mobile, providing some of the CORBA Common Object Services.

Appendix A OMG IDL

This section contains the OMG IDL for the GIOP and IIOP modules.

A.1 GIOP Module

```
module GIOP { // IDL extended for version 1.1

     struct Version {
     octet major;
     octet minor;
     };

#ifndef GIOP_1_1

     // GIOP 1.0
     enum MsgType_1_0{ // rename from MsgType
          Request, Reply, CancelRequest,
          LocateRequest, LocateReply,
          CloseConnection, MessageError
     };

#else
     // GIOP 1.1
     enum MsgType_1_1{
          Request, Reply, CancelRequest,
          LocateRequest, LocateReply,
          CloseConnection, MessageError,
          Fragment // GIOP 1.1 addition
     };
#endif

     // GIOP 1.0
     struct MessageHeader_1_0 { // Renamed from
     MessageHeader
          char magic [4];
          Version GIOP_version;
          boolean byte_order;
          octet message_type;
          unsigned long message_size;
     };


     // GIOP 1.1
     struct MessageHeader_1_1 {
          char magic [4];
          Version GIOP_version;
          octet flags; // GIOP 1.1 change
          octet message_type;
          unsigned long message_size;
```

```
    };
    };// GIOP 1.0

    struct RequestHeader _1_0 {
         IOP::ServiceContextList service_context;
         unsigned long request_id;
         boolean response_expected;
         sequence <octet> object_key;
         string operation;
         Principal requesting_principal;
    };
    // GIOP 1.1
    struct RequestHeader_1_1 {
         IOP::ServiceContextList service_context;
         unsigned long request_id;
         boolean response_expected;
         octet reserved[3]; // Added in GIOP 1.1
         sequence <octet> object_key;
         string operation;
         Principal requesting_principal;
    };

    enum ReplyStatusType {
         NO_EXCEPTION,
         USER_EXCEPTION,
         SYSTEM_EXCEPTION,
         LOCATION_FORWARD
    };

    struct ReplyHeader {
         IOP::ServiceContextList service_context;
         unsigned long request_id;
         ReplyStatusType reply_status;
    };

    struct CancelRequestHeader {
         unsigned long request_id;
    };
    struct LocateRequestHeader {
         unsigned long request_id;
         sequence <octet> object_key;
    };

    enum LocateStatusType {
         UNKNOWN_OBJECT,
         OBJECT_HERE,
         OBJECT_FORWARD
    };

    struct LocateReplyHeader {
         unsigned long request_id;
         LocateStatusType locate_status;
    };
};
```

A.2 IIOP Module

```
module IIOP { // IDL extended for version 1.1

     struct Version {
     octet major;
     octet minor;
     };

     struct ProfileBody_1_0 { // renamed from ProfileBody
          Version iiop_version;
          string host;
          unsigned short port;
          sequence <octet> object_key;
     };

     struct ProfileBody_1_1 {
          Version iiop_version;
          string host;
          unsigned short port;
          sequence <octet> object_key;
          sequence <IOP::TaggedComponent> components;
     };
};
```

Bibliography

[Chan'97] Terence Chan. Unix System Programming using C++.
1997
Prentice Hall

[Chen'97] Larry T. Chen et al. Designing Mobile Computing Systems Using Distributed Objects.
1997.
IEEE Communications Magazine, February 1997.

[Colouris'94] George Colouris et al. Distributed systems concepts and design.
1994
Addison-Wesley

[Fowler'97] Martin Fowler et al. UML Distilled.
1997.
Addison-Wesley

[Gamma'94] Eric Gamma et al. Design Patterns – Elements of Reusable Object Oriented Software.
1994.
Addison-Wesley

[GIOP'98] Chapter 13, CORBA Specification.
1998
http://www.omg.org/corba/cichpter.htm

[Hall'93] Martin Hall et al. Winsock API.
1993.
www.medusa.uni-bremen.de/intern/knowhow/winsock/

[Joseph'97] Anthony D. Joseph et al. Mobile Computing with the Rover Toolkit.
1997.
IEEE Transactions on Computers: Special Issue on Mobile Computing.

[Kemp'96] Peter Kemp et al. Design of MASE V2.
1996.
http://www.sics.se/~onthemove/docs/OTM_d33.doc

[Kernighan'88] Brian Kernighan et al. The C programming language.
1988.
Prentice Hall.

[Meyer'96] Michael Meyer et al. Design of MASE V1.
1996
http://www.sics.se/~onthemove/docs/OTM_d17.doc

[OMG'98] CORBA Specification.
1998
www.omg.org/corba/c2index.htm

[Perkins'96] Perkins et al. IP Mobility Support.
1996
http://info.internet.isi.edu:80/in-notes/rfc/files/rfc2002.txt

[Petersen'97] Karin Petersen et al. Flexible Update Propagation for Weakly Consistent Replication.

1997
http://www.parc.xerox.com/csl/projects/bayou/pubs/sosp-97/

[Pham'96] Thuan Q. Pham. Multithreaded programming with Windows. 1996.
Prentice Hall.

[Raatikainen'97] Raatikainen et al. Service Machine Development for an Open Long-term Mobile and Fixed Network Environment.
1997
DOLMEN Consortium. ACTS Ref: AC036 DOLMEN.

[Stevens'90] Richard W. Stevens. Unix network programming.
1990
Prentice Hall

[Stroustrup'97] Bjarne Stroustrup. The C++ programming language. 1997
Addison Wesley