CORBA Middleware for a Palm Operating System

Mary Connolly

B.E.

A dissertation submitted to the University of Dublin,

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

September 2001

Declaration

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

Signed: ____________________

Mary Connolly

September 2001

Permission to lend and/or copy

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

Signed: ____________________

Mary Connolly

September 2001

Acknowledgements

Many thanks to my supervisor, Mr. Alexis Donnelly, for his guidance and advice throughout the course of this project.

Thanks to Raymond for his willingness to help me with any questions and problems that I had.

Thanks to the other members of the MSc class for their friendship and for making the year so enjoyable.

Finally, a special thanks to my family and friends whose constant support throughout the year was much appreciated.

Abstract

Typically, Computer Networks are heterogeneous and therefore require special middleware applications in order to enable communication across their diverse platforms. Middleware applications make the task of writing software applications for heterogeneous systems easier, by applying platform-independent models and abstractions, and by hiding as much low-level complexity as possible without unduly sacrificing performance.

The Common Object Request Broker Architecture (CORBA) standard provides a set of rules for writing such platform independent middleware. CORBA applications require lots of functionality in order to unite diverse platforms within a heterogeneous system, and are therefore bulky and computation intensive. Generally, they are used on machines with considerable memory and processing resources, that can cope with them.

The challenge posed by this project was to write a condensed and extensible piece of CORBA middleware, that could operate effectively on a resource restricted handheld device, thus providing a portable data access device, that can conveniently fit into a shirt pocket.

Table of Figures

Fig 2.1 CORBA Architecture diagram 15
Fig 2.2 Structure of a value of type Any 18
Fig 2.3 Structure of a TypeCode pseudo-object 19
Fig 2.4 Minimum CORBA architecture.......................................27
Fig 2.5 Components of the TAO ORB 29
Fig 2.6 GIOP Message types 32
Fig 3.1 Client ORB Architecture.......................................38
Fig 3.2 Client / Server ORB Architecture.......................................38
Fig 3.3 ORB Initialisation classes 39
Fig 3.4 GIOP Messaging classes 42
Fig 3.5 Stub Implementation classes 44
Fig 3.6 Downcall classes.......................................45
Fig 4.1 Request Invocation.......................................58
Fig 4.2 Object Reference contents.......................................61
Fig 4.3 Basic structure of a GIOP Message 64
Fig 4.4 GIOP 1.2 message header 65
Fig 4.5 GIOP Request message 67
Fig 4.6 GIOP Reply message 70
Fig 6.1 Wrapper Facade.......................................79

Table of Contents

1. INTRODUCTION 1

1.1 FUTURE OF HANDHELD DEVICES IN DISTRIBUTED ENVIRONMENTS 1
1.2 PROJECT GOAL 2
1.3 ROADMAP 3

2. CORBA AND HANDHELD DEVICES 5

2.1 LIMITATIONS OF HANDHELD DEVICES 7
2.2 CORBA MIDDLEWARE BACKGROUND 9
2.1.1 OMG 10
2.2.2 CORBA Architecture 10
2.2.3 Corba Features 12
2.2.3.1 IDL 12
2.2.3.2 Language Mappings 13
2.2.3.3 ORB Interface 13
2.2.3.4 Operation Invocation and Dispatch Facilities 14
2.2.3.5 Object Adapters 14
2.2.3.6 Inter-ORB Protocol 15
2.2.4 Interface Repository 16
2.2.5 Implementation Repository 17
2.2.6 The Any Type and TypeCodes 17
2.2.7 ORB Transparencies 19
2.2.8 Common Data Representation Format 21
2.2.9 Interoperable Object Reference 22
2.2.10 Server Side of an ORB 22
2.3 CORBA VS. MINIMUM CORBA 23
2.3.1 MinimumCORBA Omissions 25
2.4 EXAMPLE IMPLEMENTATIONS OF CORBA MIDDLEWARE 27
2.4.1 TAO 27
2.4.2 PalmORB 29
2.5 IIOP AND GIOP 31

3. ORB DESIGN 34

3.1 DESIGN GOALS 34
3.2 STRIPPED DOWN CORBA STANDARD TO FIT PALM III DEVICE 35
3.3 ORB CLASS DIAGRAMS 39
3.3.1 ORB Initialisation 39
3.3.2 GIOPMessaging 42
3.3.3 Stub Implementation 44
3.3.4 ORB Downcalling 45
3.4 EXTENSIBILITY OF ORB DESIGN 46

4. CODE IMPLEMENTATION 48

4.1 DEVELOPMENT ENVIRONMENT 48
4.2 NETWORK CONNECTIVITY 49
4.3 EVENT DRIVEN PROGRAMMING 50

4.4 MEMORY MANAGEMENT FOR THE PALM DEVICE 50
4.4.1 Memory Allocation of Strings *51*
4.4.2 Memory Allocation of Classes *53*
4.4.3 Reference Counting *53*
4.4.4 Directional Attributes *54*
4.5 USE OF TEMPLATES 55
4.6 BIG ENDIAN VS. LITTLE ENDIAN 56
4.7 REQUEST INVOCATION 57
4.7.1 Stubs and Skeletons *58*
4.7.2 Creating an Object Reference from an IOR *59*
4.7.3 Implementation of IIOP and GIOP *63*
4.7.3.1 GIOP Message Header 64
4.7.3.2 Request Message Format 66
4.7.3.3 Reply Message Format 69

5. EVALUATION 72

5.1 CRITIQUE OF DESIGN 72
5.2 CRITIQUE OF IMPLEMENTATION 73
5.3 OVERALL CORBA ORB INTEROPERABILITY 75

6. CONCLUSIONS 76

6.1 SUMMARY OF WORK 76
6.2 KNOWLEDGE GAINED 77
6.3 FUTURE WORK 77

7. BIBLIOGRAPHY 81

CHAPTER 1

1. INTRODUCTION

The original idea behind handheld devices was to produce a small (pocket sized), portable, easy to use device that could be used as an extension to a less portable desktop computer. Handhelds provide a window to desktop data. Once desktop data has been downloaded onto a handheld, it can be viewed away from the desk, conveniently and speedily. Applications that make most use of this tend to be of the personal organiser type. Examples include address books, to do lists and memo pads. Email applications can also exploit handhelds in the same fashion, so that users do not necessarily have to be sitting at a desktop in order to read their electronic mail.

Over the years, handhelds have evolved at quite a high rate. For example, the earlier Palm devices that date back to 1996 were very primitive, affording only 128KB of RAM, and little or no support for communication with other devices. More recent models like the Handspring Visor have up to 8MB of RAM and can communicate with other devices using Infrared and TCP/IP.

Although handheld devices are still quite limited, it is obvious that they are certainly becoming powerful enough for broader and more sophisticated applications than those of a mere electronic organiser.

1.1 Future of Handheld Devices in Distributed Environments

One interesting advance for a handheld device is to broaden its capabilities to enable it to operate in a distributed computing environment, and to provide mobile data access to an entire system, rather than just a single desktop computer. After all, as we enter an age

that endeavours to achieve anywhere, anytime, anyhow computing, the concept of a portable distributed systems becomes crucial.

Such extended capabilities transform a handheld device into something very powerful indeed. For example, remote invocation could be used to control operations located on other devices within a distributed computing system. Scope for exciting development is certainly provided for. Ponder the notion of a pocket size computer that can cross multiple programming languages, and multiple operating systems!

1.2 Project Goal

It is apparent that while the handheld device makes a reasonable effort in its role as remote desktop window, modern PDAs offer sizeable margins for capitalising on their enhanced capacity, so that they can be used for something much more powerful. The aim of this project is to make a contribution to the task of closing the gap between using a handheld as an extension to a desktop computer, and using it as a portable access point to a distributed computing environment, or indeed fully incorporating it into a distributed system by including server side functionality so that the device itself can actually *implement* some of the system functionality.

In particular, the goal of the project is to implement a version of the Object Management Groups (OMG) Common Object Request Broker Architecture (CORBA) standard middleware to achieve a cross-platform, cross-programming language ORB implementation that enables a Palm III client application to communicate not only with a server application built on an identical computing architecture, but also with applications built on several different computing architectures. It should also enable the intercommunication of applications implemented in diverse programming languages.

1.3 Roadmap

The remaining chapters in this thesis catalogue the project phases that were performed in order to realise the aforementioned goals.

Chapter 2 CORBA and Handheld Devices

This Chapter introduces handheld devices, their characteristics, and of course their limitations. These are things that must be considered when designing applications for PDAs. It also introduces the architecture and key features of the OMG's CORBA standard, followed by a discussion on how the full CORBA standard can be stripped down to produce a minimal but compatible, standard set of implementation rules. The latter refers to the MinimumCORBA standard, which is also a recognised OMG standard that has been specially constructed for CORBA implementations on devices with limited resources.

Chapter 3 Design

The proposed extensible design of the CORBA ORB middleware, which adheres to the MinimumCORBA standard where possible, is described here. All omissions from the latter standard are documented and explained within this chapter.

Chapter 4 Implementation

This presents the important ORB implementation features and issues. It includes a section on the techniques employed to achieve consistent memory management of the Palm III device, which is key to the successful implementation of applications on such a limited piece of electronics.

Chapter 5 Evaluation

The evaluation chapter critically analyses the design and the implementation of the ORB application

Chapter 6 Conclusions

A summary of the work done and knowledge gained is provided in the conclusion. This chapter also puts forward several ideas, which could be used to carry out further work on this CORBA ORB.

CHAPTER 2

2. CORBA AND HANDHELD DEVICES

As already discussed, handheld devices such as those running Palm OS provide a useful extension to desktop computers. Even though it is indeed possible to perform more complex tasks with handhelds, their main purpose is for viewing data and entering small amounts of data, rapidly and easily. Typically a desktop user will remain at their desk for a prolonged period of time, working on a dedicated task that is enabled by their desktop. Handheld devices on the other hand, tend to be used as an aside, which can be conveniently and speedily referred to, while almost all attention is focused on another *core* task. For this reason, handhelds require the following key features:

- **Small Size**

They must be small enough to be conveniently carried anywhere, for example, in a shirt pocket.

- **Ergonomic Interface**

A handheld device should have a fast and easy to use user interface. A handheld user should be able to comfortably and rapidly navigate the device during meetings, at business lunches and in situations where there is no convenient place to mount the device.

- **Desktop Integration**

It should be easy to synchronise a handheld device with a desktop computer. This serves several purposes. Firstly, it backs up important data. It also enables a user to input large amounts of data comfortably and rapidly on a desktop machines using a mouse and keyboard. This data can subsequently be transferred electronically to the handheld

device. This process avoids lengthy manual data entry on the latter's limited input interface.
[Foster ' 00]

Clearly, a handheld device can be utilised valuably in conjunction with a desktop computer, to enable the easy use of address book, memo-pad, and other helpful organisational applications, away from the actual desk itself. It is now time to progress towards a discussion on the incorporation of handhelds into a distributed environment that encompasses multiple platforms, for example Unix or Windows, running applications implemented in diverse programming languages like Java, COBOL or C/C++. It is also of immense importance to highlight the restrictions of handheld devices. These restrictions must be considered when endeavouring to implement on a handheld, the manner of bulky and intricate, low level middleware code, that enables such interoperation.

PalmORB [Roman ' 99], is a CORBA compliant middleware application that has been developed at the University of Illinois, with a design that allows it to fit onto limited resource devices. PalmORB provides a seamless mobile data access mechanism using handheld devices and wireless links. It essentially extends a handheld device from something that simply acts as a smart organiser with stripped down versions of widely used desktop programs, to a device that is seamlessly integrated into a sophisticated distributed computing environment. This provides users with a unified image of a distributed system from a device that can conveniently fit inside a shirt pocket.

Furthermore, the design of PalmORB, for use within the user centric 2K distributed computing environment [Roman ' 99], which is also under development at the University of Illinois, makes for what appears like an even more powerful handheld device. This unique amalgamation enables computation intensive tasks to be processed away from the handheld, on a separate machine within the 2K system.

Such technology has enabled sophisticated mechanisms such as video streaming to run on handheld devices. PalmORB is discussed in further detail in section 2.4.2.

The ORB implemented for this thesis is different in that it was designed to run *entirely* on any handheld device, to which it has been commissioned, with no outsourcing of complex and demanding tasks. Such an ORB facilitates the possibility of a flexible handheld device that can interoperate within diverse distributed systems. The design is discussed in depth in chapter 3.

2.1 Limitations of Handheld Devices

Since the capabilities of desktop computers and handheld devices differ significantly, the approach to designing a handheld application is much different to that for a desktop application. The following limitations should be kept in mind when designing for a handheld.

- **Performance Requirements**

Useful information should be available instantly, since unlike a user at a desktop, who is likely to remain at that machine for some time performing dedicated tasks, a handheld user is typically performing another more important task, and merely requires the handheld for a few small but crucial functions, like retrieving a telephone number, or jotting down key points at a meeting. Since there is a limit to the processing power of a handheld device, it is very important to ensure that their applications are small and efficient. Typically, handheld applications are implemented in C or C++ for its efficiency.

- **Battery and Processor Power**

Since handheld devices rely on batteries for power, they are limited to smaller processors than a plugged-in desktop computer. Such a processor is not ideal for running computation intensive applications.

One possible way to overcome this short coming would be to arrange for any intensive number crunching operations to be executed on a more powerful machine that is remote from the handheld. This approach is taken by PalmORB.

- **Limited Memory**

Limited memory space on a handheld means that things like deeply recursive routines. Large numbers of global variables, and huge dynamically allocated data structures are not handheld device friendly.

- **RAM as Permanent Data Storage**

Unlike desktops, where vast amounts of data can be stored on hard drives, with a handheld device all data must be stored in RAM, which means that storage space is much more limited. The reason for this is that data entry and access must be very fast. Space limitations mean that handheld applications must be as small as possible, and that infrequently used features should be left out. Also, any data to be stored persistently on a handheld should be packed tightly before being written to memory. The Palm III device used for this project has only 2MB of RAM.

Other elements that should be kept in mind when writing handheld applications include the limited input methods and the small screen size. The former element makes it tedious to input large amounts of data. For example, the Graffiti handwriting recognition software system that comes with PalmOS, is faster than many forms of handwriting recognition, but at a top speed of about thirty words per minute, it is still too slow for entering large amounts of data. Small screen size makes it difficult to display large amounts of information, and complex user interfaces are out of the question. It becomes essential to strike a balance between showing an adequate amount of information, and keeping the interface looking uncluttered and easy to use.

Writing CORBA middleware is a difficult process. Obviously, writing such middleware for a restricted handheld device adds further complications. Such software tends to be bulky, and to involve the use of complicated structures and functions, and all other things that spell trouble when programming a limited resource, pocket sized device. The following subsections take a look at the CORBA standard, and then at the MinimumCORBA standard. The latter details ways of cutting down the former, in order to come up with a compliant set of rules for writing a minimal CORBA ORB, that fits onto a small handheld device, while remaining compatible with all other fully implemented, compliant CORBA ORBs.

2.2 CORBA Middleware Background

Typically, Computer Networks are heterogeneous. A network may, for example, have UNIX workstations for the support of software development, or a mainframe to handle database transactions, and of course, personal computers that run Windows and provide general office tools.

One of the main reasons for this heterogeneity is the change in technology over time. The best technologies from different time periods tend to end up co-existing on networks. Another reason for network heterogeneity is that different combinations of computers, operating systems, and networking platforms will work best for different subsets of the computing activities performed within diverse networks.

Of course, developing software for a heterogeneous distributed system is very complicated. The difficulties of application development for heterogeneous distributed systems can be eased to a large extent, by applying platform-independent models and abstractions to software

development, and by hiding as much low-level complexity as possible without unduly sacrificing performance.

The Common Object Request Broker Architecture (CORBA) standard provides a set of rules for writing such platform independent middleware, in order to hide some of the difficulties associated with writing applications for distributed and heterogeneous systems. [Baker ' 97]

2.1.1 OMG

The Object Management Group (OMG) was formed in the late 1990s to address the problems of developing portable distributed applications for heterogeneous systems. The CORBA specification, written and maintained by the OMG, supplies a balanced set of flexible abstractions and concrete services needed to realise practical solutions for the problems associated with distributed heterogeneous computing. The CORBA standard has been reviewed on several occasions. The most up to date version is CORBA 2.3. [Henning ' 99]

2.2.2 CORBA Architecture

Object Request Broker middleware provides a means for writing distributed systems that can use different programming languages and operating systems, and integrate applications to provide new systems. ‘On-the-wire’ format is a standard language and platform independent message format, that is used when transmitting messages for remote object invocation in a distributed system.

The OMG has also defined CORBAservices and CORBAfacilities, which essentially hang from the ORB internal infrastructure, to extend the built-in support for applications.

The purpose of the CORBAservices, is to provide a set of utilities that are useful for objects or low level distributed applications. A subset of these services have been grouped into categories and described below.

- **Distributed systems-related services:**

Naming Service: to allow a client to find remote objects that have been registered with the naming service. This extra level of indirection means that target objects can easily be moved from one host to another without breaking any of the references to that object that are held by clients. The only place where the host destination details will need updating, is at the naming service itself.

Event Service: to allow a client or server to send a message or event to any number of receivers.

Security Service: to ensure that only suitable privileged users can call specified operations on particular objects

- **Database-related services:**

Concurrency Service: to provide a locking mechanism to control the access to an object by concurrent callers.

Transaction Service: to control the commitment and abortion of transactions that span multiple databases, of the same or of different types.

Persistent Object Service: to define an abstract framework for how a database and an object should communicate to store and restore the object to and from the database.

- **General services:**

Licensing Service: to allow an object's data to be converted to and from a stream of bytes, so that it can be copied to another location

Time Service: to find the time of day or to obtain an event call after a specified time.

CORBAfacilities, on the other hand provide a higher level of support for applications. The latter refers to a new area of CORBA which has been designed to address information management, system management, task management and user interfaces. [Baker ' 97]

2.2.3 Corba Features

2.2.3.1 IDL

In order to be able to invoke operations on a distributed object, a client must first of all know the interface related to the target object. Such an interface is composed of the operations it supports and the types of data that can be passed to and from those operations.

The CORBA standard defines a set of rules for writing these object interfaces. These rules constitute what is known as the Interface Definition Language (IDL). IDL is not a programming language like C++ or Java, in the sense that objects and applications cannot be implemented in IDL. The latter merely allows object interfaces to be defined in a fashion that is independent of any particular programming language. This facilitates the interoperation of applications implemented in different programming languages, which is vital to the CORBA goal of supporting heterogeneous systems and integrating separately developed applications.

2.2.3.2 Language Mappings

CORBA Language mappings specify exactly how the IDL object interface definitions are translated into each of the different programming languages supported by the CORBA standard. For each IDL construct, a language mapping defines which features of the programming language are used to make the construct available to applications. For example, for languages that support the 'class' construct, IDL interfaces are mapped to classes, and operations are mapped to the member functions of those classes. Implementation languages currently supported by the CORBA standard are C, C++, Smalltalk, COBOL, Ada and Java. This cross language support enables the implementation of different portions of a distributed system in different languages. For example, a server application requiring speed and efficiency in order to cope with large amounts of data, could be implemented in a fast language like C or C++, while its clients could be written using languages, such as Java or Visual Basic, which have strong support for the development of aesthetically pleasing graphical user interfaces.

Most state of the art ORB implementations that use the CORBA standard have an IDL compiler associated with them, which generates stub and skeleton classes from the IDL definitions. These stubs and skeletons, which are discussed in further detail in section 4.7.1, provide the link between the client and server applications, and the ORB itself.

2.2.3.3 ORB Interface

The ORB interface provides a point at which a client can interface with the underlying ORB for purposes other than sending messages. For example, a client can pass an interoperable object reference, section 2.2.9, for a remote object to the ORB via the ORB interface. Message invocations, on the other hand, go through client stubs.

2.2.3.4 Operation Invocation and Dispatch Facilities

CORBA applications invoke operations on remote objects by sending requests to the target CORBA objects. At the server side, the request is processed and dispatched to the correct object adapter and servant combination. A reply is then sent back to the requesting client. A servant is an entity that implements one or more CORBA objects. Object adapters are discussed in further detail in section 2.2.3.5.

The two general approaches to request invocation and dispatch are *static* and *dynamic*. With static invocation OMG IDL is translated into language-specific stubs (client-side request invocation functions) and skeletons (server-side request dispatch functions). Dynamic invocation is more complicated. It requires the construction and dispatch of CORBA requests at run time rather than at compile time. The creation and interpretation of requests requires the use of a mechanism such as an Interface Repository, discussed in section 2.2.4, to provide run time access to IDL definitions, and their interfaces and types.

The latter approach can be useful for applications such as gateways or bridges, that receive and forward requests without having compile time knowledge of the types and interfaces involved. However, the static invocation approach provides a better programming model for application development in statically defined languages such as C++.

2.2.3.5 Object Adapters

An object adapter is an object that adapts the interface of one object to a different interface which is expected by a caller. It allows a caller to invoke requests on an object without knowing that object's true interface. The three principal functions of CORBA object adapters are to create object references that allow clients to address objects, to ensure that every object is incarnated by a servant, and to direct requests to the servant that implements the object to be invoked. Object

Adapters facilitate the development of scalable, high performance server applications.

Until version 2.1, CORBA contained specifications only for the Basic Object Adapter (BOA). The BOA was the original CORBA object adapter. CORBA 2.2 introduced the Portable Object Adapter (POA) as a means of improving the portability and capabilities of object adapters [OMG ' 01] chapter 11. The POA replaced the BOA.

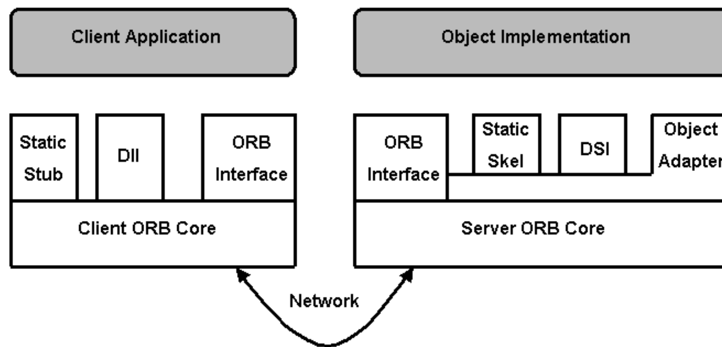

Fig 2.1 CORBA Architecture diagram

2.2.3.6 Inter-ORB Protocol

In order to be CORBA compliant, an ORB implementation must be able to communicate with all other CORBA compliant ORBs, using a protocol known as the Internet Inter-ORB Protocol (IIOP). IIOP is defined to run on the widely available Transmission Control Protocol/Internet Protocol (TCP/IP). IIOP uses a messaging format called the General Inter-ORB protocol (GIOP). That is, IIOP is the GIOP message format sent over TCP/IP. GIOP can also be layered on other transport protocols, including specialised protocols for proprietary networks. This means that any CORBA client can communicate with any CORBA object to which it has the necessary

access privileges. However, to be CORBA compliant, an ORB must be able to use IIOP when communicating with objects on other ORBs.

IIOP request packets contain the identity of the target object, the name of the operation to be invoked, and the parameters. This information is used automatically at the server side to find the target object, and call the correct function on it. [Henning ' 99]

2.2.4 Interface Repository

An Interface Repository (IFR), can be implemented as a component of a CORBA ORB to provide persistent storage for all IDL types such as modules and interfaces. The purpose of such a storage facility is to provide clients with runtime access to an object's type information (and other information about that type), so that a client can invoke an operation on a remote object, without always needing to have compile time knowledge of the objects characteristics.

A client application, wishing to invoke on a remote object, without having compile time knowledge of the objects type information, can use a Dynamic Invocation Interface (DII) to do so. The DII accesses information stored within the Interface Repository in order to construct a request message at runtime.

The IFR provides a set of functions that enable a DII to browse and list its contents, and to determine an object's type information.

A Dynamic Skeleton Interface (DSI) is really the server-side equivalent of a DII, in that it allows a server to receive an operation invocation on an object, even one whose IDL interface is unknown at compile time. Instead of the server being linked with the skeleton code for an interface, it can use the DSI which will be informed of an

incoming operation invocation. The DSI then determines the identity of the object being invoked, the name of the operation, and the types and values of each of the parameters being passed. At that stage, it is possible for the operation being requested by the client to be executed, and the result returned. [Baker ' 97]

2.2.5 Implementation Repository

An Implementation Repository allows an ORB to locate and activate implementations of objects. The Implementation Repository maintains a mapping from a registered server name to the file name of the executable code which implements that server. The advantage of registering servers with an Implementation Repository, is that if an operation invocation is made on a object whose server is not running, or if a client attempts to bind to such an object, an ORB can automatically launch the server by consulting the Implementation Repository to obtain the servers executable code file name. [Baker ' 97]

2.2.6 The Any Type and TypeCodes

The IDL Any type provides a universal type the can hold a value of arbitrary IDL type. The Any type allows for values whose types are not fixed at compile time, to be sent and received at runtime. Values of type Any maintain type safety, for example, the receiver of an Any type must treat its contents exactly as the sender intended. If the sender placed a float value in the Any type, the receiver must extract that value as a float type, other wise a runtime error will be generated.

A value of type Any consists of two members, see figure 2.2. The first member is the actual value contained inside the Any. The second member is the TypeCode of the value (described below). [Baker ' 97]

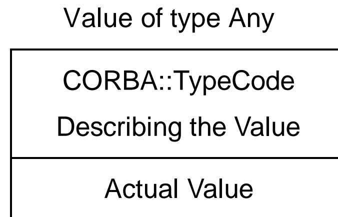

Fig 2.2 Structure of a value of type Any

TypeCodes are used to carry runtime descriptions of IDL types. They are important for the dynamic aspects of CORBA such as type Any, the DII and the DSI.

TypeCodes provide several useful functions. They help to preserve the type safety of CORBA, by ensuring that type mismatches are detected at run time. In addition, TypeCodes provide *introspection*. Given an Any containing a value whose type is unknown, this unknown type can be determined by extracting the TypeCode from the Any and interrogating it. Introspection is vital for programs requiring dynamic typing. Also, TypeCodes provide an ORB runtime at the receiving end with the information required to correctly unmarshal the values off the wire.

A TypeCode, figure 2.3, essentially contains two values, the TCKind member and a description of the TypeCode. The TCKind is an enumeration that records the kind of type that is described by the TypeCode, for example a null, float, object reference or struct kind, among others. The description of the TypeCode depends on the values of TCKind. For example, if the TCKind is a struct, the description contains the name of the struct and the name and type of each member of the structure. [Baker ' 97]

Value of type TypeCode
TCKind
Description

Fig 2.3 Structure of a TypeCode pseudo-object

An example of an area where Typecodes and the Any type are very useful, is in the CORBA Event Service, where it must be possible to transport values whose IDL types are unknown to the service. Using Any types, the events can simply be values of type Any, and the Event Service can then act as a transport for these values without requiring compile time knowledge of the actual types contained in them. At the receiving end, Typecode introspection can be used in order to determine the type contained in the Any value.

2.2.7 ORB Transparencies

Request invocation for an ORB requires the following characteristics:

- **Location transparency**

The client should not need to know whether the target object is a local object in the same or a different address space, or if is implemented in a process on a different machine. Server processes do not necessarily remain on the same machine forever, it should be possible for them to be moved around from machine to machine without clients becoming aware of it. If a server process is moved, new IORs containing the new location details should be generated for each of the objects it supports. Once the client has obtained the updated IORs, the ORB should simply obtain the new server destination details, and use this information to send the client request to the correct location.

- **Server transparency**

The client should not need to know which objects are implemented on which servers

- **Language independence**

The client should not be concerned with what language is used by the server. As an example, a C client should be able to call a Java implementation without being aware of the fact that it is invoking on a Java object.

- **Implementation independence**

The client should not need to be aware of how the object implementation works on the server.

- **Architecture independence**

The client should be unaware of the CPU architecture that is used by the server and should be shielded from such details as byte ordering and structure padding. This facilitates Palm / desktop communication, since Palm uses big-endian while intel based Pentium machines use little-endian.

- **Operating system independence**

The client should not be required to know what operating system is used by the server. The use of the standard on-the-wire message format means that a received message can be understood by any operating system.

- **Protocol independence**

The client should not need to know what communication protocol is used to send messages. Generally, if several protocols are available to communicate with the server, an ORB should transparently select a protocol at run time. In order to be CORBA compliant, this ORB should be able to communicate IIOP messages over the TCP/IP protocol. [Henning ' 99]

2.2.8 Common Data Representation Format

The Common Data Representation, defined by GIOP, is required for the binary layout of IDL types for transmission across a network. CDR-encoded data should be tagged to indicate the byte ordering of the data, which can be either big-endian or little-endian. This is necessary so that both big-endian and little-endian machines can send data in their native format, with the onus being on the receiver to undertake byte-swapping if it uses a different byte order to the sender.

All data types require well-defined encodings in order to ensure interoperability between ORBs.

CDR requires the alignment of primitive data types along their natural byte boundaries. For example, short values should be aligned on a 2-byte boundary, long values on a 4-byte boundary and double values on an 8-byte boundary. Strings and wide strings should be aligned as unsigned long types (aligned on a 4-byte offset), that indicates the length of the string, including its terminating NULL byte, followed by the bytes of the string, terminated by a NULL byte. Structures should be aligned as a sequence of structure members in the order in which they are defined in the IDL. This kind of alignment means that data can be marshalled and un-marshalled simply by pointing at a value stored in memory in its natural binary representation. This approach avoids expensive data copying during marshalling.

CDR encoding requires an agreement between sender and receiver about the types of data that are exchanged. This agreement is established by the IDL definitions that are used to define the interface between sender and receiver. If the agreement is violated, the receiver has no way to prevent misinterpretation of the data.

Because CDR supports both little-endian and big-endian representations and aligns data on natural boundaries, it makes marshalling both simple and efficient. [Henning ' 99]

2.2.9 Interoperable Object Reference

Object references are the only way for a client to reach target objects. A client cannot communicate unless it holds an object reference. References are published by servers in several ways. The most common way for a client to acquire object references is to receive them in response to an object invocation. In that case, object references are parameter values and are no different from any other type of value, such as a string. Clients simply contact an object, and the object returns one or more object references. In this way, clients can navigate an “object web” in much the same way as following hyperlinks. Another common way for clients to obtain object references is for servers to advertise references in some well-known service, such as the Naming Service.

Regardless of the origin of object references, they should always be created by the ORB server run time on behalf of the client, to which they should subsequently be made available. [Henning ' 99]

2.2.10 Server Side of an ORB

Whenever a server application creates an IOR object reference, the server-side run time embeds object key information inside the object reference, that supports binding of the object to the servant that implements it. An IOR is also provided with an IP address (or host

name) and TCP port number, to allow a client to correctly locate the host in which the remote object is implemented. The contents of an IOR are discussed in more detail in section 2.2.9. A server can insert its own address and port number into a reference to facilitate direct binding. A server can also employ indirect binding, which involves the use of an external location broker known as an Implementation Repository.

At the server side of a request invocation, the ORB locates the IOR information that is encoded within the request message. If the server application for the object being invoked is not already running, the ORB activates it. The server side object adapter uses the IOR information retrieved from the request message, to dispatch the request to the servant that incarnates the target object. Any arguments that have been provided by the client invocation are also passed to the object, and the operation is invoked. If any of the arguments are out or inout values, they are returned to the client in a reply message, along with the return value. Out and inout parameters are further discussed in section 4.4.4. If the call fails, an exception, including any data contained in the exception, is returned to the client.

2.3 CORBA Vs. Minimum CORBA

The MinimumCORBA standard describes a subset of the CORBA standard, and is designed for systems with limited resources. Implementations of the full CORBA standard are too large to fit PDAs and other devices with limited resources. Acceptable performance levels are also an issue when considering the implementation of a CORBA ORB on a small device. Devices with resource restrictions require a cut-down version of CORBA, and this is provided by the MinimumCORBA standard.

The minimumCORBA specification supports all of OMG IDL. This allows maximum compatibility between minimumCORBA and full CORBA applications.

Many of CORBAs features have much value in typical large scale CORBA applications, however there are also some cases where these features use up so many resources that their inclusion cannot be substantiated. Minimum CORBA omits many of the resource intensive features that are not typically essential to a basic CORBA implementation. However it is of course possible to implement such features within a minimal CORBA ORB, if they are required.

Omitting features from CORBA represents a trade-off between usability and conserving resources. Obviously, an implementation of the full CORBA standard has a greater degree of user-friendliness, but minimumCORBA facilitates the conservation of limited resources.

The MinimumCORBA specification defines a single profile that preserves the key benefits of CORBA (portability of applications and interoperability between ORBs). The following goals were recognized when choosing this profile:

- MinimumCORBA provides a profile that reserves broad applicability within the world of limited resource systems.
- MinimumCORBA should interoperate easily with CORBA so that applications running on either kind of ORB can interoperate as part of a larger system.
- MinimumCORBA should support full IDL so that any CORBA application can be executed on either full CORBA or on minimumCORBA.

- Features that support the dynamic aspects of CORBA are omitted, as the systems for which minimumCORBA is targeted tend to make commitments at design-time rather than at runtime.

There are several features included within the minimumCORBA profile that incur considerable cost, in terms of static ORB size and stub code size, even when they are not being used by the applications. These include TypeCodes, user and system exception features, and inheritance features. [OMG ' 01] chapter 23.

2.3.1 MinimumCORBA Omissions

- **ORB Interface omissions**

A number of omissions are made from the ORB interface, particularly in areas to do with the dynamic features of CORBA.

Operations related to accessing the Interface Repository are omitted since the majority of the Interface Repository itself is omitted.

Operations that facilitate runtime type checking are omitted, as MinimumCORBA is only required to support design time resolution of type checking.

- **DII, DSI and dynamic Anys**

The entire Dynamic Invocation Interface, Dynamic Skeleton Interface, and dynamic Anys are omitted from minimumCORBA, as they support dynamic aspects of CORBA.

- **Interface Repository & TypeCodes**

The majority of the Interface Repository is omitted from minimumCORBA, as it is part of the dynamically typed programming model.

However, part of the TypeCode interface is retained for sending and receiving IDL types that are known at build time. The latter is used with the Interface Repository.

- **Portable Object Adapter**

MinimumCORBA supports a subset of the interfaces and policies defined by CORBA for the Portable Object Adapter.

Features required for reasons of portability and interoperability are included. However, features that support a dynamic mode of POA operation are omitted. What remains is sufficient to achieve portability and interoperability between different minimumCORBA implementations and between minimumCORBA and full CORBA implementations.

- **Policies**

Only a subset of the server side policies are used in MinimumCORBA. Among these policies are all of the default policy values from the CORBA specification. In all other cases, only the policies required for basic ORB operation, portability and interoperability are included.

- **Interoperability**

The minimumCORBA specification has the same conformance criteria regarding interoperability as CORBA. See section 2.5.

- **Language mappings**

MinimumCORBA implementations must support at least one language mapping as defined by the OMG.

The CORBA Architecture and Specification document can be consulted for further details on feature omissions within MinimumCORBA. [OMG ' 01] chapter 23.

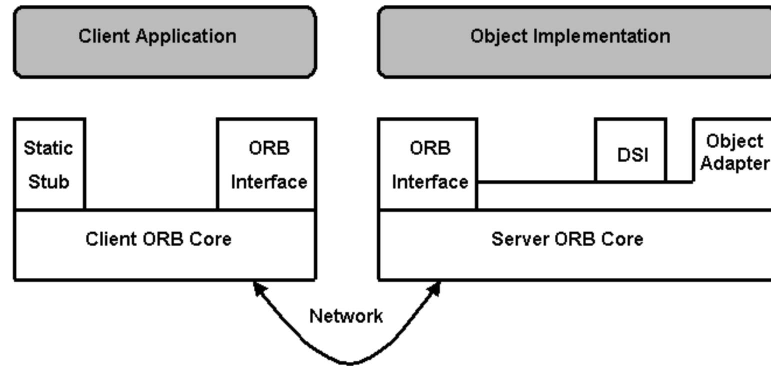

Fig 2.4 Minimum CORBA architecture.
DII and DSI have been removed

2.4 Example Implementations of CORBA Middleware

For illustration purposes, two very different existing CORBA ORB implementations are presented here. TAO is a high-performance, real-time ORB, which offers a fully CORBA compliant implementation, along with some of its own additional features which are added to enhance its middleware capabilities. On the other hand, PalmORB is an ORB that implements only a subset of the CORBA standard features, in order to produce a middleware application that will fit comfortable on a restricted handheld device.

2.4.1 TAO

While experience would suggest that CORBA is well suited for standard RPC style applications that afford "best effort" quality of service (QoS), it is not really suited for high-performance, real time applications for a number of reasons. For example, there is no QoS specification interface to enable clients to indicate the relative priorities of their requests, and no QoS enforcement measures to prevent low priority requests from blocking the execution of higher

priority requests. CORBA also lacks real time programming features that could, for example, notify a client when transport level flow control occurs. If implemented, this could help prevent network congestion problems. TAO is an example of a CORBA ORB implementation that attempts to extend its ORB capabilities in order to cater for some of CORBA's weaknesses.

As well as implementing all of the standard CORBA features discussed in chapter 2.2, TAO adds its own enhancements to the CORBA ORB specification, to enable clients to specify their QoS requirements to it, and to enforce QoS guarantees. TAO also endeavours to provide end-to-end latency, bandwidth and reliability guarantees to distributed applications, by integrating schemes for I/O subsystem architecture optimisations, into its middleware. These I/O subsystems are responsible for mediating ORB and application access to low-level network and OS resources such as device drivers, protocol stacks, and CPUs. In addition to all of the latter extras and enhancements, TAO also possesses a run-time scheduling service. This service is responsible for allocating CPU resources to meet the QoS requirements of the applications that share the ORB endsystem. It provides service guarantees for real time applications with deterministic QoS requirements, and tries to meet service guarantees within the desired tolerance, for real time applications with statistical QoS requirements.

TAO's ORB Core is based on the high-performance, cross platform ACE components [Schmidt ' 98] such asAcceptors and Connectors, Reactors, and Tasks. These components help to provide a suitable connection and concurrency model for predictably sharing the collective processing capacity of ORB endsystem components among the operations in one or more threads of control. This ORB Core can deal with multiple concurrent client requests and server replies, sending request and reply messages to the correct destinations, and passing requests at the server side to the object adapter for dispatch.

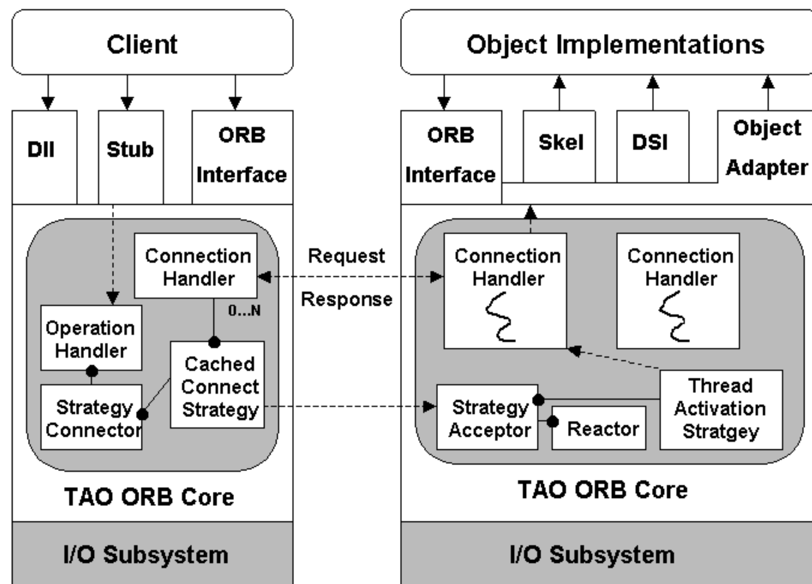

Fig 2.5 Components of the TAO ORB

It is evident that a huge amount of time and resources must have been spent on the construction of the TAO ORB. The entire TAO implementation took 50 person years to build, and spans over 680,000 lines of code. Also, TAO's large footprint requires powerful systems to run it. However its high level of sophistication means that is can be used in extreme mission critical applications. TAO has been employed by Boeing for use within its air traffic control system. Other significant users of TAO include Ericsson, Bellcore, Lucent, Motorola and Siemens. [Schmidt ' 98].

2.4.2 PalmORB

The University of Illinois have constructed PalmORB [Roman ' 99], a stripped down implementation of the CORBA standard for use on handheld devices. The client side CORBA features have been included in this ORB, but all server side functionality has been omitted, since the Palm is used mostly as a client. PalmORB is a very compact ORB, consisting of only 6000 lines of code, and occupying a mere 50KB of

memory in its compiled form. This is ideal for a limited memory device.

The 2K distributed environment has also been implemented at Illinois. 2K provides a user centric organisation of a distributed system, by persistently storing user specific information in objects called an 'environments'. Its operating system has been built on top of another fully implemented CORBA ORB (which is in fact, a modified version of TAO). The 2K system has been designed to dynamically adapt itself to the requirements of specific users that access it, providing each user with a customised view of the system that depends on their environment details. It also provides adaptable proxies, which can alleviate constrained devices from the execution of computation intensive software. 2K can decide what the original device and what the proxy should do, according to the hardware capabilities and available resources of the device.

One of the key ideas behind the 2K system is that it can be accessed from numerous different platforms like Windows NT, Solaris and Palm OS. The fact that the Palm OS platform can be used to access 2K, is of particular interest here. Originally a gap existed between PalmORB enabled handhelds and the adaptable 2K distributed system with its customisable CORBA based services and resources, however a bridging application called PalmShell was specially designed to fill this gap. The result offers flexible and powerful Palm clients, which can dynamically add and remove components from their own tailored environments. Not only can Palm applications interoperate as part of an extensive cross platform distributed system, providing a mobile data access mechanism, but they can also avail of 2K's adaptable proxies which are of particular use to such restricted devices. Proxies enable the Palm to run more powerful applications than could otherwise be deemed possible for such a small device.

PalmORB on its own permits a Palm device to communicate across diverse platforms with relatively simple applications. A more sophisticated type of application that has been enabled on a Palm device as a result of its integration into the 2K environment is that of video streaming. 2K proxies take care of the intensive decompression of video frames, and also reduce the size, rate and colour of these frames to something that can be handled with relative ease by a handheld device.

2.5 IIOP and GIOP

GIOP is defined as the basic interoperability framework for CORBA, that enables all CORBA compliant ORB communication. GIOP is not a concrete protocol that can be used directly to communicate between ORBs. Rather, it describes how specific protocols can be created to fit within the GIOP framework. The Internet Inter-ORB protocol (IIOP), which is specific to TCP/IP, is one solid realisation of GIOP. All CORBA 2.0 compliant interoperable ORBs must implement GIOP and IIOP, and almost all contemporary ORBs do so.

Not only does ORB interoperability require a network communication protocol, it also requires standardised object reference formats. Object references are opaque to applications. In fact, they are even partially opaque to client side ORBs wishing to invoke on the object that the reference refers to, but they also contain information that ORBs need in order to establish communication between clients and target objects. The standard object reference format, called the Interoperable Object Reference (IOR), can store information for almost any inter-ORB protocol imaginable. An IOR identifies at least one supported protocol and, for each protocol supported, contains information specific to that protocol. New protocols can also be added to CORBA without breaking existing applications.

For IIOP, an IOR contains a host name, a TCP/IP port number, and an object key (the opaque part). The object key is used by the portable object adapter at the host referred to in the IOR (which is also the host that created that IOR), in order to identify the target object at that host name and port combination.

There are three versions of GIOP: versions 1.0, 1.1 and 1.2. GIOP and IIOP were initially defined by CORBA 2.0. They were revised with CORBA 2.1 in order to provide support for message fragmentation. A subsequent revision with CORBA 2.3 added support for bi-directional communication. The latter enables role reversal of the client and server, without the need to open a separate connection that may be blocked by a firewall.

GIOP has eight message types.

Message Type	Originator
Request	Client
Reply	Server
Cancel Request	Client
Locate Request	Client
Locate Reply	Server
CloseConnection	Server
MessageError	Client or Server
Fragment	Client or Server

Fig 2.6 GIOP Message types

Request and Reply type messages are by far the most commonly used because they implement the basic RPC mechanism.

The Request message is sent from the client to the server, and is used to invoke an operation or to read or write an attribute.

A Reply message is always sent from the server to the client, and only in response to a previous request. It contains the result of an operation invocation. If an operation raises an exception, the Reply message contains the exception that was raised. [Henning ' 99]

CHAPTER 3

3. ORB DESIGN

A good place to start with the discussion of the ORB design would be to compare it with the two ORBs that have been illustrated in section 2.4, namely TAO and PalmORB. TAO provides a very advanced and complicated QoS oriented CORBA implementation. It implements the entire CORBA standard, along with some of its own added features and enhancements. While TAO is crucial for real-time mission-critical applications, it is obvious that much of its functionality would be absolutely superfluous for use on a handheld device, even if it was small enough to fit. A much more realistic comparison can be drawn between the designs for this ORB and those for PalmORB. By omitting server side functionality, PalmORB uses a subset of the features provided by the MinimumCORBA standard. It does however implement a minimal *client* side ORB that can invoke operations on remote CORBA objects. The ORB design for this thesis includes a similar set of ORB features to PalmORB. The main difference being that a PalmShell type application design, to integrate the ORB into a distributed environment was not included.

3.1 Design Goals

One of the main considerations taken while designing the ORB, was that it had to fit comfortably on to a Palm III device with only 2MB of memory. This memory had to provide storage for application data, like address book information, as well as for dynamic memory. This put limits on the size of the executable code that could be downloaded onto the handheld device, and on the kind of memory intensive structures that could be employed to construct the ORB.

Another key design goal was to implement an extensible ORB, to which additional middleware functionality could be added with relative ease, for future use on a more resourceful Palm device than the Palm III, for instance, a Handspring Visor with 8MB.

3.2 Stripped down CORBA standard to fit Palm III Device

- **Server-side omission**

In a similar fashion to the PalmORB design, the MinimumCORBA client side functionality was included in this ORB, while server side functionality was eliminated. The Palm III presents itself as one of the more primitive of the Palm OS family of handheld devices, and while server side functionality would indeed provide a strong edge on PalmORB, it was deemed unsuitable for the device, which would serve much better as a client in most situations. A decision was therefore made, to concentrate on implementing a solid client side ORB. However an *extensible* ORB design provided a degree of compensation, with its aim to enabling the easy addition of extended ORB functionality for future use on one of the more modern Palm devices.

Another motivation for implementing just a client side ORB, was to avoid the 32KB restriction which puts yet further limitations on the development of applications for Palm devices. This restriction prevents Palm applications whose compiled code exceeds 32KB, from operating correctly on the Palm device. The Motorola DragonBall processor that is used in Palm OS handhelds uses 16-bit memory addresses, which limits it to relative jumps of 32KB. If an application tries to call a function located more than 32KB away from it within the same code resource, the call will fail. While this 32KB restriction can be lifted to an extent, by changing the link order of the source files to avoid jumps exceeding 32KB, an absolute limit of 64KB still remains for any code resource.

Of course, applications exceeding 64KB are also possible for Palm handhelds, but their code must be divided into multiple segments so that no segment exceeds 64KB in its compiled form. This multi segmentation technique also requires the use of special runtime libraries that take up even more space. It was very important therefore to try to avoid using multi segmentation on the Palm III, in order to preserve as much of its 2MB of memory as possible, for permanent data storage, and for the dynamic allocation required by running applications. Omitting server side functionality helped to keep the ORB footprint small. [Henning ' 99]

- **Omission of DII**

Features that support the dynamic aspects of CORBA are omitted by MinimumCORBA, since the systems for which the standard is targeted make commitments at design-time rather than runtime. Thus, a decision was made to abide by MinimumCORBA and omit the DII. [OMG ' 01] chapter 23.

- **Omission of dynamic Anys**

The dynamic Any types outlined in section 2.2.6, are omitted from the MinimumCORBA standard, and hence they are also excluded from this ORB design.

- **TypeCode, exception and inheritance omissions**

There are several features included within the minimumCORBA profile that incur considerable cost, in terms of static ORB size and stub code size, even when they are not being used by the applications. Among these features are TypeCodes, user and system exception features, and inheritance features.

Some limited support for TypeCodes is included as part of the MinimumCORBA standard, however the support provided was not considered to be of particular use to this ORB. Thus, a decision was made to omit TypeCodes entirely from the design, in favor of saving on RAM.

System exception features were minimized for this ORB. In any case, the ORB code generated had to be quite small in order to fit onto a handheld, and small code requires reduced exception processing functionality.

The use of inheritance was minimized within the ORB design, however there were certain cases where it had to be included, one of the main areas being in the design of the stub classes. Inheritance provided a clean and easy way of seamlessly linking application specific stub code with ORB stub code, so that any application which implemented its own stubs could inherit from the ORB's stubs, and hence use the ORB to invoke operations on remote objects. The stub design is discussed in further detail in section 3.3.

- **Other omissions**

Limited memory space on a handheld means that things like deeply recursive routines, large numbers of global variables, and huge dynamically allocated data structures are not handheld device friendly. The ORB was designed to avoid the over use of global variables, and recursive routines. Data structures were designed to be as small as possible, only including features that are required for basic ORB operation.

Figure 3.1 presents the resulting ORB architecture with excess features removed.

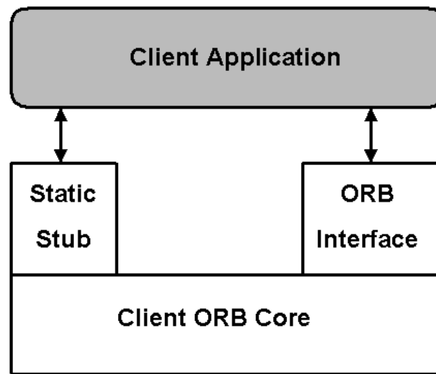

Fig 3.1 Client ORB Architecture

While server side ORB functionality has not been included in this design, figure 3.2 depicts a MinimumCORBA compliant client/server architecture. The Portable Object Adapter offers a subset of the functionality provided by a full CORBA implementation. Items that support the dynamic mode of the POA are omitted, in keeping with the minimum standards aim to omit dynamic aspects of CORBA. The dynamic skeleton interface is also omitted.

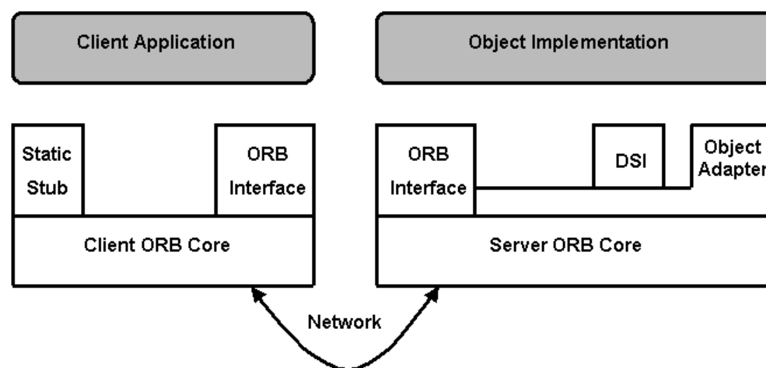

Fig 3.2 Client / Server ORB Architecture

3.3 ORB Class Diagrams

The key components of the ORB design have been spread over the following four different class diagrams. The design has been broken down into these four major areas in order to enhance the clarity of presentation.

3.3.1 ORB Initialisation

Before the client can invoke on a remote operation, it calls on the methods that initialise an ORB instance. Once the ORB has been initialised, the client can call operations that kick-start the ORB into processing remote object references, and then marshalling a message to send to the target object.

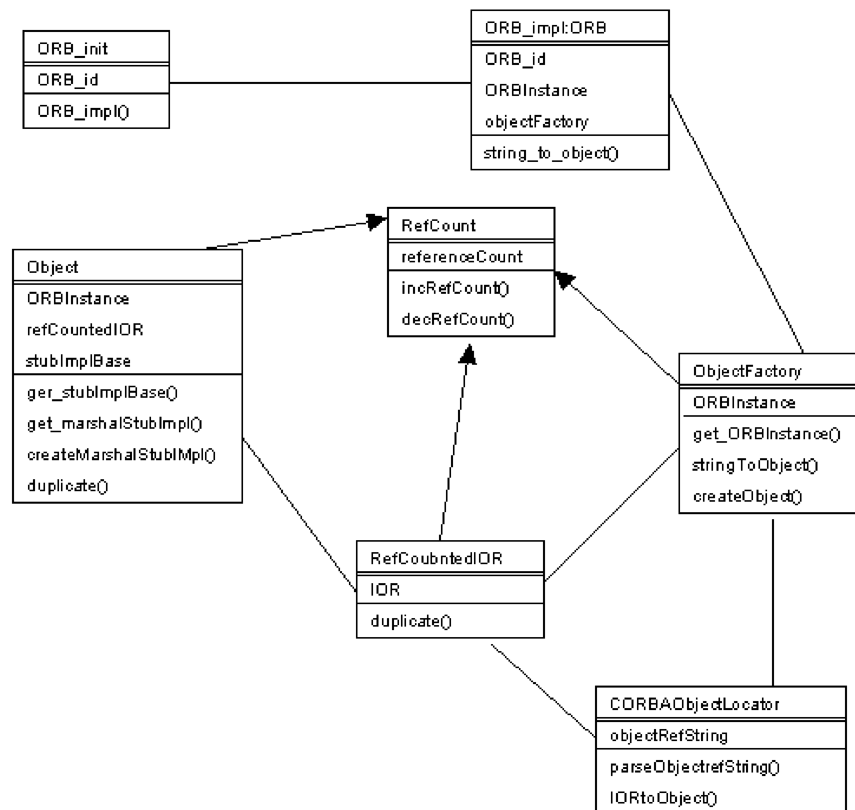

Fig 3.3 ORB Initialisation classes

- ORB_init is called by the Client application. It initialises the ORB run-time and returns a reference to the ORB object. ORB_init expects three arguments, argc, argv and orb_identifier. Argc holds the number of entries in argv, which in turn holds ORB-specific options. Orb_identifier identifies the particular ORB to be initialised. These arguments are useful if an application needs to initialise more than one ORB run-time environment. For example, it is possible for different ORB instances to have different policies and services associated with them, each instance being of particular advantage to a possible subset of client invocation requests. Argc, argv and orb_identifier, were not fully utilised by the ORB developed for this thesis however, since its client application only required a single ORB. However, they were included in the code in order to facilitate the easy extension of the ORB, to produce multiple different ORB instances for a single client application, as mentioned above, for a Palm device with more RAM and processing power than the Palm III.

- The ORB_impl class provides the ORB run-time with access to an object factory class that can be used by the ORB to create an IOR object from an IOR string. This class has been structured to allow for the addition of things like more factory classes, the Initial Service Manager, and the POA Manager [OMG ' 01] chapter 11, for server side functionality, on a more advanced Palm device.

- All proxy objects (i.e. local interfaces that represent remote objects, having an identical signature), inherit from the Object class. This allows generic operations that expect object types, to accept and return object references to these arbitrary proxy interface types. An Object instance contains references to the ORB, the IOR it corresponds to, and to its Stub Implementation, see section 4.7.1. An Object instance can be narrowed to represent its proxy subclass. The proxy retains the ORB, IOR and Stub Implementation references, which are

used to marshal request messages and to send then to the correct destination.

- The ObjectFactory generates IOR objects from their string representations. The ORB uses these IOR objects to direct request messages to the correct destination. To do this, the ObjectFactory implements the StringtoObject operation, that calls the CORBAObjectLocator class, which converts the string to an object. This conversion process is discussed in further detail in the implementation section.

- The RefCount class provides a painless mechanism for keeping a count of the number of references that have been made to objects that are referenced frequently throughout the ORB code. These reference counts can then be monitored, so that objects can be deleted when their count drops to zero, thus freeing up valuable memory.

- The RefCountedIOR class contains a reference counted IOR object reference.

3.3.2 GIOPMessaging

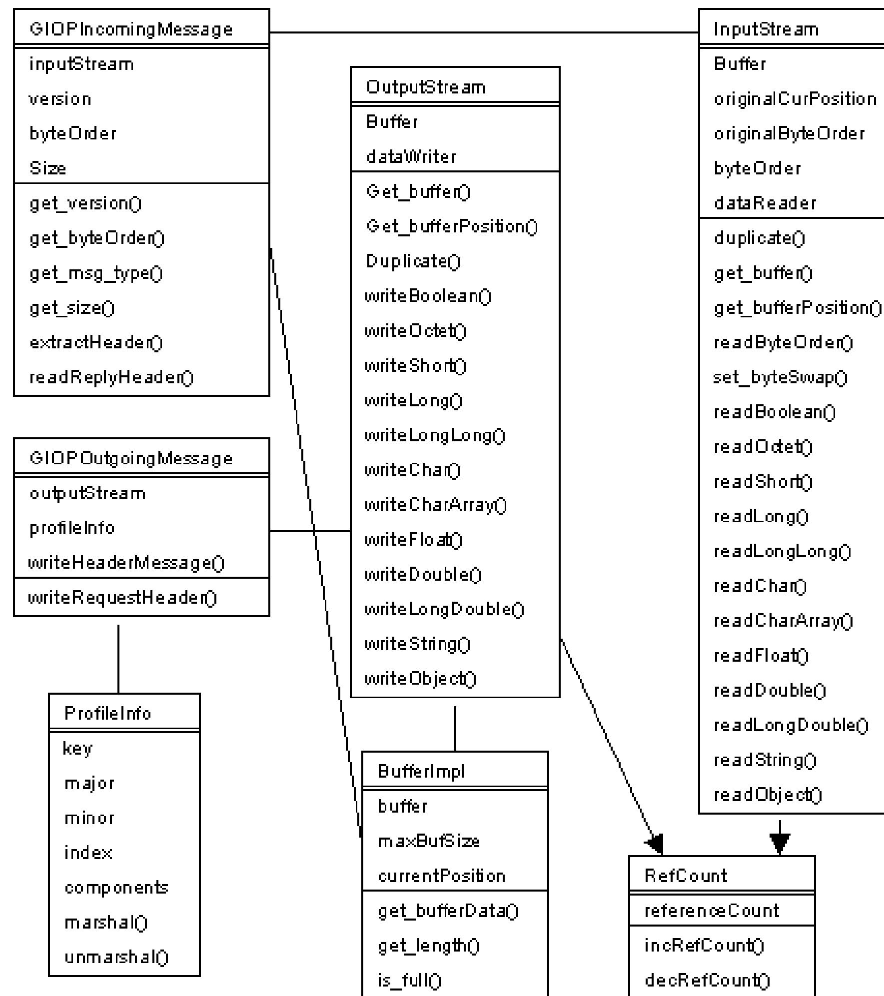

Fig 3.4 GIOP Messaging classes

- The GIOPOutgoingMessage class is responsible for writing GIOP request messages to the output stream. These messages are then sent from the output stream to the server side ORB.

- The GIOPIncomingMessage class is responsible for reading, from the input stream, a GIOP reply message that has been sent to it by the server side ORB. This message is then interpreted to see if the corresponding remote object request has been executed successfully, or if problems were encountered.

Details of the GIOP request and reply message formats are discussed in detail in section 4.7.3.

- The OutputStream class contains methods to align all data types on their natural boundaries when writing them to the output stream. This kind of alignment is necessary so that the message can be correctly de-marshalled when sent to the server side. The InputStream class contains methods that can read naturally aligned data types, from reply messages received into the input stream.

Data alignment is discussed in further detail in section 2.2.8.

- The input and output streams use instances of the BufferImpl class to store data. This class also contains useful methods and holders for determining buffer data positions, in order to facilitate reading of messages from, and the writing of messages to, the input and output streams.

- The ProfileInfo structure is constructed from the profile information retrieved from the IOR string. Profile information is included in all client request messages. When a request message is picked up by the server side ORB, the POA uses the profile information to determine which object the request is intended to invoke on.

3.3.3 Stub Implementation

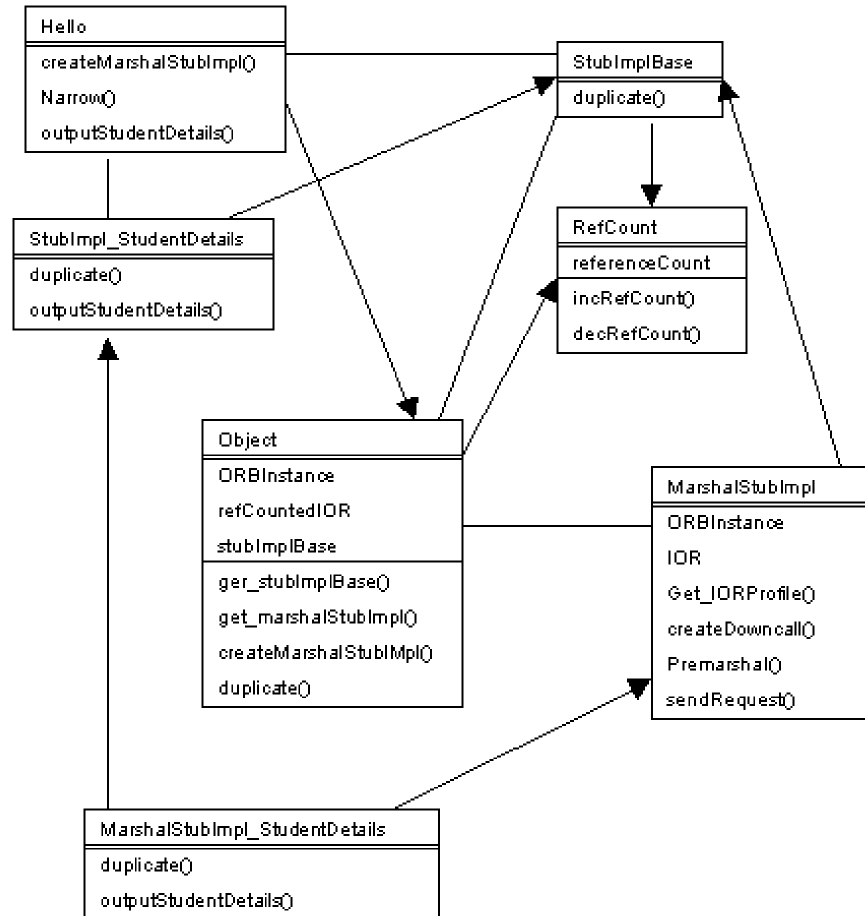

Fig 3.5 Stub Implementation classes

- StudentDetails is the name of the target object that can be invoked remotely. The StudentDetails interface class (not shown in figure 3.5) on the client side, provides the interface between the client application and the stub code that passes a request for a remote object to the ORB core. It contains the outputStudentDetails operation that has a signature identical to that of the remote operation being targeted.

- The StubImplBase class provides a reference counted base class for all stub implementation classes. A stub is a client side function that allows a request invocation to be made via a normal looking local function call.

- The MarshalStubImpl_StudentDetails class implements the client application specific proxy code that initiates the passing of a remote object invocation request to the ORB. MarshalStubImpl_StudentDetails inherits from the StubImpl_StudentDetails class, which provides a facility for keeping a reference count on the former's instances. It also inherits from the MarshalStubImpl class, which is a *generic* ORB stub class which implements operations that pass a remote object request down to the ORB core, for marshalling and dispatching.

3.3.4 ORB Downcalling

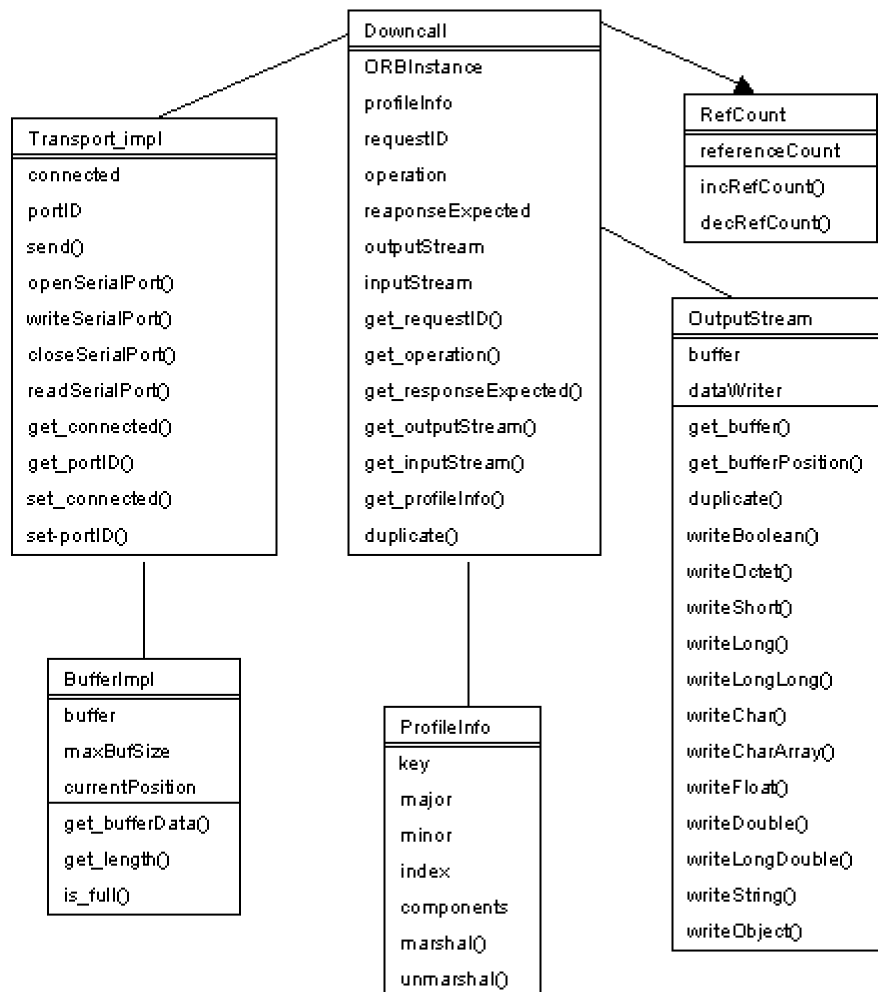

Fig 3.6 Downcall classes

- The Downcall class gives a request message an identification number. It also prompts the writing of a request message to the output stream, and subsequently calls the transport layer to send the message to the server side ORB.

- Transport_impl implements methods to open, write to, read from, and close the Palm device's serial port. This serial port provides a link from the Palm to a remote application that implements a socket connection, for IIOP, to a server side ORB, as detailed in section 4.2. Transport_impl also interprets replies to request messages that are received from the server side ORB, in order to determine whether or not the request was successfully honoured. Transport_impl effectively acts as the ORBs transport layer, sending messages to, and receiving messages from a server side ORB.

3.4 Extensibility of ORB Design

- Section 3.3.1, on the ORB initialisation design, explains how the initialisation code was designed in order to facilitate the easy extension of the ORB, to enable single clients to use different ORB instances to deal with various remote object invocations, that could require slightly varying ORB characteristics, during the same execution session.

The ORB initialisation classes were also structured to simplify the addition of things like more factory classes, the Initial Service Manager, and the POA Manager [OMG ' 01] chapter 11, to provide server side functionality for a Palm device with more resources available to it than the Palm III.

- This ORB was designed to support GIOP 1.1 and 1.2. However, the structure of the GIOP class would allow for its easy extension, to include future versions of the standard, or indeed to

support the older version if required. This would merely require the insertion of a new *case* into the GIOP messaging method, to enable the construction of a message under the new version of GIOP. The IOR and reply message interpretation code could also be extended in a similar fashion.

- At the time of writing, this PalmOS ORB was only capable of interpreting IIOP messages related to the TCP/IP communication protocol. However it did have the capacity to read IORs containing multi protocol data. This was achieved by creating a sequencing data structure into which the IOR profile data could be read. Each entry in the sequence would contain information relating to a different communication protocol. Thus, extending the ORB to interpret communication protocols other than just TCP/IP, would simply require the insertion of additional C++ functions, to interpret these protocols.

- The transport class was put in place within the ORB code in order to implement the serial / socket connection. Replacing the latter with a pure socket connection would simply involve modifying the transport class code to implement pure socket functions, rather than serial functions.

CHAPTER 4

4. CODE IMPLEMENTATION

Before delving into the actual implementation details of this ORB middleware, it is worth emphasising the fact that implementing such a low level application is a particularly frustrating and exhausting task. To make matters worse, little previous knowledge of the CORBA standard or indeed of the C/C++ programming languages was had prior to undertaking the project, which created even more difficulties. However, in the end, hard work and persistence overcame these obstacles, and all to make life easier for the writers of distributed applications!

4.1 Development Environment

As previously mentioned, a Palm III device, with a Motorola MC68328 "DragonBall" processor was used to develop the middleware for the purposes of this thesis. This provided a big challenge, since, while a Palm III possesses much more in terms of resources than some of its predecessors, it is still limited to a rather diminutive 2MB of memory. The ORB had to be made small enough, to fit easily onto this Palm III device, while leaving a reasonable amount of free memory available for the development of other regular Palm applications, and indeed for other client applications to be used in conjunction with the ORB itself.

Metrowork's CodeWarrior for Palm Computing platforms was used for the code development. CodeWarrior, which is the official development environment supported by Palm Computing, provides a number of useful development tools along with the basic source code editor. Those used for the purposes of this project included:

- **Constructor for Palm OS**

This is a resource editor with a graphical interface, that facilitates the development of Palm user interface elements. These elements are combined with the source code to create the finished product.

- **Palm OS Emulator**

This is also known as POSE. It imitates most of the hardware and software functions of an actual Palm handheld, and can be downloaded onto a desktop computer for use. One of POSE's most useful features is its accurate emulation of the processors used in a range of Palm devices (including the Palm III). This means that real Palm OS applications can be loaded directly onto POSE for debugging purposes. Debugging on the emulator is faster than on the actual Palm device, and of course, it provides worthwhile savings on batteries!

4.2 Network Connectivity

The Palm III device for which the ORB was developed did not possess a direct networking facility to enable TCP/IP connectivity to other computers. In order to emulate such a network connection, a combination of serial and TCP/IP communication capabilities were employed. The Palm device was connected to COM1 of a desktop computer via its serial cable. TCP/IP Sockets were implemented on the desktop to provide the applications on the Palm with indirect access to all other TCP/IP enabled devices on the network. It is obvious that the latter implementation was not ideal, in that it reduced the mobility of the Palm since it had to be physically connected to a TCP/IP enabled device in order to communicate with other devices on the network. However, the ORB was developed to be extensible and to operate on any Palm OS device, and so, it would be relatively easy to install it on a larger more up to date, TCP/IP enabled Palm device.

4.3 Event Driven Programming

Palm OS applications are event driven, receiving events from the OS and either handling them or passing them back to be handled by the OS itself. An event structure describes the type of event that has taken place (for example, a stylus tap on a screen button), as well as information related to that event, such as the screen coordinates of a stylus tap. During a normal application launch, execution is passed to the application's event loop, which retrieves events from the event queue and dispatches them according to the type of event. The event loop passes most events back to the OS, because the system already has facilities for dealing with common tasks such as displaying menus or determining what button on the screen was tapped. Those events that are not handled by the OS go to the application's own event handler, which either handles the events if they are interesting to the application, or passes them back to the event loop.

An application event loop was incorporated into the client application that was written to use this ORB. A CodeWarrior facility was used to generate the bones of an application loop. Functions were then written to handle the event loop events that required processing by the application. These events were handled in a way that was specific to the client application's requirements. For example, if the remote invocation application's main GUI icon was tapped, that event prompted the opening of the main application form, which indicated what objects were available for invoking on remotely. If such an object was selected, this prompted the initialisation of the ORB. The ORB subsequently interpreted that object's IOR and marshalled a GIOP message, which it then sent to the correct destination.

4.4 Memory Management for the Palm Device

Memory management is very important for a Palm OS device with limited memory resources. It is imperative that as much memory as possible be available for allocation at all times. The following paragraphs outline a number of memory management features that were put in place, in order to minimise the possibility of memory leaks, and to ensure that memory would be freed as soon as it was no longer required by either the ORB, or the client application.

4.4.1 Memory Allocation of Strings

For normal desktops, writing to incorrect memory addresses can cause dramatic application failures, but won't normally affect permanently stored data, because it resides on a separate storage device from the systems main memory. For Palm OS devices on the other hand, the same RAM is used for both data storage and for dynamic memory.

Using RAM for storage provides faster access to data, however, a rigorous means of memory management had to be availed of to prevent the possibility of corrupting permanently stored data. Palm OS APIs were used for allocating memory for strings, in order to prevent the loss of permanent data.

Two different types of memory manipulation functions for Palm OS devices are available for allocating memory for strings, and other straightforward data structures. These are *pointer* functions and *handle* functions. The MemPtrNew and MemPtrFree functions are provided to allocate and de-allocate pointers, in place of the C standard library calls, malloc and free. However, a decision was taken to implement the handle functions, based on the following argument:

The Palm operating system has the ability to efficiently manage the small amount of dynamic RAM it has available, by shifting chunks of data around with an aim to creating large chunks of contiguous space.

This is obviously preferable to having lots of small fragments of free space scattered around the memory, which cannot be used if a new data record fails to fit into any of them individually. Pointers use only unmoveable memory chunks, and so they do not avail of the latter memory management facility. However, handle functions allow applications to manipulate chunks of memory that may be moved by the operating system. If the operating system needs to allocate memory, it can move handles around until there is enough contiguous memory for the new data to be allocated. New memory can be allocated with the MemHandleNew function and freed using the MemHandleFree function. Also, because the operating system may freely decide to move the memory associated with a handle at any time, the handle must first be locked with the MemHandleLock function before it can be read from, or written to. While the handle is locked, the operating system will not move its memory to another location. The handle can subsequently be unlocked using the MemHandleUnlock function, when the read or write operation has been completed. This approach greatly increases the efficient use of the limited memory on a Palm device. [Foster ' 00]

In addition to using Palm OS handles for allocating memory for strings, a String_var class has also been implemented to provide a memory management wrapper for char *. [Henning ' 99]. Indeed, the internals of the String_var allocation mechanisms utilise memory handles to ensure the safe and efficient allocation of memory for strings. String_vars can be used in situations where keeping track of the number of references to allocated stings is difficult for the ORB programmer. This class stores a pointer to a memory allocated string in a private variable, and takes responsibility for managing the string's memory. The String_var uses a destructor to ensure de-allocation of memory when a string that it wraps goes out of scope.

4.4.2 Memory Allocation of Classes

The new and delete operators in C++ facilitate the dynamic allocation of memory for structures whose sizes aren't known until runtime. Since support for the latter operators was included in the version of CodeWarrior used to develop this ORB (though not in previous versions), they were both used in the allocation and de-allocation of memory for class structures.

However, as for string referencing, creating references to and removing references from such classes complicates the issue of *when* dynamic memory can be de-allocated. Normally, a programmer must keep track of the number of references to an object, so that the object can be explicitly de-allocated when there are no remaining references to it. This point represents a memory management issue that puts extra workload on the programmer, and runs the risk of producing memory leaks.

A set of smart pointer classes known as var types, that use the same principles as String_var, have been implemented [Henning ' 99], to alleviate the burden of having to explicitly de-allocate variable-length structures and to make memory leaks less likely. This works by associating a _var class with each normal class type. The class provides the required functionality, while the _var class acts as a memory management wrapper around the former. This means that the _var class takes care of de-allocating normal instances at the appropriate times. This de-allocation process uses reference counters to keep track of the number of references to each of its class instances.

4.4.3 Reference Counting

Each CORBA object has a reference count that indicates the number of local references that refer to it. This reference count is incremented each time a new local reference to the object is created, and

decremented when a local reference to the object is deleted. The idea behind this is that once the object's reference count falls to zero, it is automatically de-allocated. This helps to prevent memory leaks.

- _ptr types: When assigning between two _ptr type object references , the reference count is incremented explicitly using a _duplicate function. A _duplicate function has been implemented for all of the objects that require its use. A release function is used to explicitly decrement the reference count for _ptr types.

- var types: _var types can handle reference counts automatically. Therefore, for objects with associated _var types, the burden of explicitly incrementing and decrementing reference types is eliminated. In a _var type, the '=' operator is overloaded, so that when such a type is assigned a value using the '=' operator, its reference count is automatically incremented. Also, when the _var type goes out of scope, its reference count is automatically decremented in its destructor. [Henning ' 99]

4.4.4 Directional Attributes

In compliance with the CORBA standard, three types of parameter attributes were made available to clients who wish to pass parameters as part of a remote invocation. These are:

- **in**

The in attribute indicates that the parameter is sent from the client to the server.

- **out**

The out attribute indicates that the parameter is sent from the server to the client

- **inout**

The inout attribute indicates a parameter that is initialised by the client and sent to the server. The server can modify the parameter value, so after the operation completes, the client parameter value may have been changed by the server.

Directional attributes are required for two reasons. They are necessary in order to know when a parameter must be sent from a client to a server or vice versa. This enables some savings in transmission costs. Also, directional attributes are required to assist in memory management. For example, the client application implemented as part of this thesis includes inout string parameters to be read and/or modified by the target object on the server side. This means that ownership of the string parameters in this application is temporarily given to the server so that the server can de-allocate and re-allocate the strings in order to modify them. After the invocation however, ownership of the strings is returned to the client. To ensure de-allocation of the string parameters after target object invocation, the strings are declared as String_vars, so that they will be *automatically* de-allocated when they go out of scope.

Memory management for operation parameters varies with the direction and type of parameter. Directional attributes control whether the client or the server is responsible for allocating and de-allocating memory for parameters. Memory management details for in and out parameters can be obtained from most common CORBA textbooks. [Baker ' 97]

All of the above memory management mechanisms help to prevent memory leakage.

4.5 Use of Templates

Template classes, which provide one of C++'s most powerful capabilities for software reuse, were implemented to enable the specification, within a single code segment, of an entire range of related classes.

At runtime, templates are used to create the _var objects, described in section 4.4.2. Rather than reproducing similar _var type code for all objects that require a _var wrapper, these objects were divided into object groups containing similar characteristics. A template was then produced for each such group. At runtime, any object requiring a _var wrapper for memory management purposes, is passed into its respective _var template class, where it is subsequently assigned a private variable that holds a pointer to the class type that the wrapper encapsulates. This approach reduced the amount of code that had to be written and also minimised the resulting code footprint.

4.6 Big Endian Vs. Little Endian

An important fact that had to be kept in mind during the Palm OS program design and implementation, was that all data entering or leaving the device was arranged in Motorola's big-endian byte order. In other words, multi-byte data types such as long integers were arranged with their most significant bytes at the lowest memory address, and vice versa. This detail was very important when connecting to Intel-based machines, all of which use little-endian byte ordering.

The CORBA standard states that byte swapping should only occur on the receiver side of a sent message. A device sending a message, can therefore send using its own natural byte-order format, however, a special message field is set aside to allow the sender to indicate the byte-order that they use. At the receiver end, this byte-order field is examined. If the byte-order for sender and receiver are the same, the receiver has nothing to worry about, but if the byte-orders are different,

then the burden is on the receiver to swap the bytes so that the intended message can be correctly interpreted.

A byte swapping facility was introduced into the ORB code in order to deal with the inevitable byte-ordering problem that arose when the Motorola processor attempted to communicate with an Intel device. It was implemented using bitwise left shift and right shift operators, and the bitwise-inclusive-or operator. [Lazzarotto]

4.7 Request Invocation

A client is an entity that invokes a request on a CORBA object.
A client application was written for the Palm device. This client manipulates a remote object by sending messages to it. The ORB sends the message to the object whenever the client invokes an operation on the object. To send the message, the client needs to hold an object reference (IOR) for the object. The object reference uniquely identifies the target object and encapsulates all of the information required by the ORB to send the message to the correct destination. IORs are discussed in further detail later in this chapter. [Henning ' 99]

The entire request invocation mechanism was implemented to be completely transparent to the client, for whom a request to a remote object looks like an ordinary method invocation on a local C++ object:

```
Student->outputStudentDetails(name, student_number);
```

In the above example, Student corresponds to the remote object, outputStudentDetails to the remote operation, and name and student_number are the parameters that were passed to the remote operation. A client side call to the remote operation (as above), results in a call to a client side stub, which in turn passes the request to the ORB, which then marshals the object name, operation and parameters, and sends the resulting request package to the correct server machine,

where it was de-marshalled and serviced. Stubs and skeletons are discussed in detail in the following section. [Baker '97]

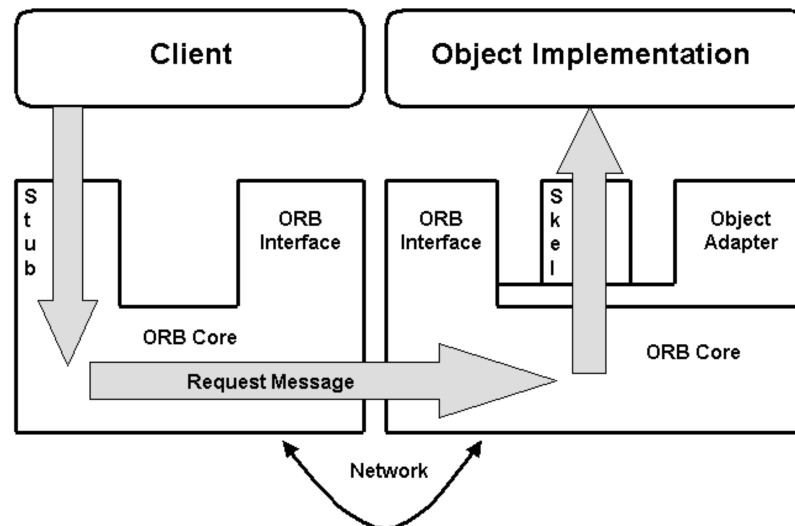

Fig 4.1 Request Invocation

4.7.1 Stubs and Skeletons

To invoke an operation on a remote object the client first instantiates a proxy object in its own address space. The proxy is a C++ instance that provides the client with an interface to the target object. It contains references to the ORB, the IOR and to the stub Implementation. The signature on the proxy interface is the same as the signature on the implementation of the remote object on the server side.

A reference to the proxy object is analogous to a C++ class instance pointer, but denotes an object implemented in a different process, and on another machine. When the client invokes an operation on the proxy via the proxy reference,

```
Student->outputStudentDetails(name, student_number);
```

the proxy’s stub then sends a corresponding message to the remote servant via the ORB.

```
MarshalStubImpl_ Student::outputStudentDetails (CORBA::Long& _ob_a0, char*&
_ob_a1, char*& _ob_a2)
```

It is actually the ORB that marshals the message, locates the server and establishes network connections transparently on behalf of the client. [Baker ' 97]

```
void  MarshalStubImpl_ Student:: outputStudentDetails (CORBA::Long& _ob_a0,
char*& _ob_a1, char*& _ob_a2)
{
        OB::Downcall_var _down = createDowncall("outputStudentDetails ", true);

        OB::OutputStreamImpl* _out = _preMarshal(_ob_down);

        _out -> write_long(_ob_a0);
        _out -> write_string(_ob_a1);
        _out -> write_string(_ob_a2);

        _postMarshal(_ob_down);
        _request(_ob_down);
}
```

The above code creates a down call to the ORB to marshal a request message. It also writes the parameters that are to be passed to the remote object to the output stream.

Once the request data has been marshalled and aligned on the output stream buffer, it is then sent by the ORB via a serial / socket connection to the server implementing the required object. The destination host and port details are obtained from the object’s IOR. [Baker ' 97]

4.7.2 Creating an Object Reference from an IOR

In order to use an object reference, the ORB takes the string representation of the IOR, that is provided by the server that supports the object to be invoked. This IOR string, which is initially in hexadecimal format, is then converted into an IOR object using the object_to _string ORB operation. This involves firstly converting the IOR string from hexadecimal to decimal format. The decimal string is then converted to its ASCII representation by casting it to an unsigned char type. The contents of the unsigned char string are then used to create the IOR *object*. From this point on, the IOR is in a format that can be used directly by the ORB. An IOR object generally contains three major pieces of information. In this case, the Repository ID information was omitted from the IOR object since an interface repository facility was not implemented. The two pieces of information contained by this minimal ORB's IOR objects were as follows:

- **Endpoint information**

This field provides the ORB with all of the information it needs in order to establish a physical connection to the server implementing the target object. The endpoint information indicates which protocol to use when attempting to invoke an operation on the object represented by the IOR. It also contains physical addressing information appropriate for a particular transport. Since this ORB uses IIOP only, the endpoint information contains an Internet domain name or IP address and a TCP port number. The Endpoint Information field could also contain the address of an Implementation Repository to be consulted to locate the correct server on which the requested object runs. This extra level of indirection would enable server processes to move from one machine to another, without breaking existing references held by clients. However, due to the memory constraints of the Palm III device used here, the Implementation Repository was omitted from this ORB, and so the Endpoint Information field directly contains the address and port number of the server that implements the object. Thus if a server process moves location to a different machine, a new IOR containing

the new object location details would had to be provided for the Palm device.

The CORBA standard also allows information for several different protocols and transports to be embedded in the reference, permitting a single reference to support more than one protocol. Since the ORB developed for our Palm device used only IIOP, the TCP/IP information alone was extracted from the IOR for use.

- **Object key**

Unlike the endpoint information, which is standardised, the object key contains proprietary information. The arrangement and usage of this information is unique for different ORB implementations. All ORBs have an application-specific object identifier that is embedded inside the object key by the server, when the server creates the reference. When the object identifier is received by the server-side ORB from a client request message, it is used by that ORB and its object adapter (or one of its object adapters) to identify the target object in the server, upon which an operation invocation had been requested, from within the message. The client-side simply sends the key as a transparent block of binary data with every request it makes. Since for all intents and purposes, the key remains an opaque block of information to the client, it does not matter that the reference data is in proprietary format. It is never looked at by any ORB, except the ORB hosting the target object (i.e. the very ORB that created the object reference with the proprietary object key in the first place).

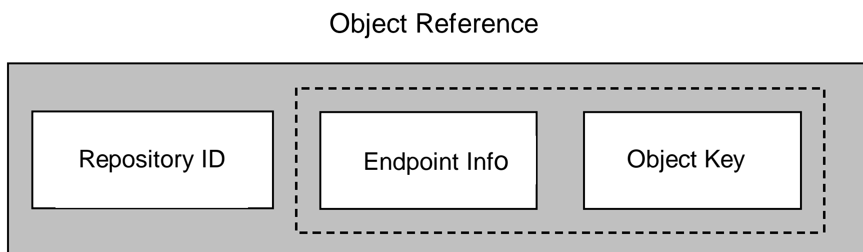

Fig 4.2 Object Reference contents

The following pseudo IDL shows how the information required to send a request to the correct target object, is encoded within an IOR object that has been generated by the ORB from an IOR string.

```
module IOP{

        typedef unsigned long           ProfileId;
        const ProfileId                 TAG_INTERNET_IOP=0;
        const ProfileId                 TAG_MULTIPLE_COMPONENTS=1;

        struct TaggedProfile{
                ProfileId               tag;
                Sequence<octet>         profile_data;
        };

        struct IOR{
                string                  type_id;
                TaggedProfile           profile
        };

};
```

[OMG '01] chapter 13

struct IOR, which is the main data type within the generated IOR object, defines the basic encoding of an IOR as a string followed by a sequence of profiles. The type_id string contains the interface type of the IOR in a standard CORBA format. The profiles field specifies the IIOP protocol profile that is to be used to send all messages to the object referenced by the IOR. An ORB that supports multiple profiles could contain a sequence of protocol profiles within the profiles field, one for each protocol supported by the target object and the client itself.

This ORB supports only IIOP, so the type id is followed by a *single* profile containing a structure of type TaggedProfile. A tagged profile contains a tag field and an octet sequence that contains the profile body

identified by the tag. As an example, for IIOP 1.1 and IIOP 1.2, the tag is TAG_INTERNET_IOP (zero), and the profile_data member encodes a structure of type IIOP::ProfileBody as shown below.

```
Module IIOP{

        struct Version {
                octet major;
                octet minor;
        };

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

};
```

[OMG ' 01] chapter 15

This ORB supports CORBA versions 1.1 and 1.2. The Version field enables the ORB to identify what version of CORBA generated the IOR. This information is used by the ORB when deciding how to marshal a request to a server object.

The object host and port information is used to send CORBA requests over TCP/IP. And, as already mentioned, the object_key field, which is included in all CORBA request messages, contains information on how to identify the POA and that servant that implements the object at the server side.

4.7.3 Implementation of IIOP and GIOP

Support for GIOP and IIOP versions 1.1 and 1.2 was implemented within the ORB code. The implementation followed the CORBA specification exactly. This support facilitates the transformation of IOR strings to IOR objects, and the generation of request messages, for inter-ORB communication using either version. It also facilitates, for both versions, the interpretation of reply messages received from the server-side, which determines if a remote operation invocation has been successful or not. The CORBA Architecture and Specification [OMG ' 01] can be consulted for details of version 1.0 of the latter standards, if desired.

Figure 4.3 shows the basic structure of a GIOP 1.1 or GIOP 1.2 message.

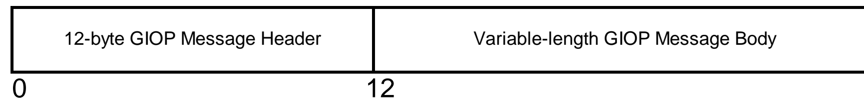

Fig 4.3 Basic structure of a GIOP Message

Request and Reply type messages only, were implemented since these two are by far the most commonly used, and because they alone implement the basic RPC mechanism. Also, support for fragmentation (introduced in GIOP 1.1) and for bi-directional communication (GIOP 1.2) was omitted. However, it was felt that given more time, and a more powerful PalmOS based PDA, both of these features could have been implemented without undue effort.

4.7.3.1 GIOP Message Header

The following describes the implementation of the message header in pseudo IDL. It is the same for versions 1.1 and 1.2:

```
module GIOP {

        struct Version {
```

```
                octet      major;
                octet      minor;
        };

        enum MsgType_1_1 {
                Request
        };

        struct MessageHeader_1_1 {
                char                    magic[4];
                Version          GIOP_version;
                Octet                   flags;
                Octet                   message_type;
                Unsigned long           message_size;
        };

};
```

[OMG ' 01] chapter 15

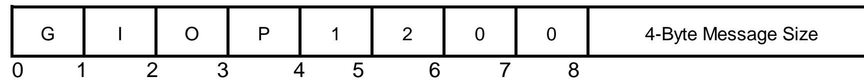

Fig 4.4 GIOP 1.2 message header

The message header layout is as follows:

- The first 4 bytes of a message header are always the characters GIOP, which indicate that the message is a GIOP message. They also serve to define message boundaries.
- The 4th and 5th bytes are the major and minor version numbers represented as 8-bit binary values.

- The 6th byte is a flag byte. The least significant bit of the flag byte is used to specify whether the remainder of the message is in big-endian or little-endian format. The Palm device uses big-endian (indicated by a 0). The second-least significant bit indicates whether or not a message uses fragmentation. A value of 1 indicates that the

message is a fragment of a larger message, and that there are more fragments to follow. A value of zero (as in the above case, figure 4.4) indicates that the message is a complete message or that it is the last message in a sequence of fragments.

- The 7th byte indicates the message type. For example, the value 0 indicates a Request message.

- Bytes 8-11 contain a 4-byte unsigned value that indicates the size of the remainder of the message, which constitutes the GIOP message body, (e.g. for a Request message, these bytes would indicate the size of the Request Header and the Request Body combination).

The GIOP message body consists of the message header and body type, that are specific to the type of message that it encompasses. For example, for a Request message, the GIOP message body consists of the Request message Header and the Request message Body.

The implementation of Request and Reply messages is described in the following subsections.

4.7.3.2 Request Message Format

The Request message formats for GIOP 1.2 and 1.1 differ slightly, however the ideas behind both are similar. For this reason, a description of how the more recent version 1.2 was implemented, has been included in this document, while details of the implementation of version 1.1 have been omitted. If desired, they can be obtained form the CORBA Architecture and Specification document [OMG ' 01].

The Request message consists of three parts as shown in figure 4.5.

Fig 4.5 GIOP Request message

The Request message, which contains a Request header and a Request body, follows the GIOP header. The Request header is structured as follows:

```
Module GIOP {

        struct RequestHeader_1_2 {

                unsigned long                   request_id;
                octet                           response_flags;
                octet                           reserved[3];
                TargetAddress                   target;
                string                          operation;
                IOP::ServiceContextList    service_context;
        };
};
```

[OMG ' 01] chapter 15

The fields within the Request header are as follows:

- **request_id**

The client uses this field in order to relate a request with its response. The request_id is set to a unique number when a request is being sent. Reply messages also have a request_id field, in which they include the identification number of the request that they are responding to. This means that the client can have replies for more than one request outstanding at any one time.

- **response_flags**

The response_flags field can be set to indicate whether or not a reply message is to be expected from the server-side of an invocation.

- **reserved**

As part of the CORBA standard, the three bytes of the reserved field are reserved for future use and are always set to zero for GIOP 1.1 and 1.2.

- **target**

The target field is a union type, which identifies the object that is the target of the invocation. It contains a key address, profile address and reference address.

The key address contains the object_key field, obtained form the transport specific IOR generated by the target objects server. It is only meaningful to the server and is not interpreted by the client.

The profile address field is the transport specific GIOP profile selected from the target's IOR by the client ORB. It indicates to the server side ORB the type of transport being used by the client side ORB.

The reference addressing information contains the full IOR of the target object. It is used by the server to identify the POA and servant of the object on which an operation is to be invoked.

- **operation**

This field contains a string that indicates the name of the operation to be invoked.

- **service_context**

The service_context field contains ORB service data being passed from the client to the server. It could contain, for example, data for transaction services, codeset negotiations services, or bridging services. This field is not used by this ORB. It is set to the value of 0, so that it will be skipped over by the ORB at the server side.

In GIOP version 1.1, request bodies *immediately* follow the Request Header. In GIOP version 1.2, the message body is always aligned on an

8-octet boundary. Since GIOP specifies that the maximum alignment for any primitive type is 8, this guarantees that the request body will not require any re-marshalling if the message header or request header are modified. The data for the request body includes all in and inout parameters, marshalled as if they were members of a structure, in the order in which they are specified in the operations OMG IDL definition, from left to right.

The request body for the operation used to demonstrate this CORBA ORB:

void outputStudentDetails(inout string name, inout long student_number);

Would be equivalent to the following structure:

```
struct outputStudentDetails _body{

        string      name;            //leftmost inout parameter
        long        student_number; // rightmost parameter

};
```

4.7.3.3 Reply Message Format

A Reply message is sent from a server to a client in response to a client's Request message, provided that the response_expected flag of the request is set to true. Since this minimal ORB implements only client side functionality, it does not need to generate Reply messages to send to other hosts. However, the ORB implementation is capable of interpreting and processing Reply messages received from the host machines of objects that it invokes.

As with Request messages, Reply messages in GIOP versions 1.1 and 1.2 are slightly different. Functionality to interpret both message versions has been included in the code, but only version 1.2 is

described in detail in this document. For details on version 1.1, see the CORBA Architecture and Specification document [OMG ' 01] chapter 15.

Like a Request message, a Reply message is also made up of three parts. The Reply header and body follow the GIOP header, and together form the GIOP message body.

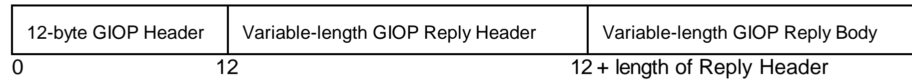

Fig 4.6 GIOP Reply message

The following defines the Reply header structure:

```
Module GIOP {

        Enum ReplyStatusType_1_2 {

                NO_EXCEPTION,
                USER_EXCEPTION,
                SYSTEM_EXCEPTION,
                LOCATION_FORWARD,
                LOCATION_FORWARD_PERM,
                NEEDS_ADDRESSING_MODE

        };

        struct  ReplyHeader_1_2 {

                Unsigned long                   request_id;
                ReplyStatusType         reply_status;
                IOP::ServiceContextList service_context;

        };

};
```

When a Reply message is received from a server by the Palm client, its reply_status field value is extracted in order to determine whether or not the remote operation invocation was successful. For example, a NO_EXCEPTION value would indicate that the request completed successfully, while a USER_EXCEPTION request would imply a user exception.

Extraction of the Request_id field from the Reply message allows the Palm client to associate that Reply with one of its own Request messages.

The Reply body would contain the return value of the remote operation, followed by all of the operations out and inout parameters if applicable.

As with a Response body, GIOP 1.2 also aligns Reply bodies on an 8-byte boundary rather than directly after the Reply header.

CHAPTER 5

5. EVALUATION

This chapter critically examines the design and implementation of the application that was developed for the purposes of this thesis, which aimed to produce an ORB for operation on a restricted Palm OS device. Strong areas are highlighted, and areas that offer room for further development are also discussed, along with suggestions on how to address them.

5.1 Critique of Design

- The key to designing middleware for the Palm III device lies in keeping the code small and avoiding the use of complex structures and mechanisms that eat heavily into memory. This essential design factor can reason out many of the shortcomings associated with the ORB design. For example, the use of server side functionality would have been particularly motivating, especially since it was not included in PalmORB. This would have given our ORB a definite edge over the latter. Also, considering the speed at which modern technology is continually increasing the power of electronic devices, it is only a matter of time before the notion of a two-way communications, pocket sized device, will be capitalised on. Had our ORB been developed on one of the newer Palm devices, such as a Handspring Visor bearing 8MB RAM and a more powerful processor, then implementing server side capabilities would have been a lot more feasible. As mentioned on several occasions throughout this thesis, a valuable alternative was instead employed, in the form of an extensible design, fashioned to accommodate the relatively easy addition of server side functionality.

A second viewpoint could however, argue the advantages of application development on a device with such low memory resources as the Palm III. This kind of constraint forces the design of a very compact ORB application. A handheld having more RAM available to play about with, could have resulted in the development of a looser middleware application. The restrictions imposed by a 2MB device forces developers to keep a very close eye on conserving resources.

It is hoped that this ORB implementation will provide the best of two worlds. That is, a finely tuned and condensed middleware application with lots of scope for functionality extensions.

It is also worth mentioning that the final code in its compiled form, occupies 55KB of memory (similar to that of PalmORB which requires 50KB RAM). This implies that the ORB application uses three percent of the available RAM, leaving a comfortable proportion available for permanent data storage, other applications, and of course for the dynamic memory allocation of running applications. The implementation of server side functionality, which would likely break the 64KB barrier, resulting in multi-segmentation and the need for resource intensive runtime libraries, as discussed in section 3.2, might not leave such a secure proportion of free RAM. Again, for the more modern Palms, the extra memory required for these runtime libraries would become less of an issue, and the ORB application could be divided into as many segments as necessary so that each segment would be less than 64KB.

5.2 Critique of Implementation

There are several areas of the implementation to be examined here.

- Firstly, much of the implementation effort focused on memory management issues, since preserving free memory is of major importance to a handheld device. Techniques employed to facilitate

successful management and freeing up of memory have been discussed at length in section 4.4. Extensive use is made of references in order to avoid making multiple copies of data structures that use up valuable memory resources.

The circular theory refers to the situation where two object references are left pointing to each other. The result of this, is that neither reference will ever be broken, and so the two objects will remain in existence indefinitely. This theory would suggest that using references to the large extent that this ORB does, could result in the latter kind of situation, thus giving rise to a memory leak. However, since a Palm III is an extremely limited device in terms of memory (among other things), it was deemed more important to try to ensure that measures were taken to reduce dynamic memory usage as much as possible, by using references rather than making expensive copies of data structures. It was also argued that if the circular object problem were to occur, it would happen only very rarely, and if after an extended period of time, performance problems were encountered then it would be possible to clear out the Palm memory by executing a soft reset. This was regarded as acceptable, since it is believed that such a reset would, at worst, be required only on very rare occasions.

- Next of all the absence of a network card meant that the Palm device could not be used in a truly mobile fashion. Instead, the Palm had to be physically connected to a network enabled device using its docking cradle. This placed the onus on the second device, to actually make the socket connection with the server side ORB. To achieve true mobile data access using the handheld would require a suitable network card, along with some relatively minor changes to the transport implementation code.

- Finally, the current ORB implementation consists of a Palm OS platform specific executable file that must be integrated into each new

client application that requires platform interoperability. This manner of code replication causes major memory resource problems when more than one client application needs to use the ORB, undoing all of the hard work that went into creating resource saving middleware. The good news is that this problem can be easily remedied. The ORB application could be saved as a Palm OS shared library class. Such classes can be used by any number of programs, thus eliminating the need for a separate copy of the ORB for each client program.

5.3 Overall CORBA ORB Interoperability

The overall ORB implementation was tested with the Orbacus ORB from Iona technologies. The purpose of this was to see if our ORB could interoperate with another CORBA compliant ORB. The result was a success. A client application on the Palm device was able to remotely invoke an operation on an object that was implemented by a servant running on the Orbacus ORB. A client request was sent to the server side, and a reply was subsequently received from the server by the client, indicating that the invocation had been a success. This result would also suggest that the ORB should be capable of interoperating with any CORBA compliant ORB.

CHAPTER 6

6. CONCLUSIONS

The purpose of this chapter is to summarise the work carried out during the course of the project, along with the knowledge that was gained with regard to writing CORBA middleware. Ideas for future work that could be carried out on the ORB application are also suggested.

6.1 Summary of Work

The ORB implementation was condensed enough to ensure that it would not consume an undue amount of the Palm III's memory. This left room for the development of other Palm applications, and for plenty of dynamic memory allocation. It also ensured that there would be no shortage of free memory for the entry of Palm application data such as address book data, e-mail messages, meeting minutes etc.

Extensive memory management capabilities were put in place to help ensure the maximum possible availability of memory at all times.

ORB stub code was implemented in a way that allowed all client applications to seamlessly interface with the ORB application. This meant that the ORB could be used with diverse client applications without ever requiring any modifications to its code.

Functionality was incorporated to allow the ORB to interpret IOR strings provided by server side ORBs. This was required so that the ORB could determine where target objects were hosted.

The GIOP/IIOP protocol was fully implemented to allow the ORB to interoperate with other CORBA ORBs that use the TCP/IP communication protocol. The ORB was capable of marshalling request messages, and interpreting replies messages, in order to determine if requests were successfully honoured. This was demonstrated by successfully sending a GIOP/IIOP request from this ORB to Iona's Orbacus ORB.

6.2 Knowledge Gained

An in depth knowledge of the client side features of the MinimumCORBA standard was gained during the course of this project. This also covered low level mechanisms like CDR data alignment and byte swapping. Preliminary reading of the original CORBA specification also provided a solid grasp on the principles behind the complete standard, and the components required to build a fully compliant CORBA ORB. A high level understanding of the CORBA services that could be implemented to enhance an ORB was also achieved.

Tackling the Palm OS style of programming provided good experience in event-based programming which is very different to purely class based or procedural programming.

6.3 Future Work

Writing a complete CORBA compliant ORB is an immense task. Writing a minimal ORB reduces the required effort somewhat, and leaves plenty of scope for extending functionality, given sufficient hardware resources.

- The most obvious piece of future work would of course involve extending the ORB to include server side functionality, particularly as

handheld devices are continually being developed to offer greater memory resources and processing power.

- Dynamic features like the DII and DSI are omitted from the MinimumCORBA standard. However, it would be very interesting to add some dynamic capabilities to a handheld, and move away from the idea of a PDA as a device that only makes decisions at design time. The ability to make run time decisions would certainly broaden the use of handheld devices in large distributed environments, to which new components are constantly being added. TypeCodes and Any types could be used to facilitate the implementation of the DII and DSI. The DII could then flexibly consult an interface repository catalogue, to invoke on remote objects of which it has no compile time knowledge. The DSI could use an implementation repository in a similar fashion, to implement objects not known to it at compile time.

Another area for development could involve incorporating some of the CORBA Services outlined in section 2.2.2. The Naming Service might be a useful one to start with. It would be remote from the Palm itself, and therefore not impact seriously on RAM resources. A Naming Service would allow a client to find references to all objects that have been registered with it, eliminating the need for IORs to be explicitly transferred to the ORB when required for remote invocation. It would also provide an extra level of indirection, allowing target objects to easily move around from host to host, without breaking any of the references to that object, that are held by clients. The only place where the host destination details would need updating would be at the Naming Service itself.

- The ORB developed for the purposes of this thesis has been designed to run solely on Palm OS platforms. It would however, be nice if the ORB could be ported to other diverse handheld devices, such as the IPAQ pocket PC. One way of extending this ORB to enable

cross platform portability, would involve utilising the Wrapper Facade pattern [Schmidt ' 99] to encapsulate low-level functions and other Palm OS specific functions and data structures, with object-oriented class interfaces. Wrapper facades provide methods that forward client invocations to non-portable functions, so that such functions do not have to be accessed directly. See figure 6.1.

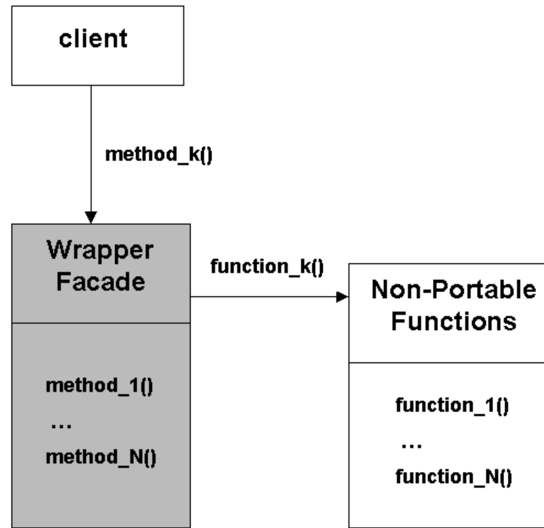

Fig 6.1 Wrapper Facade

Another area for investigation would involve implementing interpreted stubs and skeletons with dynamic qualities, to replace the current compiled stubs and skeletons that have a static knowledge of the types they marshal and unmarshal. This interpretation concept is examined and evaluated in an INFOCOM '99 paper [Gokhale ' 99]. The evaluation determined that the code size for stubs and skeletons that use interpretive schemes is smaller in size compared to the compiled form. This point is particularly interesting when considering devices with limited memory.

As a final note, the time constraints on this thesis did not leave any time for implementing an IDL compiler specific to this ORB. As it

stands, application programmers using the ORB would be required to know how to write client application specific stub interfaces, in order to interface with the ORB. An interesting project would involve implementing an IDL compiler to remove this burden from the client application programmer.

7. Bibliography

[Baker ' 97] Sean Baker, CORBA Distributed Objects Using Orbix, Addison-Wesley, 1997

[Foster ' 00] Lonnon R. Foster, Palm OS Programming Bible, IDG Books Worldwide, 2000

[Gokhale ' 99] Aniruddha Gokhale, Douglas C. Schmidt, Techniques for Optimizing CORBA Middleware for Distributed Systems, INFOCOM March 1999

[Henning ' 99] Michi Henning, Steve Vinoski, Advanced CORBA Programming with C++, Addison-Wesley, 1999

[Lazzarotto] Patrick Lazzarotto, Bitwise Logical Operations in CA-Visual Objects, http://www.cavo.com/newsletter/vo199912/bitwise.pdf

[OMG ' 01] Object Management Group, The Common Object Request Broker Architecture, February 2001

[Rhodes ' 99] Neil Rhodes, Julie Mc Keehan, Palm Programming, The Developers Guide, O' Reilly, 1999

[Roman ' 99] Manuel Roman, Ashish Singhai, Dulcineia Carvalho, Christopher Hess, Roy H. Campbell, Integrating PDAs into Distributed Systems: 2K and PalmORB, HUC 1999

[Schmidt] Douglas C. Schmidt, TAO Architecture, http://www.cs.wustl.edu/~schmidt/TAO-architecture.html

[Schmidt ' 98] Douglas C. Schmidt, David L. Levine, Chris Cleeland, Architectures and Patterns for Developing High-

Performance, Real-time ORB Endsystems, Advances in Computers 1999

[Schmidt ' 99] Douglas C. Schmidt, Wrapper Façade, A structural Pattern for Encapsulating Functions within Classes, C++ Report Magazine Feb 1999

[Schmidt ' 99] Douglas C. Schmidt, Chris Cleeland, Applying a Pattern Language to Develop Extensible ORB Middleware, Design Patters in Communication, Cambridge University Press 2000