CORBA based Middleware for Cooperating Mobile Robots

Colin Ryan
B.Tech.

September 2000

A Dissertation submitted in partial fulfilment of the requirements for the Degree of M.Sc. in Computer Science

University of Dublin
Trinity College Dublin

Declaration

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

_____

Colin Ryan

September 2000

Permission to Lend and/or Copy

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

_____

Colin Ryan

September 2000

Abstract

An embedded system is an autonomous information processing system that determines or controls to a large extent the behaviour of a larger system. The proliferation of embedded systems applications is increasing daily, yet most implementations are largely proprietary and utilise very few existing software standards in implementing their external interfaces. For a particular class of embedded system, those in use in mobile environments, the use of wireless communications protocols is a fundamental requirement.

The aim of this research is to investigate the applicability of the Object Management Group's Common Object Request Broker Architecture (CORBA) to designing and implementing middleware (ORBs) to present interfaces to embedded systems. The CORBA standard enables the construction of distributed systems of multiple components with complex interactions and hence supports the building of distributed architectures modelling real-world systems. The project also uses a wireless protocol to assess the suitability of CORBA to embedded systems that operate in mobile environments. The applicability of the CORBA standard is assessed with a canonical application utilising the Lego Mindstorms Robotics kit.

Having designed and built the test system outlined above, the suitability of the utilised technologies to the project environment, and hence to that of embedded systems in mobile environments, is assessed. Conclusions are drawn as to system performance and robustness as well as exploration of potential shortcomings of the design and scope for further research.

Acknowledgements

I would like to thank my supervisor, Dr. Vinny Cahill, for all his help and guidance through the year. Thanks to Jim, Andy and Ray of the DSG group for their assistance at various stages of this work and all my classmates for a great year and barrel of laughs (and several of beer).

Thanks to Ms. Hughes for the smashing dinners and the ultra-reliable taxi service. Most of all thanks to my family for their perpetual, and unconditional, support. And lastly I thank myself, without whom none of this would have been possible …

Table Of Contents

Table of Figures

Chapter One

Introduction

The overall goal of this report is to investigate the suitability of CORBA middle-ware technology to resource-constrained embedded systems with a particular focus on mobile environments. The aim being to implement a minimal ORB (more specifically an 'environment specific' ORB), on a severely resource limited platform (the LEGO Mindstorms RCX), along with an associated Environment Specific Inter-ORB Protocol (an ESIOP) to facilitate message based CORBA communication with that ORB. This implementation is referred to throughout this document as the 'nanOrb' implementation. This chapter introduces the key areas of interest to the nanOrb project and outlines the project objectives and achievements. Finally a roadmap of the rest of this document is presented.

1.1 Embedded Systems

Embedded systems are autonomous information processing systems that determine, or control to a large extent, the behaviour of a larger system. An embedded system typically consists of a microprocessor embedded into some device for some specific purpose other than to provide general purpose computing. Continuing advances in the downsizing of computer hardware components (and the decrease in their cost) present new opportunities for the use of embedded systems in applications ranging through embedded control, multimedia, networking and information and biomedical appliances. Each of these applications imposes their own restrictions on the capabilities of the end system, but all share some typical characteristics. The most common of engineering constraints are driven by end unit cost and operating environment

specifics. These result in systems consisting only of the most minimal of hardware resources (processing power and memory) necessary to support their function. Hence the software engineering process must be highly efficient and is tightly constrained. Any form of inter-system communications is typically implemented at a very low-level and consisted of some proprietary communications protocol.

The LEGO Mindstorms Robotics Invention System was chosen as the target embedded system for this report, as it accurately reflects the characteristics of an extremely resource constrained embedded system. At the centre of this robotics toolkit is the RCX brick, which contains a Hitatchi HD6433292 micro-controller and 32k of external RAM. These resources are obviously insufficient to accommodate a full ORB implementation. The RCX brick can communicate with another RCX or indeed an appropriately equipped PC via an embedded infrared transceiver. This transceiver is quite limited in its range, having only the 9-volt power-supply of the RCX to power it. As such the environment provides a device with minimal processing and volatile memory capabilities, in conjunction with a rudimentary communication mechanism.

1.2 Mobile Devices

The advent of wireless communications technology created a new model for the interconnection of electronic devices. Communicating devices no longer need be tied to a physical location, or indeed to be stationary. It is now possible to support communication for electronic devices as they are in motion or located in areas of minimal physical infrastructure. This technology coupled with the aforementioned increase in availability of small-sized, low-cost microprocessors has seen the phenomenal growth in both inter-connected small-scale mobile devices (embedded systems) and connectivity with conventional tethered services. This of course has important ramifications for the software that resides on these devices. It is now charged

with implementing more and more functionality, and consequently manifests much greater complexity. There is hence a recent shift towards applying the more high-level abstract design procedures and supporting infrastructures that are used in conventional distributed software systems to these mobile applications and (embedded) devices with a view to facilitating easier and more standardised development.

1.3 Common Object Request Broker Architecture (CORBA)

The Object Management Groups Common Object Request Broker Architecture (CORBA) specifies an infrastructure that "provides interoperability between software objects in a heterogeneous, distributed environment" [1] and, to a large extent, transparent to the programmer. The aim of this architecture is to allow computer systems using different hardware, operating systems, and programming languages to communicate transparently and reliably. Hence a COBRA implementation enables the transparent communication between software objects (via a set of well defined interfaces), which may be implemented using any third generation programming language (for which an IDL mapping has been defined) and ultimately hosted on any distributed operating system.

An implementation of the CORBA specification ultimately provides a "virtual bus". Once all communicating devices conform to the CORBA specification and it's interfaces, the internal implementation details may be very different for each device. Once in place, any device can make objects available on this bus and all of the other devices on the underlying communications network may access them in a transparent way. The implementation of this virtual bus is actually the responsibility of all participating devices on the network. Those devices making objects available on the bus are deemed to be 'servers' while those availing of the objects are deemed to be clients (these roles are relevant only to the particular object and

apply only for the duration of the time the object is in use). A device making a server object available may indeed by a client of another, and vice versa.

The central component of any CORBA implementation is the Object Request Broker (ORB). It implements the communications infrastructure necessary to facilitate, identify and locate objects, handle connection management and reliably deliver data between these objects. This ORB Core is the most crucial part of a CORBA implementation; it is responsible for implementing the communication of requests and their results. In addition to the ORB itself, the current CORBA 2 specification [1] describes various augmenting services, providing functionality such as transaction support and object naming resolution, which although complementary to the core functionality are not ultimately required by it.

1.4 Applying CORBA to Embedded Systems

There are numerous ORB products available in the marketplace today, many of which have been built with a particular focus on performance. Although these implementations may successfully facilitate the building of reliable distributed applications, they are typically designed to implement the full CORBA specification. The difficulty with these implementations, in the context of embedded systems, is the excessive demands they make on the host systems and other environmental resources. These far exceed those available to the majority of embedded applications. Hence there is a need for more efficient and/or specifically tailored CORBA implementations if CORBA is to be used in the embedded environment.

The OMG's '*minimumCORBA'* specification, as described in the *'Minimum CORBA RFP'* [2], specifies a CORBA model for environments in which the resources available to the implementation are constrained by the very nature of the applications, referred to in the document as "embedded systems". Minimum CORBA provides a reduced CORBA core specification

which implements basic client/server functionality, whilst removing the aforementioned peripheral services, with the intention of bringing CORBA into the domain of the embedded application. The specification ultimately specifies features of the full CORBA specification, which may be omitted in a reduced ORB implementation.

For even smaller embedded systems, which may not have enough system resources to accommodate the still relatively sizeable (approx 50k) Minimum CORBA implementation, another option exists. CORBA enabling libraries have recently emerged on the commercial market. These small footprint libraries can be as small as 15k in size [3], as much as ten times smaller than a full ORB implementation. These libraries, often called ‘engines’ [3], enable CORBA communication at a much lower level than conventional CORBA clients and servers. It is important to point out the difference between the simple byte-streaming service of a standard TCP/IP protocol stack implementation and the much greater service that CORBA functionality provides, enabling the networking of application software objects and their invocation data in a distributed environment.

In all cases a CORBA implementation is dependent on an underlying transport protocol and implicitly a network. It ultimately assumes a minimum capability of this and the system hardware. The majority of current implementations are based on the TCP/IP protocol stack, which provides the underlying connection-oriented transport that the CORBA defined inter-ORB messaging specification, the General Inter-ORB Protocol (GIOP) expects. Indeed a CORBA 2.0 compliant implementation must implement an IP mapping of the GIOP. This implementation is called the Internet Inter-ORB Protocol (IIOP) and is the standard protocol used by all compliant ORBs, hence enabling programs built with different ORBs to communicate.

The most constrained of GIOP, and hence CORBA, implementations ultimately must relax some of the specifications in order to accommodate their environment. They are somewhat customised to suit the underlying hardware and transport facilities. These implementations are termed “environment

specific" implementations [4]. Environment specific protocols enable the use of CORBA over transport protocols other than TCP/IP and ultimately allow the use of protocols that are optimised for specific environments.

1.5 Project Objective

The key objective of this project is to investigate the suitability of CORBA to embedded systems through exploring its implementation on a severely resource constrained system. To investigate this, an Environment Specific Inter-ORB Protocol (an ESIOP) and embedded ORB ("nanOrb") are designed, implemented and accessed using a simple application. Through documenting and analysing the design process, the different configuration and customisation options within the implementation are illustrated. The implementation of this system not only ultimately demonstrates that the CORBA specification can be applied in this domain, but also highlights the difficulties involved.

The end goal of any COBRA implementation is to allow the application developer to build a distributed application where, once the CORBA IDL interface is defined, only the application specific code needs development and the underlying infrastructure supporting distributed object invocations is largely transparent. Any embedded ORB implementation, and hence this one, should therefore provide the same functionality, allowing the developers of both client and server functionality to concentrate development efforts on the application specific code. Hence the nanOrb environment specific messaging and ORB functionality is implemented in such a way as to allow the programmer to utilise it in creating new applications, with as little restructuring as possible (a certain amount of customisation is necessitated on the embedded device in order to preserve the efficient implementation of the ORB functionality).

1.6 Design Overview

The end design of an embedded CORBA implementation should facilitate normal CORBA clients making invocations on the embedded service via the conventional IIOP mechanisms. Therefore some form of bridging function is needed between the CORBA specified Internet Inter-ORB Protocol (IIOP) and the Environment Specific Inter-ORB Protocol (ESIOP). This bridging should ideally be transparent to the client (that is the client should not require any special knowledge of the environment specific implementation). In this way the client is appears to directly invoke operations on the embedded server.

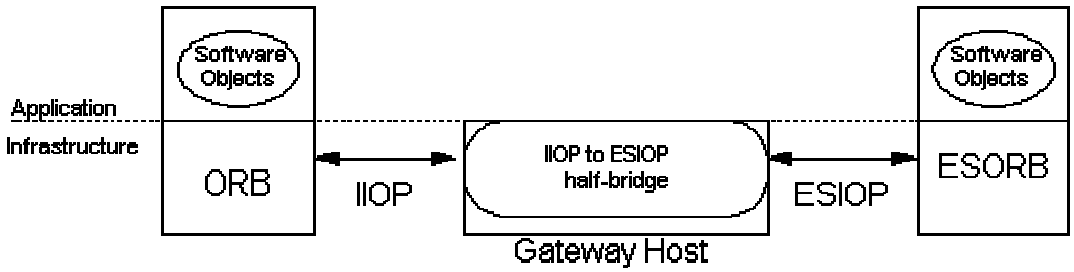

Figure 1 The Target Architecture

The main effort of the design is in defining a specific message-set for the Environment Specific Inter-ORB Protocol (ESIOP) and the “on-the-wire” format of these messages as well as providing the embedded ORB functionality on the target system. This ORB functionality should facilitate the encoding and decoding of supported IDL data types and the sending and receiving of the ESIOP messages.

1.7 Project Achievements

A design for enabling the aforementioned functionality has been produced and facilitates the transparent invocation of embedded server object methods, from the client. The work in this report has been kept focussed on implementing the ESIOP and embedded ORB functionality. The bridging functionality has been implemented as a CORBA server application (as opposed to the ORB encompassed solution described in Figure 1) in order to demonstrate the design. Whereas a more complete solution would implement a non application-specific bridge as part of an ORB implementation, the time constraints involved would not permit this. This architecture necessitates that the gateway implements the specified IDL interface, but this should not be the case in a more complete solution.

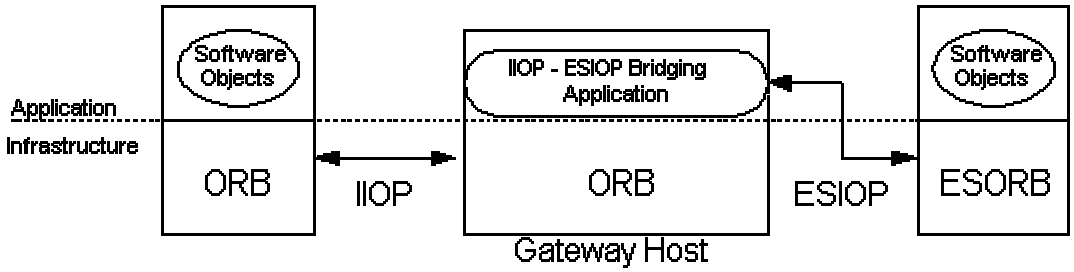

Figure 2 The nanOrb Architecture

This current implementation illustrates the embedded device acting as a server for normal CORBA clients by translating between the necessary IIOP and ESIOP messages. It defines a message-set for the ESIOP implementation and the data encoding rules. It also supports the demonstration on inter-embedded system invocations via the ESIOP. It does not as yet facilitate the RCX making client requests of normal CORBA servers.

1.8 Roadmap

The remainder of this document is divided, by chapter, according to the different work objectives of this research. Chapter two provides a background for each of the technologies utilised. The Common Object Request Broker Architecture is described in more detail, with particular attention to its more recent application to Embedded Systems, including several recent projects in this area. The GIOP specification is also described in more detail, detailing its individual components and messaging functionality. The LEGO Mindstorms Robotics Invention System is presented in more detail and the various development tools and environments available for it are reviewed. This chapter does not endeavour to fully explain each of these fields in its entirety, instead references are provided, where appropriate, to facilitate further reading.

Chapter three outlines the design objectives involved in mapping the GIOP specification onto an Environment Specific Inter-ORB Protocol and implementing embedded ORBs. Each of the design objectives is explored in detail and then developed through applying it to the Mindstorms environment. The intention is to illustrate the various configurations and customisations that are possible, and how they will ultimately effect the implementation.

The implementation of a test application (“nanOrb”) is described in chapter four. This application is used to verify the design presented previously. The technologies used and the application architecture is described in detail. Each of the components of the final application is documented along with the difficulties encountered and how they were overcome.

The application is analysed in Chapter 5 with a view to determining CORBA’s applicability to the embedded system domain. The advantages and disadvantages of using CORBA to provide standardised interfaces to embedded systems are examined. There are many design decisions and

optimisations that can be made when implementing an ESIOP. The justifications for each of these are investigated and the ramifications explored. Ultimately this chapter describes the degree to which this work was successful in verifying CORBA's success.

Finally Chapter 6 summarises the successes and failings of the nanOrb project. Conclusions are drawn from the work presented in previous chapters. The completed work is summarized and suggestions for further development and/or improvements are also made.

Chapter 2

Background

This chapter describes the main technologies relevant to this research, as outlined in the previous chapter. A number of relevant research papers and commercial products are also outlined, with a view to establishing the current ‘state-of-the-art’ in these areas. The overall aim of this chapter is to give the reader a context within which to place this work and aid further reading and/or development.

2.1 Embedded Systems

2.1.1 Overview

The IEE defines embedded systems as “…devices used to control, monitor or assist the operation of equipment, machinery or plant. ‘Embedded’ reflects the fact that they are an integral part of the system” [5]. The same literature further states, “all embedded systems are or include computers. Some of these computers are however very simple systems as compared with a PC”. Thus, it is true to say that many embedded systems do not look like traditional computers. Embedded programmable microprocessors can be found in consumer-electronics devices, kitchen appliances, networking equipment, and industrial control systems in one form or another -- from 8-bit micro-controllers to 32-bit Digital Signal Processors (DSPs). Though they're most often associated with desktop computers, the most pervasive use of microprocessors today is by far in embedded systems from the most simple of devices through to extremely sophisticated systems such as large

manufacturing systems and even such safety critical systems as airplane avionics.

The very simplest embedded systems are typically charged (programmed) with performing a simple function or set of functions to meet a single predetermined purpose. In the more complex embedded systems the operation of the embedded system is determined by some compiled code (a program), which enables the embedded system to do execute the logic of a specific application. This ability to program the system means that the same system, where flexible enough, can be used for a variety of different purposes. In some cases a microprocessor may be designed in such a way that application software for a particular purpose can be added to the basic software in a second process, after which it is not possible to make further changes: this is sometimes referred to as ‘firmware’.

The growth in utilization of programmable processors in embedded systems has largely been caused by the increase in availability of powerful, inexpensive processors and low-cost memory. However, perhaps the most exciting catalyst to this growth is the utilization of application oriented embedded systems within the Internet.

Though this presents a diverse spectrum of potential platforms and end applications for an embedded system, these are not of direct concern to this research. It is the software design process as distinct from technology, and the facilitation of high-level intercommunication for embedded devices, along with their influencing factors, that is focused upon. External design constraints based, for example, on cost pressures, long life-cycle requirements, real-time requirements, reliability requirements etc. are not explored, although it is recognised that these are the defining factors for any embedded system’s hardware and hence, implicitly, its software (indeed embedded systems in many cases must be optimised for life-cycle and business-driven factors rather than for maximum computing throughput). Thus, to reiterate, in this paper we are addressing the fundamentally common physical attributes of all such systems, that is those common physical resource constraints that

influence all embedded software implementations and hence how they might communicate.

2.1.2 Engineering Constraints

It is the physical resources of the embedded system that ultimately drive its final software implementation (be they originally cost or otherwise constrained). These constraints include specifications such as the target processor and its instruction set, memory availability (both RAM and ROM), and others arising from embedded operating systems/firmware and input and output capabilities and requirements [6]. These constrained systems are a direct result of the nature of the end product (or device) in which the system will operate. Household appliances for example, are more and more likely to utilise embedded system technology, yet due to the extremely competitive, cost-driven markets in which these manufacturers operate, the systems themselves are as economic, and implicitly constrained, as possible.

2.1.3 Embedded Systems in Distributed Environments

In the context of this report we will describe a distributed embedded system as “a system in which individual embedded systems running applications and communicating via some network medium are physically separated” [7]. As embedded systems are used more and more, (“Approximately 3 billion embedded CPUs are sold each year, with smaller (4-, 8-, and 16-bit) CPUs dominating by quantity and aggregate dollar amount” [8]), they increasingly need to communicate and interoperate with desktop and client-server installations [9].

The processors in an embedded system can be connected via any number of proprietary or standard buses, LANs, and WANs. When the degree of spatial distribution is relatively local, the processors are typically hard-wired together via a shared memory bus or similar hardware. Processors can also

be linked together via a separate high-speed serial bus such as the Controller Area Network (CAN) bus, which facilitates a greater degree of distribution and supports data rates of up to 1 Mbps. In order to support still further distribution, WAN protocols, such as the prevalent TCP/IP stack, are implemented in suitable devices. It is still, however, often considered far too expensive for smaller systems. Ultimately, although the underlying communications technology may not utilize a physical-connection, as the demand for mobility support increases, modern networks are based more and more on wireless communication.

Embedded Systems operating in distributed environment are also frequently subject to real-time constraints. As such, these systems must be designed so that tasks are always executed by a specified deadline. The particular deadline may a specific time, a time interval or indeed a discrete event [10]. This last statement describes the two key approaches to real-time systems design, the event-driven and time-driven models. The greater the frequency of these tasks and the potential for failure on missing the deadlines, the more the application exhibits ‘hard real-time’ requirements. Conversely, if missing a deadline will not necessarily compromise the system, the application is said to have ‘soft real-time’ requirements [10].

In discussing the applicability of the CORBA to embedded systems, the focus is implicitly on distributed embedded systems, independent of the underlying communications mechanism and more specifically those that execute some form of application logic. Whether these systems are subject to real-time constraints or not, is less important, although more likely in the context of a mobile environment.

2.2 The Common Object Request Broker Architecture (CORBA)

2.2.1 Overview

The Object Management Group (OMG) was established in 1989 to create standards for distributed object computing. Its standards were intended to "allow interoperability of objects, component, and applications in a heterogeneous networked environment" [11]. Early work resulted in the Object Management Architecture (OMA), an abstract object model, providing concepts of object concepts and terminology. The Common Object Request Broker Architecture (CORBA) is an open distributed-object computing infrastructure that specifies a concrete object model, based on the OMA model. Its basic task is to handle requests between clients and object implementations, in a distributed environment. CORBA automates many common network programming tasks such as "object registration, location, and activation; request de-multiplexing; framing and error handling; parameter marshalling and de-marshalling; and operation dispatching" [1].

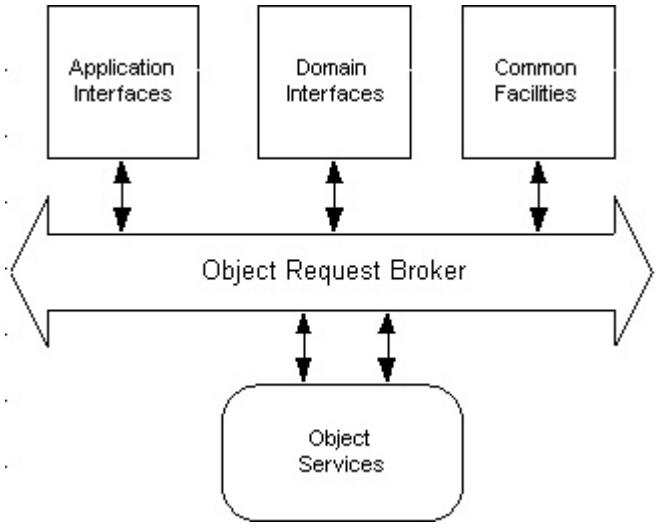

Figure 3 The Object Management Architecture

CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by the OMG and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). It did not however, provide for the inter-working of different vendors ORBs, which typically used proprietary data representation and marshalling schemes.

CORBA 2.0 [1], adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate using the General Inter-ORB Protocol (GIOP) specification, the standard implementation of this specification being the Internet Inter-ORB Protocol (IIOP), which utilises the TCP/IP protocol stack. All CORBA 2.0 compliant ORB implementations must support IIOP. This support facilitates a standard inter-orb communications mechanism.

2.2.2 The CORBA ORB

The core of CORBA is the ‘ORB core’, or middleware (*‘Middleware’ is the software that resides between an application program and the base operating systems and networking functions. Its purpose is to shield application developers from complex low-level coding*). The OMG defines the ORB as “the middleware that establishes the client-server relationships between objects” [11], it further explains how, “Using an ORB, a client can transparently invoke a method on a server object, which can be on the same machine or across a network. The ORB intercepts the call and is responsible for finding an object that can implement the request, pass it the parameters, invoke its method, and return the results” [11]. The client does not have to be aware of where the object is located, its programming language, its operating system, or any other system aspects that are not part of an object's interface.

In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments.

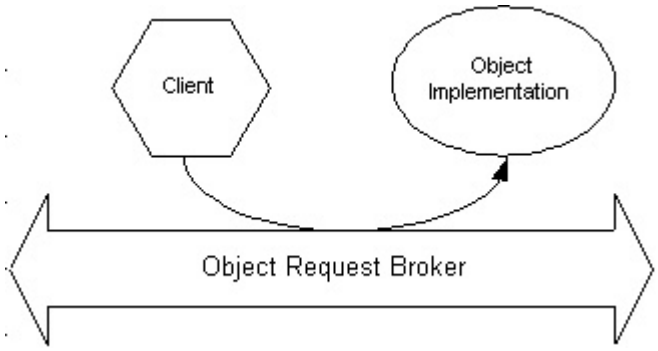

Figure 4 An Object Request Being Sent Through the Object Request Broker

2.2.3 Interface Definition Language

The IDL was originally part of the Open Software Foundation's Distributed Computing Environment (DCE). It described function interfaces for Remote Procedure Calls (RPCs), so that a compiler could generate proxy and stub code that marshalled function parameters between machines. This same model is used in CORBA to define interfaces to remote objects and hence generate stub and proxy classes, which can be used by the programmer to provide the distributed object functionality. The programmer uses these classes in the normal way, without having to worry about their internal functionality, and object distribution is achieved.

The Interface Definition Language (IDL) defined in CORBA facilitates the use, and inter-working, of multiple third generation programming languages. The IDL clearly defines the interface of a CORBA object according to a set off well-defined data types. This coupled with the GIOP and CDR (see below), which define the actual data representation, marshalling and transport rules, allows the seamless inter-working of components, which can potentially be created with different languages, on different operating systems, having different data representation rules.

2.2.4 CORBA Services

At the heart of every CORBA implementation, is the ORB core, that which provides the basic objects references and invocation functionality, hence enabling a client to transparently invoke the services of distributed objects implementations. In addition to this basic middleware functionality, the OMG has adopted a number of value-adding functions represented by middleware services called the Common Object Services (COS). CORBA services greatly extend the functionality provided by the core and some of are essential for the development and deployment of distributed applications. These services provided include, amongst others, centralised object name resolution, distributed event services and transactional support.

The OMG Naming Service is the simplest of the standard CORBA services. It essentially provides a mapping from object names to references. It is essentially a well-known repository that stores named object references. The key benefit of the OMG Naming Service is its distributed capabilities, and that it allows for stored object references to be accessed through a CORBA environment. Servers advertise themselves with the Naming Service by providing an object reference and an associated name at run-time, hence enabling clients to use the Naming Service to locate objects in a CORBA environment.

The OMG Event Service provides support for event-driven communication in the CORBA environment. It essentially implements a publisher-subscriber model via the concept of “Event Channels” and supports both push and pull operations. The ORB core allows for synchronous and asynchronous requests. With synchronous requests, the client application is blocked until the request is returned to the client, whereas with an asynchronous request, the client continues to execute, however if a response is needed, the client must periodically poll the ORB for that response until it is ready. There is no provision in this model for event-driven communication. The Event Channel is an object that provides this. The Event Channel accepts

connections from one or more suppliers, and one or more consumers. An event is defined as any piece of data that has been generated as a result of some activity. The key is that any event received from one of the suppliers is transmitted to every consumer.

The Object Transactional Service enables transactional functionality in the CORBA environment. It is, in essence, a distributed transaction manager. It supports the inter-working of object-oriented and procedural transactional applications and includes support for the industry X/Open transactional standard.

2.3 The General Inter-ORB Protocol

2.3.1 Overview

The biggest shortcoming of the early release of CORBA (pre CORBA 2.0) was its lack of a protocol specification. In order to facilitate inter-ORB communications each ORB vendor typically implemented their own proprietary inter-ORB protocol, hence complicating (if not disabling) any form of inter-vendor ORB communications. This problem was solved with the release of the CORBA 2.0 specification, which described an abstract protocol that facilitated inter-operability. This protocol, the General Inter-ORB Protocol, specified a standard set of messages, and their encoding specifics, which could ultimately be mapped onto any connection oriented transport mechanism (the aforementioned IIOP being the TCP/IP mapping of the abstract GIOP model).

The GIOP specification can be conceptually divided into three primary components:

1) The Common Data Representation
2) The GIOP Message Set
3) Transport Requirements

2.3.2 The Common Data Representation

The Common Data Representation (CDR) provides a common syntax for the transfer of IDL defined data. It defines the low-level binary, “on-the-wire” format of inter-ORB communications (ultimately byte streams).

The CDR standard itself has three key features:

1) Variable Byte Addressing - The CDR supports both ‘little-endian’ and ‘big-endian’ architectures as there is no guarantee, in the heterogeneous distributed environment in which CORBA operates, that any two communicating devices will use the same byte addressing rules. Using CDR, the sender does not have to perform any byte swapping; this is the sole responsibility of the receiver. The actual byte-order of a message is flagged in the message protocol header, so the receiver knows how the message contents must be interpreted.

2) A complete IDL mapping – The CDR defines the representation of all IDL defined data types, hence freeing programmers from having to marshal their data. Constructed data types such as structures, strings, arrays etc., are all built from the primitive types using OMG defined rules. The elements of a structure, for example, are always encoded in the order of their declaration.

TYPE	Description
boolean	An 8-bit value with the range [0-1]
char	An 8-bit value with a mapping into the ISO Latin-1 8859.1 character set.
octet	An 8-bit value with the range [0-255] that is not marshalled
short	A 16-bit integer with the range [-2exp15, 2exp15-1]
unsigned short	A 16-bit integer with the range [0, 2exp16-1]
wchar	An 8-bit, 16-bit , or 32-bit value that represents international character data
long	A 32-bit integer with the range [-2exp31, 2exp31-1]
unsigned long	A 32-bit integer with the range [0, 2exp32-1]
long long	A 32-bit integer with the range [-2exp63, 2exp63-1]
unsigned long long	A 32-bit integer with the range [0, 2exp64-1]
float	A 32-bit value conforming to the ANSI/IEEE 754-1985 floating-point standard
double	A 64-bit value conforming to the ANSI/IEEE 754-1985 double-precision floating-point standard
long double	A 128 bit value conforming to the ANSI/IEEE 754-1985 double-precision floating-point standard

Figure 5 GIOP Defined Primitive Data Types

3) Naturally Aligned Primitive Types – THE CDR specifies that primitive data types should be aligned on their natural byte boundaries (that is the way most machine architectures would align them). Whereas this process is somewhat inefficient in its consumption of bandwidth, it is ultimately more efficient than a more compact representation, as data can be unmarshalled by simply ‘pointing’ at its binary value in memory. The alignment of a primitive data type is equal to the size of that type in bytes. Hence, a type of size N bytes must be positioned in an octet stream where the index is a multiple N. A gap may also be inserted in the stream to preserve this alignment.

TYPE	ALIGNMENT (Bytes)
char	1
byte	1
short	2
unsigned short	2
long	4
unsigned long	4
long long	8
unsigned long long	8
float	4
double	8
long double	8
boolean	1
enumeration	4

Figure 6 Octet Sizes of Primitive Data Types

It is important to note that CDR encoded data is not self-identifying. Upon receipt, the marshalled data is nothing more than a sequence of octets (bytes). The receiver must know in advance how these marshalled octets are to be decoded. This knowledge is facilitated by the IDL definitions of an interface, which informs as receiver how data is to be interpreted.

2.3.3 The GIOP Message Set

2.3.3.1 Overview

The GIOP specification defines a set of eight messages (since version 1.1), which are considered sufficient to accomplish the functional objectives of CORBA. Whereas, s only two of these messages are actually required to achieve the basic remote invocation objectives of CORBA, the remaining six provide various, complimentary, functionality.

Message Type	Issuer	enum Value
REQUEST	Client	0
REPLY	Server	1
CANCEL_REQUEST	Client	2
LOCATE_REQUEST	Client	3
LOCATE_REPLY	Server	4
CLOSE_CONNECTION	Server	5
MESSAGE_ERROR	Both	6
FRAGMENT	Both	7

Figure 7 The 1.1 GIOP Messages

The FRAGMENT message was added in version 1.1 of the GIOP specification, to allow for the more efficient marshalling of data by the sender.

2.3.3.2 The GIOP Protocol Header

In order to transmit a GIOP message, the sender will fist include the protocol message header, a 12-byte header consisting of five fields, and then the message body, which is specific to the type of message being sent.

```
Module GIOP {
        struct Version {
                octet           major;
                octet           minor;
        };
        enum MsgType_1_1 {
                Request, Reply, CancelRequest, LocateRequest,
                LocateReply, CloseConnection, MessageError, Fragment
        };
        struct MessageHeader_1_1{
                char            magic[4];
                Version         GIOP_version;
                octet           flags;
                octet           message_type;
                unsigned long   message_size;
        };
};
```

Figure 8 The GIOP Protocol Header IDL

- The first four bytes always contain the characters ‘GIOP’, to indicate the message type.
- The fourth and fifth bytes contain the Major and Minor version numbers as binary values.
- The sixth byte is a flag, the least significant bit of which signifies whether the message is in big endian (0) or little-endian(1) encoding. The second bit is used to flag fragmentation.
- The seventh byte is used to indicate the type of the GIOP Message and corresponds to the numeric value of the appropriate **MsgType_1_1** enumeration.
- The last four bytes are used to indicate the sized of the message, excluding the twelve bytes of this header.

2.3.3.3 The Request Message

The request message is used to encode object invocations and send them to the server. A complete request message consists of the aforementioned GIOP protocol header, a request message header and also the request message body. The latter two forming the GIOP message body.

```
module GIOP {

        struct RequestHeader_1_1 {
                IOP::ServiceContextList     service_context;
                unsigned long               request_id;
                boolean                     response_expected;
                octet                       reserved[3];
                sequence<octet>             object_key;
                string                      operation;
                Principal                   requesting_principal;
        };

};
```

Figure 9 The GIOP 1.1 Request Header IDL

- The ‘service_context’ is an IDL defined structure used by services such as transactional and security services to implicitly pass service information with requests and replies, transparent to the client.
- The ‘request_id’ field is an unsigned long integer value, used to uniquely identify a particular request, and also to relate response messages to their request messages.
- The ‘response-expected’ field is used to indicate whether a server response is expected in reply to a particular request. The value is set to false (0) for IDL specified ‘one-way’ functions.
- The ‘reserved’ octet is reserved for future use.

- The ‘object_key’ field is the object key from the Interoperable Object Reference (IOR). This is a server specified value and has no relevance outside of the server’s scope.
- The ‘operation’ field contains a string value which indicates the relevant method to be invoked on an object
- The requesting principal field is used for security purposes in order to identify the requester. It is now deprecated, as the service context provides this information.

The body of the request message is an octet sequence containing the encoded IDL specified ‘in’ and ‘out’ parameters for the requested operation.

2.3.3.4 The Reply Message

The reply message is sent by a server in response to a client request. A complete reply message consists of the GIOP protocol header, a reply message header and also the reply message body. The latter two forming the GIOP message body. The reply message indicates the success or failure of a request, and (in the case of the former) also includes any IDL defined ‘out’ parameters of the associated method invocation.

```
module GIOP {
        enum ReplyStatusType {
                NO_EXCEPTION, USER_EXCEPTION,
                SYSTEM_EXCEPTION, LOCATION_FORWARD
        };

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

Figure 10 The GIOP 1.1 Request Header IDL

- The ‘service_context’ field is used in the same way in the reply header as in the request header.

- The ‘request_id’ field returns the uniquely identifier of the associated request, hence a client does not have to wait for once request to complete before making another.

- The ‘reply_status’ field contains one of the ‘ReplyStatusType’ enumeration values and indicates the result of the request. The ‘LOCATION_FORWARD’ reply is used when a server cannot fulfil a particular request, but advices the client to try another address.

The body of the reply contains an octet sequence containing any ‘out’ parameters for the requested operation.

2.3.3.5 The Cancel Request Message

The ‘Cancel Request’ message is sent by a client to a server to indicate that the client no longer requires, or expects, a response to a particular request. A complete ’Cancel Request ‘message consists of the GIOP protocol header and the ‘Cancel Request’ message header.

```
module GIOP{
        struct CancelRequestHeader{
                unsigned long           request_id;
        };
};
```

Figure 11 The GIOP Cancel Request Header IDL

The message header contains only the unsigned long ‘request_id’ field, which indicates the particular request to be cancelled. The server does not acknowledge the ‘Cancel Request’ message.

2.3.3.6 The Locate Request Message

The ‘Locate Request’ message is sent by a client to a server to determine whether a particular Interoperable Object Reference is valid. More specifically, it is a more bandwidth efficient method (as opposed to sending a complete request message) of determining of whether or not an object is available at a particular address. A complete ’Locate Request ‘message consists of the GIOP protocol header and the ‘Locate Request’ message header.

```
module GIOP{
        struct LocateRequestHeader{
                unsigned long           request_id;
                sequence <octet>        object_key;
        };
};
```

Figure 12 The GIOP Locate Request Header IDL

The message header contains an unsigned long ‘Request Identifier’ field and an object key (octet sequence). This message is often used in conjunction with a CORBA ‘interface repository’, which acts as a central service for dispatching client ‘look-up’ requests to server implementations[11].

2.3.3.7 The Locate Reply Message

A server sends the ‘Locate Reply’ message to a client in response to the ‘Locate Request’ message. A complete ‘Locate Reply’ message consists of the GIOP protocol header, the ‘Locate Reply’ message header and the ‘Locate Reply’ message body.

```
module GIOP{

        enum LocateStatusType{
                UNKNOWN_OBJECT,
                OBJECT_HERE,
                OBJECT_FORWARD
        };

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

Figure 13 The GIOP Locate Reply Header IDL

The message header contains an unsigned long ‘Request Identifier’ field and the ‘locate_status’ field, indicating the result of the ‘Locate Request’ message. The message body then contains the Interoperable Object Reference (IOR) of the requested object

2.3.3.8 The Close Connection Message

The ‘Close Connection’ message is only sent by the server. The message consists only of a GIOP header. If the client wishes to send further requests, these must be sent on a new connection. This message is typically used when a server has reached its maximum number of concurrent connections.

2.3.3.9 The Message Error Message

The ‘Message Error’ message can be sent by wither the client or server. It is sent when the protocol header of a received GIOP message indicates a protocol version that is not supported by the recipient.

2.3.3.9 The Fragment Message

If a GIOP client decides to use fragmentation, the first part of a request or response message is sent with the fragment bit in the protocol header set to true. The ‘Fragment’ message is used after these messages to pass further fragments of encoded data and also indicate whether more fragments are to follow. The ‘Fragment’ message exists to avoid necessitating the client marshalling of large messages in their entirety, before sending them.

2.3.4 Transport Assumptions

The GIOP specification makes certain assumptions of the underlying transport protocol [1]:

- It provides a reliable connection oriented service. A connection-oriented transport allows a host to open a connection to by specifying the address of the receiver. This process will typically return an identifying handle to that connection, which can then be used for the duration of communication without the need to specify the address for every the message sent.

- Connections must be full duplex. Upon a connection being established, either communicating parties should be able to use that connection to send messages without needing the address of the originator.

- The transport is reliable. The transport should ensure that any messages sent over a connection are received at the destination without duplication.

- The transport provides a byte-stream abstraction. The transport should ultimately be viewed as a ‘data-pipe’, once established. In this way, a communicating host can view a connection as a continuous stream of bytes and not have to deal with underlying networking issues such as fragmentation and re-transmission.

2.4 Object References

The CORBA specification describes an object reference as “an object name that reliably denotes a particular object” [11]. The ultimate aim of this reference is to facilitate the client utilizing the object in a location and implementation transparent way. The General Inter-ORB Protocol uses the Interoperable Object Reference (IOR) to identify objects.

The IOR is a data structure that provides information on the type of object it references, the underlying transport protocols that support contacting it and optional further service information. The IOR is structure is defined in a flexible manner. This flexibility is intended to facilitate the addition of support for multiple transport protocols and their associated optional data.

An IOR consists of three key components. The first, a ‘type_id’ is a scoped string indicating the most derived type of the represented objects IDL defined interface. The IOR will secondly always contain a sequence of one or more ‘TaggedProfile’ structures. Each of these contains endpoint information indicating how an object may be contacted via a specific protocol. Within each of these a third value, the Object_ID is an octet sequence used by the server to identify the particular object. The ‘TaggedComponent’ structure may

optionally be included within a ‘TaggedProfile’ and is used to communicate extra service specific information.

```
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;
                sequence <TaggedProfile>        profiles;
        };

        typedef unsigned long          ComponentId;

        struct TaggedComponent {
                componentId            tag;
                sequence<octet>                 component_data;
        };

        typedef sequence<TaggedComponent> MultipleCoponentProfile;
};
```

Figure 14 The Interoperable Object Reference IDL

Hence an IOR can be defined for a server object, which contains end-point information for each specific protocol the server supports as well as optional extra ‘TaggedComponent’ data. In this way a server implementation can support existing and future protocols in a single published IOR.

The default ‘profileId’ (as a consequence of IIOP support being mandated by the COBRA 2.0 specification) is the ‘TAG_INTERNET_IOP ’ ‘profileId’. In

order to identify a communications end-point using the TCP/IP transport, this will contain Hostname or IP address and a port number, thus enabling a client to locate a server implementation via the TCP/IP protocol.

2.5 CORBA and Embedded Systems

2.5.1 Overview

As has been described previously, the majority of implementations of the full CORBA specification consist not only of the ORB core functionality, but also several of the aforementioned CORBA services. Hence, these implementations tend to be quite large, in terms of their storage and memory requirements, and also computationally intensive to execute, even on modern day desktop computers. These implementations are hence not suitable to use in heavily resource constrained embedded systems.

For this reason, alternatives have recently emerged. These alternatives use various different approaches to enabling the use of CORBA in the embedded environment. The first, and perhaps most simplistic, of these consists simply implementing a CORBA enabled gateway or proxy service for client's wishing to interact with embedded devices. In this model, CORBA is used between the client and the gateway, and the gateway in turn communicates with the embedded system using a proprietary protocol, typically some low-level proprietary protocol. This clearly is not an embedded system CORBA implementation. The truer embedded CORBA implementations fall into two broad categories, those implementing the OMG's 'minimumCORBA' specification [2], and those implementing some form of protocol 'engines', derived from the GIOP specification, the latter enabling CORBA compliant communication at a much-lower level.

The first of the two aforementioned options, the OMG Minimal CORBA standard, defines a subset of the full CORBA standard, which facilitates the implementation of more efficient and smaller footprint ORBs. The core changes in the standard are the removal of most of the dynamic facilities for creating and using objects; this decision was based on the assumption that “The background of embedded systems tends to require design-time decisions on resource allocation, object location and creation. Together with pre-determined patterns of interaction, this yields a much more predictable system environment” [2].

The standard attempts to “minimise the specification of unnecessary or costly services, whilst retaining maximum compatibility” [2] with the existing full CORBA specification. Support for the full set of CORBA IDL is retained; hence there is no barrier to implementations utilising any existing external CORBA services, when running in larger CORBA systems.

The second approach to facilitating embedded CORBA, is much lower level approach utilising an efficient ‘protocol engine’ to enable CORBA (GIOP) compliant communications on an embedded system. This ‘engine’ may be in the form of a third-party library or a proprietary implementation. These ‘engines’ ultimately facilitate the basic construction and deconstruction of GIOP compliant packets along with their transmission over the underlying protocol.

A further consideration in the application of CORBA to embedded applications is the underlying transport. Whereas the de-facto modern networking protocol is the TCP/IP protocol suite, and indeed many embedded systems do require this connectivity, the processors in an embedded system can be connected via any number of proprietary or standard buses, as discussed in section 2.1.3. It is therefore often the case that embedded CORA applications will implement a messaging transport other than the Internet Inter-ORB Protocol, that is an ‘environment specific’ implementation.

2.5.2 Embedded CORBA Research

There is a lot of research work in progress in the embedded CORBA domain. The focus of these efforts cover the two main approaches outlined previously, those of the minimumCORBA specification and customised low-level GIOP communications, utilizing both the Internet Inter-ORB Protocol (IIOP) and other Environment Specific Inter-ORB Protocols (ESIOPs). This section introduces some of the most relevant works and compares them to the focus of this report.

There are several commercial and research derived high-performance CORBA products in existence. ORB products such as ORBit [12], ORBacus [13], OmniORB [14] and the ACE ORB [15] are all built with a focus on high-performance.

The ACE ORB, for example, is a product of high-performance and real-time CORBA research at U.C.L.A. [16]. The ACE ORB (TAO), pronounced "dau", is an open source CORBA 2.2 compliant, C++ implementation. It more recently includes a minimumCORBA compliant implementation, which as a result of its component structure and open-source form does lend itself to application and customisation in the embedded environment.

Lockheed Martin has developed a software infrastructure called “HARDPack middleware” to “manage data in an object-oriented, real-time, distributed environment” [17]. This product provides a CORBA 2.0 based ORB implementation including Dynamic Invocation Interface (DII) and an Interface Repository (IR) and CORBA services include naming and event communication.

The difficulty with all the aforementioned CORBA implementations is that although they are very efficient, their memory footprints and resource demands are still far too excessive for the more constrained of embedded systems.

The K-ORB Project [18] is one minimumCORBA research effort that is very relevant to this project. The project describes a minimumCORBA framework that facilitates the building of ORBs tailored to the particular requirements of the target environment. It implements a ‘pluggable framework’ facilitating the utilisation of different components of the K-ORB system as required. Using this architecture multiple networking protocols are facilitated and hence ESIOPs accommodated. This model and its architecture have been leveraged throughout this work and it is anticipated that the two projects will be integrated in the future.

For the more severely constrained of embedded systems however, all of these full minimumCORBA implementations are unattainable. GIOP compliant protocol engines can provide more efficient ORB core functionality for these devices. Products such as Sunsofts IIOP Protocol Engine [19] and IONA Technologies IIOP engine [3] enable IIOP messaging on these devices.

The Sunsoft IIOP Protocol engine is a library written in C++ and is composed of four parts: “a CDR marshalling engine, a Type Code interpreter, the engine framework (including a partial ORB implementation) and IIOP-specific modules” [19]. It also provides some run-time dynamic invocation support. The IONA IIOP engine provides a similar functionality via a “highly-efficient, low-footprint run-time library” [3] written in ANSI-C. It essentially provides an API to the GIOP functionality. Whereas these tool do address embedded CORBA functionality for Internet enabled devices, they do not support any form of environment specific implementations.

The [20] paper describes an environment specific CORBA implementation based on the Controller Area Network (CAN) bus. The report describes a “Compact Common Data Representation (CCDR)” [20] format, an optimised version of the CORBA CDR specification, which enables more efficient use of the small payload of the CAN bus (8 byte) message payloads, hence compromising processing speed for bandwidth efficiency. It further describes a customised (reduced) messages set, based on two of the eight GIOP specified messages, and message header format (again with a focus on increasing bandwidth efficiency). This work is an excellent demonstration of the environment driven constraints and resultant customisations that characterise embedded system development and hence will feature in any environment specific CORBA implementation.

2.6 The Lego Mind Storms Robotics Invention System

2.6.1 Overview

The Lego Mindstorms Robotics Invention System is a product manufactured by the Lego Company. It consists of conventional Lego bricks, along with a ‘programmable brick’ and several motors and actuators (touch and light sensors), which, collectively provide the building blocks of a simple but powerful robotics kit. Users can program and compile programs for constructed robots, using a PC based application, and download these to the robot via a wireless infrared link, which utilises the PC’s serial (RS-232) port and a small infrared transceiver (tower).

The Mindstorms product was derived from, but is ultimately considerably different to, the “Programmable Brick Project” of the Epistemology and Learning Group in MIT’s Media Lab. The Programmable Brick is part of the ongoing LEGO/Logo research project at the Epistemology and Learning

Group, which was originally started by Seymour Papert [21], the creator of the LOGO language/teaching environment.

Whereas the originally intended way to program Mindstorms robots was using the provided PC application, several language ports now exist which facilitate much greater flexibility and control in building applications. These range from Visual Basic enabling COM controls [22] to the NQC ('Not Quite C') language [23], a C like low-level language and even to replacement firmware in the form of pbForth [24]), tinyVM [25] and legOS [26].

2.6.2 The RCX Brick

The RCX brick is the battery powered programmable micro-controller that is the heart of the Mindstorms kit. It is contained in a single brick that is capable of operating three motors, three sensors, and the infrared communications interface. At the core of this brick is a Hitachi HD6433292 micro-controller that contains 16K of ROM and 512K or RAM and runs at a speed of 16Mhz. A further 32K of external RAM is also contained in the brick.

The 16K on-chip ROM contains a driver that is run when the RCX is first powered up. This driver facilitates the downloading of firmware to the RCX. The standard firmware occupies 16K of memory and facilitates downloading of user-compiled programs to the RCX that can then be interpreted and executed by the firmware.

The Mindstorms RCX is a very small scale embedded system with severe physical resource limitations. As such, it is perfectly suited to exploring the practicality and possibility of CORBA implementations on embedded systems and the associated limitations.

2.6.3 The Development Process and Tools

The standard development environment supplied with the Mindstorms kit supports some simple but useful programming. It is a graphical environment, which allows the user to drag and drop RCX actions and events as building blocks for an RCX application. Once built the program is compiled to byte-code and downloaded to the robot, where the firmware of the RCX interprets this byte-code and controls the RCX (or the robot which is connected to) accordingly. The standard firmware itself controls the hardware interrupts, multi-threading and IR Port communications but is relatively limited from the developer's perspective.

There are various tools in existence that facilitate the extension of the RCXs capabilities via replacing different portions of the architecture. The simpler varieties provide replacement compilation environments for the developer. The Not Quite C (NQC) language [23] and the 'spirit.ocx' COM control Holdren, 2000 #13] are two such examples which provide a more procedural programming environment, but still fall far short of exploiting the hardware's full capabilities. There are also some TCL and Perl language interfaces that facilitate the run-time control of the robots.

In order to take advantage of the full 32K or RAM and the full capabilities of the RCX brick, it is necessary to replace the standard LEGO supplied firmware. Three such replacements exist at present. PBForth [24] is based on the interpretive Forth language, and provides run-time control of the RCX via the IR link. The TinyVM [25] environment provides a Java based replacement firmware for the RCX. The third replacement and also one providing the most comprehensive and powerful development environment is legOS, a GNU based cross-compilation environment that facilitates assembly C and C++ programming of the Hitatchi HD6433292.

2.6.4 LegOS and the Layered Network Protocol

LegOS essentially provides a multitasking (pre-emptive) operating system for the RCX. The operating system and its programs are written in standard C (with some C++ support) and then cross-compiled for the Hitachi chip using a GNU built compiler. Once the basic operating system is compiled and downloaded to the RCX as a replacement firmware, user programs can be compiled, downloaded and executed.

The development environment offers almost complete C language support, including semaphores, multi-threading, floating point emulation and the ability to store multiple programs (these programs are dynamically linked with the underlying legOS operating system).

The most interesting feature of the more recent releases of the legOS environment however, is the Layered Network Protocol (LNP). LNP provides a simple networking abstraction to the underlying infrared transport mechanism. It facilitates the transmission of up to 255 bytes of data along with a checksum, to ensure integrity (this is known in LNP as an integrity packet). This mechanism acts like a broadcast channel for all listening hosts (where a host can be either an RCX brick or a PC enabled with the standard LEGO infrared tower). The protocol silently discards erroneous packets.

The API also provides an addressing mechanism where 2 bytes of the 255-byte payload of the integrity packet are used for addressing information, hence enabling the sending of a 253 byte addressed packet. A single byte is used for the source and destination addressing information. Each byte is then bit-masked to provide port information on each node. Hence it is possible to describe a host-port pairing to identify a communication end-point.

The programming model necessitates that a process is created to handle incoming packets on each participating hosts. This process sleeps and is awoken via an interrupt, upon receipt of an addressing packet for the port it

is assigned to, or an integrity (broadcast) packet. It is this transporting mechanism, coupled with the programming capabilities of the legOS environment, which makes the Mindstorms environment particularly suitable to an ESIOP implementation. That is, a server process can be created to handle data for a particular port on a specific host, that host-port pairing being a uniquely identifiable LNP transport end-point.

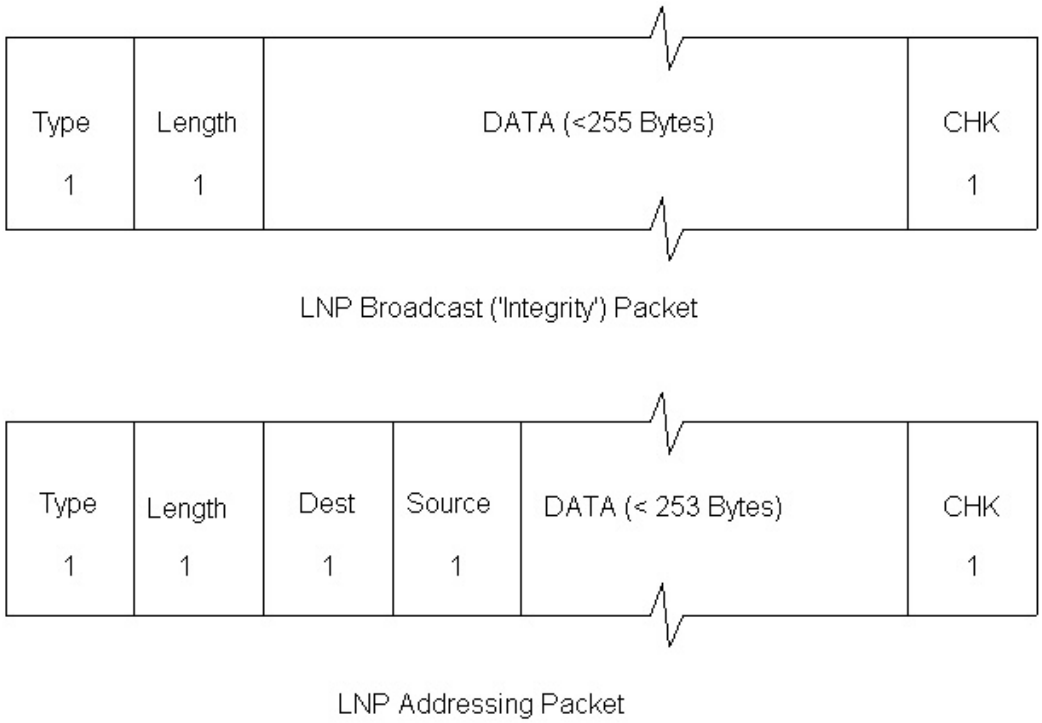

Figure 15 The structure of LNP packets

2.7 Mobile Applications

2.7.1 Overview

The advent of wireless communications technology and the subsequent proliferation of devices utilizing it have many ramifications for computer software, particularly distributed applications. There is a broad range of devices utilizing the technology, each with their individual characteristics. The one common attribute of these devices is their need for portability, if they are to support true mobility. Devices such as laptop computers, Personal Data Assistants (PDAs) and mobile phones are typical examples. These devices must be smaller in physical dimensions and lighter in weight, than their fixed-location (tethered) counterparts if they are to be used in a mobile environment and still strive to provide the same service to the user. The electronic component manufacturing industry has largely met these requirements, through the constant downsizing of chipsets and other device components, such as displays and battery-packs.

In conjunction with the requirements these technologies make of their physical attributes, they have large implications for the nature of the applications they host and indeed the way they are used. Whereas these applications can, and do, provide all the same functionality as their tethered counter-parts, they can also provide much further benefits to the user (such as location based services).

2.7.2 Communications Characteristics

Of particular relevance to this work, are the characteristics that the afore-mentioned mobile devices exhibit which are distinct to those of their tethered peers. A mobile device, or node, is ultimately unconstrained in its physical location, it can hence theoretically be utilised anywhere on the planet. The

wireless communications mechanism upon which these devices depend or not quite so unconstrained, they are subject to the limitation (or 'coverage area') of their infrastructure. A mobile phone user could for example potentially attempt to use the device anywhere on the planet whereas this attempt will only be successful if a supporting cellular network is available.

Not only must the wireless transport be available before communication can be attempted, but it must also remain available for the duration of the devices usage. This may seem like an obvious statement, but when the fact that the device may be moving (and the 'coverage area' of the underlying network is not) is considered along with the dynamic nature of most wireless communication mechanisms, it cannot be assumed. Radio-based mechanisms are subject to the inherent random nature of mobile radio devices; factors such as interference and signal reflection can practically annihilate a strong signal. This constitutes a significant departure from the assumptions of most fixed-network transports.

Various attempts have been made at overcoming these limitations of wireless technologies, indeed emerging technologies such as Code Division Multiple Access (CDMA) have done much to remove interference problems, but ultimately they cannot be removed. Wireless applications use various 'smart' techniques to reduce their affect. Digital cellular phones use specialised prediction algorithms to 'fill-in' the space created by lost or erroneous packets, hence minimising the interference noticed by the user. In other environments, with less real-time constraint, simple time-out and re-transmission policies may be applicable. Of course, depending on the capabilities of the mobile devices, and the nature of their use, a certain degree of autonomous operation may be acceptable, or even applicable. It may simply be feasible to have devices 'contact' the network periodically, perhaps simply to retrieve or deliver information relating to tasks, and then continue in a disconnected fashion. In systems where the application is less 'dispatch' oriented, disconnected operation may be supported via the caching or relevant data and subsequent re-synchronisation of this data upon re-establishment of connectivity. These implementations may support this

disconnected operation by mimicking the server functionality on the local device, with a view to concealing the disconnected operation from the dependent applications. The Rover [27], ALICE [28] and Dolmen [29] projects discuss support for such disconnected operation.

Chapter 3

Design

This chapter introduces the design goals relevant to building an Environment Specific Inter-ORB Protocol (ESIOP) and further develops these in the context of the LEGO Mindstorms environment. The aim is to not only introduce the generic ESIOP design process, but also to detail its application in the context of a specific environment.

3.1 Overview

The overall goal of this project is to investigate the suitability of CORBA middleware technology to resource-constrained embedded systems with a particular focus on mobile environments. The aim being to implement a minimal ORB (more specifically an ESIOP engine), on a severely resource limited platform (the LEGO Mindstorms RCX), along with an environment specific messaging protocol (an ESIOP implementation) to facilitate CORBA based communication with that ORB.

The design enables a standard CORBA (IIOP) client to send requests to a CORBA compliant server, which acts as an IIOP half-bridge (gateway) and translates these calls into the environment specific format, for communication with the embedded system. At all times the messaging protocol is an environment- specific implementation (or ‘functional subset’ of the GIOP).

Hence this design must describe an Environment Specific ORB messaging protocol that includes:

1. A data representation syntax that specifies which of the standard IDL defined data types are supported and how they are encoded for transmission.

2. An ORB message set that is suitable to the environment. Some form of mapping between the GIOP message set and this set is also required if full CORBA functionality is to be supported.

3. An ORB transport mechanism which can provide the reliable byte-streaming service that CORBA implementations expect

4. A communication end-point mechanism such that individual server implementations can be reliable located.

5. An Object Addressing format that facilitates the reliable addressing of objects in the environment.

Once these, and hence the messaging protocol, have been defined the embedded ORB implementation can be addressed. The design of this embedded ORB, nanOrb, must support the aforementioned message set and provide as consistent an API as possible to the embedded applications it supports.

A gateway function is also required, to facilitate the conversion of IIOP to ESIOP requests, and vice-versa. This gateway must be capable of deconstructing messages from one transport and encoding them for the other, along with performing any complementary functionality and optimisations that may be appropriate to the environment.

3.2 The Data Representation Syntax

The data representation standards for the ESIOP implementation must be defined. These must typically identify a distinct sub-set (or the full set) of the IDL defined data-types to support and an encoding syntax of each of these. The possibility of using a compact format (similar to that described in [20]) was explored, but is unnecessary due to the MTU size of the LNP transport being far more generous (253 bytes) than that of the CAN bus (8 bytes) used in the Kim project. The resource constraints in this application environment are more processing and storage than transport oriented. Hence, for the nanOrb implementation, a GIOP-like data representation scheme is implemented, preserving the natural memory alignment of data-types on 32-bit boundaries. The formats of the messages are based on version 1.1 of the GIOP standard. The full set of IDL defined primitive data types are supported.

The RCX uses a little-endian addressing architecture, as does the Intel 386-based gateway used in nanOrb project. In a truly GIOP-based solution the RCX functionality would have to include the ability to perform conversion of data received from the gateway, were it based on a big-endian architecture. CORBA specifies that the message-sender's byte ordering should be used for messages and flagged in the GIOP message header [1] (this rule exists to avoid unnecessary conversions on the server). This is not the case with this architecture because conversions are not necessary they are eliminated from the design. This highlights a potential design optimisation wherein byte ordering could always be the responsibility of the bridging host in order to reduce the load on the target embedded system. This would, of course, necessitate that the gateway is 'aware' of the destination's underlying architecture (in the context of embedded systems this is not an unreasonable assumption). Optimisations such as this, and others, are further discussed in the section of this chapter, relating to the design of the bridging function.

3.3 The ORB Message Set

In any environment specific implementation a functional subset (or the full set) of the GIOP message set must be defined. In order to consistently present full CORBA services to clients, the functionality of the GIOP messages must be supported in some way. Rather than implementing the full GIOP message set in the nanOrb design, a subset of the standard 8 GIOP 1.1 specified messages set is identified that is deemed suitable to the environment.

As a consequence of the extreme resource constraints on the RCX, the memory footprint of the ESIOP engine must be kept minimal, if it is to work at all. A minimal number of messages are implemented in order to provide basic functionality and demonstrate the applicability of the CORBA to the environment. The number of these messages may be expanded in the future, but first it is necessary to determine how much of the limited memory resources these will consume and prove the concept. It is important to remember that ultimately, when implementing embedded CORBA, there will be a trade-off between the amount of memory utilized to provide CORBA functionality and the amount available for implementing application logic.

3.3.1 Client Initiated Messages

REQUEST – This message encodes an object invocation from a client to a server. The object invocation/operation is encoded and sent to the server, along with any IDL defined ‘IN’ and ‘OUT’ parameters (see chapter 2). The response to this, from the server, is contained in the Response message (if the ‘RESPONSE_EXPECTED’ flag is set in the request header). The REQUEST message must be implemented as it facilitates the fundamental CORBA functionality, that of remote object invocation. Within the nanOrb implementation, some of the fields in this message header (such as Service Context and Principal information) are included, in order to comply as much as possible with the GIOP specification, but not populated. They are not necessary to provide the core CORBA functionality. They could of course be

developed in future implementations, to provide, for example, security and transactional support.

Note; It is should be noted that this service contextual information could potentially be processed on the gateway (bridging) host, hence keeping the embedded ORB footprint to a minimum.

LOCATE_REQUEST – This message is used to check the validity of an object reference and if a server will support a particular reference. The server may reply with a LOCATION_FORWARD type reply if it does not support an object locally. In particular this message is used in conjunction with an 'Interface Repository' (the Interface Repository provides run-time resolution of server implementations), so that a client can use a single 'well-known address' to resolve object references via the LOCATION_FORWARD reply. The request message will perform the same functionality although in a more expensive manner, that is, with a greater message payload (which contains all the invocation data, as opposed to just the Object Reference). It is, however, more expensive in the context of the Mindstorms environment, and many embedded systems (as our primary constraints are memory-footprint and processing oriented), to implement the processing necessary to facilitate the functionality of another message, than to send a greater sized packet over the underlying transport. Hence this message is not implemented in this design. It is worth noting that the 'LOCATE' functionality could be used by an IIOP client when talking to a gateway (or discovering it) to facilitate mobility support.

CANCEL_REQUEST – This message is sent by a client to cancel a previous request. The request ID is used to identify the particular request. It is advisory only and the server is not obliged to acknowledge it. This message is not particularly suited to real-time control applications and hence is not implemented. Again, it may be of use in facilitating an application where client requests are valid for a longer period of time, for example if the client was to dictate a particular behaviour to an embedded device. Its appropriateness in an embedded context is really application specific.

3.3.2 Server Initiated Messages

REPLY – This message is sent by the server if the 'RESPONSE_EXPECTED' flag is set in the request message. This can be used to return the results of an object invocation (where 'OUT' parameters are specified in the IDL) or simply a status. There is a REPLY_STATUS field in the header that can be used to indicate NO_EXCEPTION, hence reducing the need for a MESSAGE_ERROR message. This message is implemented. It is important, however, that the gateway's call, to an RCX based object, is non-blocking. That is, the server should not wait for the receipt of a REPLY message before continuing. This is a departure from the GIOP model, but necessary due to the inherent unreliability of the infrared environment. Were the underlying transport of the ESIOP reliable, this optimisation would not be necessary.

LOCATE_REPLY - This message is sent from the server in response to LOCATE_REQUEST message. It contains the results of a location attempt. It is not implemented, for the aforementioned reasons. It would not add any extra functionality in the RCX, and accommodating it would only serve to increase the memory consumption of the embedded ORB implementation. Again, it may be useful in facilitating location forwarding for IIOP clients using a gateway to request services of embedded devices (such as the RCX) in a mobile environment, where these devices may be in communication with different gateway's at different times, depending on their location.

CLOSE_CONNECTION – This message is used to inform a client that it should not expect to receive any further information in relation to a particular request. It is used to facilitate the 'clean' closing of a long running connection between the client and the ORB server. It is implemented in this implementation, as we are evaluating the server as a simple client controlled device. It would not be practical to rely on long-running connections in a mobile Infrared environment. It may well be applicable to an application where the embedded server expects to support multiple simultaneous connections

and hence may need to notify clients as it reaches its concurrency limit and wants to cease offering its service temporarily.

This highlights another potential optimisation in an ESIOP application. If a single, or multiple interconnected, gateway(s) exist, which are aware at all times of the number of connections an embedded server is supporting, these gateways could assume responsibility for managing maximum numbers of connections, hence ‘protecting’ the servers from overloading, without the need for the server’s to implement this functionality.

3.3.3 Common Messages

MESSAGE_ERROR – This message is sent when either party detects an error-condition as a result of a message. This is usually because of the incorrect formation of the message, or it’s containing an unsupported version number. This message is typically implemented, although the REPLY message does contain the functionality of flagging message errors. This message is implemented in the nanOrb environment, to facilitate the raising of exceptions when packets are incorrectly assembled and sent to the RCX. This will allow for the differentiation between the lack of an IR link and a ‘bad packet’ being the cause of a non-response from the RCX.

MESSAGE_FRAGMENT – This message is used when request or reply messages need to be fragmented due to either transport constraints or buffering efficiency on the client. Due to the unlikelihood that large encoded packets will be sent, it is unlikely fragmentation will be required for the simple application that we have targeted. Hence this message is not implemented. It is also highly unlikely that a severely resource constrained device would have the resources necessary to support the buffering and reassembly of larger packets.

3.4 The ORB Transport Protocol

Whereas the GIOP standard does not specify the transport protocol to be used, it does make certain assumptions as to the nature of this underlying transport mechanism. It expects a connection-oriented protocol, which the RCX network protocol, the Layered Network Protocol (LNP), does not provide. Through the provision of a connection-oriented transport, the need for acknowledgements of GIOP messages is alleviated. The lack of this functionality has some important implications for designs of this nature.

Any RCX based ESIOP implementation cannot make these same reliable transport assumptions, as the underlying communications medium (Infra-red) and environment cannot reliably facilitate a true connection-oriented service. A ‘best-effort’ implementation was attempted (whilst assessing the technology) involving an extra layer of abstraction between the ES-IOP and LNP layers, providing a TCP like timeout and re-transmit function. This did not provide and significant improvements over the simple LNP functionality, once the RCX was sufficiently out-of-range or subject to destructive interference, re-transmission provided no improvement.

Another solution, which is more inclined to correcting IR transmission problems, would involve the RCX based robot reorienting itself between attempts to establish communication (to aid finding a better line of transmission to the tower) or perhaps even retracing its navigation steps so as to return to the last known location of reliable communications (the cost of buffering commands and how to retrace them then becomes a constraining factor). These approaches are not included in the current implementation.

An altogether different solution, perhaps for exploration in further development of this research, would involve using a ‘mobility layer’ (as described in the ALICE paper [28]), or a similar approach to provide this service. This would assume that the RCX is always in range of a particular

tower, but could perhaps provide a good demonstration of mobility support in a CORBA environment.

3.5 Communication End-Points

The concept of a communications end-point is necessitated within the underlying transport of any GIOP implementation. This end point must, typically, uniquely identify a server process on a particular host, which implements the functionality specified in the IDL interface for an object. Without this functionality, it is difficult to envisage an embedded device being able to provide a CORBA service to clients. In the case of IIOP this endpoint ultimately consists of an IP address and port number. LNP provides a very similar addressing mechanism (as described in chapter 2) where communication end-points are specified as host address, host port pairings. Using this system an individual packet can be addressed to a specific port (and hence a specific process) on a specific LNP host. An LNP host can be either an LNP enabled PC (using the LEGO Infra-Red tower) or an LNP enabled RCX brick. This system can be related very clearly to the IIOP addressing model, and facilitates the specification of LNP supporting Interoperable Object References.

3.6 An Object Addressing Format

The CORBA specification denotes an Object reference as “an object name that reliably denotes a particular object” [1]. The object reference provides “a handle to a specific implementation of an IDL derived object” [1]. An object that is accessible via a GIOP implementation is identified by an Interoperable Object Reference (IOR).

The format of an IOR includes a specific ORB's internal object reference as well as a transport based address for locating that ORB and hence the object (the IOR data structure is covered in more detail in chapter 2).

Any environment specific ORB implementation must specify a format for IOR's. In the nanOrb architecture for example, not only does the client require an IOR for the gateway in order to communicate requests, but a means is required to locate object implementations within the embedded environment. As previously stated, the underlying transport must provide some means of identifying communication end-points, these are used to facilitate the protocol specific 'Tagged Profile' in an environment specific IOR format.

As a result of the current architecture implementing application-specific gateway functionality, it is not actually necessary to implement any further IOR support. The IIOP IOR for this gateway provides all the addressing information the client requires in order to make an object invocation on the RCX and the gateway is aware of the available servers. It is however necessary to further develop this architecture if true embedded CORBA functionality is to be supported in the nanOrb environment.

The format of an embedded CORBA IOR should ultimately facilitate nanOrb server implementations advertising their supported objects along with the relevant addressing information for each protocol through which they can be reached (this being the intended function of the IOR). This 'advertising' would most likely be at design-time as opposed to run-time due to the constraints of the system (this is further discussed in chapter 6). Hence an IOR format is specified for the nanOrb ESIOP, which not only advertises the TCP/IP addressing information for the gateway, but also the LNP specific communication end-points for a server.

There were two possible IOR formats considered in this design process. The first involves extending the IIOP based IOR for the IIOP-ESIOP gateway host. This would involve adding a new ‘Tagged Component’ to the IIOP Profile ID, which would contain LNP addressing specifics enabling the gateway host to resolve RCX object references to specific RCX Host Address-Port pairs. Whereas this would enable the accurate addressing of the RCX from the Gateway, it is not a very extensible or flexible solution. At the very least it mandates that clients of an application must be IIOP capable in order to request a nanOrb service (unless LNP clients parsed the IIOP ‘Tagged Profile’ for this information).

The second format involves describes a new, LNP specific, ‘Tagged Profile’ for the IOR. The latter provides a truer CORBA functionality and defines a communication end-point specific to the LNP transport. In this way an IOR for any nanOrb service could contain the IIOP addressing specifics of a gateway host as well as the LNP addressing specifics of the target object implementation. Thus enabling a client to use this service via either transport, independently (assuming the client is capable of building messages foe either transport).

The three main components composing an IOR were introduced in chapter 2 of this document:

1) A ‘Type ID’, indicating the most derived type (or version) of the object, for example: “IDL:nanOrb/nanOrbDemoApp/1.0”

2) End-Point Information (Tagged Profile Data) for each specific transport supported by the implementation, for example, an IIOP profile would contain the Host Address (a DNS alias or IP address) and the TCP port number

3) The server specified Object Key, which allows the server to internally identify specific IDL implementations. (Note: An optional sequence of Tagged Components can also be included)

Hence, a Tagged Profile for an LNP based Object Implementation can be specified:

```
struct ProfileBody{

        Version             naniop_version;
        unsigned short      hostAddress;
        unsigned short      addressingmask; //facilitates resolution of Port ID
        sequence            objectKey;

};
```

Figure 16 A sample nanOrb Tagged Profile

Once this Tagged Profile is included in an IOR a client can theoretically access a nanOrb server either directly (if LNP and enabled and ESIOP capable) or via the gateway (if IP enabled). It is more likely in the context of an embedded CORBA implementation that the end client would use the IIOP related portion of the IOR to contact the gateway, which would in turn use the ESIOP portion to contact the relevant server (the assumption is that the IOR is available to both parties). In this model the client knows only of a standard CORBA IOR and its IIOP based implementation and the gateway determines the specific embedded implementation to contact. This approach to defining per-protocol connectivity information for IOR's is consistent with the CORBA IOR model and provides the greatest flexibility to clients.

3.7 The IIOP to ESIOP Bridge

When implementing an ESIOP a complimentary bridging function (or gateway) is typically required in order to facilitate normal IIOP enabled clients making requests of ESIOP servers and also the reverse. If IIOP based communication is not supported, the implementation is not CORBA 2.0 compliant [1]. This gateway should accept requests from any IIOP enabled CORBA client and convert these to form suitable to transmission over the ES-IOP and vice versa. In CORBA terminology, this function is termed an IIOP to ESIOP 'half-bridge'. The "half-bridge" term is used in discussing Inter-ORB Bridges. According to the [1] "mediated bridges" are those that use an agreed median message format when translating between proprietary ORB protocols. This median message format is IIOP. In the case of the nanOrb architecture, where one ORB is using IIOP, the bridge between IIOP and an ESIOP is termed a "half-bridge".

Hence for the nanOrb project a gateway is needed that will perform the bridging between IIOP and nanOrb ESIOP messages, or between the IP and LNP transports. There are some interesting design optimisations that can be made to such an implementation when it pertains to embedded systems, particularly if it can be assumed that a certain amount is known of the architecture and capabilities of the target devices as well as the application characteristics. These optimisations are ultimately aimed at removing some of the functionality (and hence processing) from the embedded device and making it the responsibility of the gateway.

It has already been discussed how a gateway host might assume all responsibility for ensuring the correct byte-ordering of messages in the ESIOP implementation, thus reducing demands on the embedded device. Other functionality, such as that provided by the Service Context information (see chapter 2) in GIOP messages, could also be made the responsibility of the gateway, to further free the embedded system of perhaps unnecessary processing effort. The security information contained in the Service Context

field of an IIOP client REQUEST, could conceivably be validated by the gateway, rather than the target system. Transactional support could conceivably be made more of the gateways responsibility in the same way. There are of course certain implicit assumptions as to the trusting of the gateway host. The only optimisation that is currently implemented in the nanOrb design is that of responsibility for byte-alignment, which is assumed to be the gateways.

Although this bridging function should be transparent to the client and non-application specific, the latter is not the case for the nanOrb implementation. In order implement the functionality in a transparent fashion, as in the architecture outlined in chapter 1, it is necessary to build the functionality into an ORB implementation. This was not done in the nanOrb architecture purely as a result of time constraints. A simple CORBA server application builds the packets explicitly. Upon receipt of invocations and dispatches these to the RCX.

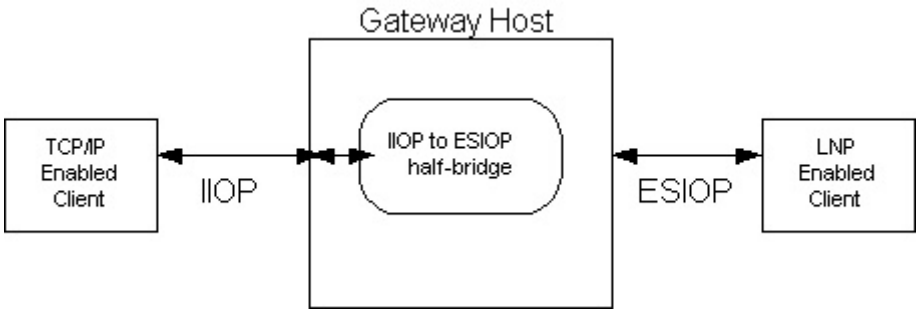

Figure 17 The IIOP to ESIOP Half-Bridge

3.8 The Environment Specific ORB on the RCX (nanOrb)

The ultimate goal of this project is to investigate if, and to what extent, ORB functionality can be implemented on the RCX, and hence any similar embedded system.

The core ORB in any embedded implementation must be as efficient as possible, in order to reduce memory consumption, and still allow application logic to be facilitated. It will also most likely be an application tailored implementation, in order to maximise efficiency (that is tailored to support only the functionality required by the application it supports). Irrespective of the final implementation, from a development perspective, the ORB should endeavour to provide as consistent an API as possible to the application (it is conceivable that some altering of this API will be necessitated for different applications, due to the minimal implementation nature of the ORB).

Within the context of the aforementioned design, the nanOrb embedded ORB implementation must at the very least implement a server process to listen for incoming object invocation request on an advertised port (in this case an LNP port). This port may be advertised in an environment specific IOR or a well-known port. Upon receipt of an ESIOP (LNP) packet, the packet must be de-marshalled, in order to obtain the data in each of its fields. The relevant fields must then be validated to ensure that the packet is correctly formed, before the actual invocation on the relevant object is called (with any necessary arguments, as defined in the application IDL) and a response message is built, encoded, and sent to the requesting client. If any of the aforementioned procedures fail a MESSAGE_ERROR message must be built and marshalled onto a buffer before being sent back to the client. It should be mentioned again, that where possible, the gateway can be used to perform “server tasks” and hence reduce the load on the embedded system.

These key requirements can be summarized as follows:

- Support the full IDL of the target application
- Create processes to listen on the relevant ports for requests
- Unmarshal received data
- Validate message structures to ensure they are correctly formed
- Pass the relevant unmarshalled arguments to object invocations
- Build and encode RESPONSE messages as appropriate to the interface IDL
- Build and encode MESSAGE_ERROR messages when necessary

The embedded ORB would of could of course require much more functionality beyond the aforementioned, if it were intended to support more CORBA functionality.

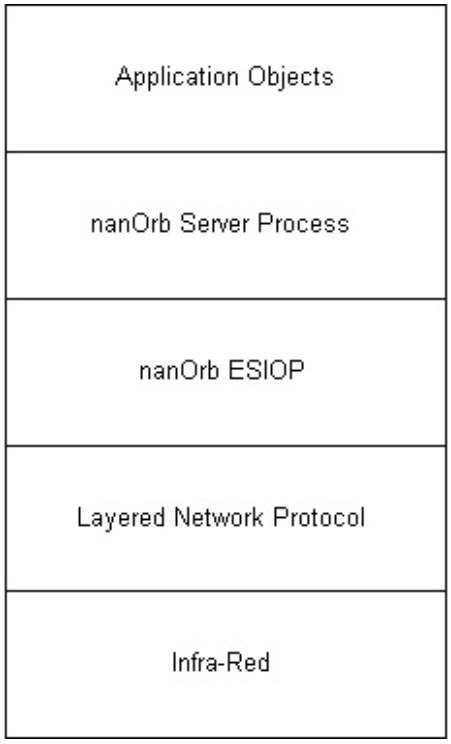

Figure 18 The nanOrb Hierarchy

Whatever the functionality that is supported by the ORB implementation, it must ultimately present the user with a consistent set of API's that facilitate

the ‘easy’ linking of application code with the ORB server. The nanOrb implementation facilitates this by defining a ‘handling’ function for each of the incoming IDL operations, within which the relevant application logic can be executed.

The nanOrb architecture starts with the rudimentary infrared transport. LNP then provides an addressing mechanism and byte stream abstraction to this. The nanOrb ESIOP defines how GIOP messages, and hence object invocation data, can be mapped onto this transport protocol. The nanOrb architecture then implements server processes that listen on LNP defined end-points for incoming ESIOP packets. It is on top of all these layers that the application code resides. Object implementations are hence located within the aforementioned handling functions for the incoming IDL operations.

3.9 Comparison to other related designs

It can be seen from the previous design steps, that a certain amount of customisation of, or variation from, the GIOP standard has taken place. This is typically the case when implementing CORBA in embedded systems.

The Kim [20] paper describes an Environment Specific CORBA, which not only is a subset of functionality of the 8 GIOP messages implemented in an ESIOP, but the data encoding standard is also customized (and referred to as the “Compact Common Data Representation)” in order to efficiently use the small, eight byte, payload of the Controller Area Network (CAN) bus. The implementation does maintain full IDL support. The nanOrb design, in contrast to this, does not modify the data encoding rules defined in the CORBA CDR standard (as the LNP payload is far more generous), the same byte-boundary alignments are implemented. There is an implicit assumption here that the payload of any nanOrb message is not likely to include any more than a small number of arguments, on account of the processing capabilities of the RCX being very limited). Hence an IIOP based message payload, could conceivably be copied from the TCP/IP input buffer in a gateway

implementation, to the LNP output buffer, once the underlying architectures used similar byte ordering and the maximum payload of the LNP packet (253 bytes) was not overrun.

Whereas the nanOrb design does not modify some of the basic encoding rules defined in the CORBA specification, it does fall short of a minimumCORBA implementation, it only implements three of the eight GIOP defined messages and does not support Service Context and Principal information. This is not to say that a more complete (and even fully minimumCORBA compliant) implementation is not possible, but the ultimate aim of this report is to investigate the suitability of the CORBA to the environment and the design goals have been kept accordingly simple.

Chapter 4

Implementation

This chapter describes the development of the simple “proof-of-concept” application that was implemented to verify the design introduced previously. The application architecture is introduced and described, along with the underlying nanOrb infrastructure and the IDL specification. The application is successful in demonstrating the gateway functionality and the final invocation of a client request on an RCX. The difficulties encountered and assumptions and optimisations made are also presented.

4.1 Implementation Goals

The aim of implementing this application was to validate the design presented in the preceding chapter. Ultimately the application should facilitate a remote client, using the Internet Inter-ORB Protocol (IIOP) to make an object invocation on the RCX, via the gateway. This is accomplished though the client invoking an IDL derived object invocation on the nanOrb gateway (half-bridge), which in turn relays this to the RCX, via the infrared transport mechanism, having made the necessary packet conversions. These packets are then received by the RCX, unmarshalled and the appropriate application code is called. The application also demonstrates that the nanOrb ESIOP implementation can be used between RCXs to remotely invoke operations. The application design was formulated to clearly demonstrate the RCX responding to the client’s invocations and that one RCX can make an invocation on a second via the nanOrb ESIOP.

4.2 The Application and its IDL specification

The demonstration application used consists of the remote client sending user-specified spatial navigation commands to an RCX controlled robot, deemed to be the “Master”, via the gateway. The gateway converts these commands into the nanOrb defined ESIOP message format and sends them to the RCX. The set of operations consists of “forward”, “reverse”, “left” and “right”, and all take a time metric (in seconds) as an argument indicating the intended duration of the movement. This RCX then relays these invocations to a second RCX, deemed to be the “Slave” via the ESIOP, hence demonstrating inter-RCX object invocation. A simple read-only attribute is also included in the IDL specification to facilitate the reading of the Master RCX’s Host_ID (as used in LNP addressing).

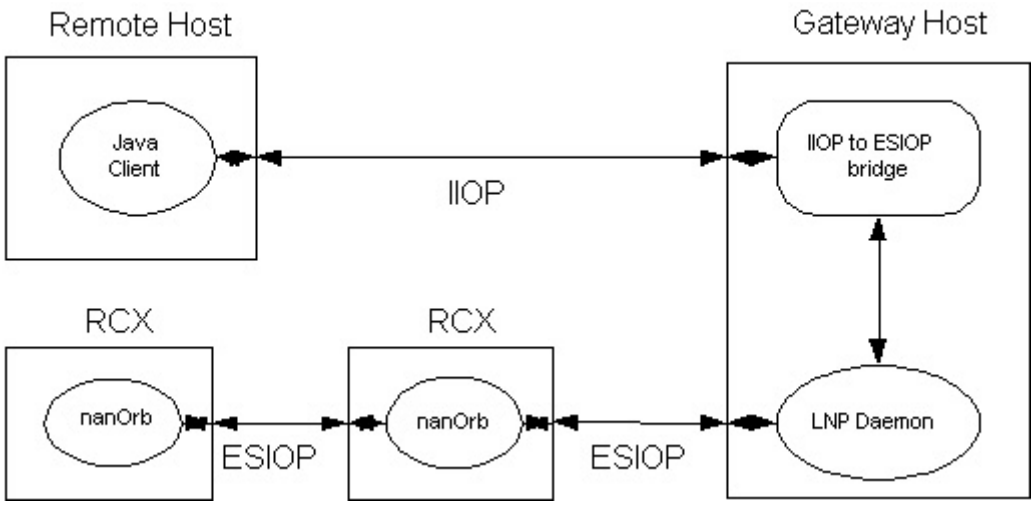

Figure 19 The nanOrb Application Architecture

The Interface Definition Language (IDL) specification of the demonstration application is defined as shown in figure 18:

```
interface nanOrbDemo{

        readonly attribute short currentMaster;

        void forward(in short forwardMetric);
        void reverse(in short reverseMetric);
        void left(in short leftMetric);
        void right(in short rightMetric);

};
```

Figure 20 The nanOrb application IDL

4.3 The Client Implementation

The client functionality is written using the Java language. The Java Development Kit, version 1.2, includes support for CORBA, specifically IIOP, functionality as well as GUI Development. The Sun “idltojava” tool is used to generate the client application stub classes. The *org.omg.CORBA.ORB* class is used to provide the core ORB communications functionality, along with the *org.omg.CosNaming* class for providing CORBA Common Object Services (COS) Naming functionality. The Java ‘Swing’ classes are used to construct the user interface.

The final application contacts a COS Naming Server (OmniNames, which is implemented on the same host as the gateway for this project) to resolve the Interoperable Object Reference (IOR) for the nanOrb gateway server. That is the IDL generated stub classes are used to create a “nanOrbDemo” object and invoke methods upon it. These invocations are hence relayed to the nanOrb gateway server via the IIOP, as though it were a normal IIOP server.

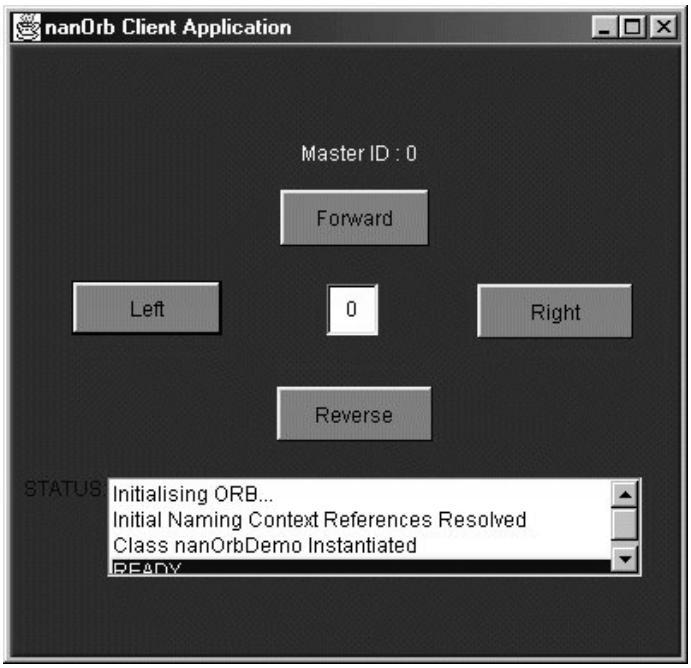

Figure 21 The Client Application

4.4 The Gateway Implementation

The nanOrb gateway performs the "half-bridge" functionality discussed in the previous chapter, translating between the IIOP and the nanOrb ESIOP message formats. Whereas this bridging functionality would typically be built into the core of an ORB implementation, for the purposes of demonstration and to avoid unnecessarily building a full ORB implementation, the bridging functionality is implemented at the application layer (this is known as 'request-level bridging' [1], as opposed to 'inline bridging' where the bridging functionality is implemented within the ORB). Through implementing it in the application layer, it is possible to better demonstrate the process of building the nanOrb ESIOP packets and encoding them, before sending them to the RCX.

The gateway is implemented on the Linux operating system (using Red Hat version 6.2). The AT&T "OmniORB 3.0" ORB implementation (which has been tested and certified CORBA 2.1 compliant [14]) is used to provide the

CORBA Server functionality, in conjunction with the “OmniNames” COS Naming implementation. The former was chosen as it is a fully IIOP compliant high-performance ORB and is also supplied with source code, thus facilitating debugging and/or further development, if required. “OmniNames”, which facilitates run-time object registration and look-up, is used to provide more flexibility in client and server development, that is to reduce the need to copy “stringified IORs” between machines.

Once compiled and configured, using the GNU “binutils 2.91” and “egcs-1.1.2” compilation environment [30], OmniORB provides the CORBA development environment, and OmniNames, the run-time support, for the gateway server processes. All code is written using the C++ language and compiled using the aforementioned GNU tools.

The gateway application, implements the ‘nanOrbDemo_I’ interface, derived from the IDL generated skeleton class. Hence, upon receipt of an IIOP request from the client, the gateway server invokes the appropriate method of this ‘nanOrbDemo’ implementation. It is within the call to each of these methods that the nanOrb bridging functionality is implemented.

This bridging functionality utilises the C++ classes outlined in the next section, which provide the ESIOP messaging and marshalling/unmarshalling functionality. These classes facilitate the building of request messages according to the design specification and their encoding for transmission over the LNP transport.

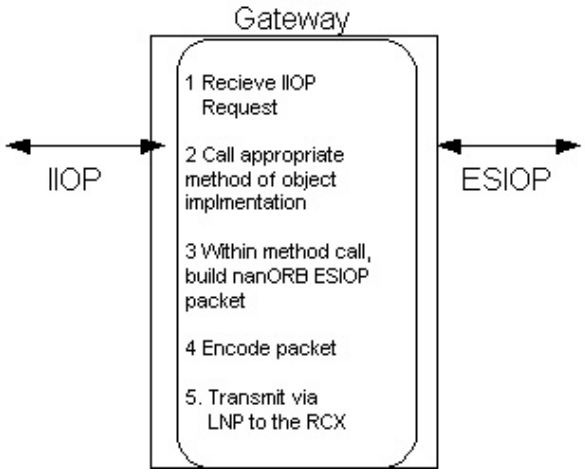

Figure 22 Gateway translating IIOP to ESIOP requests

For the purposes of demonstration, once the ESIOP packet for a particular request is built, the gateway echoes the packet structure to the screen, in order to illustrate the process (Note: The garbled data after the Magic field of the header is a result of it not containing a '\0' character).

```
Method right(6) invoked

Protocol Header Fields:
MAGIC:                  GIOPx-'@3
MAJOR:                  1
MAJOR:                  1
FLAGS:                  0
Message Type:           REQUEST
Message Size:           47
Request Message Header Fields:
SERVICE CONTEXT LENGTH: 0
REQUEST ID :            0
RESPONSE EXPECTED:
RESERVED:               000
OBJECT KEY LENGTH:      11
OBJECT KEY:             nanOrbDemo
OPERATION LENGTH:       6
OPERATION:              right
PRINCIPAL LENGTH:       0
SERVICE CONTEXT:        0
Request Message Payload:
ARGUEMENT:              6

Sending Marshalled Data...

OK
```

Figure 23 Gateway output in response to a "right (6)" invocation

4.5 The nanOrb ESIOP implementation

The ESIOP functionality is implemented using a series of C++ classes, several of which are derived from those used to provide the same functionality in the ALICE [28] and KORB [18] projects. Within the implementation of the IDL specified “nanOrbDemo” server, packets are constructed within the respective interface methods and then sent to the appropriate nanOrb servers.

These packets are built using the argument from the client request. Two simple C++ classes, ‘simplenanIOPHeader’ and ‘simplenanIOPRequest’, provide the protocol header and REQUEST message functionality, respectively. Once the nanOrb ESIOP message is created it is marshalled onto a data buffer, using another C++ class, ‘simpleEncoder’ which provides the basic CDR-derived, byte-aligned data encoding functionality. This marshalled data is then sent to a final wrapper class for the LNP Daemon (‘simpleLNPTransport’) for transmission to the “Master” RCX. The application logic within these classes is used again in the implementation of the nanOrb server.

4.6 The RCX ORB implementation - nanOrb

Although the gateway is implemented using the C++ language, it is not possible to use it for the RCX implementation. Whereas the GNU cross compilation environment for the Hitatchi HD6433292 does provide C++ language support, it is not as extensive as its C language support, hence the latter is used for the implementation of all nanOrb functionality on the RCX. As a consequence of this, much of the object-oriented design work done for the gateway can not be re-used in the RCX environment, but a lot of the more generic C language code can.

The “Master” RCX’s port number 8 is used as an address for the nanOrb implementation (with the Host address of 1). This is specified as a ‘well known’ address within the gateway implementation. The previous chapter discussed how this might further be developed to support the exporting of nanOrb IOR. Hence the TAGGED_PROFILE portion of the application’s IOR would be constructed as illustrated:

```
nanOrb_ProfileBody{

        Version.major = 1;
        Version.minor = 0;
        HostAddress = 0x18;
        AddressingMask = 0xF0;
        objectKey = “nanOrbDemo”

};
```

Figure 24 The nanOrbDemo IOR

Upon receipt of data on this port, a hardware interrupt is raised which awakens a process assigned to that particular port. The incoming data is copied to a global array, before a second packet processing routine is awakened via a semaphore. This separation of receiving and processing logic is necessary as the execution of any significant instructions within any hardware- interrupt driven routine is inherently unstable.

The packet processing routine is responsible for ensuring the validity of the packet, through checking that all the necessary protocol and message header fields are correctly formed. Once the message has been validated, the invocation data is removed and the application code (the implementation of an IDL specified method) is invoked and passed these client specified arguments. The mapping between invocations and their demarshalled parameters and their actual implementation is facilitated by a short section of switching logic, which the application developer is expected to modify in order to support the application code. If a packet is incorrectly formed (that is it does not comply with the nanOrb packet structure), a MESSAGE_ERROR routine

is invoked. The full functionality for building and returning the MESSAGE_ERROR and REPLY messages has not yet been completed. This is due to some of the difficulties highlighted in the next section (the linker error).

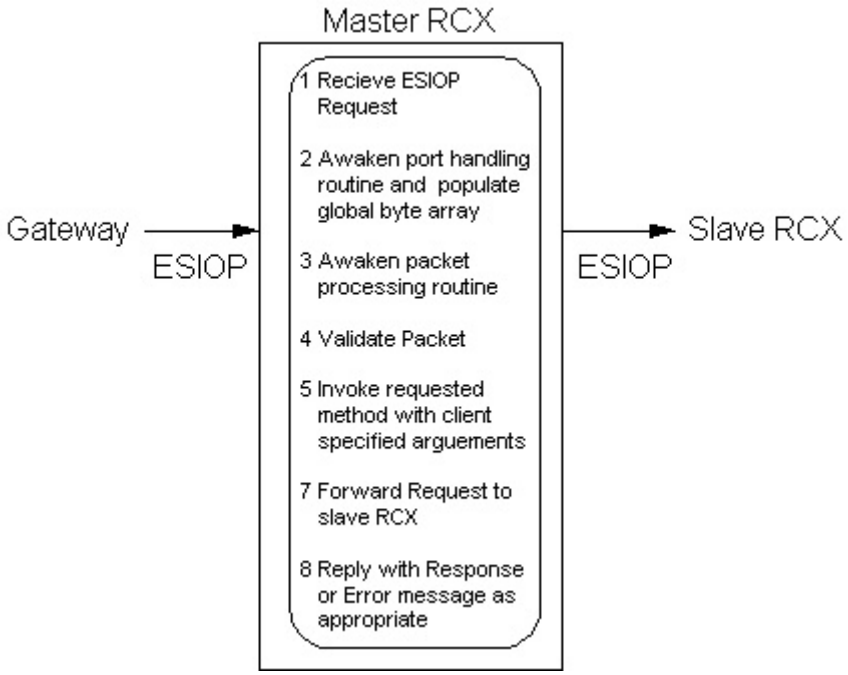

Figure 25 nanOrb Processing Client Requests

Once the actual IDL method implementations are invoked, the a short string (constructed using the first and last characters of the invocation string) is presented on the RCX's LCD display. The value of the first argument is also displayed, thus demonstrating the successful communication of the client request.

4.7 Difficulties Encountered

Due to the nature of the application, particularly the fact that it involved embedded systems programming with an unsupported toolset, a number of difficulties were encountered during the implementation. The more serious difficulties are outlined in this section along with their resolutions, where possible.

It was originally intended to use the Microsoft Windows Operating System as the platform for the gateway implementation, in conjunction with the WinLNP COM server [31]. The ‘Cygwin’ application [32] provides a GNU based compilation environment for the Windows operating system, which enables the use of the GNU cross-compilers for the Hitatchi HD6433292. Whereas, it was possible to use this tool to facilitate the programming of the RCX using legOS, the WinLNP COM control proved not to be as mature and reliable as the Linux based LNP Daemon (LNPD). It was therefore decided, after initially beginning development on the Windows platform, to migrate to the Linux platform.

There were however some difficulties with enabling the LNP Daemon. These were eventually found to be caused by the Universal Asynchronous Receiver Transmitter (UART) chip in the Linux machine’s serial port. The 16550AF UART chip has a documented transmit problem that can result in the random loss of a byte. If the FIFO functionality is enabled, it can occasionally fail to transmit a character. The character does not transmit and no interrupts are generated, hence the ‘user’ is not aware of any failure. By turning this FIFO functionality off (using the Linux “setserial ” command), the problem is resolved and the LNP Daemon functions properly.

A more serious difficulty was when using the previously mentioned “liblnp.h” library within the OmniORB environment to request LNP functionality from the LNP Daemon process. The problem encountered involved any call to the LNP Daemon to transmit data, from within an OmniORB process, blocking

indefinitely and not returning. The data was sent to the LNPD process and received by the RCX, but the call never completed within the OmniORB server. This LNP functionality was provided by the aforementioned C++ wrapper (‘simpleLNPTransport’) class, which worked correctly when instantiated outside of OmniORB, but failed when called as described. This issue has not as yet been resolved. A second standalone application is also implemented, which facilitates the demonstration of the gateway functionality and the RCX’s response to invocations.

The most serious difficulty encountered in the implementation of the application involved the GNU cross-compilation environment for C code. When certain simple C code statements were compiled to object code, incorrect linking information was generated, which caused the linking process to fail. This problem was caused by the compiler optimisation of certain relatively simple code structures. When these were simplified, through breaking them into simpler assignments and evaluation statements, the code compiled and linked correctly. This problem manifested itself on several occasions and, in each case, took considerable effort to locate the problem code and resolve. The debugging process when developing on the RCX involves writing hexadecimal information to the LCD screen on the RCX. Every time a change is made to application code, it must be recompiled in the cross-compilation environment and then downloaded (via the infrared link) to the RCX before it can be tested. The entire development process is hence iterative and extremely exhaustive.

It can be seen that whereas the implementation ultimately succeeded, it was not without its difficulties and shortcomings. These were largely due to the lack of support and prior knowledge in this domain. Indeed these problems are typical characteristics of the embedded system development process and can ultimately be resolved.

Chapter 5

Evaluation

This chapter reviews the nanOrb design and implementation, with a view to determining the extent to which it and therefore CORBA is suited to the Mindstorms environment and to hence draw more general conclusions as to the applicability of CORBA to the embedded system domain. The efficiency of the nanOrb implementation is explored and the justification for each of the design decisions made is investigated and the ramifications explored. The design is then compared to the other embedded CORBA approaches outlined in chapter 2 and the potential for improving the nanOrb architecture is explored.

5.1 Overview

The nanOrb implementation enabled the Java-based IIOP client, and hence any IIOP enabled CORBA client to make requests of the nanOrb gateway, which were in turn relayed to the RCX based environment specific ORB (nanOrb) and successfully interpreted and executed. The fact that any intermediate bridging function or transport mechanism other then IIOP was in use was transparent to client. The application also demonstrated the use of the ESIOP implementation to facilitate inter-RCX messaging. It was, in this capacity, successful in demonstrating that CORBA can be used to present simple interfaces to embedded systems.

The approach taken in designing the application was to sacrifice anything other than basic COBRA functionality in order to free more physical resources on the embedded system for application logic. This trade-off is

characteristic of almost any embedded-ORB implementation. The more CORBA functionality provided, the more resources consumed, ultimately reducing those available to the supported applications and conversely the less CORBA functionality implemented, the more resources available to supporting application logic.

Whereas the unreliable nature of the infrared transport was accommodated as much as possible, the nanOrb implementation and demonstration application do not fully explore the mobility capabilities of the RCX and hence its effects on an ORB implementation. This is due to the time constraints to which this work was subjected. It is anticipated that further development might explore this in more detail and even enable mobility support, perhaps using an architecture similar to that described in the ALICE paper [28].

5.2 Efficiency

The final nanOrb implementation, supporting request, reply and error messaging, is approximately 5k in size (note: this approximation is due to the fact that the functionality required for the latter two messages is not yet debugged, but is completed enough to make a reasonable approximation). This 5k program is hosted on the legOS firmware, which occupies a further 20k or memory. Thus approximately 7k of memory remain available for supporting application logic on the RCX. It is possible to further reduce the size of either the legOS firmware or indeed the nanOrb functionality. By removing the legOS support of RCX various functionality, such as button events and the different sensor inputs, and recompiling the firmware, it was possible to further reduce its size to approximately 16k (it was actually possible to further reduce this footprint, but this necessitates removing LNP and LCD support). The nanOrb footprint was also experimentally reduced through removing code supporting reply and error messages and simply silently discarding erroneous messages. This yielded a footprint of only 3.5k.

5.3 Optimisations and Consequences

Whereas the nanOrb implementation does facilitate the development of CORBA based applications in the Mindstorms environment and potentially other embedded environments, it is not without its shortcomings. The design chapter outlined several optimisations (or customisations) that were made to the CORBA speciation in order to accommodate the environment. These compromises ultimately constrain the capabilities of the architecture. The key objective of the CORBA specification is to facilitate the distribution of software objects in a heterogeneous environment and facilitate the transparent development of these. The nanOrb architecture does facilitate the former, but not to the full extent that a complete CORBA implementation does. It also compromises some of the transparency that the developer might expect of a CORBA implementation.

The ESIOP message set was reduced to facilitate the basic object invocation functionality of CORBA, but little else. The remaining five messages were excluded for performance reasons (the justification for which is provided in chapter 3). It is possible that these messages could provide functionality required in a different embedded environment than the Mindstorms environment, supporting applications of a different nature to the presented “nanOrbDemo” application. The lack of this functionality (although it may be masked by the gateway implementation) reduces the client’s capability to transparently send CORBA messages to the server. It is clear that an ESIOP implementation facilitating a more complete message set is possible. This would of course provide a more complete CORBA functionality to clients but ultimately limit what the RCX server applications would be capable of, as they would have minimal memory remaining to support their logic.

The IDL and corresponding marshalling/unmarshalling support designed is limited to the set of IDL defined primitive data-types. This means that a programmer wishing to perform operations utilising the more complex

data types must ultimately perform some of the nanOrb marshalling/unmarshalling explicitly, using the provided API. This support could easily be added to the existing implementation, it was not included in the current design as it was not deemed to be of importance to the projects objective, that being to demonstrate CORBAs applicability to the embedded domain.

The assumptions that CORBA makes of the underlying transport mechanism were also relaxed in the nanOrb implementation. The Mindstorms environment (explicitly the infrared transport mechanism) cannot, by its very nature, guarantee the delivery of data, as CORBA expects. Chapter 3 outlined some of the implications of these constraints, most noticeably that a client of the embedded ORB (the gateway) should not block on making a request and also the need for the ‘concealment’ of these failings from regular CORBA clients. It is quite likely that embedded ORB implementations in environments other than the Mindstorms environment will be able to provide this connection-oriented service that CORBA implementations expect.

The focus of this project was on allowing CORBA compliant clients to make object invocations on embedded systems; the design has not addressed the inverse of this model. The current model does not facilitate embedded devices making client requests of normal CORBA compliant servers. The nanOrb implementation does demonstrate the RCX acting as a client to a second RCX. True embedded ORB implementations should of course facilitate the embedded applications interacting with normal CORBA services as both clients and servers. The existing nanOrb implementation would require additional functionality in the gateway as well as some means of resolving IORs if it were to support the RCX’s acting as clients to normal CORBA services.

The final optimisation involved the assigning of certain (CORBA specified) server responsibilities to the gateway implementation. This is not considered to be an unreasonable design optimisation in the context of embedded systems. It is not unreasonable to assume that the gateway

controlling access to CORBA enabled devices will be aware of their underlying architecture. The assignment of byte-alignment compliance to the gateway is therefore not considered a huge compromise. In much, the same way, the conjecture that the gateway might ultimately handle all service context processing, is also not unrealistic, although not implemented.

5.4 Architecture Comparison

There are several approaches to providing CORBA interfaces to resource-constrained devices; these have already been introduced in chapter 2. The first of these is the comparatively simple approach of implementing a CORBA enabled proxy, which in turn communicates with the embedded device via some proprietary serial protocol, but still providing the object abstraction to the client. This approach whilst perhaps facilitating CORBA control of the devices, does not ultimately bring CORBA to the embedded system (rather it brings the embedded system to CORBA). Clients must use this proxy to communicate with the system, and the embedded systems themselves are not aware of the CORBA oriented services they provide. The IDL defined application interface is actually implemented on the proxy and the necessary translations are made to messages so the embedded device may respond. This approach is used in various applications, for providing standard CORBA interfaces to network elements for example. It facilitates amore centralised management architecture, without actually absorbing the cost (in terms of hardware and software engineering) of implementing CORBA on the device. It does not ultimately make the embedded device “CORBA enabled”.

The nanOrb implementation provides a more flexible architecture than this and actually “CORBA enables” the embedded device. Through defining a transport specific ‘tagged profile’, any LNP enabled client can contact the RCX, or alternatively an IP capable client can contact the gateway to relay requests. In both cases the invocation data is relayed to the RCX, where the

implementation of the IDL defined interface is located. Hence the RCX is ‘CORBA aware’. It is important to note that the RCX based nanOrb implementation is ultimately application specific, although every effort has been made to provide a consistent platform (or API), it must ultimately be somewhat modified be due to the resource constraints of the RCX. The implementation does however provide a consistent platform upon which to develop applications, where extraneous functionality can be removed from the final implementation.

The Object Management Groups (OMGs) ‘*minimumCORBA*’ standard is intended to facilitate a full CORBA functionality for embedded devices. It is a far more costly implementation than the nanOrb implementation described in the previous chapter. Whereas the specification removes some of the more unnecessary CORBA services, it still implements a large set of functionality. This report has demonstrated that embedded devices can be made available to standard CORBA clients without this full implementation. The nanOrb application illustrates that the embedded devices often do not need this complete functionality; much of it can be facilitated on a bridging host. This approach may be viewed as a hybrid of both of the previous approaches (using a CORBA enabled proxy and implementing minimumCORBA), but still enables the CORBA interface to the device. The assumption within this statement is that the gateway application is already necessitated, through the utilization of a transport other than IIOP, to perform the aforementioned ‘half-bridge’ function. By keeping the ESIOP messages consistent with the format of their GIOP counterparts, albeit only a subset, it is also possible to facilitate clients, that are capable of using the embedded environments’ transport mechanism, directly request service of the devices. This support is, of course, only for the messages that the embedded device recognises.

5.5 Improving the nanOrb Architecture

It has been illustrated that the process of designing embedded CORBA implementations is frequently one of compromising. That is, in order to accommodate the specifics of the embedded domain, certain CORBA expectations or features are ultimately relaxed or optimised as appropriate. The previous sections have highlighted what the nanOrb implementation has achieved, in terms of providing CORBA services on embedded devices and also the portions of the CORBA specification it has implicitly removed or altered. Each of these presents an opportunity for improvement. However, the more any specific CORBA functionality is facilitated, the less resources are ultimately available for other improvements and/or application logic.

Perhaps the biggest single improvement to the specification would be to develop the gateway functionality to make it a more generic, as opposed to application specific, and to facilitate nanOrb devices acting as clients to regular CORBA services. This functionality, perhaps coupled with a better definition of a nanOrb API, would facilitate the development of CORBA applications in the Mindstorms environment without the need to modify the infrastructure beyond some simple reconfiguration. This type of functionality is of course, what all CORBA implementations aim for. It is however doubtful that any embedded CORBA infrastructure can be completely transparent to the application developer, due to the nature of embedded programming and the constraints it imposes.

Chapter 6

Conclusion

6.1 Work Completed

The aim of this report, as stated in the abstract, is to investigate the applicability of the Object Management Groups Common Object Request Broker Architecture to presenting interfaces to severely resource constrained embedded devices. This investigation is facilitated through the implementation of an environment specific CORBA application. The design process appropriate to implementing an Environment Specific Inter-ORB Protocol (ESIOP) has been presented, in conjunction with several optimisations, which can be made to facilitate the efficient implementation of an embedded ORB. This process has been further developed and demonstrated through its application to the demonstration environment, the LEGO Mindstorms kit, and ultimately illustrated by the “nanOrbDemo” application. Lastly the successes and failings of the implementation have been reviewed.

The overall process has involved customising (reducing) the GIOP message set in order to enable a minimum footprint on the RCX brick as well as relaxing some of the GIOP transport assumptions, due to the nature of the underlying infrared transport mechanism. Certain portions of the ORB functionality have been removed from the embedded device and implemented on the bridging host, again to reduce resource consumptions.

The final nanOrb application clearly demonstrates that CORBA can be applied to embedded devices, enabling the inter-communication of normal CORBA requests to and their execution. It also illustrates alternative approaches to full providing this functionality and optimisations that can be exploited to better accommodate the embedded system. The inverse

relationship between the amount of CORBA functionality supplied and the amount of application logic that can hence be accommodated has been highlighted at various stages of this report.

The previous assessment chapter highlighted the various assumptions made (and their implicit compromises) in designing the nanOrb architecture. It is conceivable that the application of this architecture to a different embedded environment, supporting different distributed applications, would result in the imposing of different operating characteristics and end-functionality requirements on its implementation. The applicability of any embedded CORBA solution is ultimately largely dependent on its environment.

6.2 Work Remaining

The implementation of the REPLY and ERROR message functionality specified in the nanOrb design must be completed. This is just a debugging process; the core application logic is in functional.

6.3 Further Research

Whereas this report does demonstrate the provision of basic CORBA request functionality on the RCX, and hence embedded systems, it does not fully explore the extent to which this functionality can be extended. Whereas it was not attempted to implement a full *minimumCORBA* implementation (that was not the ultimate focus of this project), there is scope for the implementation of more than the aforementioned basic request functionality. Further work might implement this functionality in a logically progressive fashion and ultimately draw conclusions as to the relationship between the provision of functionality and the consumption of resources, hence illustrating the ‘balance’ that can be achieved with implementations.

It may also be possible to attempt a full *minimumCORBA* implementation on the RCX. This would ultimately consume all physical resources (memory) and hence most likely only facilitate a CORBA interface to the underlying hardware, as opposed to accommodating considerable application logic. It is anticipated that this activity will be pursued in conjunction with the K-ORB project [18].

The nature of the Mindstorms robots, their operating characteristics and the available toolset provides a rich environment for the exploration of mobility support within the embedded CORBA field. The infrared transport and addressing mechanism could enable the use of multiple PC's acting as mobility gateways to the RCX's as described in the Alice paper [28]. The application might also explore autonomous disconnected operation characteristics.

Bibliography

[1] OMG, “The CORBA Specification”, 2000
http://www.omg.org/technology/documents/formal/corba_2.htm

[2] The Object Management Group, “The *MinimumCORBA* Specification”, 1999
http://www.omg.org/homepages/realtime/rfp/real-time_minimum_corba.html

[3] Iona Technologies, “The IONA IIOP Engine”, 2000
http://www.iona.com/products/embsys/whitepaper.html

[4] The Object Management Group, “OMG Formal Documents” 2000
http://www.omg.org/technology/documents/formal/

[5] IEE, “The Millenium Problem in Embedded Systems” 1997
http://www.iee.org.uk/2000risk/old/archive/emb.htm

[6] P. J. Koopman, “Embedded System Design Issues” : Carnagie-Mellon University, 1996

[7] N. Furihata, “Distributed Embedded System Design” : University of Southern California, 1999
http://www-scf.usc.edu/~furihata/research.html

[8] B. Cole, “Architectures overlap applications,” *Electronic Engineering Times*, pp. 40,64-65., 1995

[9] T. Wong, “CORBA in the Embedded Systems Context” : Highlander Communications, 1999

[10] J. Ganssle, “The Challenges of Real-Time Programming” : Embedded Systems Programming, 1998
http://www.embedded.com/98/9807fe.htm

[11] The Object Management Group, “What is CORBA”, 2000
http://www.omg.org/corba/whatiscorba.html

[12] Redhat Corporation, “The ORBit Project Homepage”, 2000
http://www.labs.redhat.com/orbit/

[13] Object Oriented Concept Inc., “The Orbacus Homepage”, 2000
http://www.ooc.com/ob/performance.html

[14] AT&T Research Labs Cambridge, “OmniORB Home”, 2000
http://www.uk.research.att.com/omniORB/

[15] The OCI Host Site, “The ACE ORB”, 2000
http://www.theaceorb.com

[16] D. C. Schmidt, “High-Performance CORBA”, 2000
http://www.cs.wustl.edu/~schmidt/corba-research-performance.html

[17] Lockheed Martin, “The HardPack Fact Sheet”, 2000
http://www.hardpackorb.com/factsheet.pdf

[18] J. Dowling & A. Edmunds, “The K-ORB Project”, Trinity College Dublin, 2000
http://www.dsg.cs.tcd.ie/research/minCORBA/index.html

[19] A. S. Gokhale and D. C. Schmidt, “ Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems” Washington University 1998

[20] K. Kim, G. Jeon, S. Hing, S. Kim, and T. Kim, “Resource-Conscious Customisation of CORBA for CAN-based Distributed Embedded Systems” IEEE 2000

[21] F. Martin, “The MIT Programmable Brick Project”, 1999 http://el.www.media.mit.edu/projects/programmable-brick/

[22] S. Holdren, “Spirit OCX”, 2000 http://www.holdren.com/scott/legos/

[23] D. Baum, “Not Quite C”, 2000 http://www.enteract.com/~dbaum/nqc/index.html

[24] R. Hempel, “Forth for Mindstorms”, 1999 http://www.hempeldesigngroup.com/lego/pbFORTH/

[25] J. Solorzano, “The TinyVM Project Homepage”, 2000 http://sourceforge.net/projects/tinyvm

[26] M. L. Noga, “The legOS Project Homepage”, 2000 http://www.noga.de/legOS/

[27] A. D. Joseph, “Mobile Computing with the Rover Toolkit,” *IEE Transactions on Computers: Special Issue on Mobile Computing*, 1997

[28] R. Cunningham, “Architecture for Location Independent CORBA Environments (ALICE)”, Trinity College Dublin, 1998

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

[30] The GNU Software Foundation, “The GNU Homepage” 2000

http://www.gnu.org

[31] Y. Jeannnotat, “The WinLNP Home Page” 2000
http://www.geocities.com/WinLNP

[32] CygnusSolutions Inc., “Cygwin Homepage” 2000
http://www.cygnus.com/cygwin/