INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]

Flyover – a Technique for Achieving High Performance CORBA-Based Systems

by

Wai Keung Wu, B.Sc (CS)

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Applied Science

Ottawa-Carleton Institute for Electrical Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

August 27, 2001

© Copyright 2001, Wai Keung Wu

**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-66896-7

Canada

The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

**Flyover – a Technique for Achieving High
Performance CORBA-Based Systems**

submitted by Wai Keung Wu, B.Sc (CS),
in partial fulfillment of the requirements for
the degree of Master of Applied Science

Chair, Department of Systems and Computer Engineering

Thesis Supervisor

Carleton University

August 27, 2001

Abstract

Common Object Request Broker Architecture (CORBA) is becoming increasingly popular as a distributed object computing middleware which provides an interoperable and reusable architecture for the development of distributed applications. Most existing middleware-based systems have focused on functional correctness and have ignored system performance. Although some CORBA implementations are optimized for performance, they do not obtain the performance gain by exploring limited heterogeneity in systems. In a system with limited heterogeneity, some of the communicating components are made of similar hardware and software. Examples of such systems include embedded and real-time systems, and telecommunication systems.

This thesis describes the design and implementation of a flyover based on the commercially available CORBA compliant ORB called Orbix-MT 2.3. The flyover applies optimization strategies to improve the performance when a client-server pair are built using similar software and hardware platforms. Moreover, the flyover preserves the client and server application transparency and location transparency, without any source code modification of the Orbix ORB. Based on the performance measurement of the flyover on a network of workstations under a synthetic workload, in comparison to commercial-off-the-shelf ORB products, we observe that the flyover shows a substantial response time improvement on transferring short messages. A lesser improvement is observed when large messages are transferred. This thesis also shows that even on systems in which a small subset of communicating components exhibit similarity, the flyover still ameliorates the overall system performance.

Acknowledgments

Firstly, I would like to express my sincere gratitude to my thesis supervisor Professor Shikharesh Majumdar for his invaluable guidance, continuous support, and numerous encouragements during the past two years.

Also I would like to express my deep appreciation and thanks to my parents, sister and brother for their patience and strong support.

Last but not least, I am very thankful to our system administrator, Narendra Mehta and the support of my friends especially Majid, Van, and Khalid.

Table of Contents

Chapter 1 Introduction	1
1.1 Background	1
1.2 Motivation	2
1.3 Goals and Contributions of the Thesis	3
1.4 Thesis Outline	4
Chapter 2 CORBA Overview	5
2.1 An Overview of Distributed Systems	5
2.2 Development Techniques for Distributed Systems	6
2.2.1 Lower Level Programming Interfaces	6
2.2.2 Sockets and Transport Layer Interfaces (TLI)	7
2.2.3 ACE Wrappers interface	10
2.2.4 The Distributed Computing Environment (DCE)	11
2.3 The Common Object Request Broker Architecture (CORBA)	12
2.3.1 ORB Core	14
2.3.2 ORB Interface	14
2.3.3 CORBA Interface Definition Language (IDL)	15
2.3.4 IDL Compiler	15
2.3.5 IDL Stub and Skeleton	15
2.3.6 Object Adapter	16
2.3.7 Interoperable Object Reference (IOR), Object Key and Object Identifier	17
2.3.8 GIOP and IIOP	18
2.3.9 Naming Service and Object Binding	20
2.3.10 Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI)	20
2.3.11 Interceptors	21
2.4 Performance Issues in CORBA	22
2.5 Real-time CORBA	25
Chapter 3 Design of the Flyover	30
3.1 Systems with Limited Heterogeneity	30
3.2 Previous Works on CORBA-based Systems with Limited Heterogeneity	31
3.3 Overview of the Flyover	32
3.3.1 Goals and Key Requirements of the Flyover	33
3.4 Design Challenges and Solutions	38
3.5 Overview of a CORBA Request-Reply Transaction in the Flyover	41
3.5.1 Operations Performed During a CORBA Request-Reply Transaction in the Flyover	42
3.6 Case Study	43
3.6.1 Case Study: CORBA IDL Interfaces	45
3.6.2 Execution Trace of a Invocation of Interface Method in Orbix-MT 2.3	47
3.7 Execution Trace of a Invocation of Interface Method in the Flyover	53
3.7.1 Step 1: Server Initialization	54
3.7.2 Step 2: Establishing a socket connection to the server	58

3.7.3	Step 3: Method Invocation on the Client Stub	64
3.7.4	Step 4: Marshalling of Method Invocation into the Optimized GIOP	67
3.7.5	Step 5: Message Demultiplexing and Demarshalling on the Server Side	76
3.7.6	Step 6: Message reply	79
3.8	Data Types	79
3.8.1	Object Reference	79
3.8.2	String, Array, Union, Struct and Interface Inheritance	81
3.9	Implementation Issues	83
3.9.1	Implementation of IDLP and the Flyover	83
3.9.2	Switching Between CORBA and the Flyover at Run Time	85
3.9.3	Limitations of the Flyover	86
Chapter 4	Performance Measurement and Instrumentation	87
4.1	Experiments	87
4.1.1	Single Client/Server System under No Load Condition	87
4.1.2	Single Client/Server System with Poisson Arrival of Requests	88
4.1.3	Modelling Client-Server Systems with Limited Heterogeneity	89
4.1.4	Measurement of Binding Time	90
4.1.5	Measurement of System Throughput	90
4.2	Parameters	90
4.2.1	Data Types	90
4.2.2	Data Size	91
4.2.3	Arrival Rate of Requests	92
4.2.4	Degree of Heterogeneity (p)	92
4.3	Automation of Performance Measurement	92
4.4	Experimental Environment	93
4.5	Performance Metrics	93
4.5.1	Response Time	93
4.5.2	Mean System Throughput	95
4.5.3	Systems Calls for Elapsed Time Measurement	95
4.6	Accuracy of Measured Data	96
Chapter 5	Experimental Results	98
5.1	Single Client/Server System under No Load Condition	98
5.1.1	CORBA Array	98
5.1.2	CORBA Primitive Types	107
5.1.3	CORBA String	108
5.1.4	CORBA Sequence	109
5.1.5	CORBA Struct	110
5.1.6	CORBA Object Reference	111
5.1.7	CORBA Enumeration	112
5.2	Measurement of Binding Time	113
5.3	Single Client/Server System with Poisson Arrival of Requests	114
5.4	Client/Server Systems with Limited Heterogeneity	117
5.5	Summary of Experimental Results	119
Chapter 6	Conclusions	122

6.1 Summary	122
6.2 Future Research	123
References	126
Appendix A	131
A.1 Marshalling and Demarshalling of a CORBA Struct in the Flyover	131
A.2 Marshalling and Demarshalling of a CORBA Union in the Flyover	133
A.3 Interface Inheritance in the Flyover	137
Appendix B	140
B.1 Additional Performance Measurement Results for CORBA Array	140
B.2 Additional Performance Measurement Results for CORBA Sequence	141

List of Figures

Figure 2-1 Development techniques of distributed applications	6
Figure 2-2 Socket function calls using a connection-oriented protocol	8
Figure 2-3 Socket function calls using a connection-less protocol	8
Figure 2-4 ACE IPC_SAP	10
Figure 2-5 General architecture of DCE	11
Figure 2-6 General architecture of CORBA	13
Figure 2-7 Interceptors in a request-reply CORBA transaction	22
Figure 3-1 General architecture of the flyover	32
Figure 3-2 Request demultiplexing in CORBA	34
Figure 3-3 Optimization techniques applied to the flyover	35
Figure 3-4 General CORBA applications development paradigm	40
Figure 3-5 Workflow of generating client and server executable codes using IDLP	40
Figure 3-6 High-level view of a CORBA request-reply transaction in the flyover	41
Figure 3-7 The BOAImpl approach to define a C++ implementation class	44
Figure 3-8 The TIE approach to define a C++ implementation class	45
Figure 3-9 Sample source codes for case study	46
Figure 3-10 Class Diagrams of IDL interface interfaceA in Orbix-MT 2.3	48
Figure 3-11 Sequence diagram for client method invocation in Orbix-MT 2.3	49
Figure 3-12 Class diagram of dispatch class in Orbix-MT 2.3	51
Figure 3-13 Sequence diagram for operation dispatching in Orbix-MT 2.3	52
Figure 3-14 Server object registration	54
Figure 3-15 Modified class diagram of interfaceA on the server side	55
Figure 3-16 Sequence diagram for server object registration	56
Figure 3-17 Sequence diagram for initialization of server listener	57
Figure 3-18 IDL interface of the Port Locator object	58
Figure 3-19 Modified class diagram of interfaceA on the client side	59
Figure 3-20 Sequence diagram for locating the target object using the bind function	60
Figure 3-21 Adding interface method GET__OBJKEY__() by IDLP	61
Figure 3-22 Modification of server skeleton code in getting target object ID	62
Figure 3-23 Sample client codes for locating target object in the flyover using i) CORBA Naming Service and ii) Stringified Object Reference	63
Figure 3-24 Class Message in the flyover	64
Figure 3-25 Sequence diagram for client invocation on the modified client stub	65
Figure 3-26 Padding bytes in a data stream for the CORBA struct paddingdemo	68
Figure 3-27 Format of message_properties in optimized GIOP	73
Figure 3-28 Sequence diagram for request demultiplexing and demarshalling in the server	77
Figure 3-29 Example of passing object reference in the flyover	80
Figure 3-30 Sequence diagram for marshalling of an object reference in the flyover	80
Figure 3-31 Marshalling and demarshalling functions of string and array in the flyover	82
Figure 3-32 Example of marshalling CORBA array in the flyover	83
Figure 3-33 Structure of IDLP	84
Figure 3-34 Block diagram of the operations performed by IDL parser inside IDLP	84

Figure 3-35 Block diagram of the operations performed by C++ parser inside IDLP	84
Figure 4-1 Pseudo code to measure the response time under no load condition	87
Figure 4-2 Single client-server system with Poisson arrival of requests	88
Figure 4-3 Model of client-server systems with limited heterogeneity	89
Figure 4-4 Pseudo code to measure the system throughput	90
Figure 4-5 Time measurement points in CORBA request-reply transaction	94
Figure 5-1 Response time measurement for CORBA array of long	101
Figure 5-2 Response time measurement for CORBA array of octet	101
Figure 5-3 Response time measurement for CORBA array of char	102
Figure 5-4 Response time measurement for CORBA array of double	102
Figure 5-5 Throughput of transferring CORBA octet array	106
Figure 5-6 Response time measurement for CORBA string	108
Figure 5-7 Response time measurement for CORBA sequence of octet	109
Figure 5-8 Response time measurement for CORBA sequence of long	110
Figure 5-9 Response time measurement for CORBA struct	111
Figure 5-10 Response time measurement for CORBA object reference	112
Figure 5-11 Response time measurement for CORBA enumeration	113
Figure 5-12 The relationship between net time saving and the number of CORBA invocations	114
Figure 5-13 Performance result of transferring an array long[256] on a single client/server system with Poisson arrival of requests	115
Figure 5-14 Performance result of transferring an array long[2048] on a single client/server system with Poisson arrival of requests	116
Figure 5-15 Performance result of transferring an array long[8192] on a single client/server system with Poisson arrival of requests	116
Figure 5-16 Performance result of transferring an array long[256] on a system with limited heterogeneity	117
Figure 5-17 Performance result of transferring an array long[8192] on a system with limited heterogeneity	118
Figure A-1 Marshalling of a CORBA struct in Orbix-MT 2.3	132
Figure A-2 Marshalling of a CORBA struct in the flyover	133
Figure A-3 Marshalling of a CORBA union in Orbix-MT 2.3	135
Figure A-4 Marshalling of a CORBA union in the flyover	136
Figure A-5 IDL demo for interface inheritance in the flyover	137
Figure A-6 Class diagram of single interface inheritance on the server side in Orbix-MT 2.3	138
Figure A-7 Class diagram of single interface inheritance in the flyover	139
Figure B-1 Response time measurement for CORBA array of boolean	140
Figure B-2 Response time measurement for CORBA array of short	140
Figure B-3 Response time measurement for CORBA array of float	141
Figure B-4 Response time measurement for CORBA sequence of char	141
Figure B-5 Response time measurement for CORBA sequence of boolean	142
Figure B-6 Response time measurement for CORBA sequence of short	142
Figure B-7 Response time measurement for CORBA sequence of float	143
Figure B-8 Response time measurement for CORBA sequence of double	143

List of Tables

Table 3-1 IDL data types in CORBA v2.3.1 support in the flyover	37
Table 3-2 Mapping of operation ID of interfaceA by IDLP	67
Table 3-3 CDR alignment of primitive data type	68
Table 3-4 GIOP message types and originator	69
Table 3-5 Format of GIOP message header	69
Table 3-6 Format of request header	70
Table 3-7 Format of response_expected	71
Table 3-8 Format of reply header	72
Table 3-9 Format of optimized GIOP message header and request header	73
Table 3-10 Marshalling and demarshalling functions of class Message for primitive types	75
Table 4-1 Testing IDL Parameters	91
Table 4-2 Interceptors defined in Orbix-MT 2.3 for performance measurement	95
Table 5-1 Response time measurement for CORBA array of long	99
Table 5-2 Response time measurement for CORBA array of octet	100
Table 5-3 Response time measurement for CORBA array of char	100
Table 5-4 Response time measurement for CORBA array of double	100
Table 5-5 Components of response time in a CORBA transaction of long[32]	103
Table 5-6 Components of response time in a CORBA transaction of long[8192]	103
Table 5-7 Response time measurement for CORBA primitive types	107
Table 5-8 Response time measurement for CORBA string	108
Table 5-9 Response time measurement for CORBA struct	110
Table 5-10 Response time measurement for CORBA object reference	111
Table 5-11 Extra overheads in passing object reference through the flyover (TCP)	111
Table 5-12 Performance measurement for CORBA object reference	112
Table 5-13 Time overhead of object binding	113
Table 5-14 Overhead of object binding in the flyover	114

Glossary of Terms and Symbols

AAL	Asynchronous Transfer Mode Adaptation Layer
ACE	Adaptive Communication Environment
API	Application Programming Interface
ATM	Asynchronous Transfer Mode
BOA	Basic Object Adapter
BSD	Berkeley Software Distribution
CDR	Common Data Representation
CORBA	Common Object Request Broker Architecture
DCE	Distributed Computing Environment
DCE-CIOP	Distributed Computing Environment-Common Interoperability Protocol
DCOM	Distributed Components Object Model
DII	Dynamic Invocation Interface
DSI	Dynamic Skeleton Interface
DOC	Distributed Object Computing
ESIOP	Environment-Specific Inter-ORB Protocol
GIOP	General Inter-ORB Protocol
IDL	Interface Definition Language
IDL P	Interface Definition Language Processor
IIOP	Internet Inter-ORB Protocol
IOR	Interoperable Object Reference
IP	Internet Protocol
IPC	Inter-Process Communication
IPC SAP	Inter-Process Communication Service Access Point
IR	Interface Repository
MTU	Maximum Transmission Unit
NFS	Network File Systems
NMS	Network Management Systems
OA	Object Adapter
OCI	Open Communication Interface
OMG	Object Management Group
ORB	Object Request Broker
OSI	Open System Interconnect
p	Degree of Heterogeneity
POA	Portable Object Adapter
RPC	Remote Procedure Call
RTOS	Real-Time Operating Systems
SII	Static Invocation Interface
TCP	Transmission Control Protocol
TLI	Transport Layer Interface
TSS	Thread-Specific Storage
UDP	User Datagram Protocol
UTC	Universal Coordinated Time
XNS	Xerox Network Systems

Chapter 1 Introduction

1.1 Background

The growth of telecommunication services during the past decade, such as on-line banking systems and e-commerce systems, has increased the demand for scalable, high performance and low latency distributed systems. These kinds of services must be able to deal with a large number of requests and provide fast responses back to the clients. A distributed system or a decentralized system is an increasingly popular and cost effective way to improve the server performance, particularly in an era of rapid decline in the price performance ratio of computing systems, and the availability of gigabits optical networks.

When the size and complexity of the distributed application is increasing, the Distributed Object Computing (DOC) framework becomes a very popular client/server paradigm because it combines the well-known merits of object-oriented systems, such as encapsulation, reusability and extensibility, with the high scalability of the distributed systems. Common Object Request Broker Architecture (CORBA) is a middleware standard that is well suited for applications such as e-commerce systems, which exchange richly typed data via request/response or one-way communication.

Distributed systems are inherently heterogeneous. Adding new system components, either hardware or software, often leads to diversity within the distributed systems. Thus, CORBA provides inter-operability that allows the client application to communicate with the servers on different software and hardware platforms.

However, facilitation of inter-operability in CORBA incurs significant performance penalty in the traditional CORBA request-response transaction. Moreover, although

heterogeneity in distributed systems is natural, most distributed systems are characterized by limited heterogeneity, which includes a subset of communicating components that is made of similar hardware and software. Examples of such a system include embedded and real-time systems, e-commerce systems and various types of telecommunication systems. Thus bypassing the CORBA operations when possible can enhance system performance.

1.2 Motivation

Since most of the distributed systems are characterized by limited heterogeneity, the system components that exhibit heterogeneity can communicate through the ORBs, whereas the similar components can use a “flyover” which communicates through proprietary protocols and bypasses a number of standard CORBA operations.

The previous research conducted at the Real-Time and Distributed Systems Lab at Carleton University describes three performance improvement techniques that perform such bypass operations [3]:

- The first technique prevents data conversion between the native data types used by system components and the standard data format specified in CORBA.
- The second technique saves communication time by removing the padding bytes in a CORBA message.
- The third technique compresses the message headers, thus reducing the communication overhead.

Although substantial latency improvements are observed by applying these optimization strategies, the implementation of the first technique involves source code modification. Moreover, the major drawback of the second technique described in [3] is

that the format of data buffer must be known in advance at the server side, which rarely occurs in real systems. To overcome such problems, this thesis presents the design of a flexible architecture called “Flyover” that employs a number of optimization strategies including those described in the previous paragraph. Although the flyover achieves a higher performance by exploiting limited heterogeneity in system it should preserve the client and server application transparencies, as well as location transparency, without source code modification. With the help of the flyover, the application designer can focus on their core applications rather than developing their own proprietary method to attain a better performance.

1.3 Goals and Contributions of the Thesis

This thesis describes the design and implementation of the flyover based on the commercial middleware Orbix-MT 2.3. A description of a tool prototype used for the construction of such flyover in an application system is also included. The performance improvement produced by the flyover in comparison to two other commercial middleware products under various workloads is discussed. Note that the important research contributions of the master thesis described in [3] identify three performance optimization techniques whereas this thesis describes the design, prototype implementation as well as performance evaluation of the flyover.

The main contributions of this thesis include:

- the design and implementation of the flyover model based on the commercial middleware Orbix-MT 2.3;
- the investigation of the relative performance of the flyover, and two commercial ORB products Orbix-MT 2.3 and Orbix 2000;

- the experimental evidence that the flyover exhibits a significant performance improvement for transferring short messages in systems with limited heterogeneity;
- the demonstration that the flyover prototype can be implemented without source code modification of the middleware product.

1.4 Thesis Outline

This thesis comprises six chapters. Chapter 2 introduces the development techniques of distributed systems and an overview of CORBA. It also describes the related research in high performance CORBA and real-time CORBA. Chapter 3 describes the design and implementation details of the flyover. Chapter 4 describes the experiments and performance measurements, including the latency measurement under various system loads. It also describes the experiments to model the system with limited heterogeneity. Chapter 5 evaluates and discusses the performance result of the flyover with two other commercial middleware products. Chapter 6 presents the concluding remarks and further research issues.

Chapter 2 CORBA Overview

2.1 An Overview of Distributed Systems

A distributed system is a collection of physically separated and autonomous computers linked by communication networks. It uses software and hardware to produce an integrated computing facility that gives the users an impression of a single powerful system. The decline in the price performance ratio of computing systems and the availability of high-speed networks promotes the rapid development of distributed systems and distributed applications.

Distributed systems have several advantages that include resource sharing, concurrency, scalability, fault tolerance, location transparency and cost effectiveness [28]. For instance, distributed systems allow sharing of physical devices such as disks and printers or replicated data. The distributed execution of a complex application achieves concurrency and enhances overall performance. Ideally, distributed systems have no centralized components so that there is no restriction on the maximum size to which the system can grow. Moreover, it prevents single point of failure that occurs in centralized systems.

However, a distributed system is inherently heterogeneous. It consists of different hardware, different operating systems as well as applications implemented in different programming languages. Hence, development of robust distributed systems requires a thorough understanding of hardware, communication protocol and operating systems in order to provide interoperability across different platforms. The following sections describe the development techniques for distributed systems and applications.

2.2 Development Techniques for Distributed Systems

Traditionally, developers of distributed systems have to choose between high performance, lower-level interfaces or less efficient, higher-level interfaces. Using lower level interfaces to obtain robust, extensible distributed applications requires good knowledge of communication networks, such as network address and connection management, presentation layer conversion, process and thread synchronization and operating system calls. It is easier to develop distributed applications using higher-level interfaces, but it sacrifices fine-grained control over functionality and maximal performance.

An overview of various programming techniques used in developing distributed applications, starting from lower level interfaces (e.g. putmsg/getmsg and socket system calls) followed by higher level interfaces (e.g. C++ wrapper, CORBA, DCOM and DCE) as shown in Figure 2-1, is covered in following subsection.

Figure 2-1 Development techniques of distributed applications

2.2.1 Lower Level Programming Interfaces

Inter-process Communication (IPC) mechanisms, such as FIFOs (named pipes) and Message Queues (System V) provide low-level system call interfaces to the communication subsystems. Examples of system calls in UNIX are open, close, read, write,

`putmsg` and `getmsg` [55][56]. Despite its high efficiency, using lower level interface is complicated and error-prone. In addition, these kinds of applications are not easily portable to different operating systems because they are tightly tailored to the underlying operating system kernel.

2.2.2 Sockets and Transport Layer Interfaces (TLI)

The Socket interface, first developed in 4.1 BSD (Berkeley Software Distribution), provides application program interfaces (API) to the underlying communication protocols. From 4.3 BSD, sockets and all the codes to support the communication protocols (TCP, UDP etc.) are completely implemented within the UNIX kernel [55]. Hence the socket system call interfaces are the lowest entry points above the kernel to support communication protocols.

Sockets support multiple protocol families such as UNIX domain, Internet domain and Xerox Network Systems (XNS) domain. In Internet domain, the socket supports either connection-oriented protocol (TCP/IP) or connectionless protocol (UDP/IP). Figure 2-2 and Figure 2-3 show the series of socket function calls in sending and receiving messages over the connection-oriented and connectionless transports respectively. Note that there is no system call required for connection establishment over connectionless protocol.

Figure 2-2 Socket function calls using a connection-oriented protocol

Figure 2-3 Socket function calls using a connection-less protocol

Transport Layer Interface (TLI) was introduced with System V Release 3.0. TLI provides an interface to the transport layer of the OSI model [55]. One major difference between sockets and TLI is that sockets are implemented as system calls while TLI is implemented as a set of C library functions that provide a user interface to the implementation of various networking protocols [55].

Socket and TLI provides several advantages including:

- **Support of fine-grain control over functionality; e.g. memory management, demultiplexing and connection management.**
- **Omission of unnecessary overheads such as presentation layer conversions present in CORBA (discussed in Section 2.3), particularly useful for streaming and data-intensive applications such as medical imaging, interactive teleconferencing, and video-on-demand [37][43].**
- **Providing portability of application codes to different operating systems that support socket and TLI interfaces.**

However, socket and TLI network programming interfaces are non-type safe and low-level. These complicate programming and permit subtle run-time errors. For instance, communication endpoints in the socket interface are identified by integer-typed descriptors. The socket descriptors shield applications from the internal representation of OS data structures. In UNIX and Windows NT, socket descriptors share the same name space as other handles, such as files, named pipes and terminal devices. Therefore, misusing the file descriptor as a socket descriptor in the socket system calls cannot be detected at compile-time.

Moreover, the standard socket interface is defined as a set of C functions shown in Figure 2-2 and Figure 2-3. They perform tasks such as locating address information for network services, establishing and terminating connections, and sending and receiving data [55]. However, there is a high potential for programming errors such as neglecting to use the `htons` and `ntohs` library function to convert port numbers from host byte-order to network byte-order and vice versa.

2.2.3 ACE Wrappers interface

The ACE (Adaptive Communication Environment) toolkit provides the IPC SAP (Inter-Process Communication Service Access Point) object-oriented network programming wrappers to encapsulate lower-level network programming interfaces [40] [43]. It consists of a family of classes shown in Figure 2-4 that encapsulate descriptor-based network programming interfaces such as sockets (SOCK_SAP), TLI (TLI_SAP), UNIX SVR5 STREAM pipes (SPIPE_SAP), and UNIX named pipes (FIFO_SAP). Each class category is organized as an inheritance hierarchy. Therefore, the base class IPC_SAP can represent the similarities between the category classes while the subclasses represent the differences.

Figure 2-4 ACE IPC_SAP

The ACE wrappers are designed to improve programming simplicity, reusability, correctness and portability. The IPC_SAP wrappers automate and simplify many aspects such as connection establishment and addressing, and provide object-oriented interfaces that enhance reusability. Since IPC SAP makes all the classes strongly typed, invalid function call will be rejected at compile time rather than at run time. Therefore it ameliorates programming correctness. They also improve portability by shielding applications from platform-specific network programming interfaces. In addition, they do

not introduce any significant overhead compared with using sockets directly. The performance of using SOCK_SAP C++ wrapper over ATM and Ethernet networks is very close to that of using C sockets [43].

However, ACE shares some of the same drawbacks suffered by low-level communication mechanism such as sockets. In particular, a large proportion of the code required to program with ACE is not directly related to the application. The programmer using ACE components still has to wrestle with the communications details of his/her application rather than focus on the application problem space.

2.2.4 The Distributed Computing Environment (DCE)

The OSF (Open Software Foundation) DCE is a layer between the operating systems and the distributed application shown in Figure 2-5 [6]. DCE provides the middleware services that allow a distributed application to interact with a collection of possibly heterogeneous computers, operating systems, and networks as if they were a single system. Figure 2-5 shows the various elements that comprise the DCE architecture.

Figure 2-5 General architecture of DCE

The importance features of DCE are summarized [6]:

- DCE Threads provide a user-level thread package that supports the creation, management, and synchronization of multiple threads of control within a single

process. If the host operating system (such as UNIX and WindowNT) already supports user-level threads, DCE Threads is not necessary.

- **DCE Remote Procedure Call (RPC)** provides a mechanism that clients can invoke procedures in servers. It encapsulates the details of where servers are located the network, types of hardware and operating system platforms and differences in data representation. A client can use the directory service to bind to a particular server at run time.
- **The DCE Directory Service** provides a central repository for information about resources, such as users, machines and RPC-based services, in the distributed system. These services consist of Cell Directory Service (CDS), Global Directory Service (GDS, that uses X.500), Domain Name Service (DNS), and Global Directory Agent (GDA).
- **The DCE Distributed Time Service (DTS)** provides synchronized time with Universal Coordinated Time (UTC) standard on the computers participating in DCE.
- **The DCE Security Service** provides secure communications and access control to resources. The security service includes authentication, authorization and auditing.
- **The DCE Distributed File Service (DFS)** allows users to access and share files on a file server anywhere on the network, without having to know the physical locations where the files are actually stored.

2.3 The Common Object Request Broker Architecture (CORBA)

As the size and complexity of software systems is increasing, object-oriented (OO) systems are increasing in their popularity. The merits of OO systems, such as design flexibility, encapsulation and reusability, are well known. Distributed Object Computing

(DOC) is a distributed computing paradigm that combines the advantages of OO systems with those of distributed processing systems. CORBA and DCOM (Distributed Components Object Model) are two of the most popular DOC-based middleware.

The DCOM is a protocol, based on the Open Software Foundation's DCE-RPC specification that enables software components to communicate directly over a network in a reliable and efficient manner [27]. It is developed by Microsoft and is strongly supported in Microsoft Windows platforms.

CORBA is a standard proposed by the Object Management Group (OMG) for building distributed object-oriented client-server systems in a heterogeneous environment [30]. Middleware like CORBA provides inter-operability that allows a component of an application to communicate with another component, which is implemented in a different programming language and is running on top of different operating system. Because of its flexibility and inter-operability, CORBA is widely used in many areas such as telecommunication, electronic commerce, process control systems and network management systems (NMS) [19].

A brief overview of general CORBA is presented in this section. Figure 2-6 depicts the general architecture of CORBA.

Figure 2-6 General architecture of CORBA

2.3.1 ORB Core

The ORB core is the communication infrastructure which is responsible for accepting requests from the client, finding the object implementation, transparently activating the object if necessary, handling the connection and communication between client and object, delivering the request to the object and returning a response, if any, to the client. The main feature of the ORB is providing location, communication and implementation transparency. It allows clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and hardware.

The client and server could exist in the same process, different processes or different hosts across the network. If remote, the invocation passes from the ORB of the client to the ORB of the object implementation. The ORB Core is implemented as a run-time library linked into client and server applications.

2.3.2 ORB Interface

An ORB is a logical entity that may be implemented in various ways, such as one or more processes or a set of libraries. To decouple applications from implementation details, the CORBA specification [30] defines an abstract interface for an ORB. These services are the same for all ORBs, and can be invoked by either the clients or the object implementations. Some of these services include those used by the ORB itself, such as converting object references to strings and vice versa, as well as initializing and shutting down the ORB.

2.3.3 CORBA Interface Definition Language (IDL)

The OMG Interface Definition Language (OMG IDL) is the language used to specify the operations and attributes that the interface supports, and thus defines the requests that can be made on the object. The main characteristics of the IDL are supporting object-oriented interfaces, interface inheritance and exception handling.

IDL is an object contract language [36] instead of a programming language. It is independent of the language used in object implementation and an IDL compiler is required to translate an IDL specification to the target language binding, such as C++, Java, Smalltalk and so on. Hence the use of IDL promotes programming language transparency and modularity.

2.3.4 IDL Compiler

An IDL compiler automatically transforms OMG IDL definitions into stub codes and skeleton codes that are in target application programming languages such as C++ or Java.

2.3.5 IDL Stub and Skeleton

According to the IDL definitions, the IDL compiler generates the stubs and skeletons. These stub and skeleton serve as the glue between the client and server application respectively, and the ORB.

The client stub implements the Proxy pattern [18] and provides a strongly-typed, static invocation interface (SII). It transforms typed data objects from higher-level representations to lower-level representations such as marshalling the typed parameters into a Common Data Representation (CDR), and vice versa (demarshalling the return parameters).

The skeleton is the server-side equivalent of the stub. It implements the Adapter pattern [18] and coordinates invocation with an Object Adapter (OA). It demarshals the parameters, and dispatches the method invocation to the object implementation. In general, these presentation layer conversions are a major source of bottleneck in high-performance communication subsystems [47].

Dispatching the request through the stub and the skeleton is also called Static Invocation because the IDL interfaces that the client can access are determined at compile time.

2.3.6 Object Adapter

The CORBA Object Adapter mediates between object implementations (servants) and the ORB. In general, the object adapter provides the following functionalities [47]:

- Request demultiplexing: Object adapters demultiplex each CORBA request to the appropriate servant.
- Operation dispatching: Once the object adapter locates the target servant, it dispatches the requested operation. At this point, the skeleton is used to transform the parameters in the request into the arguments passed to the intended servant operation.
- Servant activation and deactivation
- Generating and interpreting object references

2.3.6.1 CORBA Basic Object Adapter (BOA)

Over the life of the CORBA specification, only two object adapters have ever been officially adopted by the OMG. From the first version of CORBA through CORBA 2.1, the standard object adapter defined by the OMG was called the Basic Object Adapter (BOA). The BOA provides basic functions of request demultiplexing, operation dispatch-

ing and on-demand object activation. When a client issues a request, the BOA determines if the object is currently running and if so, it delivers the request to the object. If the object is not running, the BOA activates the object and then delivers the request.

Unfortunately, the BOA from the original CORBA specification is incomplete. The following are some of the key problems [47]:

- Not defining a portable way to associate skeletons with servants
- Failing to describe how servants are registered
- Completely ignoring the issues of multithreading in server process

2.3.6.2 CORBA Portable Object Adapter (POA)

In CORBA 2.2 specification, OMG defined a new standard object adapter called Portable Object Adapter (POA) [47][48][61]. The features of this newly defined adapter include persistent and transient objects, explicit and on-demand activation, and different POA policies such as multithreading policy, servant retention policy, request processing policy and life span policy.

2.3.7 Interoperable Object Reference (IOR), Object Key and Object Identifier

To invoke operations on the object, the client has to obtain the object reference, either by calling CORBA Naming Service, or by converting a stringified object reference to an object reference. Object Reference or Interoperable Object Reference (IOR) contains three types of information:

- **Type ID:** The type ID, also known as a repository ID, identifies the most derived type of the object associated with the IOR at the time the IOR was created. This repository ID is also used as an index into the Interface Repository, which allows the IDL definition for an interface to be retrieved at run time.

- **Object Key:** This information is an opaque type of binary data. It contains proprietary information of the ORB that created the IOR. The object key itself consists of two parts: object identifier and object adapter name, which uniquely identifies the particular servants within the context of object adapter and particular object adapters in the server, respectively. In general, General Inter-ORB Protocol (GIOP) does not define the contents of the object key; hence, the ORB implementations can put whatever information they need in the object key so that they can locate the target object.
- **Protocol and Address Details:** This field specifies the protocol and the addressing information appropriate for that protocol. For example, if the ORBs communicate by Internet Inter-ORB Protocol (IIOP), the addressing information consists of IP (Internet Protocol) address and TCP (Transmission Control Protocol) port number.

Object references are opaque. The information inside the IOR is not intended to be visible to application level programmers.

2.3.8 GIOP and IIOP

The General Inter-ORB Protocol (GIOP) supports protocol-level ORB interoperability over any connection-oriented transport. The GIOP specification consists of the following elements [30]:

- **Common Data Representation (CDR):** a transfer syntax that maps OMG IDL data types into a bi-canonical format for transfer between agents over networks. CDR has the following features [30]:
 - **Variable byte ordering:** The message originator determines the message byte order (big endian or little endian) and sets the flag in each GIOP message to

indicate the appropriate byte order. If the receiver has different byte order, it is responsible for swapping to match its native ordering.

- **Aligned primitive types:** Primitive OMG IDL data types are aligned on their natural boundaries within GIOP messages.
- **Complete OMG IDL Mapping:** CDR describes the representations for all OMG IDL data types, including the complex types, and various pseudo-objects.
- **GIOP Message Formats:** Defines eight types of message formats, including Request, Reply, CancelRequest, LocateRequest, LocateReply, CloseConnection, MessageError and Fragment.
- **GIOP Transport Assumptions:** Describes types of network transport layer that may be used to transfer GIOP messages. For example, the transport must be reliable, connection-oriented and should not impose arbitrary message size limitations, fragmentation, or alignments. Moreover it also describes how the connections can be managed.

Because of the popularity of the TCP/IP transport protocol, OMG introduced the Internet Inter-ORB Protocol (IIOP) in CORBA specification 2.0. Essentially, IIOP maps the GIOP into a TCP/IP transport. The IIOP specification describes how the agents open TCP/IP connections and use them to transfer GIOP messages. In addition, custom Environment-Specific Inter-ORB Protocol (ESIOP) specification is also defined [30]. For instance, the first implementation of ESIOP is DCE-CIOP (Common Interoperability Protocol) that provides interoperability between CORBA and DCE environments.

2.3.9 Naming Service and Object Binding

Before the client can invoke a method on an object in the server, it must first obtain a reference to that object. The process of associating the client object reference with the server object is called object binding, or simply binding. The client can obtain the IOR by calling the CORBA Naming Service. The Naming Service holds a database of bindings between abstract object names and object references. The server that activates an object can register it with the Naming Service, giving it a unique name that can be used by other clients or servers to locate that object. The object name registered in the Naming Service is an abstract name and is independent of any properties of the object, such as an interface name or the name of the server or host in which the object resides.

Another way of binding is converting the stringified object reference to an object reference. When the server initializes the object, it first calls the ORB function [30] to get the stringified object reference from that object and then writes it into a file. The client reads that file, usually shared by network file systems (NFS) [58], to get the stringified object reference and then converts it back to object reference.

The last way is using proprietary function calls provided by the ORB vendors. For example, Orbix provides the `_bind()` function [21] to make objects implicitly available to clients. However, this function is not defined in the CORBA standard [30].

2.3.10 Dynamic Invocation Interface (DII) and Dynamic Skeleton Interface (DSI)

The Dynamic Invocation Interface (DII) allows the client to invoke requests on an object without having compile-time knowledge of the object's interface. Instead of calling through stub code, the client should provide the request information, for example, the name of the operation or attribute and types and values of each argument. This

information can be obtained from the Interface Repository (IR). Similarly, the Dynamic Skeleton Interface (DSI) is the server-side equivalent of DII. It permits a server to receive an operation or attribute invocation on any object, even one with an IDL interface unknown at compile time.

The main reason for the introduction of DSI in CORBA 2.0 is to implement gateways [21][60]. A bi-directional gateway must be able to act as both a client and a server. It must translate requests from the foreign object system into requests on CORBA objects, and turn requests from CORBA applications into foreign object invocations.

However, the flexibility provided by DII and DSI is costly. A remote method invocation made through the DII requires a series of function calls, including creating a Request object [30], adding parameters and invoking the dynamic request. However, the same invocation through stub/skeleton can be achieved through a single call. It has been shown that passing the request through the stub/skeleton codes is far more efficient than through the DII [15].

2.3.11 Interceptors

Since CORBA 2.2 the OMG has opened up the ORB a bit by defining Interceptors that allow the interception of messages and execution of custom application codes at certain defined points within an ORB [32]. Figure 2-7 depicts the eight intercepting points in a CORBA request-reply transaction. There are two types of Interceptor defined in CORBA 2.2: request level and message level. Request-level interceptors (points 1, 4, 5 and 8 in Figure 2-7) are given access to a request object and allow modification on the structured request before and after it has been invoked. One example of request-level interceptor is the Access Control Interceptor, which uses information about the requesting

principal, for example username and access ID, and the operation name in order to make an access control decision [30].

Message-level interceptors (points 2, 3, 6 and 7 in Figure 2-7) allow access to actual message buffers of the request before and after sending across the networks. If functions such as encryption and compression are performed on messages, message-level interceptor interface is required.

Figure 2-7 Interceptors in a request-reply CORBA transaction

The Portable Interceptor introduced in CORBA 2.4 defines two new interception points. The network interceptors and object level interceptors allow the usage of different transport protocols, and the interception of lifecycle events respectively [47].

2.4 Performance Issues in CORBA

Early research on high-performance CORBA is to overcome high overheads of excessive data copying and conversion overhead in the presentation layer [13], inefficient demultiplexing [16], and memory management. On lower speed networks like Ethernet and Token Ring, this overhead is often masked. However, on high speed networks such as ATM and Fast Ethernet, empirical studies [13][14][15][17][43] show that for bulk data transfer, the performance of conventional CORBA is about 30% to 75% lower than the performance achievable using lower-level transport layer interfaces such as sockets or TLI. To overcome such high overheads, different optimization techniques have been applied on current implementation of CORBA. A brief review of a representative set of

research on performance optimization techniques for CORBA-based systems is presented.

Perfect hashing [15] and active demultiplexing strategies [16][38] can be used rather than the linear search which is known to be expensive and non-scalable. A fast and flexible message demultiplexing strategy based on dynamic code generation is discussed in [9]. Applying techniques such as replacing general purpose functions with specialized efficient methods, as well as precomputation and caching of values, can also improve performance for SunSoft IIOP over ATM networks [17][41][43]. The Real-time CORBA TAO uses thread-specific storage (TSS) [42] to minimize unnecessary data allocation and copying [38]. To overcome the high overhead in the presentation layer, an optimized IDL compiler, like USC [33] and “Flick” [8], can generate more efficient stub and skeleton codes. Moreover, a hybrid scheme is designed to combine compiled (faster but larger in size) and interpretive (slower but smaller) (de)marshalling to achieve better performance [20].

Using load balancing techniques can improve system performance by distributing the loads on replicated objects. The integration of load balancing techniques with CORBA naming service is described in [5]. Load balancing is also available in various commercial CORBA like IONA's Orbix. TAO's Adaptive Load Balancing Service [35] is designed to maximize resource utilization by gathering loading information in the replicated objects and re-distributing the requests to the underloaded objects. This load balancing service also supports application-defined load metrics and load balancing policies. Another example of using load balancing in middleware is Realize [54]. The Realize middleware replicates the server objects and allocates the objects to processors.

By measuring the usage of the resources, it schedules the requests to the replicated objects in order to achieve load balancing. Moreover, Realize maintains consistency of the replicas of the objects by using multicast messages.

Collocation optimization is a technique for transparently optimizing communication overhead when clients and server objects are configured into the same address space. The standard collocation technique uses a so-called "collocation-safe" stub to handle operation invocations on a collocated object. Because the collocated stubs must access the client ORB and the server ORB, they incur a significant amount of overhead. The direct collocation technique forwards all requests directly to the servant class without passing through the ORB, and borrows the client's thread-of-control to execute the operation. The empirical result reveals that the direct collocation technique obtains a significant performance gain [50].

Since embedded systems typically run the same ORB implementation on similar hardware, removing the redundant fields in the GIOP header and the GIOP Request header [30] can minimize the ORB protocol message footprint and enhance performance. For example, TAO's GIOPlite, which is given as an option (`-ORBgioplite`) to the client and server `CORBA::ORB_init` method, removes the GIOP magic number, version number, byte order flag, request service context and principal. Since the GIOPlite saves 15 bytes of the original GIOP, it shows a slight improvement over the standard GIOP [42]. Three performance optimization techniques that exploit systems with limited heterogeneity are discussed in [3].

The communication network protocols used for interchange of information between a CORBA server and its clients can have a significant effect on performance. The standard

IIOP protocol is based on TCP that often turns out to be too heavy for embedded applications that require a small message footprint and low latency [34]. The omniORB2 middleware is designed to be adaptable to different network transports [4]. The performance of the omniORB running over TCP/IP, shared memory, Scalable Coherent Interface (SCI) and ATM Adaptation Layer 5 (AAL5) is described in [25]. The Smart Proxies hide the communication mechanism and adapt to different communication transports and services, like TCP/IP, UDP, UDP with JPEG encoding and shared memory [22]. Because of the importance of adaptation to different communication protocols for different applications, OMG has standardized the Open Communication Interface (OCI). Through OCI, a protocol module can be plugged into the ORB so that the distributed application can be put on top of any network without changing the application code. TAO's Pluggable Protocol framework is a highly flexible and reusable design that yields high and predictable performance [34].

At the system level, the architecture of client-broker-server interaction also has a strong impact on system performance. Two ORB architectures: Handle-driven architecture (H-ORB) and Forwarding architecture (F-ORB) are described in [1]. The relative performance of the two architectures depends on system loads. The combination of the good attributes of these two architectures into an Adaptive ORB (A-ORB) that dynamically switches between the two architectures depending on system load is discussed in [53]. The paper shows that the adaptive architecture performs significantly better than an H-ORB or an F-ORB.

2.5 Real-time CORBA

Hard real-time systems, such as regulators for nuclear reactors and certain command

and control systems used by military [39], require a guarantee that the time constraints or deadline associated with transactions will be met. Failure to meet the deadlines will lead to catastrophic results. However, soft real-time systems relax the time constraints and allow some of the deadlines to be missed. Examples of soft real-time systems are telecommunication switches, routers, and various types of process control systems.

Because of the importance of end-to-end predictability in real-time applications, real-time CORBA (RT CORBA) specification is added into the COBRA 2.4 specification [31]. The main characteristics and requirements of Real-Time CORBA 1.0 specification and related research are summarized in [45]. A brief description of the summary is presented.

a. High Performance and Low Latency: Real-time CORBA requires high performance and low latency. Therefore request demultiplexing and dispatching, memory management and presentation layer operation must be highly optimized. The optimization techniques described in Section 2.4 address some of these issues.

b. Priority Scheduling and Prevention of Priority Inversion: A higher priority task should be scheduled before a lower-priority task. An empirical evaluation of multiple hard real time scheduling techniques that include static, dynamic and hybrid static/dynamic scheduling, such as Rate Monotonic Scheduling (RMS), Earliest Deadline First (EDF), Minimum Laxity First (MLF) and Maximum Urgency First (MUF) is presented in [11].

When a lower priority request blocks the execution of a higher priority request for an indefinite period, priority inversion is said to occur. To prevent priority inversion, the ORB end-systems (e.g. operating systems and real-time I/O subsystems) should implement priority inheritance or priority ceiling protocols.

c. Real-Time CORBA Priority, Priority Mapping and Priority Transform: In a heterogeneous environment, different operating systems may use their own priority schemes. To overcome the heterogeneity of OS's priority schemes, the Real-Time CORBA standard defines two priority types: CORBA and native. Each request, either one-way or two-way, can be assigned a CORBA priority. The CORBA priority can be mapped into a local priority, which is native to a particular operating system.

Priority transforms are also defined by RT-CORBA for setting the priority of invocations based on external factors such as system load, operation criticality, and the state of scheduling service. The priority transforms can be implemented as interceptors and can change the invocation priority as the requests are sent or received.

d. Thread Scheduling and Thread Pool Model: Many real-time systems use multi-threading to support task preemption since the overhead of thread context switching is much less than that for heavy weight processes. Therefore RT-CORBA defines a standard thread pool model to manage thread execution on the server-side. Features provided by the thread pool model include pre-allocation of thread pools, partitioning of threads in a thread pool into different subsets, bounding of thread usage and buffering of additional requests.

e. Global Scheduling Service: RT-CORBA allows application to define high-level, intuitive QoS requirements, such as worst-case latency, maximum bit error rate and minimum bandwidth. The global scheduling service maps those QoS requirements into lower level OS and network parameters, for example, OS thread priorities and virtual circuit bandwidth/latency parameters in ATM networks. Moreover, the scheduling service is responsible for allocating system resources to satisfy those QoS requirement for

application. One example of global scheduling service is the real-time CORBA TAO. TAO supports QoS specification for each operation by associating it with a `RT_Info` IDL struct [46]. The high-level QoS requirements that can be specified in `RT_Info` struct include worst-case execution time, period, criticality and importance. TAO scheduling service maps those requirements into different thread priorities.

f. Inter-ORB Communication Management: There are two important features in RT-CORBA for managing inter-ORB communication to provide guarantee allocation of resources.

- **Selection and Configuration of Inter-ORB Protocol (IOP) Properties:** RT-CORBA specifies a standard interface that allows the real-time application designer to specify properties for the ORB and transport protocols, such as, ATM virtual circuits or Internet RSVP traffic specification. RT-CORBA defines two types of protocol policies. The server-side protocol policy is used to select the protocol to be configured into an object reference while the client-side protocol allows the designer to select the protocol to be used when clients connect to servers.
- **Explicit Binding:** Explicit binding mechanism allows clients to pre-establish private connections to servers. This connection is not shared or demultiplexed with other client-server pairs. Using private connections help eliminating priority inversion.

g. Operating System Support for Scheduling: Real-time ORBs cannot deliver end-to-end guarantees without the support of underlying operating systems. For instance, the underlying operating systems and I/O subsystem must support explicit thread priority levels and avoid priority inversions by implementing either priority inheritance or priority ceiling protocol. General-purpose operating systems such as Windows NT, Solaris, and

Linux are not suited to meet the demands of applications with stringent QoS requirements on real-time CORBA systems [24]. The real-time operating system (RTOS) VxWorks exhibits a lack of scalability, whereas another RTOS LynxOS shows predictable and effective performance [24]. The design and performance of real-time I/O (RIO) subsystem in TAO is also addressed [23].

There is related research in real-time distributed systems. For instance, the Quality Assurance Language (QuAL) has been devised for building systems in which requests have associated QoS requirements [10]. The QuAL specifications are compiled into run time components that monitor the actual QoS delivered. Upon QoS violations, application provided exception handlers are signaled to act upon faulty events. Another related work is Quality Objects (QuO) project developed in BBN Technologies [26][59]. The QuO project provides a general-purpose framework for creating distributed applications that adapt to different QoS. Also the QuO framework can incorporate the QoS features into existing distributed systems. The details of integration of QuO into TAO are presented in [12].

Chapter 3 Design of the Flyover

This chapter consists of nine sections. Section 3.1 describes the characteristics of systems with limited heterogeneity. The previous research on the CORBA-based system with limited heterogeneity is presented in Section 3.2. Section 3.3 presents the overview of the flyover, including the goals and key requirements in the design of the flyover. Section 3.4 describes the design challenges of the flyover and the solutions. Section 3.5 presents the overview of a typical CORBA request-reply transaction in the flyover. Section 3.6 presents the case study that demonstrates the operations of the flyover. A test CORBA interface defined in Section 3.6 is used for the explanation of the execution trace in the flyover described in Section 3.7. Section 3.8 describes data types supported by the flyover. Finally, Section 3.9 discusses the implementation issues.

3.1 Systems with Limited Heterogeneity

Distributed systems are inherently heterogeneous. They may consist of different types of machines, operating systems, network interfaces and applications implemented in different programming languages. Although heterogeneity in distributed systems is natural, practically most distributed systems, in particular telecommunication and embedded systems, are often characterized by limited heterogeneity. Systems with limited heterogeneity contain subsets of “similar” communicating nodes in the systems that comprise the same types of machines and operating systems running applications written in the same programming languages. Those nodes that exhibit heterogeneity can communicate through the ORBs, whereas the similar components can communicate through proprietary protocols and bypass a number of standard CORBA operations.

3.2 Previous Works on CORBA-based Systems with Limited Heterogeneity

Some of the Commodity-Off-The-Shelf (COTS) middlewares provide proprietary protocols to improve performance. This is feasible only when both client and server are using the same vendor's ORB. For instance, Orbix gives "Orbix Protocol" as an option in addition to the standard GIOP. The real-time CORBA TAO also provides a GIOPlite [42] protocol, a lightweight version of GIOP, by removing some redundant fields in the standard GIOP header (see Section 2.4). A more complete investigation of performance improvement techniques in systems with limited heterogeneity is described in [3]. It presents three optimization techniques to bypass some standard CORBA operations.

The first performance improvement technique prevents data conversion between the native format used by the client and server and the CDR specified in CORBA GIOP. The range of savings obtained from this technique is from a minimum of 2% for octet data type to a maximum of 16.6% for double data type [3]. This technique is implemented by changing the (de)marshalling routines in the ORB source code.

The second technique removes the padding bytes so that the primitive data types do not need to be aligned with their natural boundaries within GIOP messages. This strategy is implemented with the use of message-level interceptors. On the client side, the padding bytes are removed from the output buffer; they are re-inserted on the server side. Because of the fixed overhead of calling interceptor and the variable overhead of removing and inserting the padding byte, a decrease in performance is observed when the percentage of padding is below 22 %. If the percentage of padding is higher, for example 50%, the percentage of saving is observed to be about 9% for the message considered in [3]. Nevertheless, the major drawback of the second technique is that the format of data

buffer must be known in advance on the server side.

The last strategy is to compress the message header of the GIOP message. This optimization technique is similar to the GIOPlite in TAO. This strategy is implemented by changing the source code on the client side and using message-level interceptor on the server side. The performance improvement is significant (7.9%) for smaller messages.

3.3 Overview of the Flyover

To explore the performance gain on distributed systems with limited heterogeneity, optimization techniques must be applied if the client and the server object are located on same type of machines and operating systems. For instance, in Figure 3-1, client A is running on a WinNT and Pentium-based PC whereas the server process in which the target object resides is running on a Solaris and UltraSPARC workstation. The inter-process communication in this heterogeneous environment can be mediated through the ORB. Nevertheless, the components like client B and object C that exhibit similarity can communicate directly by a low-level communication interface and proprietary protocols to bypass the ORB. Client B may take the flyover shortcut to send the request to server C instead of passing the request through the ORB.

Figure 3-1 General architecture of the flyover

The flyover is an alternative communication path that allows the client and server,

which exhibit similarity on a distributed system with limited heterogeneity, to communicate directly by low-level interfaces in order to achieve performance gain. The flyover itself is neither a self-contained CORBA nor lightweight-CORBA, but incorporates a low-level communication shortcut between the client and server into the existing CORBA applications.

3.3.1 Goals and Key Requirements of the Flyover

Using a lower level interface to build a robust distributed application is complicated and error-prone. For instance, it requires a good knowledge of communication network issues, such as network address and connection management, process and thread synchronization and operating system calls. To overcome these problems, the flyover must provide implementation transparency so that all details of low-level communication should be hidden from the application programmers: i.e. the programmers do not need to create any communication endpoints explicitly or call any application programming interface (API) functions in the application code to facilitate low-level network communication. Moreover, it should preserve the CORBA IDL syntax so that there is no change in original CORBA IDL specification files, object implementation source files, client and server application source codes. The goals and key requirements of incorporating the flyover optimization into CORBA applications are described next.

Applying optimization strategies to enhance performance: The main goal of the flyover is to achieve the performance gains on systems with limited heterogeneity: for example minimize the response time and maximize throughput. There are five optimization techniques used in the flyover which are summarized as follows (the details are illustrated in Section 3.7):

- **Preventing CDR Conversion:** If the machines in which the client and server are located use the same kind of hardware architecture, the unnecessary overhead of data conversion can be avoided.
- **Removing Padding Bytes:** The primitive data types do not need to be aligned with their natural boundaries within GIOP messages. As a result this strategy can minimize the request message size and save communication time (see Section 3.7.4.2). The number of padding bytes removed depends on the mix of data types in the message.
- **Direct Socket Communication:** The network communication between the client and server is implemented by a BSD socket [55].
- **Efficient Request Demultiplexing:** In CORBA, an object method is uniquely identified by an Object Key and Operation Name. Object adapter is responsible for demultiplexing incoming requests to the object and the corresponding server skeleton dispatches the request to the appropriate operation of that object. Figure 3-2 shows the procedures in request demultiplexing in conventional ORB. Step 1 locates the object adapter by using the object adapter name in the object key (see Section 2.3.7). Step 2 uses the object identifier in the object key to locate the object implementation and corresponding server skeleton. Step 3 locates the appropriate operation according to the request's operation name.

Figure 3-2 Request demultiplexing in CORBA

In the flyover, the Object ID replaces the object key to uniquely identify the object in the server (details given in Section 3.7.5). Moreover, the Operation ID replaces the operation name to uniquely identify the static function of the target object (details given in Section 3.7.3.1).

- **GIOP Optimization:** The original GIOP is too heavily weight for high-speed communication. To provide a smaller message footprint, the protocol used in the flyover is a highly optimized GIOP. First, all redundant fields such as the magic field, version numbers and reserved fields are removed which lead to a lower header size. Second, because the flyover use only object ID and operation ID to uniquely identify the object method, the lengthy object key and operation name in the request message header are replaced by integer typed object ID and function ID respectively. Moreover, the fields message type, response expected and byte ordering flag are combined into a single field to further minimize the message size. The details of GIOP optimization are given in Section 3.7.4.3.

Figure 3-3 Optimization techniques applied to the flyover

Figure 3-3 demonstrates on which parts of CORBA the optimization techniques can be applied. For instance, the client stub and server skeleton codes are responsible for

preventing CDR conversions, removing padding bytes and encoding/decoding the request in/from the optimized GIOP. Also, the stub and skeleton are responsible for directly communicating by low-level socket interface instead of passing the request through the ORB. Moreover, based on the object ID and operation ID, the server skeleton should also demultiplex the request to the target object's method efficiently.

Flexibility: The goal of the flyover is to provide an alternative efficient communication path between the client and the server over the existing CORBA standard. If the client and the server exhibit similarity, the client can pass the request through the flyover path. Most importantly, the client should be capable of selecting either path at run time. This is important because not all the data types and features in CORBA are currently supported in the first prototype of flyover, for example any data type (complete list of supported data types is shown in Table 3-1). If the client invokes a method with a data type not supported by the flyover, it must pass the request through the ORB by default.

Preserve CORBA features – location and communication transparency: Although the client and server are communicating directly by the socket interface, the application programmer has no knowledge of where the object is. In other words, the flyover must preserve the property of location transparency in CORBA. Location transparency greatly simplifies the client code and prevents subtle network programming errors in dealing with network addresses and related socket system calls.

To facilitate the location transparency, the flyover borrows the object binding function from CORBA standard. First, the client can get the object reference as usual through the naming service, stringified object reference or by using a proprietary function call (e.g. Orbix `_bind()` function) provided by the ORB vendors (see Section 2.3.9).

Then the client can extract the hostname and server name by using the ORB function call `_host()` and `_implementation()` [30].

In addition to location transparency, all the communication details such as connection establishment and sending and receiving data using socket interfaces must be hidden from the application programmers to prevent programming error.

CORBA IDL support: The flyover should support standard IDL features. For instance, the IDL data types currently supported in flyover as shown in Table 3-1 include primitive types, constructed data types and object reference. It also supports other IDL features, for example single interface inheritance and system exception. The client and server applications that are based on the set of IDL interfaces in Table 3-1 can be run smoothly on the flyover without any modification. In the first prototype of flyover, data types such as `wchar`, `wstring`, `typecode`, `any`, `principal`, `context` and user-defined exception are not supported.

Table 3-1 IDL data types in CORBA v2.3.1 support in the flyover

	Supported	Not Supported
Primitive data type:	char, octet, short, unsigned short, long, unsigned long, long long, unsigned long long, float, double, long double, boolean, enum	wchar
Constructed data type:	struct, union, array, bounded and unbounded sequence, bounded and unbounded string	wstring
Pseudo-Object type:	System Exception, Object Reference	TypeCode, Any, Principal, Context, User-defined Exception

Client application transparency: The incorporation of an additional communication path in CORBA applications must require little or no modification to the way in which CORBA applications are normally developed. As mentioned in Section 2.3.9, the client

can get the object reference of the target servant through the naming service, stringified object reference or by using a proprietary function call (e.g. Orbix `_bind()` function) provided by the ORB vendors. Since the flyover prototype is implemented on Orbix-MT 2.3 [21] in this research, if the client application uses the Orbix `_bind()` function to get the object reference, there is no modification required on the client application. However, if the client uses the naming service or stringified object reference to obtain the object reference, little modification on the client application is required because the source codes of the CORBA naming service or ORB interface function `string_to_object` are not available. If the source codes of those functions are available for modification, client application transparency can be preserved.

IDL specification file, object implementation and server application transparency:

The CORBA programmers need not change the original IDL files, object implementation source files and server applications to facilitate the flyover communication path. This implementation transparency is important because the application programmers do not need to know how the flyover works.

3.4 Design Challenges and Solutions

Enforcing client and server application transparency in the design of flyover optimization is the major challenge. Implementation transparency is important because from the application programmer's point of view, there is no change in the original client and server programs, object implementations and IDL specifications as well. To provide implementation transparency, two possible methods can be used.

Changing the ORB source code: A CORBA ORB is typically implemented as a run-time library linked into both the client and server. If the source code of the ORB library is

available, the marshalling/demarshalling routines and compressing GIOP headers can be modified directly. There are several open source ORBs available such as Orbacus and omniORB2. However, a major drawback of modifying ORB source code is that it is error-prone and tedious. In addition, most commercial ORB vendors do not publish their source codes. Hence this method is not appropriate in general for the implementation of the flyover.

Changing the client stub and server skeleton: This method involves the modification of the client stub and server skeleton codes, which are usually generated by the IDL compiler according to the IDL specification files. Since the stub and skeleton codes are generated as source files, rather than compiled as a library, those files can be modified to adapt to the flyover. Obviously, those modifications in stub and skeleton codes cannot be not expected to be done by the application designer because preserving implementation transparency is one of key requirements of the flyover. Moreover, most software designers do not program the stub and skeleton codes directly. Therefore the flyover includes an Interface Definition Language Processor (IDL P) to deal with all the changes in stub and skeleton codes as well as the IDL source files.

In a general CORBA application development paradigm as shown in Figure 3-4, the IDL compiler first compiles the IDL interface specification into a particular programming language mapping. Suppose that the target programming language is C++. The IDL compiler generates the header file, which contains the IDL C++ class mappings corresponding to the IDL interfaces, and the client stub and server skeleton files. Then the application programmer compiles the client source code, client stub code and IDL C++ class file to generate the executable code using C++ compiler. Similarly, to generate

server executable code, the server source code, object implementation code, server skeleton code and the IDL C++ class file are compiled.

Figure 3-4 General CORBA applications development paradigm

To incorporate the flyover into the CORBA applications, IDLP is used for compiling the IDL interface specification file, instead of the original IDL compiler. Figure 3-5 describes the procedures of generating client and server executable using IDLP.

Figure 3-5 Workflow of generating client and server executable codes using IDLP

The IDL parser first analyzes the syntax of IDL interfaces and modifies the IDL specification file. Afterwards, IDLP calls the IDL compiler to generate the header file, client stub and the server skeleton. Then, the C++ parser inside IDLP analyzes the structures of those files and modifies them to incorporate the flyover into the CORBA applications. The client and server executable can be generated as usual. From the

application programmer's point of view, there is no change in CORBA application development paradigm except that IDLP is used instead of IDL compiler in compiling IDL specification files.

3.5 Overview of a CORBA Request-Reply Transaction in the Flyover

Figure 3-6 presents a high-level picture of a CORBA request-reply transaction in the flyover. If the client wants to send a request to the server through the ORB, the client stub initializes a `CORBA::Request` object and then passes it through the ORB. Class `CORBA::Request` is a CORBA compliant class which is responsible for carrying the request/reply information between the client and server through the ORB.

Figure 3-6 High-level view of a CORBA request-reply transaction in the flyover

However, if the client wants to send the CORBA request through the flyover, the client stub initializes a `Message` object which is a derived class of `CORBA::Request`. Class `Message` has the marshalling and demarshalling functions that encode/decode the parameters and return value in/from the optimized GIOP. Moreover, the message object is sending through a private socket connection between the client and server, rather than through the ORB core.

The following section provides an overview of operations performed during a CORBA request-reply transaction in the flyover. Section 3.7 describes the design and

implementation details of the flyover.

3.5.1 Operations Performed During a CORBA Request-Reply Transaction in the Flyover

Six operations are performed during a request-reply transaction in the flyover.

- 1) When the first object is instantiated in the server, the extra codes inserted in server skeleton by IDLP initializes a hash table to store the servants (object implementations), which are instantiated in the server process. Therefore the object reference can be retrieved later based on the object ID during request demultiplexing. The server skeleton also initializes a listening thread, which is responsible for continuously listening for the optimized GIOP message from the client. Note that the flyover and the ORB are running on different threads of control on the server side. Therefore the listening and processing of requests in the flyover path will not block the execution of the underlying ORBs. The thread synchronization is done by using of mutexs and conditional variables.

At last, the server skeleton initializes a `port_locator` CORBA object which sets up an IP port number in the server. This port number is used to identify the flyover server on a TCP/IP network.

- 2) After binding the object reference, the client stub modified by the IDLP can locate the target object (obtains the address information; e.g. IP address, port number and object ID) and then establish a socket connection to the server. Recall the discussion in Section 3.3.1 that using the proprietary binding function `_bind()` to obtain the object reference preserve application transparencies in the flyover. However, using

the naming service or the stringified object reference requires little modification of the client application source files.

- 3) The client invokes the method on the object reference and the flyover path is chosen if the client and server exhibit similarity.
- 4) The modified client stub marshals the CORBA method invocation into an optimized GIOP message (without CDR conversions and padding bytes) by calling the class `Message` methods. Then the stub sends the message through the private socket connection established in step 2.
- 5) The server receives the data from the socket and demarshals the method parameters by calling the class `Message` methods. Moreover, the modified skeleton code demultiplexes the request to the target object according to the object ID and operation ID. The request demultiplexing requires the hash table initialized in step 1.
- 6) Server sends a reply message back to the client if the method is a two-way call or if an exception is raised. The reply message is encoded/decoded in/from the optimized GIOP.

3.6 Case Study

In this research, the flyover is implemented on Orbix-MT 2.3 [21], which is commercially available and one of the most popular ORBs. Orbix-MT 2.3 is a multi-threaded ORB and compliant with the CORBA v.2.1 standard. Orbix-MT 2.3 is a *Handle-Driven* ORB (H-ORB) [1][2], in which the broker returns a handle, also called object reference, back to the requesting client. Then the client uses this handle to invoke the CORBA method on the object implementation.

Orbix-MT 2.3 supports two mechanisms for relating an implementation class to its IDL interface: the BOAImpl approach and TIE approach [21]. To use BOAImpl approach, Orbix generates an IDL C++ class with the same name for each IDL interface. It also generates a second C++ class using the name of the interface appended with the letters BOAImpl (e.g. <IDL interface>BOAImpl). Each BOAImpl-class inherits from the corresponding IDL C++ class and the implementation class should inherit from the corresponding BOAImpl-class. The BOAImpl approach is shown in Figure 3-7 for the IDL interface interfaceA. The IDL compiler generates the C++ classes interfaceA and the BOAImpl-class interfaceABOAImpl. The programmer defines a new class called interfaceA_implementation to implement the functions defined in the IDL C++ class.

Figure 3-7 The BOAImpl approach to define a C++ implementation class

In the TIE approach, the server programmer can implement the IDL operations and attributes using the C++ DEF_TIE macro to tie the programmer's implementation class to the IDL interface, instead of using the BOAImpl-class. The DEF_TIE macro has the general form DEF_TIE_<IDL interface>(<implementation class>). Each call

to this macro defines a TIE-class called `<IDL interface>##<implementation class>`. For example, calling macro `DEF_TIE_interfaceA(interfaceA_implementation)` generates the TIE class `interfaceA##interfaceA_implementation`. Figure 3-8 illustrates using TIE approach to define a C++ implementation class.

Figure 3-8 The TIE approach to define a C++ implementation class

Since the flyover is heavily based on the client stub and server skeleton codes generated by the Orbix-MT 2.3 IDL compiler, it is easier to demonstrate the design details of the flyover with a case study. Note that using the BOAImpl and TIE approach to define a C++ implementation class are supported in the flyover. However, for the sake of limited space in this thesis, the details of using TIE approach are not provided. All object implementations described in the following sections are using the BOAImpl approach.

3.6.1 Case Study: CORBA IDL Interfaces

The IDL interface, object implementation, client and server source files are defined in Figure 3-9. The test IDL interface is called `interfaceA`. It contains four interface

IDL interface source file:

```
//CORBA IDL
//transfer a sequence of long integer with length of 8192 (8K) and a
//floating point number
//return an unsigned short integer and a floating point number
typedef sequence<long, 8192> strseq;

interface interfaceA {
    unsigned short memfun(in strseq a, inout float b); //test function
    void foo(in long a);
    void bar(in float a);
    void foobar(in double a);
};
```

Object implementation source code:

```
class interfaceA_implementation : public interfaceABOAImpl{
    CORBA::UShort interfaceA_implementation::memfun (const strseq& a, \
                                                    CORBA::Float& b, \
                                                    CORBA::Environment &IT_env) \
    {
        throw (CORBA::SystemException){
            b = 1.2; //set the output parameter b (float)
            return 1; //set the return value
        }
    }
};
```

Client source code:

```
#define SEQ_SIZE 8192
main(int argc, char*argv[]) {
    //define the object reference on the client side
    interfaceA_var objA;

    //argv[1] contains the server hostname
    //bind the object "firstobject" in the server "demo"
    //to object reference objA using Orbix bind function
    objA = interfaceA::_bind("firstobject:demo", argv[1]);

    strseq largedata;
    //initialize the long sequence
    largedata.length(SEQ_SIZE); //SEQ_SIZE = 8192

    for (int j = 0; j<SEQ_SIZE; j++)
        largedata[j]=j%32768;
    CORBA::UShort ret;

    //invoke the CORBA method
    ret = objA->memfun(largedata, 1.234);
}
```

Server source code:

```
main() {
    //initialize three objects in the server
    //the parameter is the object name
    interfaceA_implementation A("firstobject");
    interfaceA_implementation B("secondobject");
    interfaceB_implementation C("thirdobject");

    //start the server "demo"
    CORBA::Orbix.impl_is_ready("demo");
}
```

Figure 3-9 Sample source codes for case study

methods, namely `memfun`, `foo`, `bar` and `foobar`. Only the function `memfun` is the testing function used in the case study. The execution trace of the invocation of this testing interface function in Orbix-MT 2.3 and in the flyover are presented in Section 3.6.2 and Section 3.7 respectively.

The testing interface function `memfun` takes one *in* parameter `a` of type IDL sequence `strseq`, one *inout* parameter `b` of type `float`, and return an unsigned short integer. The user-defined type `strseq` is a CORBA IDL sequence of long integer with a length of 8192.

In the client source code, it first binds the object reference `objA` to the target object “firstobject” in the server “demo” using the Orbix binding function `_bind()`. Then it initializes the sequence and invokes the interface method on the object reference.

In the server source code, it initializes three objects in the server. The first two objects are `interfaceA` objects and the third one is the `interfaceB` object (`interfaceB` is not shown in the IDL interface source file). The function `impl_is_ready` starts the server “demo”.

3.6.2 Execution Trace of a Invocation of Interface Method in Orbix-MT 2.3

Figure 3-10 shows the class diagrams of IDL interface `interfaceA` on the client side and the server side in Orbix-MT 2.3 using BOA approach. `CORBA::Object` is the base class for all IDL C++ classes [30]. It provides functions to retrieve object reference fields in string format and to convert between object references and strings. On the client side, IDL C++ class `interfaceA`, which is generated by the IDL compiler corresponding to the IDL interface `interfaceA`, inherits from the base class `CORBA::Object`. The C++ class `interfaceA` contains the proxy function `memfun` that is the C++

mapping of the operation memfun of interfaceA. Calling the proxy function memfun on the client side will marshal the request and send it to the server through the wire.

Similarly, on the server side, the IDL C++ class also inherits from the base class CORBA::Object. However, the class hierarchy is different from that on the client side. Since the BOAImpl approach is using here, the BOAImpl-class interfaceABOAImpl is inherited from the IDL C++ class interfaceA and the implementation class is derived from the BOAImpl-class. Note that the BOAImpl-class interfaceABOAImpl is an abstract class with the pure virtual function memfun. When the ORB receives the request from the client, it will dispatch the request to the object implementation.

Figure 3-10 Class Diagrams of IDL interface interfaceA in Orbix-MT 2.3

3.6.2.1 Orbix Client

According to the interface specifications, the IDL compiler generates the client stub and server skeleton codes. The client stub contains code to implement the function memfun on the client side. Figure 3-11 shows the sequence diagram for the method invocation on the client side. In Figure 3-11, the variable objA is an object reference of

C++ class `interfaceA` on the client side (see sample source codes in Figure 3-9). Class `strseq` is the C++ mapping of the CORBA IDL sequence `strseq`. Class `CORBA::Request` is the CORBA compliant class which is responsible for carrying request information from the client to the server. The entities `objA:interfaceA`, `a:strseq` and `IT_r:Request` belong to the client stub of `interfaceA`. Each interaction in Figure 3-11 is outlined below.

Figure 3-11 Sequence diagram for client method invocation in Orbix-MT 2.3

1. When the client program calls `objA->memfun(largedata, 1.234)` (see sample code in Figure 3-9), it will invoke the operation `memfun` of C++ class `interfaceA` in the client stub with two input parameters: bounded sequence `a` and floating point number `b`.
2. The stub code marshals the first input parameter `a` by calling the function `encodeOp` on the class `strseq`, which is generated by IDL compiler. This function takes the

request object `IT_r` as a parameter and marshals the sequence `a` into the request object.

3. The operator `<<` is the overloading operator function of the class `CORBA::Request`. This function marshals the sequence length first. The variable `_length` is a member of class `strseq` that specifies the sequence length.
4. Then it marshals the content of sequence buffer. The variable `_buffer` is a member of class `strseq` that stores the sequence element.
5. The stub code marshals the second input parameter `b` using `CORBA::Request`'s overloading operator `<<`.
6. After marshalling the arguments, the client stub calls the `invoke` method on the `Request` object. This `invoke` method sends the request to the server through the wire.
7. After the server sends back the result, the `invoke` method will return. The client stub first demarshals the return value (unsigned short).
8. Afterwards, the client stub demarshals the output parameter `b` (float) using the overloading operator `>>`. Then the stub code returns the control back to the client. Until this point in time the client application remains blocked.

3.6.2.2 Orbix Server

Suppose the server objects have been initialized. When the Orbix ORB receives the GIOP message, it extracts the object key and dispatches the request to the correct object. To facilitate request demultiplexing in Orbix-MT 2.3, each `BOAImpl`-class has its own dispatch object named `m_pptr` of type `<IDL interface>_dispatch` (e.g. `interfaceA_dispatch`) which is a member of its corresponding `BOAImpl`-class. (e.g.

`interfaceABOImpl`). The interface dispatch class has a dispatch method and an object implementation pointer `IT_pp`. Figure 3-12 shows the association between the dispatch class and `BOImpl`-class for the IDL interface `interfaceA`.

Figure 3-12 Class diagram of dispatch class in Orbix-MT 2.3

The object demultiplexing is automatically done in the server ORB based on the object key of the incoming request from the client. In this case study, when the GIOP message arrives at the server, the server ORB will call the dispatch method of the `interfaceA_dispatch` object with a `Request` object as an input parameter. This `Request` object contains information about the request (e.g. object key, operation name and parameters) sent from the client. Moreover, the server ORB sets the object pointer `IT_pp` pointing to the correct object implementation (`interfaceA_implementation`). Since Orbix is not an open source ORB, the detail of object demultiplexing inside the ORB is missing (e.g. how to locate the target object based on the object key). Moreover, in accordance with the CORBA specification [30], object demultiplexing is vendor specific. The ORB vendor can put whatever information inside the object key so that the server ORB can locate the target object using their own demultiplexing methods.

However, the operation dispatching is done on the dispatch function of the interface dispatch object that resides in the skeleton code. Figure 3-13 shows the corresponding sequence diagram for the operation dispatching in the skeleton code. In Figure 3-13, the

entities `m_pptr:interfaceA_dispatch`, `IT_r:Request` and `a:strseq` belong to the server skeleton. The `IT_pp` is a `CORBA::Object` pointer which points to the target object implementation. The interactions in Figure 3-13 are outlined below.

Figure 3-13 Sequence diagram for operation dispatching in Orbix-MT 2.3

1. When the GIOP message arrives at the server, the server ORB will call the dispatch method of the `interfaceA_dispatch` object with a `Request` object as an input parameter. Moreover, the server ORB sets the `IT_pp` pointer pointing to the target object implementation.
2. The server skeleton gets the operation name from the request object by calling the function `getOperation` [30]. In this case study, the value of `IT_s` is "memfun". Orbix-MT 2.3 uses linear operation dispatching.
3. The skeleton compares the operation name to locate the corresponding marshalling function. Then it demarshals the first input parameter `a` (sequence) by calling the `decodeOp` method. This function is a class method of `strseq` which takes the request object `IT_r` as an input parameter.

4. The operator `>>` is the overloading function of the class `CORBA::Request`. This function demarshals the sequence length.
5. Then the server skeleton demarshals the content of sequence buffer.
6. The server skeleton demarshals the second input parameter `b` (float).
7. After demarshalling of the parameters, the skeleton calls the function `memfun` on the real object. Note that the dynamic (or run-time) type of `IT_pp` pointer is `interfaceA_implementation` which points to the target object implementation. The variable `IT_env` is the CORBA environment object which stores the exceptions.
8. After executing the servant's function, the skeleton code marshals the return value `IT_result` (unsigned short) using overloading operator `<<` on the request object.
9. The skeleton code demarshals the output parameter `b` and returns the request back to the client.

3.7 Execution Trace of a Invocation of Interface Method in the Flyover

Section 3.5.1 already describes the overview of operations performed during a CORBA request-reply transaction in the flyover. This section presents the execution trace of the test interface method, which is described in the Section 3.6, through the flyover in details. Moreover, this session also presents the design details of the flyover.

There are six steps involved during a request-reply transaction in the flyover, which will be covered from Section 3.7.1 to Section 3.7.6. First, Section 3.7.1 presents the details of server object registration, and initialization of the server listener and port locator object. Second, the details of establishing a socket connection to the target object are discussed in Section 3.7.2. Third, Section 3.7.3 discusses the method invocation on the client side in the flyover. Fourth, the format of optimized GIOP and the marshalling

of method invocation into optimized GIOP are addressed in Section 3.7.4. Fifth, Section 3.7.5 presents the message demultiplexing and demarshalling on the server side. Finally, Section 3.7.6 describes the details of replying message in the flyover.

3.7.1 Step 1: Server Initialization

3.7.1.1 Server Object Registration

The server must be able to store the object reference (pointer) for each object instantiated in the server process to allow demultiplexing of the request received from the flyover path to the target object. One solution is to store all object pointers into a hash table, which resides in the server skeleton as shown in Figure 3-14.

Figure 3-14 Server object registration

In Figure 3-14, there are three objects resided in the server process. These three objects correspond to objects initialized in server source code shown in Figure 3-9. These objects belong to different C++ interface classes. For example, objects A and B are of type interfaceA, and objects C are of type interfaceB. To allow all object implementations of different interface class to store in a hash table, the original BOAImpl-class interfaceABOAImpl is multiple inherited from the original IDL C++

interface class `interfaceA` and a new class called `BypassImpl`, as shown in Figure 3-15. These modifications of the class hierarchy of the interfaces defined in the header file are done by IDLP.

Figure 3-15 Modified class diagram of interfaceA on the server side

The main purpose of introduction of class `BypassImpl` is to make all the servants inherit from a common base class and to store the servant pointers into the hash table. The hash table key is an object ID of type unsigned integer and the hash table value is the object pointer of type `BypassImpl*`. The object ID is automatically assigned by IDLP when the servant registers in the server. The flyover uses this object ID to retrieve the target object from the hash table. In addition, the class `BypassImpl` provides a dispatch function for request demultiplexing (the `dispatch` method will be discussed in Section 3.7.5).

When the object is instantiated in the server (server source code in Figure 3-9), the server process will call the constructor of class `interfaceABOImpl`. The class `interfaceABOImpl`'s constructor defined in the skeleton code is modified by IDLP for object registration. This constructor first gets a unique object ID and stores the object ID into the new member variable `__objindex__` added by IDLP (see Figure 3-15). This object ID uniquely identifies the servant within the server process. Figure 3-16 shows the sequence diagram for object registration in the server.

Figure 3-16 Sequence diagram for server object registration

Figure 3-16 introduces two new classes called `BypassORB` and `ServiceHandler`. Class `BypassORB` is responsible for initialization of the listening thread. Class `ServiceHandler` stores the object references into the hash table and retrieves it back based on the object ID during request dismultiplexing. Each interaction in Figure 3-16 is explained below.

1. The constructor of class `interfaceABOImpl` gets the object ID by calling the method `getObjectIndex`.
2. It registers the object pointer into the hash table. Therefore the object pointer can be retrieved later according to the object ID during request demultiplexing.

3. It stores the object pointer into the hash table `ht`, which is implemented in STL (Standard Template Library) [52]. The hash table `ht` stores the object pointer of type `BypassImpl*` with object ID (unsigned long) as the key.

3.7.1.2 Initialization of Server Listener

The server listener is an entity in the flyover which is responsible for listening to the messages from the client and dispatching it to the servant. It is running on a different thread of control so that it will not block the operations of the underlying ORB. All the synchronization problems (e.g. the ORB and the flyover may access the object method at the same time) are resolved by using mutexs and conditional variables.

When the first object is instantiated (i.e. object A in server source code in Figure 3-9), besides registering the object into the hashtable, the class `BypassImp`'s constructor also invokes the `initServerProxy` function to set up the server listener. The sequence diagram for the server proxy initialization is illustrated in Figure 3-17.

Figure 3-17 Sequence diagram for initialization of server listener

The `initServerProxy()` function of class `BypassORB` first spawns a new thread. Hence all the work of the server listener is running on its own thread of control (`threadcontrol()` method). The first step of initialization in the server proxy is setting

up the listening socket, mutexs, conditional variables and port number object. Then it waits on the `select()` system call [55] for any incoming request from the client through the flyover path.

Note that only the first object instantiated needs to initialize the server listener; the others only need to register their `BypassImpl` pointers into the hash table `ht`.

3.7.1.3 Port Locator CORBA Object

The port locator is a CORBA object to setup the server port number. The `port_locator` has the following IDL interface:

```
//IDL interface
interface portlocator {
    readonly attribute port_num;
    //set the port number
    //return 0=success -1=error
    short setportnumber(in unsigned short port_num);
};
```

Figure 3-18 IDL interface of the Port Locator object

The port number is first randomly chosen between 1024 and 65535 and then is passed to the `setportnumber` after checking for any duplicated port number. The client stub can retrieve the port number by calling the remote method `getportnumber`.

3.7.2 Step 2: Establishing a socket connection to the server

To uniquely identify the target object on a TCP/IP network, the IP address, port number and object ID are required. The IP address and port number can identify the server process on the TCP/IP network, and the object ID is used to identify the target object (servant) within the server process. Getting this information automatically is important in order to support the requirement of location and communication transparency in the flyover. To acquire the server's address information transparently, the flyover borrows the function of object binding in CORBA.

In CORBA, the client can get the object reference by using a proprietary function call, naming service or by de-referencing of a stringified object reference (Section 2.3.9). First, consider the case of using a proprietary bind function that is presented in the following subsection.

3.7.2.1 Using Orbix Bind function

Orbix 2.3-MT provides a bind function to get the IOR for each IDL interface (e.g. `interfaceA* interfaceA::_bind("objname:servername",hostname)`) [21]. This `_bind` function is generated by Orbix IDL compiler for each IDL interface defined in the IDL interface specification file and is located inside the client stub code.

To get the IP address, port number and object ID, IDLP modifies the code of the bind function inside the client stub code. Moreover, IDLP adds two private member variables `myproto` and `__objindex_` to every IDL interface on the client class hierarchy shown in Figure 3-19.

Figure 3-19 Modified class diagram of interfaceA on the client side

The private member variable `myproto` is a struct `comm_point` that stores the address information of the target object; i.e. server IP address, port number and socket

descriptor. The other member variable `__objindex_` stores the object ID of the target object implementation.

Figure 3-20 shows the sequence diagram for getting the address information (e.g. server IP address, port number and object ID) of the target object using the bind function inside the modified client stub. The interactions are summarized below.

Figure 3-20 Sequence diagram for locating the target object using the bind function

1. The client binds the object reference using Orbix bind function as usual.
2. This is the only function call in the original bind function. It gets the object reference through the `CORBA::Factory`'s method `New` [30].
3. This function call is added by IDLP to the code of the original bind function, which resides in the client stub code. The main purpose of this call is to retrieve the IP address of the target server. Inside the function `BypassORB::_bind`, the `_host()` is the CORBA compliant function to get the server hostname in which the target object is located. After getting the server hostname, the client stub can get the IP address corresponding to the server hostname using the DNS (Domain Name Service) [55].

4. This step gets the servant's object ID and the port number of the target server by invoking the CORBA interface method `GET__OBJKEY__()`.
5. The server object gets the object ID and the server port number.

Every server object is assigned a unique object ID after registration (see Section 3.7.1.1). The function `GET__OBJKEY__()` is a CORBA interface method to get the servant's object ID and port number. The IDL parser inside IDLP (see Figure 3-5) inserts this method to each interface defined in the original IDL source files shown in Figure 3-21.

```
//to transfer a sequence of double with length of 8192
//CORBA IDL
typedef sequence<double, 8192> strseq;

interface interfaceA {
    unsigned short memfun(in strseq a, inout float b);
    ... //other interface functions

    //modified: the following interface method is added by IDLP processor
    //output: the object ID and the port number
    unsigned long GET__OBJKEY__(out unsigned short port);
};
```

Figure 3-21 Adding interface method `GET__OBJKEY__()` by IDLP

To preserve server application transparency, the implementation of this interface function on the server side is automatically done by IDLP. The application programmers do not need to implement this function. Having compiled the IDL source file, the IDLP replaces the function call `GET__OBJKEY__()` in the *server skeleton* by the function call `__get__objkey__()` as shown in Figure 3-22. The function `__get__objkey__` is a method for every interface class (see Figure 3-15), which is added by IDLP in the server skeleton code. This method returns the `__objindex_` (Object ID) of the target object.

The interface method `GET__OBJKEY__` is also responsible for returning the server port number as an output parameter. In other words, one CORBA method call is invoked to get both the object ID and port number of the target server.

<p>Original: <code>//This line will call the object implementation method GET__OBJKEY__ //but it does not exist in the object implementation ((interfaceA*)IT_pp)->GET__OBJKEY__(...);</code></p> <p>Modified: <code>//It returns the __objectindex (see Figure 3-20 sequence no. 5) ((interfaceA*)IT_pp)->__get__objkey__(...);</code></p>

Figure 3-22 Modification of server skeleton code in getting target object ID

3.7.2.2 Using CORBA Naming Service or Stringified Object Reference

If the client binds the objects by using CORBA Naming Service or by de-referencing stringified object reference, the client application programmer must call the binding function `interfaceA::_explicitbind` explicitly to create the flyover communication path to the server. As described in Section 3.5.1, if the client uses the naming service or stringified object reference to obtain the object reference, modification on the client application is required because the source codes for the Orbix CORBA naming service or for the ORB interface function `string_to_object` are not available.

Figure 3-23 (i) shows the sample codes to locate the target object by using CORBA Naming Service. To create a flyover path to the server, the client must call the `resolve` method first and then call the `_explicitbind` function that is defined in Figure 3-23 (iii). Similarly, if the client application designer wants to de-reference the stringified object reference to obtain the object reference, he/she must also call the `_explicitbind` function as shown in Figure 3-23 (ii).

If the source codes of naming service function `resolve` and ORB interface function `string_to_object` are available, client application transparency can be preserved.

i) Locate target object using CORBA Naming Service in the flyover

```
//C++
CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);

//Get naming service
CORBA_Object_var obj = orb->resolve_initial_references("NameService");

//narrow to get naming context
CosNaming_NamingContext_var nc = CosNaming_NamingContext::_narrow(obj);

CosNaming_Name n;
n.length(1);
//create object name in root context
n[0].id = CORBA_string_dup("ObjectA");
n[0].kind = CORBA_string_dup("");

//resolve name
CORBA_Object_var obj = nc->resolve(n);

//narrow to derived interface type
interfaceA_var objA = interfaceA::_narrow(obj);

//explicitly create a flyover path to the server
interfaceA::_explicitbind(objA);
```

ii) Locate target object using Stringified Object Reference in the flyover

```
//C++
CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);

//open the file "objA.ref"
const char* refFile = "objA.ref";
ifstream in;
in.open(refFile);

char s[1000];
in >> s;
in.close();

//Get the object reference
CORBA_Object_var obj = orb -> string_to_object(s);

//narrow to derived interface type
interfaceA_var objA = interfaceA::_narrow(obj);

//explicitly create a flyover path to the server
interfaceA::_explicitbind(objA);
```

iii) Definition of explicit binding function _explicitbind

```
//C++
void interfaceA::_explicitbind(CORBA::Object obj){
    obj->myproto = BypassORB::_bind(obj);

    obj->__objindex_=((interfaceA*)tmpobj)->GET__OBJKEY__(obj->myproto.port);
}
```

Figure 3-23 Sample client codes for locating target object in the flyover using i) CORBA Naming Service and ii) Stringified Object Reference

3.7.3 Step 3: Method Invocation on the Client Stub

When the client calls the object reference's member function, this call will be processed by the client stub. The client stub code implements the remote function on the client side. Section 3.6.2 already shows that the function invocation on the client stub greatly depends on the CORBA Request object. To minimize the modification on the client stub code, a new class named Message is defined. The class Message is inherited from the CORBA class Request and overrides the Request's methods. Since the interface of class Request is well-defined and compliant with the CORBA standard, all the functions of (de)marshalling and compressing GIOP header can be implemented in the class Message. Figure 3-24 shows the definition of class Message and class Request.

Figure 3-24 Class Message in the flyover

Each interface member function in the client stub is modified by IDLP in order to direct the request (Message object) to the pre-established communication path in the flyover. In the modified client stub, the original Request object in the client stub is replaced by the Message object; therefore, all the methods applied on the Request object are directed to the Message object. Figure 3-25 shows the sequence diagram for client method invocation in the modified client stub. It is similar to the sequence diagram

for a client method invocation in Orbix-MT 2.3 (see Figure 3-11), except that the Message object replaces the Request object.

Figure 3-25 Sequence diagram for client invocation on the modified client stub

The interactions in Figure 3-25 are explained.

1. The client invokes the operation memfun of IDL C++ class interfaceA as usual with two input parameters: bounded sequence a and floating point number b.
2. The stub code marshals the first input parameter a by calling the function strseq::encodeOp(Message&), which is a new method added to the class strseq by IDLP.
3. The operator << is the overloading function of class CORBA::Message. This function marshals the sequence length _length into the optimized GIOP message. The format of optimized GIOP used in the flyover is explained in Section 3.7.4.3.
4. Then it marshals the content of the sequence buffer using the encodeArray method.

5. The modified stub code marshals the second input parameter `b` into optimized GIOP message.
6. After marshalling the arguments, the modified client stub calls the `invoke` method on the `Message` object. This `invoke` method sends the request to the server through the socket connection pre-established before.
7. After the server sends back the result, the `invoke` method will return. The modified client stub first demarshals the return value (unsigned short) from the reply message.
8. Afterwards, the modified client stub demarshals the output parameter (float) by using the overloading operator `>>`.

3.7.3.1 Operation ID

In the flyover, every static interface method and variable is assigned an operation ID by IDLP. For example, the operation ID `interfaceA_memfun_INDEX` is used to identify the operation of the target object, instead of the operation name “memfun” in the case study. The rule of C++ mapping of operation names and variables, as shown in Table 3-2, is explained below.

For each member function defined in the IDL interface, there is one unique C++ constant named `<IDL interface>_<memfun>_INDEX` which is assigned a unique number (operation ID) by IDLP. This operation ID is started from one. For example, the IDL interface `interfaceA` shown in Table 3-2 (i) has five member functions from `memfunA` to `memfunE`. The corresponding C++ mappings are from `interfaceA_memfunA_INDEX` to `interfaceA_memfunE_INDEX` with values from 1 to 5. Moreover, IDLP assigns two unique operation IDs for each read-and-write interface variable. The operation ID `<interfaceA>_get_<memvar>_INDEX` is used for getting

the value of the servant's member variable `memvar`, whereas operation ID `<interface>_set_<memvar>_INDEX` is used for setting the value.

Table 3-2 Mapping of operation ID of interfaceA by IDLP

IDL Source File:	Corresponding C++ mapping of operation ID by IDLP:
(i) member functions <code>memfunA();</code> <code>memfunB();</code> <code>memfunC();</code> <code>memfunD();</code> <code>memfunE();</code>	Mapping of Operation ID for each function: <code>#define interfaceA_memfunA_INDEX 1</code> <code>#define interfaceA_memfunB_INDEX 2</code> <code>#define interfaceA_memfunC_INDEX 3</code> <code>#define interfaceA_memfunD_INDEX 4</code> <code>#define interfaceA_memfunE_INDEX 5</code>
(ii) read and write member variables <code>attribute memvarA;</code> <code>attribute memvarB;</code> <code>attribute memvarC;</code> <code>attribute memvarD;</code>	Mapping of Operation ID for each variable <code>#define interfaceA_get_memvarA_INDEX 6</code> <code>#define interfaceA_set_memvarA_INDEX 7</code> <code>#define interfaceA_get_memvarB_INDEX 8</code> <code>#define interfaceA_set_memvarB_INDEX 9</code> <code>#define interfaceA_get_memvarC_INDEX 10</code> <code>#define interfaceA_set_memvarC_INDEX 11</code> <code>#define interfaceA_get_memvarD_INDEX 12</code> <code>#define interfaceA_set_memvarD_INDEX 13</code>
(iii) read-only member variables <code>readonly attribute memvarE</code> <code>readonly attribute memvarF</code> <code>readonly attribute memvarG</code>	Mapping of Operation ID for each read-only variable <code>#define interfaceA_get_memvarE_INDEX 14</code> <code>#define interfaceA_get_memvarF_INDEX 15</code> <code>#define interfaceA_get_memvarG_INDEX 16</code>

The main purpose of using the operation ID instead of the operation name is to reduce the memory footprint of the GIOP message (see Section 3.7.4.3) and provide efficient operation dispatching (see Section 3.7.5).

3.7.4 Step 4: Marshalling of Method Invocation into the Optimized GIOP

The optimized GIOP is a communication protocol between the client and server over the socket communication path pre-established in Step 2 in the flyover. It is similar to the CORBA GIOP but it is highly optimized: for instance, redundant fields such as magic number and reserved field are removed. This step is of importance because efficient marshalling and demarshalling of the client request into the optimized GIOP is crucial in achieving low latency. Three basic optimizations are employed over CORBA GIOP.

3.7.4.1 Preventing CDR Conversions

If the machines in which the client and the server are running use the same kind of

hardware and operating systems, the overhead of data conversion can be avoided.

3.7.4.2 Removing Padding Bytes

In CDR, all primitive data types must be aligned on their natural boundaries relative to the start of the CDR-byte stream [30]. Any primitive of size n octets must start at an octet stream index that is a multiple of n . In CDR, n is one of 1, 2, 4, or 8 (see Table 3-3).

Table 3-3 CDR alignment of primitive data type

Octet Alignment	IDL Data Types
1	char, octet, boolean
2	short, unsigned short
4	long, unsigned long, float, enum
8	long long, unsigned long long, double, long double

For instance, the use of padding bytes in CDR is explained with the help of following IDL example.

```
//IDL structure
struct paddingdemo {
    char c;
    double d;
};
```

This IDL structure paddingdemo contains two primitive data types: a character c , followed by a double value d . The character c can occur anywhere in a byte stream, whereas the double must be aligned on boundary with index of multiple of 8. The format of the message stream is shown in Figure 3-26.

Figure 3-26 Padding bytes in a data stream for the CORBA struct paddingdemo

In this example, the number of padding bytes in the request body is 7. This example shows the worst case situation that 43.75% of request body is used for transferring padding bytes. Removing these padding bytes can minimize the message size and hence saves communication bandwidth and improves transmission time. Note that these two strategies (Sections 3.7.4.1 and 3.7.4.2) are similar to those described in [3].

3.7.4.3 Optimized GIOP

In CORBA, the communications between the client and the server are through GIOP messages. There are eight types of messages defined in the GIOP standard which is defined in GIOP specification [30]. They are summarized in Table 3-4.

Table 3-4 GIOP message types and originator

Message Type	Originator	Value in GIOP Message Header
Request	Client	0
Reply	Server	1
CancelRequest	Client	2
LocateRequest	Client	3
LocateReply	Server	4
CloseConnection	Server	5
MessageError	Both	6
Fragment	Both	7

Since over 90% of CORBA method calls are Request and Reply messages [62], the optimized GIOP focuses on these two types of message.

Each GIOP request message consists of three parts: 1) GIOP Message Header 2) Request Header and 3) Request Body. Similarly, each GIOP Reply message is comprised of three parts: 1) GIOP Message Header 2) Reply Header and 3) Reply Body. Table 3-5, Table 3-6 and Table 3-8 depict the format of GIOP message header, request header and reply header respectively.

Table 3-5 Format of GIOP message header

GIOP Message Header (GIOP 1.1)	Size in octet
<pre>struct MessageHeader { char magic[4] Version GIOP_version //defined below octet flags octet message_type unsigned long message_size }</pre>	<pre>4 2 1 1 4</pre>
<pre>struct Version { octet major octet minor }</pre>	<pre>1 1</pre>

The main purpose of the message header is to clearly identify GIOP messages and the

byte-ordering.

- **magic** identifies GIOP messages. The value of this field is always set to “GIOP”.
- **GIOP_version** contains the version number of the GIOP protocol. The major GIOP version number of GIOP specification v.2.3.1 is 1; the minor version number is 0, 1 or 2. That is, in GIOP v.2.3.1, there are three versions of GIOP defined: 1.0, 1.1 and 1.2.
- **flags** is an octet in which the least significant bit indicates the byte ordering (false indicates big-endian and true indicates little-endian). The second least significant indicates whether this message is the last fragment (false) or not (true). There is no meaning defined for other bits.
- **message_type** indicates one of the following message types: Request, Reply, CancelRequest, LocateRequest, LocateReply, CloseConnection, MessageError and Fragment.
- **message_size** contains the number of octets in the message following the message header (see Table 3-5).

Table 3-6 Format of request header

GIOP Request Header (GIOP 1.1 and 1.2)	Size in octet
struct RequestHeader {	
unsigned long request_id	4
boolean response_expected	1
octet reserved[3]	3
sequence<octet> object_key	*
string operation	*
IOP::ServiceContext service_context	*
Principal requesting_principal	*
}	
Remark: i) Principal not in GIOP 1.2	
ii) * means variable size	

The members of request header shown in Table 3-6 have following definitions:

- **request_id** is used to associate reply messages with request messages (including

LocateRequest messages). The client or requester is responsible for generating the request id.

- **response_expected** indicates whether it is a one-way or a two-way request. The values of **response_expected** are defined in Table 3-7.

Table 3-7 Format of response_expected

Method Invocation	value
two-way	00000011
one-way (without exception) or DII with INV_NO_RESPONSE flags	00000000
one-way with exception	00000001

- **reserved** is always set to 0. These three octets are reserved for future use.
- **object_key** identifies the target object of the invocation (see Section 2.3.7).
- **operation** is the name of the method being invoked. In case of attribute accessors, the names are **_get_<attribute>** and **_set_<attribute>**.
- **service_context** contains ORB service data Context being passed from the client to the server. A context is intended to represent information about the client which is inconvenient to pass via parameters. This function may be used to provide access control [30].
- **requesting_principal** contains the value identifying the request principal. Class **Principal** implements the IDL pseudo interface **Principal** [30], which represents information about the end-users. This information may be used to provide authentication and access control. Also, it is provided to support the **BOA::get_principal** operation. This field is not present in GIOP v.1.2.

Table 3-8 Format of reply header

GIOP Reply Header (GIOP 1.1)	Size in octet
<pre>struct ReplyHeader { unsigned long request_id ReplyStatusType reply_status IOP::ServiceContext service_context }</pre>	<pre>4 4 *</pre>
Remark: * means variable size	
<pre>enum ReplyStatusType { NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION, LOCATION_FORWARD }</pre>	

The members of reply header shown in Table 3-8 have following definitions:

- `request_id` contains the same request id value as the corresponding request.
- `reply_status` indicates the completion status of the associated request. It can be one of the following values:

`NO_EXCEPTION` - There is no exception raised during the server processing.

`USER_EXCEPTION` - The user defined exception is raised.

`SYSTEM_EXCEPTION` - The system exception is raised. It can be either a specific system exception, for example `COMM_FAILURE`, or a general system exception `SystemException`.

`LOCATION_FORWARD` - The client must reissue the request to an object at some other location.

- `service_context` contains ORB service data Context being passed from the server to the client. The Class Context implements the pseudo interface Context. A context is intended to represent information about the client which is inconvenient to pass via parameters [30].

The memory footprint of GIOP message header and request header is large. For example, the size of object key is 78 bytes in Orbix-MT 2.3. Suppose the remote method

“memfun” (the test function in the case study described in Section 3.6) is invoked without any service context and principal, the total size of the message header and request header is 105 bytes. Recall that the object key is vendor specific, therefore the size of an object key may be variable. For example, the size of an object key in Orbacus 3.0 is 36 bytes.

To reduce the size of a GIOP message header and request header, the proprietary protocol used in the flyover is highly optimized (see Table 3-9).

Table 3-9 Format of optimized GIOP message header and request header

Optimized GIOP	Size in octet
struct OptimizedMessageHeader { octet message_properties unsigned long message_size }	1 4
struct OptimizedRequestHeader { unsigned long request_id unsigned long obj_id unsigned long fun_id IOP::ServiceContext service_context }	4 4 4 *
Remark: * means variable size	

The optimization applied on the GIOP in flyover is similar to TAO’s GIOPLite protocol. In OptimizedMessageHeader and OptimizedRequestHeader, there are no redundant fields such as GIOP magic number, GIOP version and reserve bytes. Also the flag and message_type in the message header and the response_expected in the request header are combined into the field called message_properties shown in Figure 3-27. Here the flyover assumes each GIOP message can carry only one request. The definitions of message type, flag and response expected are the same in GIOP v.1.2.

Figure 3-27 Format of message_properties in optimized GIOP

In the flyover, the object ID generated by the server can uniquely identify the object

within the server. In the optimized GIOP, the object ID replaces the object key in the request header. This step reduces the size of the request message header significantly. Recall the discussion in Section 2.3.7 that the object key is to identify the target object of the invocation. The object key itself consists of two parts: object adapter name and object name, which uniquely identifies particular object adapters in the server and the particular servants within the context of object adapter respectively. However, in the flyover, the object ID is enough to identify the target object within the server process.

In addition, the operation name within the request message is substituted by the corresponding operation ID, which is automatically assigned by IDLP during compilation of the IDL interface specification (see Section 3.7.3.1). Using operation ID instead of operation name not only greatly improves the efficiency of operation dispatching (will be explained in Section 3.7.5), but also reduces the size of request header (i.e. the size of function ID is 4 bytes (unsigned long) whereas the length of the operation name is usually more than 3 characters).

The optimized GIOP is lightweight. For example, the total size of the message header and the request header of the method call `memfun` is only 17 bytes as compared to 105 bytes of standard GIOP using Orbix-MT 2.3. There is a significant 84% saving in a request message header. It is important to note that the content of the object key is vendor specific. Therefore the bytes saving depends on the ORB vendors.

3.7.4.4 Marshalling and Demarshalling of Request Parameters into the Optimized GIOP

In the case study described in Section 3.6, the test function `memfun` takes two input parameters. The first one is a IDL sequence and the second is a floating point number. The following explains the marshalling and demarshalling of i) primitive data types and

ii) CORBA sequence.

i) Primitive data types supported in the flyover (see Table 3-1): The primitive data types are encoded and decoded by the class Message << and >> overloading operators.

Table 3-10 Marshalling and demarshalling functions of class Message for primitive types

//marshalling functions	//demarshalling functions
Message& operator<<(const CORBA::Boolean&)	Message& operator>>(CORBA::Boolean&)
Message& operator<<(const CORBA::Short&)	Message& operator>>(CORBA::Short&)
Message& operator<<(const CORBA::Long&)	Message& operator>>(CORBA::Long&)
Message& operator<<(const CORBA::UShort&)	Message& operator>>(CORBA::UShort&)
Message& operator<<(const CORBA::ULong&)	Message& operator>>(CORBA::ULong&)
Message& operator<<(const CORBA::Float&)	Message& operator>>(CORBA::Float&)
Message& operator<<(const CORBA::Double&)	Message& operator>>(CORBA::Double&)
Message& operator<<(const CORBA::Char&)	Message& operator>>(CORBA::Char&)

As shown in Table 3-10, for each primitive type there are one marshalling function called operator<<(const <basic type>&) and one demarshalling function operator>>(<basic type>&). For example, the modified client stub shown in Figure 3-25 calls the overloading operator<<(CORBA::ULong) to marshal the sequence length (Figure 3-25 interaction no. 3). Also it demarshals the output parameter b using the overloading operator >> (CORBA::Float).

ii) CORBA sequence: The marshalling and demarshalling of a CORBA IDL sequence in Orbix-MT 2.3 are carried out by the stub code and skeleton code. For instance, the sample code in Figure 3-9 defines the IDL sequence strseq. The corresponding C++ mapping of IDL sequence is the C++ class strseq. The IDL compiler generates the marshalling function strseq::encodeOp(CORBA::Request) and demarshalling function strseq::decodeOp(CORBA::Request) in the client stub and server skeleton.

To override the default (de)marshalling function of a bounded sequence in the flyover, the IDLP adds two new methods called strseq::encodeOp(Message&) and strseq::decodeOp(Message&) for each C++ sequence strseq.

To marshal a sequence, the function `strseq::encodeOp(Message&)` first encodes the length of the sequence in the optimized GIOP request body, and then encodes the sequence elements. Similarly, to demarshal a sequence, it extracts the length value from the message object first and then the sequence elements.

3.7.5 Step 5: Message Demultiplexing and Demarshalling on the Server Side

In the server process, the listening thread is waiting on the `select()` call (synchronous I/O multiplexing) [55] until an optimized GIOP message arrives at the server. The server thread reads the data from the socket and instantiates a `Message` object on the server side. Afterwards, the server process will dispatch the `Message` object to the `ServiceHandler` object. Recall that the main purpose of the class `ServiceHandler` is to get the object reference back from the hash table and dispatch it to the modified skeleton code.

Figure 3-28 presents the sequence diagram for request demultiplexing (object demultiplexing and operation dispatching) and demarshalling on the server side in the flyover. The class `ServiceHandler` and the hash table belong to the flyover. The class `BypassImpl`, the dispatch pointer `m_pptr` of class `interfaceA_dispatch`, the `Message` object `a_msg` and the sequence `strseq` belong to the server skeleton. The last item in Figure 3-28 is the object implementation `interfaceA_implementation` instantiated in the server.

Figure 3-28 Sequence diagram for request demultiplexing and demarshalling in the server

The interactions in the sequence diagram are explained:

1. The ServiceHandler extracts the object ID of the target object from the Message object by invoking the getObjectkey method. This method is not present in the original class CORBA::Request.
2. The handler uses the object ID as the key to extract the target object from the hash table ht. The hash table will return the BypassImpl pointer which points to the target object (see Section 3.7.1.1).
3. The handler calls the dispatch method on the target BypassImpl pointer. Recall that dispatch(Message&) is a pure virtual function of class BypassImpl. The implementation of this function is added by IDLP in the BOAImpl-class, which is defined in the server skeleton code. (See Figure 3-15 for details).

4. The `interfaceA_dispatch::dispatch2` function is invoked to dispatch the message object to the modified skeleton code. As mentioned in Section 3.6.2, the pointer variable `m_pptr` inside the class `interfaceABOAImp1` is an `interfaceA_dispatch` pointer. The function `dispatch2` is the modified skeleton code added by IDLP for operation dispatching and (de)marshalling. It is similar to the original `interfaceA_dispatch::dispatch` function generated by Orbix IDL compiler except a few line of codes is modified by IDLP.

In the `dispatch2` function, the input parameter is of type `Message` instead of `CORBA::Request` as compared to the original skeleton code (see Section 3.6.2.2 Figure 3-13). As a result, all the marshalling and demarshalling functions are invoked on the `Message` object.

5. The modified skeleton code gets the operation ID from the `Message` object. The original skeleton code gets the operation name instead.
6. The modified skeleton code compares the operation ID with each static interface operation ID to locate the correct demarshalling routine. Then it decodes the sequence `a` from the `Message` object. Using the operation ID instead of the operation name provides more efficient operation dispatching because comparing two strings is usually more computationally expensive than comparing two unsigned long integers.
7. The operator `>>` is the overloading function of the class `CORBA::Message`. This function demarshals the sequence length.
8. The content of the sequence buffer is demarshalled.
9. The modified skeleton code demarshals the second input parameter `b` (float).

10. After the skeleton code demarshals the parameters, the skeleton calls the function `memfun` on the real object.
11. After executing the servant's function, the skeleton code marshals the return value `IT_result` (unsigned short).
12. It demarshals the output parameter `b` and returns the request back to the client.

3.7.6 Step 6: Message reply

If the method call is two-way or an exception is raised, a optimized GIOP reply is sent back to the client. The mechanism of replying a message on the flyover is similar to sending a message, except that the reply message header replaces the request message header. In the case study described in Section 3.6, there are two return parameters for the method invocation `memfun`: the return value (unsigned short) and the *out* parameter `b` (float). After invoking the method in the object implementation, the server marshals the return value and the out parameter `b`. Then the skeleton sends it back to the client through the socket. There is no CDR conversion or padding bytes.

3.8 Data Types

3.8.1 Object Reference

CORBA allows passing an object reference as an in, out, inout parameter and a return value. To allow passing the object reference in CORBA, the flyover reuses the client stub and server skeleton code generated by IDL compiler. Consider the following IDL definition shown in Figure 3-29 which passes the object reference of interface `I1` into the function `bar` of interface `I2` and returns a duplicate of the object reference of interface

I1.

```
//IDL
interface I1 {
    void foo();
};
interface I2 {
    I1 bar(in I1 p);
};

//implementation of operation I2_i::op()
I1_ptr I2_i::bar(I1_ptr p){
    p->foo();
    return I1::_duplicate(p);
}

//Client program
I1_var a= I1::_bind(marker1, hostname1); //bind the object in the server
I2_var b= I2::_bind(marker2, hostname2); //bind the object in the server
I1_var c = b->bar(a);                    //test function
```

Figure 3-29 Example of passing object reference in the flyover

In the flyover, passing object reference also passes the location information of the target object (e.g. IP address, port number and object ID) implicitly. Suppose the client passes the object reference *a* into the function *bar* in the sample code shown above. This function takes an object reference of interface *I1* as an input parameter, call the interface method, and returns a duplicate of object reference *I1*. Figure 3-30 shows the sequence diagram for marshalling an object reference in the flyover.

Figure 3-30 Sequence diagram for marshalling of an object reference in the flyover

1. The client passes the object reference *a* into the interface method *bar*. The control is

passed to the client stub code of interface I2.

2. The modified client stub code first marshals the object reference. The overloading operator `Message::operator<<(CORBA::Object&)` gets the stringified object reference using the CORBA compliant function `object_to_string` [30].
3. The modified client stub code also marshals the structure `comm_point` which contains the server IP address and the port number. This method call is added by IDLP in the client stub code.
4. It also marshals object ID into the `Message` object. Since the object index is an unsigned long integer, the stub code uses the overloading operator `Message::operator<<(Ulong&)` to marshal the object ID.
5. The modified client stub sends the message through the socket connection to the server.

On the server side, the modified the server skeleton code extracts the address information (e.g. IP address, port number and object ID) of the target object from the message. Then the modified skeleton code initializes a socket connection to the target object.

3.8.2 String, Array, Union, Struct and Interface Inheritance

Besides primitive data types, bounded and unbounded sequence and object reference, the flyover also supports IDL bounded and unbounded string, array, union, struct and interface inheritance. The techniques used in marshalling and demarshalling of the string and array data types are similar to the bounded sequence described in Section 3.7. For instance, class `Message` provides two functions `encodeStringOp` and `decodeStringOp` to marshal and demarshal a string respectively. Figure 3-31 shows the

marshalling and demarshalling functions of class Request in Orbix-MT 2.3 and the corresponding functions in class Message in the flyover.

	Class CORBA::Request	Class Message (Flyover)
String	encodeStringOp(const char*& s) decodeStringOp(char*& s)	encodeStringOp(const char*& s) decodeStringOp(char*&)
Array	encodeCharArray(char*& s, CORBA::ULong len) decodeCharArray(char*& s, CORBA::ULong& len)	encodeCharArray(char*& s, CORBA::ULong len) decodeCharArray(char*& s, CORBA::ULong& len)
	encodeUCharArray(unsigned char*& s, CORBA::ULong len) decodeUCharArray(unsigned char*& s, CORBA::ULong& len)	encodeUCharArray(unsigned char*& s, CORBA::ULong len) decodeUCharArray(unsigned char*& s, CORBA::ULong& len)
	encodeShortArray(CORBA::Short*& s, CORBA::ULong len) decodeShortArray(CORBA::Short*& s, CORBA::ULong& len)	encodeShortArray(CORBA::Short*& s, CORBA::ULong len) decodeShortArray(CORBA::Short*& s, CORBA::ULong& len)
	encodeUShortArray(CORBA::UShort*& s, CORBA::ULong len) decodeUShortArray(CORBA::UShort*& s, CORBA::ULong& len)	encodeUShortArray(CORBA::UShort*& s, CORBA::ULong len) decodeUShortArray(CORBA::UShort*& s, CORBA::ULong& len)
	encodeLongArray(CORBA::Long*& s, CORBA::ULong len) decodeLongArray(CORBA::Long*& s, CORBA::ULong& len)	encodeLongArray(CORBA::Long*& s, CORBA::ULong len) decodeLongArray(CORBA::Long*& s, CORBA::ULong& len)
	encodeULongArray(CORBA::ULong*& s, CORBA::ULong len) decodeULongArray(CORBA::ULong*& s, CORBA::ULong& len)	encodeULongArray(CORBA::ULong*& s, CORBA::ULong len) decodeULongArray(CORBA::ULong*& s, CORBA::ULong& len)
	encodeFloatArray(CORBA::Float*& s, CORBA::ULong len) decodeFloatArray(CORBA::Float*& s, CORBA::ULong& len)	encodeFloatArray(CORBA::Float*& s, CORBA::ULong len) decodeFloatArray(CORBA::Float*& s, CORBA::ULong& len)
	encodeDoubleArray(CORBA::Double*& s, CORBA::ULong len) decodeDoubleArray(CORBA::Double*& s, CORBA::ULong& len)	encodeDoubleArray(CORBA::Double*& s, CORBA::ULong len) decodeDoubleArray(CORBA::Double*& s, CORBA::ULong& len)
	encodeOctetArray(CORBA::Octet*& s, CORBA::ULong len) decodeOctetArray(CORBA::Octet*& s, CORBA::ULong& len)	encodeOctetArray(CORBA::Octet*& s, CORBA::ULong len) decodeOctetArray(CORBA::Octet*& s, CORBA::ULong& len)
	encodeBooleanArray(CORBA::Boolean*& s, CORBA::ULong len) decodeBooleanArray(CORBA::Boolean*& s, CORBA::ULong& len)	encodeBooleanArray(CORBA::Boolean*& s, CORBA::ULong len) decodeBooleanArray(CORBA::Boolean*& s, CORBA::ULong& len)

Figure 3-31 Marshalling and demarshalling functions of string and array in the flyover

Class Message also provides the encodeArray and decodeArray functions to (de)marshal the CORBA array. Suppose one of the input parameters is long[10]. On the client side, the client stub will invoke the corresponding marshalling function shown

as follows:

```
//initialize a message object
Message IT_r();
CORBA::ULong IT_len = 10;
CORBA::Long* IT_arr = (CORBA::Long*) a;
//marshal the array into the message object
IT_r.encodeLongArray (IT_arr, IT_len);
```

Figure 3-32 Example of marshalling CORBA array in the flyover

The details of marshalling and demarshalling of CORBA struct and union are explained in Appendix A.1 and Appendix A.2 respectively. The details of supporting the interface inheritance in the flyover is described in Appendix A.3.

3.9 Implementation Issues

3.9.1 Implementation of IDLP and the Flyover

3.9.1.1 Implementation of IDLP

The construction of the flyover in system is based on the modification of stub and skeleton codes. Interface Definition Language Processor (IDLP) is responsible for all the source code modification on the IDL specification files, client stub codes and server skeleton codes, and calling the Orbix IDL compiler as well. Because IDLP should analyze the complicated structure of the IDL syntax, client stub codes and server skeleton codes, Perl is chosen to implement the IDLP. Perl is a scripting language that provides flexible and powerful regular expression manipulation: for example, string matching and substitution [51].

Figure 3-33 shows the structure of IDLP that has been briefly described in Section 3.4. The IDLP consists of two main components: the IDL parser and the C++ parser. The IDL parser is responsible for analyzing the IDL interfaces defined in the IDL interface specification file and modifying this specification file. The C++ parser is responsible for

analyzing the structure of the header file, client stub and server skeleton generated by the IDL compiler and modifying them to incorporate the flyover into the CORBA applications.

Figure 3-33 Structure of IDLP

There is a data structure in IDLP shared by both the IDL parser and C++ parser. This data structure stores the information of the IDL interface specification file. For example, it stores the interface names and their interface functions defined in the source IDL specification file. This information is useful in analyzing the structures of the client stub, server skeleton and header file. Figure 3-34 and Figure 3-35 show the operations performed by the IDL parser and the C++ parser inside the IDLP respectively.

Figure 3-34 Block diagram of the operations performed by IDL parser inside IDLP

Figure 3-35 Block diagram of the operations performed by C++ parser inside IDLP

3.9.1.2 Implementation of the Flyover Prototype

The prototype of the flyover has been implemented using C++ and Orbix-MT 2.3 running on Solaris 7. The hash table is implemented by using the STL library [52]. The flyover is compiled as a run-time library using Sun CC v.4.0 compiler linked to the client and server applications.

There are five important steps in the development of the flyover: planning, design, implementation, testing and performance measurement. The first step is to plan what features should be supported in the flyover. The second and third steps are the design and implementation of the flyover which are fully discussed in Section 3.7 and Section 3.8.

The fourth step is the testing procedure, which involves testing the correctness of the functionality. For examples, to test the correctness of transferring a CORBA sequence of long integers through the flyover, the value of each element of the sequence is checked at the server side to make sure that it matches the arguments transmitted by the client. After making sure that the implementation function properly, performance measurements of the flyover is made. The description of the performance measurements and results are described in Chapter 4 and Chapter 5.

3.9.2 Switching Between CORBA and the Flyover at Run Time

The client is allowed to choose between passing the request either through the flyover path or the ORB at run time by including the function call `BypassORB::Flyover(FLYOVER_ENABLE)` just before the method invocation. If the source IDL interface specification is not supported by the flyover, the statement above has no effect because the IDLP does not generate the corresponding codes in the client stub and the server.skeleton to pass the request through the flyover path. In the first prototype of the

flyover, there is no warning message generated during IDLP processing. However, support for this is worthy of future research.

The flyover allows the client to select the path at run time because it keeps the original marshalling and demarshalling routine in the original stub and skeleton codes generated by Orbix IDL compiler. Therefore if the client/server pair exhibits heterogeneity in hardware and software platforms, they can communicate through the original stubs and skeletons.

3.9.3 Limitations of the Flyover

CORBA standard defines different types of system exceptions. For instance, the ORB generates COMM_FAILURE if the client cannot communicate with the server. However, in the flyover, there is no specific system exception. If any error occurs, it will generate the system exception UNKNOWN (identifier 10000). The main reason of using one system exception for all errors is to simplify the design and implementation of the flyover prototype. However, in later versions, specific system exceptions can be raised if different errors occur.

Moreover, the first prototype of flyover only supports request and reply messages because over 90% of CORBA method calls are those kinds of messages. Supporting other message types are worthy of future work.

Besides the simplification of system errors and message types in the flyover, the request-level interceptor (filter in Orbix) and message-level interceptor (request transformer in Orbix) are not supported in the flyover.

Chapter 4 Performance Measurement and Instrumentation

4.1 Experiments

A prototype of the flyover has been implemented based on Orbix-MT 2.3. As described in Chapter 3, the flyover employs different optimization techniques to minimize the latency on CORBA-based systems with limited heterogeneity. This section describes the experiments to measure the performance of the flyover. The performance of the flyover prototype is compared with that of Orbix-MT 2.3 and Orbix 2000. Five types of experiments were conducted to measure the mean response time and the system throughput of a two-way CORBA method and are described in the following subsections.

4.1.1 Single Client/Server System under No Load Condition

This experiment measures the response time of a two-way CORBA method call under a no load condition. In this system, a single client interacts with a single server and there is no resource contention from other system components. To model this system, each CORBA method call in the client program is separated by a significant amount of think time (300 milliseconds) as compared to the mean response time of the CORBA function call. Therefore there is no queuing delay for each request. The following is the client pseudo code to measure the response time of a two-way CORBA call.

```
#define num_of_cycles 1000
time_t starttime, stoptime;
for i := 1 to num_of_cycles {
    starttime = gethrtime(); //start the high resolution timer (Section 4.5.3)
    objA->memfun(data);      //call the remote CORBA interface
    stoptime = gethrtime();  //stop the timer
    response_time = stoptime - starttime;
    usleep(300000);          //sleep 300000 microseconds (think time)
}
```

Figure 4-1 Pseudo code to measure the response time under no load condition

4.1.2 Single Client/Server System with Poisson Arrival of Requests

Since most distributed systems are running at partial or full capacity, the system with no workload condition may not completely describe the performance of the flyover in the real systems. Consider, for example, a system in which requests arrive from the end users at the client node. The client receives the request, and if free, invokes the server method immediately; otherwise the request is queued. This experiment uses an open model with a single client and a single server. Requests arrive at the client following a Poisson process (see Figure 4-2). In the experiment, the client program consists of two threads of control. One client thread, called the request thread, generates requests with exponentially distributed inter-arrival times. Once a request is generated it is put in a FIFO queue with a time stamp recording the time of arrival. The other client thread, called the execution thread, gets the request from the FIFO queue and then invokes the test CORBA method. If the reply arrives from the server, the client execution thread measures the time of message reception and repeats the process. The response time of the method call is measured by taking the time difference between the time of message reception and the arrival time. If the queue is empty, the execution thread waits on the FIFO queue until there is at least one request.

Figure 4-2 Single client-server system with Poisson arrival of requests

4.1.3 Modelling Client-Server Systems with Limited Heterogeneity

This experiment simulates systems with limited heterogeneity. Recall from Section 3.1 that distributed systems with limited heterogeneity contains a subset of similar communicating nodes. Figure 4-3 depicts the experiment of modelling systems with limited heterogeneity.

Figure 4-3 Model of client-server systems with limited heterogeneity

A simple model is used for obtaining insights into system performance. Request generation of this model is similar to the client-server systems described in Section 4.1.2. A separate client thread generates requests with exponentially distributed inter-arrival times and the execution thread picks up the request and makes a remote CORBA call to the server. However, in this kind of system the client passes the request with probability $1-p$ through the flyover to server A. The request is passed through the ORB to server B with a probability of p . The parameter p is called the degree of heterogeneity. If p is equal to zero, the system is totally homogeneous. If p is equal to 1 the system is totally heterogeneous. The degree of heterogeneity in a system depends both on the hardware and software used in construction of the various nodes as well as the traffic distribution.

In the experiments, server process A and B are located on two different workstations but with the same system configurations.

4.1.4 Measurement of Binding Time

Since the flyover, Orbix-MT 2.3 and Orbix 2000 are H-ORB systems, the client must get the object reference first before invoking the remote CORBA method. The elapsed time in getting the object reference is called the binding time. Measurement of the binding time is important because the flyover takes one extra remote CORBA call to get the object ID and port number (see Section 3.7.2).

4.1.5 Measurement of System Throughput

This experiment measures the system throughputs of the amount of data (not include the GIOP header) sending through the ORBs or the flyover in one second. In this experiment, there is no think time. The client invokes the interface method immediately after the reply comes back from the server. Figure 4-4 is the client pseudo code to measure the system throughput.

```
#define num_of_cycles 1000
time_t starttime, stoptime;
starttime = gethrtime(); //start the high resolution timer (Section 4.5.3)
for i := 1 to num_of_cycles {
    objA->memfun(data);    //call the remote CORBA interface
}
stoptime = gethrtime(); //stop the timer
total_time = stoptime-starttime;
throughput = sizeof(data)*num_of_cycles/total_time;
```

Figure 4-4 Pseudo code to measure the system throughput

4.2 Parameters

This section describes the parameters specifying the characteristics of the applications and the systems.

4.2.1 Data Types

As discussed in Section 3.3.1, the flyover supports a large set of IDL data types

including primitive and constructed data types and pseudo-objects as well. Table 4-1 shows all the IDL parameters that are used in the experiments used for evaluation of the flyover prototype. The primitive data types for performance evaluation include char, octet, short, long, float, double and boolean. The constructed data types include bounded and unbounded sequence, bounded and unbounded string, array, struct and enum. The experiments on pseudo-object type include passing an object reference. Moreover, measurement of response time of CORBA sequences and arrays includes all primitive data types (e.g. sequence<char>, sequence<octet> and so forth).

Table 4-1 Testing IDL Parameters

	Parameters	Length
Primitive data type:	char, octet, short, long, float, double, boolean	
Constructed data type:	bounded and unbounded sequence (sequence<char>, sequence<octet>, sequence<short>, sequence<long>, sequence<float>, sequence<double>, sequence<boolean>)	1, 32, 256, 1K, 2K, 4K, 8K, 16K, 32K, 64K, 128K
	bounded and unbounded string	1, 32, 256, 1K, 2K, 4K, 8K, 16K, 32K, 64K, 128K
	array (array<char>, array<octet>, array<short>, array<long>, array<long>, array<float>, array<double>, array<boolean>)	1, 32, 256, 1K, 2K, 4K, 8K, 16K, 32K, 64K, 128K
	struct	
	enum	
Pseudo-Object type:	object reference	

In the performance measurement for different IDL data types, the method is involved irrespective of the argument passed. The client sends the request to the server and the server returns a reply message (void) back to the client immediately.

4.2.2 Data Size

Measurement of response time of the IDL sequence and array also includes different

sequence lengths and array sizes. The range of sequence length or array size is from 1 to 128K as shown in Table 4-1. Varying the data sizes measures the system latency in transferring small and large messages.

4.2.3 Arrival Rate of Requests

This is the rate of requests that arrive at the client side with Poisson distribution. The rate is expressed in number of requests arriving per second. This parameter is used for investigation of system performance with different system loads. This parameter is used in the second and the third set of experiments (see Sections 4.1.2 and 4.1.3).

4.2.4 Degree of Heterogeneity (p)

This is the probability p that the client passes the request through the ORB to the server. In other words, the request is sent through the flyover with a probability $1 - p$. The value of p is between 0 and 1. The main purpose of this parameter is to investigate the system performance by varying the percentage of the number of similar communicating nodes in the systems. This parameter is used in the third set of experiment only (see Section 4.1.3).

4.3 Automation of Performance Measurement

Performance measurement is a time-consuming task because the experiment has to be repeated many times to get an accurate result. Moreover, there are so many combinations of the IDL parameters listed in Table 4-1. For example, the measurement of a bounded sequence or array has 7×11 combinations (numbers of primitive types $\times$ numbers of sequence lengths). As well, the same experiment should be run on different ORBs in order to compare performance. Therefore, it is more convenient to have a shell script to

automate all compilation and performance measurement.

The shell script executes in two steps: 1) changes the size and data type of the IDL source interface using the Perl script and 2) recompile and execute the CORBA test programs. The shell script repeats the above two steps until it finishes all the combinations.

4.4 Experimental Environment

Experiments are performed on the “Quiet” Network located in the Real-Time and Distributed Systems (RADS) Lab. The Quiet Network comprises a set of 40 MHz Sparc-2 workstations connected by a 10Mbps Ethernet. Each Sparc-2 workstation contains 32MB RAM and is running under Solaris 7. The Maximum Transmission Unit (MTU) on the Ethernet adapter is 1500 bytes. The Orbix broker is located in a separate 200MHz Ultra-1 workstation.

4.5 Performance Metrics

To measure the flyover performance, the mean response time and the mean throughput of a two-way CORBA method call are used as the performance metrics which measure the latency and capacity of the systems respectively.

4.5.1 Response Time

The client response time is the elapsed time measured in nanosecond (ns) from the moment the request is generated at the client to the time the reply arrives from the server. In the client-server systems with Poisson arrival of requests, the response time includes the queuing delay. The mean response time is a measure of the latency of the system.

In a coarse-grained view, the response time of a CORBA method invocation

comprises of queuing delay, computation time and communication time. The queuing delay depends on the arrival rate and the service rate whereas the communication time depends on the transfer rate of the underlying networks and the size of GIOP message. In a fine-grained view, the response time of a request-reply CORBA transaction can be further divided into twelve parts shown in Figure 4-5. Measurement of each component provides an insight of which portion of end-to-end response time is spending on marshalling, demarshalling or request demultiplexing. It also allows measurement and comparison of effectiveness of the optimization techniques applied in the flyover. For example, preventing CDR conversion and optimized GIOP may improve the marshalling/demarshalling time and the communication time respectively.

To measure the delay of those twelve components, the starting time and ending time of each component are measured. Figure 4-5 also shows the points of time measurement.

Figure 4-5 Time measurement points in CORBA request-reply transaction

Getting the time at the measurement points in the flyover can be achieved by inserting calls to the timing functions in the source codes of the flyover. However, to compare the performance with Orbix-MT 2.3, the Interceptor (described in Section 2.3.11) must be

used to intercept the messages and to measure the time at the intercepting point, since the ORB source code is not available.

Orbix-MT 2.3 provides eight request-level interceptors called *filters* and four message-level interceptors called *request transformers*. A filter allows modification on the structured request just before and after (de)marshalling, whereas the request transformer allows modification on a message buffer immediately before and after transmission across the network. Table 4-2 shows the different types of filters and request transformers and their corresponding intercepting points on a CORBA request-reply transaction.

Table 4-2 Interceptors defined in Orbix-MT 2.3 for performance measurement

Intercepting Points	Interceptors
B	out request pre marshal
C	out request post marshal
D	send request transformer
E	receive request transformer
F	in request pre marshal
G	in request post marshal
H	out reply pre marshal
I	out reply post marshal
J	send reply transformer
K	receive reply transformer
L	in reply pre marshal
M	in reply post marshal

4.5.2 Mean System Throughput

The average system throughput is measured in megabits per second (Mbps). It measures the amount of data (not include the GIOP header) sending through the ORBs or the flyover in one second. The throughput is a measurement of the capacity of the system.

4.5.3 Systems Calls for Elapsed Time Measurement

There are several UNIX function calls to measure the elapsed time:

- 1) `gethrtime()` [56]: This function returns the current high-resolution real time (as a long

long integer) expressed as nanoseconds (ns) since some arbitrary time in the past. Because this high-resolution timer provides a cheap and accurate interval timing, it is ideally suited to performance measurement. This high resolution timer is used for performance measurement on the flyover by putting it in the time measurement points A to M in Figure 4-5. The CPU time consumed by this system call is approximately 200 ns. This is very small in comparison to the measured response times that are of the order of milliseconds.

- 2) `gettimeofday()` [56]: This function gets the system's current time which is expressed in elapsed seconds and microseconds since 00:00 Universal Coordinated Time (UCT), January 1, 1970. Since this function is not able to get a high-resolution time (the unit is in microseconds) and calculating the time difference for each component requires manipulation of the `timeval` structure, this timer is not used in the performance measurement on the flyover.
- 3) `clock()` [56]: This function gets the amount of CPU time (in microseconds) used since the first call to `clock()` in the calling process. Similar to the drawback of `gettimeofday()` function, it does not provide a high-resolution time. Therefore it is not used in the performance measurement on the flyover.

4.6 Accuracy of Measured Data

Although the computing nodes in the Quiet Network are dedicated for running the experiments, the network was shared with other machines in the lab. Therefore all experiments were performed during night time to minimize interference in network traffic from these other machines. Moreover, one simple test program is written to verify

the impact of the background network traffic and the stability of the measurement network.

Before running the experiment, the network speed for TCP/IP traffic was measured by using a UNIX system command known as `netstat -i` [56]. This command displays how many input and output packets are used in a user specified interval on the workstation. The measurement of network traffic in the experimental workstations before running the experiment indicated that the average numbers of input packets and output packets per second on the client workstation were 27.6 and 2.3 respectively. This corresponds to the background activities. To use the full capacity of the 10 Mbps network, a test program on the client workstation was continuously sending a packet of size 1024 bytes to the server. After receiving the packet, the server sent back a 1-byte reply to the client. The empirical result shows that the number of input and output packets per second on the client workstation jumped to 897 and 756 respectively when the test program was running. Although the `netstat` command does not report the packet size, one can see that the background traffic only constitutes 3.08 % of the total number of packets when the test program was running. In other words, the background traffic has very little impact on the performance result. Moreover, each experiment consists in repeating client cycles. The number of client cycles is chosen to be large enough to obtain a confidence interval less than or equal to $\pm 4\%$ at a confidence level of 99%.

Chapter 5 Experimental Results

In this chapter, the results of a number of experiments described in Chapter 4 are presented. First, the performance results for single client/server system under no load condition are presented in Section 5.1. Second, the result of measurement of binding time is discussed in Section 5.2. Third, Section 5.3 describes the performance result for client/server systems with Poisson arrival of requests. Fourth, Section 5.4 presents the performance characteristics for systems with limited heterogeneity. Finally, Section 5.5 summarizes the performance characteristics of the flyover. Throughout this chapter, time measurements are rounded to the nearest microsecond (μ s).

5.1 Single Client/Server System under No Load Condition

This experiment measures the response time of a two-way CORBA interface method call under no load condition, which is described in Section 4.1.1. The IDL data types used in this experiment are described in the following order: array, primitive types, string, sequence, struct, object reference and enumeration.

5.1.1 CORBA Array

Table 5-1 to Table 5-4 and Figure 5-1 to Figure 5-4 show the performance comparison of passing CORBA array of long, octet, char and double respectively. The figures show that the performance improvement achieved by the flyover does not change significantly with the data type. The array sizes in the experiments are chosen in such a way that the performance of the flyover can be compared with other systems for a range of response times (see Figure 5-1 to Figure 5-4). The array sizes used in the experiment

vary from 1 to 128K. As shown in Table 5-1, the flyover illustrates a better performance over both Orbix-MT 2.3 and Orbix 2000 in sending an array of long. For example, in transferring an array `long[32]`, the flyover achieves 73.88% and 49.30% improvements over the Orbix-MT 2.3 and Orbix 2000 respectively. Similar performance gains can also be obtained in passing the arrays with other data types. For example, passing `octet[32]` gives rise to 75.03% and 51.63% improvements (Table 5-2) while passing `double[32]` obtains 72.92% and 51.33% improvements (Table 5-4).

A significant performance gain can be obtained if the flyover is based on UDP transport. This time saving stems from skipping the establishment and termination of connection, sequencing of out-of-order packet, end-to-end reliability (e.g. checksums, positive acknowledgements and timeout) and the end-to-end flow control in TCP, which provides connection-oriented reliable transport. Note that UDP may be used on systems in which probability of a message loss is very low and end-to-end acknowledgement is provided in the application layer.

Table 5-1 Response time measurement for CORBA array of long

Array Size	Orbix-MT 2.3 (μ s)	Orbix 2000 (μ s)	Flyover (TCP) (μ s)	Flyover (UDP) (μ s)	Improvement Flyover over Orbix-MT 2.3	Improvement Flyover over Orbix 2000	Improvement Flyover TCP over UDP
1	16401	8430	3762	3010	77.06%	55.37%	19.99%
32	16453	8475	4297	3243	73.88%	49.30%	24.53%
256	17780	9937	5375	4435	69.77%	45.91%	17.49%
1K	21408	13534	8882	8601	58.51%	34.37%	3.16%
2K	24680	17501	12849	12371	47.94%	26.58%	3.72%
4K	34843	28135	22842	22297	34.44%	18.81%	2.39%
8K	54436	45857	41346	40898	24.05%	9.84%	1.08%
16K	99634	93721	88731	-	10.94%	5.32%	-
32K	183504	176321	172601	-	5.94%	2.11%	-

Table 5-2 Response time measurement for CORBA array of octet

Array Size	Orbix-MT 2.3 (μ s)	Orbix 2000 (μ s)	Flyover (TCP) (μ s)	Flyover (UDP) (μ s)	Improvement Flyover over Orbix-MT 2.3	Improvement Flyover over Orbix 2000	Improvement Flyover TCP over UDP
1	16126	8210	3740	2985	76.81%	54.45%	20.19%
32	16222	8375	4051	3123	75.03%	51.63%	22.91%
256	17042	8792	4498	3484	73.61%	48.84%	22.54%
1K	17620	10046	5294	4397	69.95%	47.30%	16.94%
2K	19493	11275	6989	5874	64.15%	38.01%	15.95%
4K	21198	13486	8783	8389	58.57%	34.87%	4.49%
8K	24803	17387	12661	12210	48.95%	27.18%	3.56%
16K	34377	28520	22682	22107	34.02%	20.47%	2.54%
32K	53778	46075	41847	40396	22.19%	9.18%	3.47%
64K	97826	93175	89441	-	8.57%	4.01%	-
128K	181658	175456	172074	-	5.28%	1.93%	-

Table 5-3 Response time measurement for CORBA array of char

Array size	Orbix-MT 2.3 (μ s)	Orbix 2000 (μ s)	Flyover (TCP) (μ s)	Flyover (UDP) (μ s)	Improvement Flyover over Orbix-MT 2.3	Improvement Flyover over Orbix 2000	Improvement Flyover TCP over UDP
1	16281	8776	3783	3067	76.76%	56.89%	18.93%
32	16355	8981	3799	3097	76.77%	57.70%	18.48%
256	16734	9012	4469	3347	73.29%	50.41%	25.11%
1K	17849	10051	5476	4391	69.32%	45.52%	19.81%
2K	20015	11235	7088	5862	64.59%	36.91%	17.30%
4K	21393	13340	8978	8361	58.03%	32.70%	6.87%
8K	25043	17482	12674	12165	49.39%	27.50%	4.02%
16K	35119	28547	22732	22128	35.27%	20.37%	2.66%
32K	55252	46298	42348	41859	23.35%	8.53%	1.15%
64K	101582	91325	88894	-	12.49%	2.66%	-
128K	186041	174456	172229	-	7.42%	1.28%	-

Table 5-4 Response time measurement for CORBA array of double

Array size	Orbix-MT 2.3 (μ s)	Orbix 2000 (μ s)	Flyover (TCP) (μ s)	Flyover (UDP) (μ s)	Improvement Flyover over Orbix-MT 2.3	Improvement Flyover over Orbix 2000	Improvement Flyover TCP over UDP
1	16540	9126	3780	2972	77.47%	59.16%	20.26%
32	16597	9234	4494	3331	72.92%	51.33%	25.88%
256	19818	11294	6993	5944	64.71%	38.08%	15.00%
1K	24748	18050	12739	12258	48.53%	29.42%	3.78%
2K	34216	28498	22674	22210	33.73%	20.44%	2.05%
4K	53389	46086	42434	41897	20.52%	7.92%	1.27%
8K	99392	92101	88837	-	10.62%	3.54%	-
16K	182395	178568	172833	-	5.24%	3.21%	-

Figure 5-1 Response time measurement for CORBA array of long

Figure 5-2 Response time measurement for CORBA array of octet

Figure 5-3 Response time measurement for CORBA array of char

Figure 5-4 Response time measurement for CORBA array of double

As shown in Table 5-1, the performance gain is less when a larger array is passing

through the flyover. For example, sending an array `long[16384]` can only obtain 10.94% and 5.32% performance gains over Orbix-MT 2.3 and Orbix 2000 respectively (Table 5-1). The major reason for this is that the communication (transmission) time becomes the dominant component in the end-to-end response time when the message size increases.

Table 5-5 and Table 5-6 present the average distribution of times in a request-reply transaction corresponding to an array transfer for the Orbix-MT 2.3 and the flyover. Since the return type of the test function is void, there is no “Reply Demarshalling” and “Reply Marshalling” in Table 5-5 and Table 5-6. Note that the communication (transmission) time includes the elapsed time in sending both request and reply over the network.

Table 5-5 Components of response time in a CORBA transaction of long[32]

	Orbix-MT 2.3 (μ s)	Flyover (TCP) (μ s)	Flyover (UDP) (μ s)
Out Request Marshalling	493	331	328
Out Request Processing	1485	168	171
In Reply Processing	2334	317	315
Communication time	4875	3138	2086
In Request Dispatching	6434	7.45	7.44
In Request Demarshalling	483	322	322
Out Reply Processing	349	14	13.8
Total	16453	4297	3243

Table 5-6 Components of response time in a CORBA transaction of long[8192]

	Orbix-MT 2.3 (μ s)	Flyover (TCP) (μ s)	Flyover (UDP) (μ s)
Out Request Marshalling	3856	3057	3069
Out Request Processing	1364	158	150
In Reply Processing	2598	385	386
Communication time	35992	34532	34112
In Request Dispatching	6466	9	9.12
In Request Demarshalling	3729	3160	3156
Out Reply Processing	431	16.35	16.23
Total	54436	41346	40898

From Table 5-5 and Table 5-6, one can see that execution times of “Out Request Marshalling” and “In Request Demarshalling” in the flyover are reduced because of preventing CDR conversions that occur in GIOP. The marshalling/demarshalling overheads vary with the numbers of bytes in the messages. For example, the time consumed in demarshalling the input parameter `long[32]` and `long[8192]` on Orbix-MT 2.3 are 483 μ s and 3279 μ s respectively. This time difference in demarshalling is because of the larger data size being transferred.

Table 5-5 and Table 5-6 also show the improvement in communication time as a result of using the optimized GIOP and removing padding bytes. Optimized GIOP reduces the sizes of GIOP message header and request header (Section 3.7.4.3) while removing padding bytes minimizes the size of the request and reply (Section 3.7.4.2). For example, there is a significant improvement in communication time (35.63%) in transferring an array `long[32]` in the flyover (TCP) (3138 μ s) as compared to Orbix-MT 2.3 (4875 μ s) because the total sizes of message header and request header are reduced from 105 bytes to 17 bytes (see Section 3.7.4.3).

However, as the size of the array increases, the time saving in message header and request header is amortized by the large data size being transferred. Therefore, the performance improvement in communication time is only 4.06% in transferring an array `long[8192]` (see Table 5-6).

Using UDP transport instead of TCP in the flyover can further reduce communication time because UDP is a connectionless protocol. For example, as shown in Table 5-5, transferring a small message (e.g. `long[32]`) in “flyover UDP” (2086 μ s) can improve 33.5% of the communication time over the “flyover TCP” (3138 μ s). This improvement

primarily occurs because of the time saving in bypassing the establishment and termination of connection, packet sequencing, end-to-end reliability and flow control. An early paper on network performance measurement shows that the maximum throughput of UDP in an Ethernet environment is double than that of TCP [7]. Note that for using UDP, some form of end-to-end acknowledgement may be required at a higher layer.

It is important to note that in the flyover, there is a upper bound on message size when UDP is used because the maximum IP packet size is 65535 bytes, and the maximum size of IP header and UDP header is 60 bytes and 8 bytes respectively [55] [57]. The maximum payload for UDP datagram is 65467 bytes (assume maximum IP header). If the user transfers a GIOP message larger than this value, the flyover raises a system-defined exception UNKNOWN. For example, transferring a message of `long[16384]` is not allowed in the flyover on UDP, because the total size of the message parameter and the optimized GIOP request header is greater than 65467 bytes. Note that if the data size is larger than the upper bound of UDP diagram size when the message is transferred through the flyover UDP, the response time measurement is marked with “-” as shown in Table 5-1 to Table 5-4.

The communication time of using UDP is close to that of using TCP when the message size gets larger. For example, from Table 5-6, transferring a large message `long[8192]` in flyover UDP (34112 μ s) can obtain only 1.2% improvements in communication time over the flyover TCP (34532 μ s).

The overhead of request dispatching in the flyover is negligible compared to the response time. It only constitutes less than 0.02% of the response time in sending an array `long[8192]` (see Table 5-6). This time improvement in request demultiplexing and

operation dispatching occurs because the object reference can be retrieved from the hash table directly according to the object ID in the message object. As well, dispatching the request to the corresponding member function of the object implementation is based on comparing the operation IDs (see Section 3.7.3.1), instead of the operation names, which is known to be computationally expensive.

Figure 5-5 shows the throughput analysis of transferring a CORBA octet array in the throughput experiment described in Section 4.1.5. The label “C” is the performance of C sockets over the Ethernet. The performance of C sockets (two-way call) peaks for a 4K array size. When the array size exceeds 4K, performance starts leveling off at around 7.83 Mbps with 32K data buffers. One of the factors causing this behavior is the MTU (Maximum Transmission Unit) size of the Ethernet network, which is 1500 bytes. When the array size exceeds the MTU size, they are fragmented at the sender side and assembled at the receiver side. Therefore, these extra overheads level off the throughput.

Figure 5-5 Throughput of transferring CORBA octet array

The throughputs of flyover, Orbix-MT 2.3 and Orbix 2000 increase as the array size

increases. Moreover, the throughput of the flyover is higher than that of Orbix-MT 2.3 and Orbix 2000 for different array sizes and is closer to the maximum throughput of the TCP in sending a large array. It is because the overheads of (de)marshalling and request demultiplexing are lower in the flyover as compared to those of Orbix-MT 2.3 and Orbix 2000. This high throughput is useful for data intensive and streaming applications.

The performance results of the CORBA array with other data types (e.g. short, float, and boolean) are shown in Appendix B.1.

5.1.2 CORBA Primitive Types

In this experiment, only one primitive type (e.g. one long integer) is passed as an argument for a method invocation. Table 5-7 depicts the performance comparisons of passing CORBA primitive data types. As shown in Table 5-7, the flyover prototype demonstrates a performance improvement from the lowest 54.25% (long) to the highest 56.32% (double) as compared to Orbix 2000 because of the optimizations applied on the flyover.

Table 5-7 Response time measurement for CORBA primitive types

Primitive data types	Orbix-MT 2.3 (μs)	Orbix 2000 (μs)	Flyover (TCP) (μs)	Flyover (UDP) (μs)	Improvement Flyover over Orbix-MT 2.3	Improvement Flyover over Orbix 2000	Improvement Flyover TCP over UDP
long	16656	8442	3862	3010	76.81%	54.25%	22.06%
octet	16425	8210	3740	2985	77.23%	54.45%	20.19%
char	16456	8376	3720	2960	77.39%	55.59%	20.43%
boolean	16430	8280	3743	2981	77.22%	54.79%	20.36%
short	16667	8465	3835	3015	76.99%	54.70%	21.38%
float	16728	8547	3897	3026	76.70%	54.41%	22.35%
double	16828	8930	3901	3034	76.82%	56.32%	22.23%

This significant improvement is beneficial when short control messages are used for achieving synchronization among different system components.

5.1.3 CORBA String

Table 5-8 and Figure 5-6 show the performance result achieved with a CORBA string. Because the marshalling/demarshalling function of string is similar to the array of char, the response time of passing the string is close to that of an array of char with the same length (Figure 5-3).

Table 5-8 Response time measurement for CORBA string

CORBA String	Orbix-MT 2.3 (μ s)	Orbix 2000 (μ s)	Flyover (TCP) (μ s)	Flyover (UDP) (μ s)	Improvement Flyover over Orbix-MT 2.3	Improvement Flyover over Orbix 2000	Improvement Flyover TCP over UDP
1	16440	8652	3776	3089	77.03%	56.36%	18.19%
32	16572	8692	3812	3121	77.00%	56.14%	18.13%
256	16827	8760	4489	3398	73.32%	48.76%	24.30%
1K	17885	9971	5531	4378	69.07%	44.53%	20.85%
2K	20135	11338	7056	5883	64.96%	37.77%	16.62%
4K	20868	13603	8898	8389	57.36%	34.59%	5.72%
8K	25544	18002	12653	12130	50.47%	29.71%	4.13%
16K	35216	29296	22586	22215	35.86%	22.90%	1.64%
32K	55439	48249	42269	41934	23.76%	12.39%	0.79%
64K	102992	90325	88798	-	13.78%	1.69%	-
128K	187369	173456	172310	-	8.04%	0.66%	-

Figure 5-6 Response time measurement for CORBA string

5.1.4 CORBA Sequence

Since Orbix-MT 2.3 gives an unstable result in passing a lengthy sequence, all performance comparisons for CORBA IDL sequence are performed with Orbix-2000, “flyover TCP” and “flyover UDP” only. Figure 5-7 and Figure 5-8 illustrate the performance comparisons of octet and long sequence. All other response time measurements for CORBA sequences are shown in Appendix B.2. As expected, a significant performance improvement is achieved by using a flyover.

Figure 5-7 Response time measurement for CORBA sequence of octet

Figure 5-8 Response time measurement for CORBA sequence of long

5.1.5 CORBA Struct

Figure 5-9 demonstrates the performance result of passing the following CORBA structure.

```
struct cd {
    char c;
    double d;
}
```

As shown in, the flyover also achieves significant improvements in response time over Orbix-MT 2.3 (77.08%) and Orbix 2000 (55.77%).

Table 5-9 Response time measurement for CORBA struct

Orbix-MT 2.3 (μs)	Orbix 2000 (μs)	Flyover (TCP) (μs)	Flyover (UDP) (μs)	Improvement Flyover over Orbix-MT 2.3	Improvement Flyover over Orbix 2000	Improvement Flyover TCP over UDP
16485	8542	3778	3041	77.08%	55.77%	19.51%

Figure 5-9 Response time measurement for CORBA struct

5.1.6 CORBA Object Reference

Figure 5-10 and Table 5-10 illustrate the experimental result of passing an object reference. Recall the discussion in Section 3.8.1 that passing an object reference in the flyover includes passing the target port number as well as the object ID of the target object. In addition, the server skeleton is required to demarshal and extract the port number and object ID from the request object and then explicitly create the socket path to the target object. Table 5-11 shows the average distribution of the extra overheads of the marshalling and demarshalling routines, and the establishment of a socket connection in passing an object reference in the flyover.

Table 5-10 Response time measurement for CORBA object reference

Orbix-MT 2.3 (μ s)	Orbix 2000 (μ s)	Flyover (TCP) (μ s)	flyover (UDP) (μ s)	Improvement Flyover over Orbix-MT 2.3	Improvement Flyover over Orbix 2000	Improvement Flyover TCP over UDP
20259	18065	17816	15172	12.06%	1.38%	14.84%

Table 5-11 Extra overheads in passing object reference through the flyover (TCP)

Execution time (μ s)	
Marshal the port number and the IP address	23
Demarshal the port number and the IP address	20
Extra Communication time	38
Establish the socket connection	7454
Total Extra Overhead	7535

Figure 5-10 Response time measurement for CORBA object reference

Despite the extra overhead incurred, there are still performance improvements of 12.06% and 1.38% in passing object reference through the flyover because the savings from the optimizations (e.g. optimized GIOP header and efficient request demultiplexing) offset the extra overhead.

5.1.7 CORBA Enumeration

Figure 5-11 and Table 5-12 show the performance result of passing the CORBA enum. Since the IDL enum is mapped to a long integer, a similar performance pattern is observed as compared to response time of passing a long integer (Section 5.1.2).

Table 5-12 Performance measurement for CORBA object reference

Orbix-MT 2.3 (μs)	Orbix 2000 (μs)	Flyover (TCP) (μs)	Flyover (UDP) (μs)	Improvement Flyover over Orbix-MT 2.3	Improvement Flyover over Orbix 2000	Improvement Flyover TCP over UDP
16723	8432	3899	3020	76.68%	53.76%	22.54%

Figure 5-11 Response time measurement for CORBA enumeration

5.2 Measurement of Binding Time

This experiment measures the binding time of the flyover, Orbix-MT 2.3 and Orbix 2000 (see Section 4.1.4). Table 5-13 shows the time of binding the object reference to the target object implementation. The binding time measurement in Orbix-MT 2.3 and the flyover prototype are based on the Orbix-MT 2.3 proprietary bind function, while in Orbix 2000 the time measurement is based on de-stringifying an object reference because Orbix 2000 does not provide this bind function.

Table 5-13 Time overhead of object binding

Orbix-MT 2.3 (μs)	Orbix 2000 (μs)	Flyover TCP (μs)
2246	3017	28694

The substantial increase in binding time in the flyover is because the extra overhead of using one CORBA method call `GET__OBJKEY__` to obtain the port number and object ID of the target object, and establishing the socket connection (see Section 3.7.2.1). illustrates the average distribution of the overhead of object binding in the flyover. Calling the `GET__OBJKEY__` function and establishing the socket connection constitute the primary sources of overhead (66% and 26% respectively).

Table 5-14 Overhead of object binding in the flyover

Elapsed time (μ s)	
Call Orbix _bind() function	2230
Call GET_OBJKEY__()	19069
Establish the socket connection	7395
Total	28694

However, in many distributed systems, particularly telecommunication systems, the location of the object is rather static. Therefore, the increase in binding time can be amortized by the time saving of multiple CORBA method invocations on the same object reference in the flyover. For example, four CORBA calls which transfer a double through the flyover are enough to overcome the extra overhead of the object binding. The net time saving against the number of CORBA invocation is shown in Figure 5-12. Note that if access to ORB source code were available, the naming service program could be changed so that the extra call is avoided. This would lead to a substantial reduction in binding overhead.

Figure 5-12 The relationship between net time saving and the number of CORBA invocations

5.3 Single Client/Server System with Poisson Arrival of Requests

Experiments under no load condition were described earlier. The effect of system load controlled by adjusting the rate of request arrivals is considered next.

This experiment measures the response time of a two-way CORBA interface method

call with Poisson arrival of requests, which has been described in Section 4.1.2. Figure 5-13, Figure 5-14 and Figure 5-15 show the simulation results of transferring the arrays long[256], long[2048] and long[8192] on the client/server system with Poisson arrival of requests. As the arrival rate increases, the response times of both the flyover prototype and Orbix increase because some of the requests are put into the queue before processing by the client application. The figures show that the relative performance of the flyover with respect to Orbix improves substantially as the system load increases. For example, when the arrival rate is 4 requests per second in transferring long[8192] (Figure 5-15), flyover shows only 25.54 % and 9.39 % improvements over the Orbix-MT 2.3 and Orbix 2000 respectively. However, at a rate of 14 requests per second, the flyover demonstrates a 50.07% and a 18.41% improvements respectively. A similar trend can be expected with other kinds of data types.

Figure 5-13 Performance result of transferring an array long[256] on a single client/server system with Poisson arrival of requests

Figure 5-14 Performance result of transferring an array long[2048] on a single client/server system with Poisson arrival of requests

Figure 5-15 Performance result of transferring an array long[8192] on a single client/server system with Poisson arrival of requests

5.4 Client/Server Systems with Limited Heterogeneity

Figure 5-16 and Figure 5-17 show the simulation results of transferring the array `long[256]` and `long[8192]` by varying the degree of heterogeneity from 0 to 1. This experiment models the system with limited heterogeneity, which has been described in Section 4.1.3. When the degree of heterogeneity p is 1, the system is totally heterogeneous. The response time curves are similar to performance curves of Orbix-MT 2.3 in Figure 5-13 and Figure 5-15. However, p equal to 0 indicates the system is totally homogeneous. Therefore, it demonstrates a similar performance of passing parameters in the flyover (see Figure 5-13 and Figure 5-15).

Figure 5-16 Performance result of transferring an array `long[256]` on a system with limited heterogeneity

Figure 5-17 Performance result of transferring an array long[8192] on a system with limited heterogeneity

When p is between 0 and 1, the response time curve is between the lower bound (the flyover prototype) and the upper bound (Orbix-MT 2.3). As the degree of heterogeneity decreases, greater number of CORBA invocations are passing through the flyover. Hence the performance curve is shifting down and moving close to the curve of homogeneous systems.

This experiment shows that even when a small subset (e.g. degree of heterogeneity $p=80\%$) of similar communicating components exists within the systems, the overall system performance can still be improved. For example, in sending an array long[8192] at a rate of 14 requests per second, the flyover shows a 20.23% response time improvement when p is equal to 80% .

5.5 Summary of Experimental Results

A comprehensive experimental study on the relative performance of the Orbix-MT 2.3, Orbix 2000 and the flyover (on TCP and UDP transport) for different CORBA IDL data types and system load is presented in the previous sections. A summary of the experimental results is presented.

The performance comparison for IDL array, primitive type, string, sequence, struct, object reference and enum is covered in details in Sections 5.1.1 to 5.1.7. The flyover demonstrates a significant latency improvement in sending a short message. This can be attributed to the optimized GIOP, preventing CDR conversions, removing padding bytes, efficient request demultiplexing and dispatching, as well as using a direct socket communication. Table 5-5 and Table 5-6 show that the overhead of request demultiplexing in the flyover is very small in comparison to the response time. In addition, the marshalling and demarshalling overheads are 33 % less than the Orbix-MT 2.3 because of preventing CDR conversions.

The significant response time improvement in sending short messages in the flyover is beneficial if short control messages are used for achieving synchronization among different system components.

As message size increases, the communication time becomes the dominant component in the end-to-end response time. The overhead saving by the optimizations is amortized over the large data size being transferred in the flyover. Therefore, the performance improvement is less. However, it has been observed that the performance of the flyover is still higher than Orbix-MT 2.3 and Orbix 2000 for passing long arrays of character (7.42% and 1.28% respectively in passing an array `char [131072]`).

The throughput analysis of transferring a CORBA octet array demonstrates that the throughput of the flyover is higher than that of Orbix-MT 2.3 and Orbix 2000 for different array sizes. Moreover, as shown in Figure 5-5, when the message size increases, the throughput of the flyover is closer to the throughput of the TCP.

Passing an object reference in the flyover requires two more tasks: 1) marshalling and demarshalling of the port number and the Object ID; 2) establishing the socket connection to the target object. Although the extra overhead is incurred, passing an object reference is still efficient.

Before invoking a CORBA method on the target object, the client should obtain an object reference for the target object. The major drawback of the flyover is the substantial increase in the binding time. Since the flyover uses one extra CORBA method call to obtain the port number and the object ID, the binding time is more than the original Orbix-MT 2.3 binding function.

However, in many telecommunication systems, the object locations are static. The extra overhead in object binding can thus be amortized by the time saving achieved on multiple CORBA method calls on the same object reference. Section 5.2 indicates that passing four short messages through the flyover can offset the extra overhead of object binding.

To demonstrate the performance of the flyover running at higher system load, the experiment of a client/server system with Poisson arrival of requests is performed. As described in Section 5.3, the experimental results show that the relative performance of the flyover improves substantially when system load increases.

Section 5.4 presents the performance results for distributed systems with limited

heterogeneity. As the degree of heterogeneity decreases, greater numbers of CORBA invocations pass through the flyover. Therefore the response time measurement is closer to the homogeneous systems. This experiment also demonstrates that even with a small subset of communicating components that exhibits similarity in software and hardware platforms, the flyover still ameliorates the overall system performance.

Chapter 6 Conclusions

6.1 Summary

The main focus of this thesis is the design, implementation and analysis of the performance prototype of the flyover. The flyover prototype is based on the Orbix-MT 2.3, which is a commercially available middleware product. The main goal of the flyover is to achieve performance gain on systems with limited heterogeneity. There are five optimization techniques used in the flyover, including preventing CDR conversions and padding bytes, GIOP optimization, direct socket communication and efficient request demultiplexing. It also preserves location transparency, client and server application transparencies. In addition, the flyover prototype is implemented without modification of the middleware source code. The flyover changes the client stub and server skeleton codes, which are automatically modified by the interface definition language processor (IDL) written in Perl. The major contributions of the thesis are included in Section 1.3.

The performance comparisons of the flyover with Orbix-MT 2.3 and Orbix 2000 for a variety of different data types flowing between the client and the server are described in Chapter 5. The flyover demonstrates a substantial response time improvement for short messages and a lesser improvement for large messages, when the client and the server use similar software and hardware platforms. For examples, transferring a long integer through the flyover achieves a 54.25% response time improvement as compared to that of Orbix 2000. Moreover, the experiment shows that the flyover can achieve a better system throughput than Orbix-MT 2.3 and Orbix 2000. A performance improvement of approximately 5% was observed for larger data sizes when a single client communicates

with a single server (see Section 5.1.1). A higher performance improvement is expected at higher system load when multiple clients contend for the system resources. It is also encouraging to observe that the throughput of the flyover is close to that of a raw socket for low (less than 10K bytes) and high (more than 100K bytes) data sizes.

6.2 Future Research

This research leads to a few future research questions that are presented.

- One of the important directions of research is to allow the flyover to detect whether the client and server are comprised of similar technologies (hardware and software) at run time. One possible method is to obtain this information a priori from the system designers. However such an approach is not useful when the system architecture changes frequently or when objects move from one node to another. Using a central registry system for storing the hardware and software characteristics of a node and querying it at run time looks promising. Further research is warranted.
- The runtime flyover library needs to be compiled for each CPU architecture. Automatic downloading of the library routines from a server and its recompilation when the machine is booted forms an interesting direction for future research.
- The first prototype does not support some of IDL features, for example, wchar, wstring, type code, any, principal, context and user-defined exception. Developing support for these is worthy of future work.
- Since the object binding in the flyover requires one extra CORBA method invocation in order to obtain the port number and object ID of the target object, this incurs extra overhead in object binding (see Section 5.2). One method of preventing this substantial overhead is to piggyback the port number and object ID on the first reply

message after the first CORBA invocation. In other words, the first CORBA invocation must be passed through the ORB and then the server attaches the port number and the object ID to the reply after processing the request. Afterwards, the client can establish the explicit socket connection to the target server so that the subsequent requests can be passed through the flyover. This approach can save one CORBA invocation in the flyover.

- Figure 5-5 illustrates that data (octet) throughput through the flyover is higher than Orbix 2.3 and Orbix 2000, and is closer to that of a raw socket. In other words, the flyover is useful for data intensive application. There is related research in streaming service in the CORBA. The “Blob” Streaming framework [37] combines the flexibility of high-level distributed object computing middleware (e.g. CORBA) with the efficiency of low-level transport mechanisms (e.g. sockets). It uses the sockets to achieve high performance for bulk data transfer and uses CORBA to provide the flexibility needed for distributed electronic medical imaging systems. The OMG also defines CORBA A/V Streaming Service specification [29]. The A/V streaming service uses CORBA for streaming control (e.g. play, stop and rewind) and flow control (e.g. buffering) while it utilizes the external communication path for bulk audio/video transfer between two end-points. The A/V stream service shows that the throughput of A/V streaming service is close to the maximum throughput of TCP.

As shown in Figure 5-5, the performance of the flyover is close to that of a raw socket even for large data sizes. Thus the flyover can also be used for bulk data transfer (octet stream) in a heterogeneous environment, while the CORBA is used for synchronization such as streaming control and flow control. Since the flyover

provides communication and application transparencies, the application designer does not need to deal with low-level socket programming and related system calls. The performance comparison of using flyover for bulk data transfer with the CORBA A/V streaming service warrants further investigation.

References

- [1] I. Abdul-Fatah and S. Majumdar, "Performance Comparison of Architectures for Client-Server interactions in CORBA," in *Proceedings of International Conference on Distributed Computing Systems (ICDCS '98)*, Amsterdam, Netherlands, May 1998, pp. 2-10.
- [2] R. Adler, "Distributed Coordination Models for Client/Server Computing," *IEEE Computer*, Vol. 28, No. 4, April 1995, pp. 14-22.
- [3] I. Ahmad and S. Majumdar, "Achieving High Performance in CORBA-Based Systems with Limited Heterogeneity," in *Proceeding of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2001)*, Magdeburg, Germany, May 2001, pp. 350-359.
- [4] AT&T Laboratories, "omniORB2," Cambridge, 1999.
- [5] T. Barth, G. Flender, B. Freisleben, F. Thilo, "Load Distribution in a CORBA Environment," in *Proceedings of International Symposium on Distributed Objects and Applications 99*, Edinburgh, United Kingdom, September, 1999.
- [6] T. J. Brando, "Comparing DCE and CORBA," Technical Report, MITRE Corporation, March 1995.
- [7] L. Cabrera, E. Hunter, M. J. Karels, and D. A. Mosher, "User-Process Communication Performance in Networks of Computers," *IEEE Transactions on Software Engineering*, Vol. 14, No. 1, July 1988, pp. 38-53.
- [8] E. Eide, K. Frei, B. Ford, J. Lepreau and G. Lindstrom, "Flick: A Flexible, Optimizing IDL Compiler," in *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, Las Vegas, NV, June 1997, pp. 44-56.
- [9] D. R. Engler and M. F. Kaashoek, "DPF: Fast, Flexible Message Demultiplexing using Dynamic Code Generation," in *Proceedings of ACM SIGCOMM '96 Conference*, Stanford University, California, USA, August 1996, pp. 53-59.
- [10] P. G. Soares Florissi, "QuAL: The Quality of Service Assurance Language," Ph.D. Thesis, Department of Computer Science, Columbia University, 1998.
Available from: <http://www.cs.columbia.edu/~pgsf/qual.html>
- [11] C. D. Gill, D. L. Levine and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service", *International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, Vol. 20, No. 2, March 2001, pp. 117-154.

- [12] C. D. Gill, D. C. Schmidt, J. P. Loyall, R. Schantz, D. L. Levine, and F. Kuhns, "Applying Adaptive Middleware to Manage End-to-End QoS for Next-generation Distributed Applications", *Computer Communications, special issue on QoS-Sensitive Network Applications and Systems*, 2000.
- [13] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks", in *Proceedings of SIGCOMM '96*, Stanford, California, August 1996, pp. 306-317.
- [14] A. Gokhale and D. C. Schmidt, "Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks", in *Proceedings of GLOBECOM '96*, London, England, November 1996, pp. 50-56.
- [15] A. Gokhale and D. C. Schmidt, "Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks," in *Proceedings of 17th International Conference on Distributed Computing Systems (ICDCS 97)*, Baltimore, USA, May 1997, pp. 401-410.
- [16] A. Gokhale and D. C. Schmidt, "Evaluating the Performance of Demultiplexing Strategies for Real-Time CORBA," in *Proceedings of IEEE GLOBECOM 97' Conference*, Phoenix, November 1997, pp. 1729-1734.
- [17] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA Inter-ORB Protocol (IIOP) Engine for Minimal Footprint Embedded Multimedia Systems," *IEEE Journal on Selected Areas in Communications, special issue on Service Enabling Platforms for Networked Multimedia Systems*, Vol. 17, No. 9, September 1999, pp. 1673-1706.
- [18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [19] P. Haggerty and K. Seetharaman, "The Benefits of CORBA-Based Network Management", *Communications of the ACM*, Vol. 41, No. 10, October 1998, pp.73-79.
- [20] P. Hoschka and C. Huitema, "Automatic Generation of Optimized Code for Marshalling Routines," in *Proceedings of IFIP Conference of Upper Layer Protocols, Architectures and Applications (ULPAA '94)*, Barcelona, Spain, June 1994, pp. 131-146.
- [21] IONA Technologies, "Orbix-MT 2.3," 1997.
- [22] R. Koster and T. Kramp, "Structuring QoS-Supporting Services with Smart Proxies," in *Proceedings of IFIP/ACM Middleware 2000 Conference*, Palisades, New York, April 2000, pp. 273-288.
- [23] F. Kuhns, D.C. Schmidt and D.L. Levine, "The Performance of a Real-time I/O Subsystem for QoS-enabled ORB Middleware," in *Proceedings of the Multimedia Computing and Networking (MMCNO0)*, San Jose, California, January 2000.

- [24] D. L. Levine, S Flores-Gaitan, D.C. Schmidt, "An Empirical Evaluation of OS End System Support for Real-time CORBA Object Request Brokers," in *Proceedings of Real-time Technology and Applications Symposium (RTAS)*, Vancouver, British Columbia, Canada, June 1999.
- [25] S. Lo and S. Pope, "The Implementation of a High Performance ORB over Multiple Network Transports," in *Proceedings of IFIP Middleware '98*, The Lake District, England, September 1998, pp. 157-172.
- [26] J. Loyall, J. Gossett, C. Gill, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, and D. Karr, "Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications," in *Proceedings of 21st International Conference on Distributed Computing Systems (ICDCS-21)*, Phoenix, Arizona, USA, April 2001, pp. 625-634.
- [27] Microsoft Corporation, "Distributed Component Object Model Protocol – DCOM/1.0," January 1998,
Available from: <http://www.microsoft.com/com/tech/DCOM.asp>.
- [28] S. Mullender, *Distributed System*, ACM Press, 1989.
- [29] S. Mungee, N. Surendran, Y. Krishnamurthy and D. C. Schmidt, "The Design and Performance of a CORBA Audio/Video Streaming Service," in *Proceedings of the HICSS-32 International Conference on System Sciences, Multimedia DBMS and the WWW*, Hawaii, January 1999.
- [30] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, ver 2.3.1, October 1999.
- [31] Object Management Group, *Real-Time CORBA 1.2 FTF Report*, August 2000.
- [32] P. Narasimban, L. E. Moser and P. M. Melliar-Smith, "Using Interceptors to Enhance CORBA," *IEEE Computer*, Vol. 32, No.7, July 1999, pp. 62-68.
- [33] S. W. O'Malley, T. A. Proebsting, and A. B. Montz, "USC: A Universal Stub Compiler," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, London, UK, August 1994, pp. 295-306.
- [34] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware," in *Proceedings of IFIP/ACM Middleware 2000 Conference*, Palisades, New York, April 2000, pp. 372-395.
- [35] O. Othman, C. O'Ryan and D. C. Schmidt, "An Efficient Adaptive Load Balancing Service for CORBA," *Distributed Systems Engineering Journal*, March 2001.
- [36] A. Pope, *The CORBA Reference Guide*, Addison-Wesley, 1998.

- [37] I. Pyarali, T. H. Harrison and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Speed Electronic Medical Imaging," in *Proceedings of 2nd Conference on Object-Oriented Technologies and Systems*, Toronto, Canada, June 1996.
- [38] I. Pyarali, C. O'Ryan, D. Schmidt, N. Wang and V. Kachroo, "Applying Optimization Principle Patterns to Design Real-time ORBs," in *Proceeding of 5th Conference on Object-Oriented Technologies and Systems (COOTS '99)*, San Diego, California, USA, May 1999.
- [39] M. J. Robillard, H. R. Callison and J. Maurer, "Extending the DII COE for Real-Time," *CrossTalk: The Journal of Defense Software Engineering*, September 1999.
- [40] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6th USENIX C++ Technical Conference*, Cambridge, Massachusetts, April 1994.
- [41] D. C. Schmidt, "Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance", in *Proceedings of Hawaii International Conference on System Sciences*, January 1998.
- [42] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, Vol. 37, No.4, April 1999, pp. 54-63.
- [43] D. C. Schmidt, A. S. Gokhale, T. H. Harrison, and G. Parulkar, "A High-Performance End System Architecture for Real-Time CORBA," *IEEE Communications Magazines*, Vol. 35, No. 2, February 1997, pp. 72-77.
- [44] D. C. Schmidt, T. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of 1st Conference on Object-Oriented Technologies and Systems*, Monterey, California, June 1995.
- [45] D. C. Schmidt and F. Kuhns, "An Overview of the Real-time CORBA Specification", *IEEE Computer special issue on Object-Oriented Real-time Distributed Computing*, Vol. 33, No. 6, June 2000, pp. 56-63.
- [46] D. C. Schmidt, D. L. Levine, S. Mungee, "The Design of the TAO Real-Time Object Request Broker," *Computer Communications*, Vol. 21, No. 4, April 1998, pp. 294-324.
- [47] D. C. Schmidt and S. Vinoski, "Object Adapters: Concepts and Terminology", *SIGS C++ Report*, Vol. 9, No. 11, November/December, 1997.
- [48] D. C. Schmidt and S. Vinoski, "Object Interconnections: Using the Portable Object Adapter for Transient and Persistent CORBA Objects," *SIGS C++ Report*, Vol. 10, No. 4, April 1998.
- [49] D. C. Schmidt and S. Vinoski, "Object Interconnections: C++ Servant Classes for the POA," *SIGS C++ Report*, Vol. 10, No. 6, June 1998.

- [50] D. C. Schmidt, N. Wang and S. Vinoski, "Collocation Optimization for CORBA," *SIGS C++ Report*, September 1999.
- [51] R. L. Schwartz and T. Christiansen, *Learning Perl*, O'reilly And Associates, 1998.
- [52] SGI, "Standard Template Library (STL)," 1997.
- [53] E-K. Shen, S. Majumdar and I. Abdul-Fatah, "The Performance of Adaptive Middleware Systems," in *Proceedings of ACM Principles of Distributed Computing Conference (PODC '00)*, Portland, Oregon, July 2000, pp. 199-207.
- [54] P. Melliar-Smith, L. Moser, V. Kalogeraki and P. Narasimhan, "The Realize Middleware for Replication and Resource Management," in *Proceedings of IFIP Middleware '98*, The Lake District, England, September 1998, pp. 123-138.
- [55] W. R. Stevens, *UNIX Network Programming*, Prentice Hall, 1990.
- [56] Sun Microsystems, *Solaris 7.0 Reference Manuals*, 1999.
- [57] A. S. Tanenbaum, *Computer Networks*, Prentice Hall, 1996.
- [58] A. S. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, 1995.
- [59] R. Vanegas, J. A. Zinky, J. P. Loyall, D. Karr, R. E. Schantz, D. E. Bakken, "QuO's Runtime Support for Quality of Service in Distributed Objects," in *Proceedings of IFIP Middleware '98*, The Lake District, England, September 1998, pp. 207-222.
- [60] S. Vinoski, "CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environment," *IEEE Communication Magazines*, Vol. 35, No. 2, February 1997, pp. 72-77.
- [61] S. Vinoski, "New Features for CORBA 3.0," *Communications of the ACM*, Vol. 41, No. 10, October 1998, pp. 44-52.
- [62] S. Vinoski and M. Henning, *Advanced CORBA Programming with C++*, Addison Wesley, 1999.

Appendix A

A.1 Marshalling and Demarshalling of a CORBA Struct in the Flyover

In CORBA, IDL compiler compiles a CORBA struct into a target programming language mapping. Suppose that the target programming language is C++. A CORBA struct is mapped directly to a C++ struct such that each member of the CORBA struct maps to a corresponding member of the C++ struct. Consider the following:

```
//IDL
struct cd {
    char c;
    double d;
};
```

This maps to:

```
//C++
struct cd {
    CORBA::Char c;
    CORBA::Double d;

    void encodeOp (CORBA::Request &IT_r) const;
    void decodeOp (CORBA::Request &IT_r);
    void decodeInOutOp (CORBA::Request &IT_r);
    cd(const cd &);
    cd();
    ~cd();
    cd& operator= (const cd&);
};
```

The functions `encodeOp(CORBA::Request&)` and `decodeOp(CORBA::Request&)` are the marshalling and demarshalling functions of the C++ struct respectively. Figure A-1 shows the sequence diagram for marshalling of the C++ struct `cd` mentioned above in Orbix-MT 2.3. To marshal a CORBA struct, the client stub or server skeleton calls the struct's member function `cd::encodeOp` with the `CORBA::Request` object as an input parameter. This function will marshal each member of the CORBA struct into the

CORBA::Request object in the order of its declaration. In the sample code mentioned above, the first member c (char) is marshalled by the overloading operator <<(CORBA::Char&). Afterwards, the second member d (double) is marshalled by the overloading operator <<(CORBA::Double&).

Figure A-1 Marshalling of a CORBA struct in Orbix-MT 2.3

Similarly, to demarshal a CORBA struct from the request, the client stub or server skeleton calls the struct `cd::decodeOp` function with the `CORBA::Request` object as an input parameter. The function `cd::decodeInOutOp` is used for demarshalling the CORBA struct as an `inout` parameter.

In the flyover, the client stub or server skeleton needs to marshal/demarshal the struct into/from the `Message` object. Recall that class `Message`, which is derived from the class `CORBA::Request`, is responsible for carrying the request and reply information in the optimized GIOP format between the client and server through the flyover. To allow marshalling and demarshalling the parameters and return values into/from the `Message` object, IDLP adds three functions to the C++ mapping of the CORBA struct, namely, `encodeOp(Message&)`, `decodeOp(Message&)` and `decodeInOut(Message&)`.

To marshal a CORBA struct in the flyover, the client stub or server skeleton calls the struct member function `cd::encodeOp(Message&)` with the `Message` object as an

input parameter, instead of the CORBA::Request object. Figure A-2 shows the sequence diagram for marshalling the CORBA struct in the flyover.

Figure A-2 Marshalling of a CORBA struct in the flyover

The interactions shown in Figure A-2 is similar to that in Figure A-1 except that the parameter of the marshalling function `encodeOp` is a `Message` object, which marshals each member of the CORBA struct into the optimized GIOP format.

The functions `decodeOp(Message&)` and `decodeInout(Message&)`, which are added by IDLP, are used for demarshalling the CORBA struct as out parameters and inout parameters in the flyover respectively.

A.2 Marshalling and Demarshalling of a CORBA Union in the Flyover

A CORBA union maps directly to a C++ struct. Consider the IDL declarations:

```

//IDL
typedef sequence<long, 1024> seq;
typedef long vector[1024];
typedef string<1024> str;

union u switch(long) {
    case 1: float f;
    case 2: seq b;
    case 3: str s;
    default: vector v;
};
    
```

The IDL compiler maps to the following C++ struct:

```

//C++
struct u {
private:
    //The discriminant
    CORBA::Long __d;
    union {
        CORBA::Float _f_;
        seq * _b_;
        CORBA::String_mgr * _s_;
        vector_slice * _v_;
    };
public:
    void encodeOp (CORBA::Request &IT_r) const;
    void decodeOp (CORBA::Request &IT_r);
    void decodeInOutOp (CORBA::Request &IT_r);

    CORBA::Long _d() const;
    ...
};

```

In the C++ mapping of the CORBA union, the value of the discriminant `__d` indicates the type of the value that the union currently holds. The function `_d()` returns the current value of the discriminant.

In Orbix-MT 2.3, marshalling of the CORBA union uses corresponding marshalling function of the current type of the value that the union holds. Figure A-3 shows the sequence diagrams of the marshalling of the CORBA union in Orbix-MT 2.3. Each interaction in Figure A-3 is outlined below.

1. The client stub or server skeleton marshals the union by calling the method `u::encodeOp(CORBA::Request&)`.
2. The marshalling function `u::encodeOp(CORBA::Request&)` first marshals the discriminant `__d` into the `CORBA::Request` object `IT_r`.
3. If the discriminant `__d` is equal to 1, then it calls the overloading operator `<<(CORBA::Float)` to marshal the float value `_f_`.

4. If the discriminant `__d` is equal to 2, then it marshals the sequence `_b_`. For more information about the marshalling and demarshalling of the CORBA sequence, please see Section 3.7.4.4.
5. If the discriminant `__d` is equal to 3, then it marshals the string `_s_` by calling the `Request::encodeStringOp` method. For more information about the marshalling and demarshalling of the CORBA string, please see Section 3.8.2.
6. If the discriminant `__d` is equal to 4, then it marshals the array `_v_` by calling the `Request::encodeLongArray` method. For more information about the marshalling and demarshalling of IDL array, please see Section 3.8.2.

Figure A-3 Marshalling of a CORBA union in Orbix-MT 2.3

To allow marshalling and demarshalling the parameters and return values into/from the Message object, IDLP adds three functions to the C++ mapping of the CORBA

union, namely, `encodeOp (Message&)`, `decodeOp (Message&)` and `decodeInout (Message&)`. The `encodeOp (Message&)` marshals the union into the `Message` object in the optimized GIOP format whereas the `decodeOp (Message&)` demarshals the union from the `Message` object. The last function `decodeInout` is used for demarshalling the union as an inout parameter.

The sequence diagram for marshalling a CORBA union into the optimized GIOP format is shown in Figure A-4. The interactions in Figure A-4 are similar to those in Figure A-3 except that the `Message` object is an parameter of the marshalling function `u::encodeOp (Message&)` instead of the `CORBA::Request` object so that all marshalling functions are invoked on the `Message` object.

Figure A-4 Marshalling of a CORBA union in the flyover

A.3 Interface Inheritance in the Flyover

The flyover also supports IDL single interface inheritance. Interface inheritance allows a new interface to be defined by extending the functionality. Consider the following IDL example:

```
//IDL
//interfaces to demonstrate the single interface inheritance in the flyover
interface A {
    void foo();
};
interface B : A{
    void bar();
};
```

Figure A-5 IDL demo for interface inheritance in the flyover

Figure A-6 is the corresponding C++ class hierarchy on the server side in Orbix-MT 2.3 based on the IDL interfaces defined in Figure A-5. In Figure A-6, IDL C++ class A and class B are both generated by Orbix compiler corresponding to the IDL interfaces. Since interface B inherits from interface A, the IDL C++ class B also inherits from class A. Suppose the BOA-Impl approach is using here. The BOA-Impl class inherits from the corresponding IDL C++ class. For example, class ABOAImpl inherits from class A and class BBOAImpl inherits from class B. To implement the IDL interface in Orbix, the implementation class derives from the BOA-Impl class. For example, the implementation class A_implementation inherits from the class ABOAImpl. However, since interface B derives from interface A, the implementation class B_implementation needs to inherit from both its BOA-Impl class BBOAImpl and class A_implementation.

Figure A-6 Class diagram of single interface inheritance on the server side in Orbix-MT 2.3

Figure A-7 shows the modified class hierarchy that incorporate single interface inheritance in the flyover. The modified class diagram is similar to original one as shown in Figure A-6, except that the BOA-Impl classes inherit from both their IDL C++ classes and the class `BypassImpl`. This modification is carried out automatically by IDLP. Note that interface A and interface B share the common variable `__objindex__`, therefore during server object registration, these two values must be set to the same values so that it can be retrieved from the hash table in request demultiplexing.

Figure A-7 Class diagram of single interface inheritance in the flyover

Appendix B

B.1 Additional Performance Measurement Results for CORBA Array

Figure B-1 Response time measurement for CORBA array of boolean

Figure B-2 Response time measurement for CORBA array of short

Figure B-3 Response time measurement for CORBA array of float

B.2 Additional Performance Measurement Results for CORBA Sequence

Figure B-4 Response time measurement for CORBA sequence of char

Figure B-5 Response time measurement for CORBA sequence of boolean

Figure B-6 Response time measurement for CORBA sequence of short

Figure B-7 Response time measurement for CORBA sequence of float

Figure B-8 Response time measurement for CORBA sequence of double