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[®]

Performance Enhancement Techniques for CORBA-Based Systems with Limited Heterogeneity

by

Imran Ahmad

**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

January, 2002

© Copyright 2001, Imran Ahmad

**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.

0-612-68308-7

Canada

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

**Performance Enhancement Techniques for CORBA-Based
Systems with Limited Heterogeneity**

submitted by Imran Ahmad,

in partial fulfillment of the requirements for the degree of

Master of Applied Science

Chair, Department of Systems and Computer Engineering

Professor Shikharesh Majumdar, Thesis Supervisor

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
23rd August, 2001

Abstract

Applications and services for next-generation distributed systems must be interoperable, reusable, and capable of providing scalability as well as low latency to delay-sensitive systems that include telecommunication and other distributed systems running over high-speed networks. Requirements for interoperability and reusability motivate the use of object-oriented middleware such as Common Object Request Broker Architecture (CORBA) compliant products. However, unless CORBA can be implemented efficiently, it will not be widely used in realtime and other latency-sensitive distributed applications. This thesis presents three performance enhancement techniques for CORBA-based middleware. Two of these exploit limited heterogeneity in systems. In such systems a standard CORBA protocol is used when clients and servers interacting with one another are implemented by using different programming languages and/or operating systems. However, when a similar client-server pair, built using the same technology, communicates, a number of CORBA operations are bypassed, thus reducing the communication overhead. Based on a commercial middleware product, and measurements made on a performance prototype running on a network of workstations, this research demonstrates that there is strong potential for achieving a significant performance improvement by incorporating these techniques into the middleware. Two of the performance enhancement techniques presented in the thesis are appropriate for data intensive applications, whereas the third is effective when short control messages are exchanged between the client and its server.

Acknowledgments

Some three and half years ago, while browsing different North American Universities on my newly installed Internet system in Pakistan, I came across the name of Carleton University and Professor Majumdar. It was summer of 1997. I casually wrote an email to the professor explaining my area of interest, and inquired whether he would be interested in supervising me for graduate studies. I had no idea that this casual interaction would become a significant milestone for my professional career. I would like to express my sincere gratitude to Professor Majumdar for the opportunity he provided to me while I was thousands of miles away, and for his valuable guidance, advice and encouragement throughout my research. His pivotal role in this whole project cannot be overstressed.

I am also indebted to my family in Pakistan, especially my father, for his continuous long distance encouragement.

I would like to thank the Israr family, especially Junaid and Tauseef, for their support throughout this research.

Last but not the least, I would like to sincerely thank my wife Naheed for her constant encouragement. This whole research would not have been possible without her consistent support.

Table of Contents

CHAPTER 1: INTRODUCTION.....	1
1.1 AN OVERVIEW OF DISTRIBUTED SYSTEMS.....	1
1.2 DISTRIBUTED OBJECT COMPUTING	2
1.3 MOTIVATION FOR THIS THESIS	3
1.4 GOALS OF THE THESIS.....	4
1.5 STATE OF THE ART	4
1.6 PERFORMANCE OF CORBA BASED SYSTEMS	7
1.7 THESIS OUTLINE	8
CHAPTER 2: CORBA	9
2.1 CORBA AND DISTRIBUTED COMPUTING.....	9
2.2 STRUCTURE OF AN OBJECT REQUEST BROKER (ORB).....	10
2.2.1 <i>Role of Clients</i>	12
2.2.2 <i>Dynamic Invocation Interface</i>	13
2.2.3 <i>Interface Repository</i>	13
2.2.4 <i>Implementation Repository</i>	14
2.2.5 <i>Example ORBs</i>	14
2.3 STRUCTURE OF AN OBJECT IMPLEMENTATION	16
2.4 GENERAL REQUEST FLOW	17
2.5 OMG INTERFACE DEFINITIONS LANGUAGE	18
2.6 LANGUAGE MAPPINGS.....	22

2.7 OBJECT ADAPTERS	23
2.7.1 <i>Implementation of Object Request Broker</i>	26
2.7.2 <i>Reference Acquisition</i>	28
2.8 PROXIES.....	28
2.9 GENERAL CORBA APPLICATION DEVELOPMENT.....	29
CHAPTER 3: CHARACTERIZATION OF MIDDLEWARE	32
3.1 INTRODUCTION	32
3.2 MIDDLEWARE REQUIREMENTS	33
3.2.1 <i>Scalability and Load Balancing</i>	33
3.2.2 <i>Synchronization and Co-ordination</i>	34
3.2.3 <i>Endpoint Binding Transparency</i>	34
3.2.4 <i>Ease of Use</i>	34
3.3 HETEROGENEITY OF A SYSTEM	35
3.3.1 <i>Concept of Domains in a System</i>	35
3.3.2 <i>Analysis of a Typical System</i>	37
3.3.3 <i>Characterization of Heterogeneity</i>	37
3.3.4 <i>Concept of Flyovers in a System with Limited Heterogeneity</i>	39
3.4 GIOP	39
3.4.1 <i>Introduction</i>	39
3.4.2 <i>GIOP's Transport Assumptions</i>	40
3.4.3 <i>The Common Data Representation</i>	42
3.4.4 <i>Connecting Client and Server</i>	43

3.4.5 <i>Marshalling/Unmarshalling</i>	45
3.5 SPECIALIZING GIOP INTO IIOP.....	46
3.5.1 <i>Connection Mechanism</i>	47
3.6 DETAILS OF GIOP MESSAGES.....	48
3.7 CORBA: NOT AN IDEAL MIDDLEWARE.....	57
3.7.1 <i>The reality of IIOP Interoperability</i>	57
3.7.2 <i>CORBA Messaging</i>	58
3.7.3 <i>Performance</i>	59
CHAPTER 4: PERFORMANCE ENHANCEMENT TECHNIQUES	60
4.1 OVERVIEW.....	60
4.2 METHODS USED IN PERFORMANCE ENHANCEMENT.....	60
4.2.1 <i>Using Interceptors for Customized Event Handling</i>	61
4.3 PERFORMANCE ENHANCEMENT STRATEGY 1: PREVENTING NATIVE-NEUTRAL DATA CONVERSIONS.....	66
4.3.1 <i>Background</i>	66
4.3.2 <i>Implementation</i>	67
4.4 PERFORMANCE ENHANCEMENT STRATEGY-2: REMOVING PADDING	67
4.4.1 <i>Background</i>	67
4.4.2 <i>Implementation</i>	69
4.5 PERFORMANCE ENHANCEMENT STRATEGY-3: COMPRESSING HEADERS.....	70
4.5.1 <i>Background</i>	70
4.5.2 <i>Implementation</i>	71

**CHAPTER 5: RESULTS OF EXPERIMENTS ON THE PERFORMANCE
ENHANCEMENT TECHNIQUES..... 73**

5.1 METHODS USED FOR PERFORMANCE MEASUREMENTS 75

5.1.1 Using JNI..... 75

5.1.2 Measuring Marshalling Time at the Client Using Interceptors..... 77

5.2 EXPERIMENTAL ENVIRONMENT 78

5.3 PERFORMANCE ENHANCEMENT STRATEGY 1:..... 80

5.4 PERFORMANCE ENHANCEMENT STRATEGY-2: 83

5.4.1 Results of Experiments 83

5.5 PERFORMANCE ENHANCEMENT STRATEGY-3: COMPRESSING HEADERS..... 84

5.6 OBSERVING OPTIMIZED BEHAVIOR WITH VARYING DEGREES OF HOMOGENEITY 86

5.7 OBSERVING EFFECTS OF INCREASED SYSTEM LOAD 89

5.7.1 Observing the Performance Enhancement in a Real World Scenario 89

5.7.2 Results of Experiment..... 92

5.8 DISCUSSION 95

CHAPTER 6: CONCLUSIONS 99

6.1 CONTRIBUTIONS OF THIS RESEARCH 99

6.2 PERFORMANCE ENHANCEMENT TECHNIQUES: CONCLUSIONS 99

6.2.1 Performance Enhancement Technique 1..... 100

6.2.2 Performance Enhancement Technique 2..... 100

6.2.3 Performance Enhancement Technique 3..... 101

6.2.4 Future Research..... 102

REFERENCES 103

List of Figures

Figure 2.1: Common Object Request Broker Architecture	27
Figure 2.2: Proxy represents the actual Server in CORBA environment	30
Figure 3.1: The concept of CORBA domains	35
Figure 3.2: IIOP IOR Structure	48
Figure 3.3: GIOP Messaging	50
Figure 3.4: Definition of GIOP message Header	51
Figure 3.5: Order of a Request Message Stream	54
Figure 3.6: GIOP Request Header	54
Figure 4.1: Different Stages in a typical CORBA transaction	63
Figure 5.2: ClientRequestInterceptor interface	64
Figure 5.1: JNI provides an interface between Java and C worlds	76
Figure 5.2: Result of technique 1	81
Figure 5.3: Performance Optimization Strategy 2	85
Figure 5.4: Division of nodes into domains according to their homogeneity	88
Figure 5.5: Event Flow in Single Server – Single Client System	92
Figure 5.6: Effect of System Load	93

List of Tables

Table 3.1: GIOP Messages	49
Table 4.1. CDR alignment of primitive fixed-length types	68
Table 4.2: GIOP fields that are needed in a transaction between two similar nodes	72
Table 5.1: Improvement in marshalling/unmarshalling from performance enhancement Technique-1	80
Table 5.2: Experimental Results of Enhancement technique 3.	86
Table 5.3: Effect of varying degree of Homogeneity on Enhancement Technique 2	89
Table 5.4: Results of applying Enhancement strategy 2: Original Response Time	94
Table 5.5: Results of applying Enhancement strategy 2: Response Time after applying Enhancement	94

Glossary of Terms

A-ORB	Adaptive ORB
BOA	Basic Object Adapter
CCM	CORBA Component Model
CDR	Common Data Representation
CORBA	Common Object Request Broker Architecture
DCE	Distributed Computing Environment
DII	Dynamic Invocation Interface
DOC	Distributed Object Computing
DSI	Dynamic Skeleton Interface
F-ORB	Forwarding ORB
H-ORB	Handle-Driven ORB
IDL	Interface Definition Language
IOR	Interoperable Object Reference
IIOP	Internet Inter-ORB Protocol
IPC	Interprocess Communication
JNI	Java Native Interface
JVM	Java Virtual Machine
OMA	Object Management Architecture
OMG	Object Management Group
ORB	Object Request Broker

OO

Object Oriented

RPC

Remote Procedure Call

CHAPTER 1

Introduction

1.1 An Overview of Distributed Systems

A distributed computing system is a system having multiple nodes that can perform computation. Each of these computing nodes may be a PC, a mainframe, or other types of computing device. Various network communication devices connecting these distributed nodes make the distributed computing possible.

An alternative to distributed computing is the centralized computing system in which all the computing power is within a central mainframe node. Network overheads were always thought to be a bottleneck for distributed computing, and centralized computing used to be the preferred choice. Over the last few years the advances in network communication technologies have helped make distributed computing a better choice over the centralized systems. Today, distributed computing has many advantages over centralized computing including: reliability, performance, scalability, extensibility and cost effectiveness [WRU-00].

Distributed systems can be seen in all spheres of life. They range from the omni-present Internet to railway tracking and monitoring systems. In spite of the current downturn in global economy, technologies related to distributed systems continue to grow and evolve at a rate that is perhaps higher than any other area of technical research [ZDN 01].

1.2 Distributed Object Computing

Over the last decade Object-Oriented (OO) computing has proven its worth and has become a very popular computing paradigm [MST-95]. The underlying concepts of polymorphism, inheritance and data encapsulation give it an advantage over procedural language based systems. It facilitates the reusability of code as modular, self-contained independent blocks, which can considerably reduce the total programming efforts, and eliminate the 're-inventing the wheel' phenomenon [EGA-95].

Distributed Object Computing (DOC) combines the advantages of distributed systems and OO technology. There are many benefits that we obtain by combining these two technologies including:

- Object-oriented features such as polymorphism, inheritance and encapsulation are now available in the distributed environment.
- Interaction between distributed applications can be realized at a higher level of abstraction.
- Different conceptual software modules can be grouped as independent self-contained blocks and can be reused. This offers an opportunity for the vendors to

develop such components as libraries that users can use as commercial-off-the-shelf (COTS) products.

A major advantage of using DOC is that it can provide location transparency, thus making it as easy to access and use an object on a remote node, as an object on a local node. Location transparency involves the following:

- Locating and loading remote classes
- Locating remote objects and providing references to them.
- Enabling remote method calls, including passing of remote objects as arguments and return values
- Notifying programs of network failures and other problems.

1.3 Motivation for this Thesis

Distributed computing environments are characterized by diversity in hardware and software. The need for interoperability among these heterogeneous systems is rapidly increasing. The notion of middleware has been introduced to provide interoperability in these heterogeneous distributed environments. The Common Object Request Broker Architecture (CORBA), proposed by Object Management Group (OMG), is one of the commonly used standards for middleware architectures. CORBA, which is based on DOC, brings all the advantages offered by a distributed computing and object oriented paradigm in a heterogeneous world. But all these advantages come at a price, and CORBA introduces additional overheads. In certain performance sensitive applications,

these overheads may become a stumbling block. This research is motivated by the desire to optimize the CORBA systems.

1.4 Goals of the Thesis

CORBA standard has emerged to provide interconnectivity between distributed nodes. It assumes that all of the nodes within a system are heterogeneous and do not provide any advantage if most of the nodes happen to be of the same type. This thesis has tried to discover ways to optimize the system performance in such scenarios. The goal is to develop performance enhancement techniques that can improve system performance, when a subset of the components in the system uses the same programming language, operating system and hardware.

1.5 State of the art

Conventional implementations of CORBA incur considerable overheads when used for performance-sensitive applications over high-speed networks. Research is ongoing in order to make CORBA as fast as possible so that it can be used in various performance thirsty systems. A representative set of papers that focus on performance issues in CORBA are discussed. A more detailed survey is provided in [SMA-00.1].

One dimension of recent research has been in innovative client-middleware-server architecture configuration in order to improve system performance. These architectures refer to the way a request from the client is routed to the server and a reply from the

server is sent back to the client. For example, in a Handle-Driven (HORB) architecture, when a client wants to request a service it sends the server name to an entity called, the agent, in the middleware system. The agent performs a name to object reference (IOR) mapping, and sends an IOR or handle back to the client. The client uses the handle to contact the server and receive the desired service. A number of COTS ORBs use such a handle-driven architecture. In a Forwarding (F-ORB) architecture the client sends the entire request for service to the agent that locates the appropriate server and forwards the request to it. The server performs the desired service and sends a response back to the client. A detailed report on the investigation of adaptive middleware is presented in [ESH-00]. This paper presents a process planner architecture that uses a concurrency in server execution to improve performance. The combination of the good attributes of the H-ORB and the F-ORB are combined into an ORB called the Adaptive ORB (A-ORB), in which, depending on the system load, the interaction architecture switches between a handle-driven and a forwarding mode. The adaptive architecture is observed to perform significantly better than an H-ORB or an F-ORB. A performance comparison of these architectures is presented in [IAB-98.1] [IAB-98.2].

The research team at the University of Washington has developed techniques for improving the performance of conventional CORBA IIOP [DSC-98.2]. Several sources of overhead in IIOP implementations, such as excessive marshalling/ unmarshalling operations, data copying, and high level function calling, are identified. The optimization techniques suggested by the authors include optimizing for the common case, eliminating waste of CPU cycles that occurs during data copying, such as, replacing general purpose

methods with specialized efficient methods, as well as pre-computation and caching of values. Applying these techniques produced a high performance improvement for various types of messages transmitted over an ATM network. A summary of these efforts can be found in [DSC-98.1]. It should be noted that they work towards development of performance enhanced ORB from scratch, as opposed to the development of techniques to improve the performance of existing ORBs that this thesis has focused on.

In addition to the construction of the middleware, the underlying operating system plays a crucial role in the determination of overall system performance. An appropriate set of operating system services is important for the middleware to deliver the desired level of performance. A measurement-driven methodology for identifying the sources of these overheads is described in [DLE-99].

Using load-balancing techniques on systems that consist of replicated objects can also lead to an improvement in distributed system performance. The integration of load balancing with name service is proposed in [TBA-99]. CORBA has been used in the context of various applications. The performance of a group membership protocol based on CORBA is discussed in [SMI-00]. This paper provides a performance comparison of three implementations of group membership protocol: one implementation uses UDP sockets, while the other two use CORBA for communication.

Another area of research is to make CORBA compliant to real time system needs. A real time system is a system that receives stimuli from the external environment and produces

the required responses in a timely manner. A request for service on real time systems often has a quality of service (QoS) associated with it. The quality of service may include several attributes demanded from the response, such as bit error rate being lower than a specified upper bound. Research is being conducted to make a real-time ORB using classical designed patterns at the DOC group at the University of Washington. These research efforts are summarized in [DSC-98.2].

The CORBA Component Model (CCM) proposed recently by OMG promotes software reuse and provides more flexibility for dynamic configuration of applications [BSY-99]. Although the component model addresses a number of shortcomings of the original CORBA standard, it is not yet suitable for a number of performance demanding and real time systems. CCM currently lacks mechanisms through which one can optimize system performance when several components are collocated on the same node. A more detailed description of these limitations of CCM and a number of proposed solutions are described in [NWA-00.2]. The use of components to construct QoS sensitive systems is discussed in [NWA-00.1][NWA-99].

1.6 Performance of CORBA Based Systems

Distributed object computing is a popular paradigm for implementing both business and real time applications. Heterogeneity is natural in the kinds of distributed environments in which a system component is implemented in a particular programming language running on top of a specific operating system that has to communicate with another component

that is implemented using a different language and operating system. A CORBA middleware provides interoperability in such environments and makes it possible for such diverse components to interact with each other. Although middleware provides interoperability, it adds overheads due to the fact that it introduces additional layers of abstraction. Achieving high scalability and low latency are desirable in many environments that include embedded systems. Typical distributed systems are characterized by a considerable degree of homogeneity. That is, a large subset of components interacting with one another use the same programming language and operating system. Our research is motivated by the following question.

Problem Statement

Can homogeneity between components be exploited to reduce communication delays in a distributed object environment?

1.7 Thesis Outline

This thesis is divided into six chapters. *Chapter 1* is the introduction, which also specifies the goals of this thesis. *Chapter 2* is about the CORBA standard. *Chapter 3* explains the IIOP protocol in depth. *Chapter 4* discusses the performance enhancement techniques proposed in this research. *Chapter 5* presents the experimental environment and results of experiments that demonstrate the effectiveness of the techniques. *Chapter 6* presents our conclusions, and discusses the scope of future research.

CHAPTER 2

CORBA

2.1 CORBA and Distributed Computing

As the 1990s progressed, intranet and Internet computing evolved. TCP/IP emerged as by far the most popular transport layer protocol [GVA-96]. There was a need for a standard for the technology that could link together the various distributed nodes comprised of different hardware configurations, operating systems and programming languages [GPA-95].

This is where the CORBA specification emerged. In this heterogeneous world where scalability and reliability were crucial, CORBA provided a tight, reliable, and secure coupling between distributed nodes. By the end of the 1990s, networks had transformed from a simple wire that linked client and server to a key infrastructure component that enabled the linking of distributed computing nodes in powerful ways. The use of CORBA in these distributed environments is becoming more and more popular. CORBA applications can scale to thousands of transactions per second, and power Web sites in electronic commercial and financial trading in public view, and some of the largest manufacturing installations in less visible situations [MHE-99].

The change in distributed computing requirements during the past few years meant that the CORBA specification had to change. To keep pace, CORBA standard had to evolve. For example CORBA found its use in real-time and performance sensitive applications, and specifications had to be modified to incorporate the new requirements. Another reason for the evolution of CORBA was that its specifications were too loose, and different vendors perceived and implemented them differently, resulting in non-interoperable CORBA components [JSE-01]. Different proprietary inter-ORB protocols are now in use to cope with this situation. CORBA specifications had to change and be made more precise to handle this.

2.2 Structure of an Object Request Broker (ORB)

Some basic terminology that is important in the context of CORBA is detailed below.

Client is the entity that performs an operation on the object and *Object Implementation* is the code and data that actually implements the object.

The *ORB* is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the request [SIE-98]. The interface the client sees is completely independent of where the object is located, what programming language it is implemented in, or any other aspect, which is not reflected in the object's interface.

To make a request, the Client can use the *Dynamic Invocation interface* (the same interface independent of the target object's interface) or the static OMG Interface Definition Language, (*IDL stub* (the specific stub depending on the interface of the target object) [OMG-98].

The Object Implementation receives a request either through the OMG IDL generated skeleton or through a dynamic skeleton. The Object Implementation may call the Object Adapter and the ORB while processing a request or at other times. Definitions of the interfaces to objects can be defined in two ways. Interfaces can be defined statically in an interface definition language, called the OMG IDL. This language defines the types of objects according to the operations that may be performed on them and the parameters to those operations. Alternatively, interfaces can be added to an Interface Repository service; this service represents the components of an interface as objects, permitting run-time access to these components. In any ORB implementation, the Interface Definition Language and the Interface Repository have equivalent expressive power [OMG-98] [OMG-01].

The client performs a request by having access to an Object Reference for an object, and knowing the type of the object and the desired operation to be performed. The client initiates the request by calling stub routines that are specific to the object or by constructing the request dynamically.

The dynamic and static stub interfaces for invoking a request satisfy the same request semantics, and the receiver of the message cannot tell how the request was invoked [OMG-98]. The ORB locates the appropriate implementation code, transmits parameters, and transfers control to the Object Implementation through an IDL skeleton or a dynamic skeleton. Skeletons are specific to the interface and the Object Adapter. In performing the request, the object implementation may obtain some services from the ORB through the Object Adapter. When the request is complete, control and output values are returned to the client. The Object Implementation may choose which Object Adapter to use. This decision is based on what kind of services the Object Implementation requires [DSC-95].

2.2.1 Role of Clients

A client has access to an object reference for the object, and invokes operations on it. A client knows only the logical structure of the object according to its interface, and experiences the behavior of the object through invocations[AGO-96.1]. Although we will generally consider a client to be a program or process initiating requests on an object, it is important to recognize that anything relative to a particular object is a client. For example, the implementation of one object may be a client of other objects. Clients generally see objects and ORB interfaces through the perspective of a language mapping, bringing the ORB right up to the programmer's level. Clients have no knowledge of the implementation of the object, which object adapter is used by the implementation, or which ORB is used to access it.

2.2.2 Dynamic Invocation Interface

An interface is also available that allows the dynamic construction of object invocations; that is, rather than calling a stub routine that is specific to a particular operation on a particular object, a client may specify the object to be invoked, the operation to be performed, and the set of parameters for the operation through a call or sequence of calls. The client code must supply information about the operation to be performed and the types of the parameters being passed (obtaining it from an Interface Repository or other run-time source). The nature of the dynamic invocation interface may vary substantially from one programming language mapping to another.

2.2.3 Interface Repository

The Interface Repository is a service that provides persistent objects that represent the IDL information in a form available at run-time. The Interface Repository information may be used by the ORB to perform requests. Moreover, using the information in the Interface Repository, it is possible for a program to encounter an object whose interface was not known when the program was compiled, yet, be able to determine what operations are valid on the object and make an invocation on it. In addition to its role in the functioning of the ORB, the Interface Repository is a common place to store additional information associated with interfaces to ORB objects. For example, debugging information, libraries of stubs or skeletons, routines that can format or browse particular kinds of objects, etc., might be associated with the Interface Repository [AGO-96.2].

2.2.4 Implementation Repository

The Implementation Repository contains information that allows the ORB to locate and activate implementations of objects. Although most of the information in the Implementation Repository is specific to an ORB or operating environment, the Implementation Repository is the conventional place for recording such information. Ordinarily, installation of implementations and control of policies related to the activation and execution of object implementations is done through operations on the Implementation Repository.

In addition to its role in the functioning of the ORB, the Implementation Repository is a common place to store additional information associated with implementations of ORB objects. For example, debugging information, administrative control, resource allocation, security, etc., might be associated with the Implementation Repository [GBR-01].

2.2.5 Example ORBs

CORBA specification does not exactly specify how to implement ORB specifications. Different vendors take different approaches. Some of them, described in [OMG 98], are summarized below:

(a) Client- and Implementation-resident ORB

If there is a suitable communication mechanism present, an ORB can be implemented in routines resident in the clients and implementations. The stubs in the client either use a location-transparent IPC mechanism, or directly access a location service to establish communication with the implementations.

(b) Server-based ORB

To centralize the management of the ORB, all clients and implementations can communicate with one or more servers whose job it is to route requests from clients to implementations. The ORB could be a normal program as far as the underlying operating system is concerned, and normal IPC could be used to communicate with the ORB.

(c) System-based ORB

To enhance security, robustness, and performance, the ORB could be provided as a basic service of the underlying operating system. Because the operating system could know the location and structure of clients and implementations, it would be possible for a variety of performance enhancements to be implemented, such as, avoiding marshalling/unmarshalling when both are on the same machine.

(d) Library-based ORB

For objects that are light-weight and whose implementations can be shared, the implementation might actually be in a library. In this case, the stubs could be the actual methods. This assumes that it is possible for a client program to get access to the data for the objects, and that the implementation trusts the client not to damage the data.

2.3 Structure of an Object Implementation

An object implementation provides the actual state and behavior of an object. The object implementation can be structured in a variety of ways. Besides defining the methods for the operations themselves, an implementation defines procedures for activating and deactivating objects, using other objects or non-object facilities to make the object state persistent, to control access to the object, as well as to implement the methods. The object implementation interacts with the ORB in a variety of ways to establish its identity, to create new objects, and to obtain ORB-dependent services. It does this primarily via access to an Object Adapter, which provides an interface to ORB services that is convenient for a particular style of object implementation [DSC-97].

When an invocation occurs, the ORB Core, object adapter, and skeleton arrange that a call is made to the appropriate method of the implementation. A parameter to that method specifies the object being invoked, which the method can use to locate the data for the object. Additional parameters are supplied according to the skeleton definition. When the method is complete, it returns, causing output parameters or exception results to be transmitted back to the client.

When a new object is created, the ORB may be notified so that it knows where to find the implementation for that object. Usually, the implementation also registers itself as implementing objects of a particular interface, and specifies how to start up the implementation if it is not already running.

2.4 General Request Flow

In a CORBA environment, the client application makes requests and the server application receives them and acts on them. Requests flow down from the client application, through the ORB, and up into the server application in the following manner.

1. The client can choose to make requests either using static stubs or using the Dynamic Invocation Interface (DII). Either way the client directs the request into the ORB core linked into its process.
2. The client ORB core transmits the request to the ORB core linked with the server application.
3. The server ORB core dispatches the request to the object adapter that created the target object.
4. The object adapter further dispatches the request to the servant that is implementing the target object. Like the client, the server can choose between

static and dynamic dispatching mechanisms for its servants. It can rely on static skeletons, or its servants can use the Dynamic Skeleton Interface (DSI).

5. After the servant carries out the request, it returns its response to the client application.

CORBA supports several styles of requests.

- When a client invokes a synchronous request, it blocks while it waits for the response. These requests are similar to conventional remote procedure calls.
- A client that invokes a deferred synchronous request sends the request, continues processing, and then later polls for the response. Currently, this style of request can be invoked only using the DII.
- CORBA also provides a oneway request, which is a best-effort request that may not actually be delivered to the target object and is not allowed to have responses. ORBs are allowed to silently drop oneway requests if network congestion or other resource shortages would cause the client to block while the request is delivered.

2.5 OMG Interface Definitions Language

To invoke operation on a distributed object, a client must know the interface offered by the object. An object's interface is composed of the operations it supports and the types of

data that can be passed to and from those operations. Clients also require knowledge of the purpose and semantics of the operations they want to invoke.

In CORBA, object interfaces are defined in the OMG Interface Definition Language (IDL). Unlike C++ or Java, IDL is not a programming language, so objects and applications cannot be implemented in IDL. The sole purpose of the IDL is to allow object interfaces to be defined in a manner that is independent of any particular programming language. This arrangement allows applications implemented in different programming languages to interoperate. The language independence of IDL is critical to the CORBA goal of supporting heterogeneous systems and the integration of separately developed applications [OCC-99].

OMG IDL supports built-in simple types, such as signed and unsigned integer types, characters, boolean, and strings, as well as constructed types such as enumerated types, structures, discriminated unions, sequences (one-dimensional vectors), and exceptions. These types are used to define the parameter types and return types for operation, which in turn are defined within interfaces. IDL also provides a module construct used for name-scoping purposes.

The following example shows a simple IDL definition:

```
interface Employee
{
    long number( );
};-
```

This example defines an interface named `Employee` that contains an operation named `number`. The `number` operation takes no arguments and returns a `long`. A CORBA object supporting the `Employee` interface is expected to implement the `number` operation to return the number of the employee represented by that object.

Object references are denoted in IDL by using the name of an interface as a type. For example:

```
interface EmployeeRegistry {  
    Employee lookup (in long emp_number);  
};
```

The `lookup` operation of the `EmployeeRegistry` interface takes an employee number as an input argument, and returns an object reference of type `Employee` that refers to the employee object identified by the `emp_number` argument. An application could use this operation to retrieve an `Employee` object, and then use the returned object reference value to invoke `Employee` operations.

Arguments to IDL operations must have their directions declared so that the ORB knows whether their values should be sent from client to target object, or vice versa, or both. In the definition of the `lookup` operation, the keyword, “in”, signifies that the employee number argument is passed from the client to the target object. Arguments can also be declared, “out”, to indicate that, like return values, they are passed from the target object back to the client. The “inout” keyword indicates an argument that is initialized by the

client and then sent from the client to the target object; the object can modify the argument value and return the modified value to the client [DSC-98.2].

A key feature of IDL interfaces is that they can inherit from one or more other interfaces. This arrangement allows new interfaces to be defined in terms of existing ones, and objects implementing a new derived interface can be submitted where objects supporting the existing base interfaces are expected. For example, consider the following Printer interfaces:

```
interface Printer {  
    Void print ( ) ;  
};  
  
interface ColorPrinter : Printer {  
    enum ColorMode {BlankAndWhite, FullColor };  
    void set_color (in ColorMode mode);  
};
```

The ColorPrinter interface is derived from the Printer interface. If a client application is written to deal with objects of type Printer, it can also use an object supporting the ColorPrinter interface, because such objects also fully support the Printer interface.

IDL provides one special case of inheritance: all IDL interfaces implicitly inherit from the Object interface defined in the CORBA module. This special base interface supplies operations common to all CORBA objects.

2.6 Language Mappings

Because OMG IDL is a declarative language, it cannot be used to write actual applications. It provides no control constructs or variables, so it cannot be compiled or interpreted into an executable program. It is suitable only for declaring interfaces for objects and defining the data types used to communicate with objects.

Language mappings specify how IDL is translated into different programming languages. For each IDL construct, a language mapping defines which facilities of the programming language are used to make the construct available to applications. For example, in C++, IDL interfaces are mapped to classes, and operations are mapped to member functions of those classes. Similarly, in Java, IDL interfaces are mapped to public Java interfaces. Object references in C++ map to constructs that support the operator-> function (that is, either a pointer to a class or an object of a class with an overloaded operator-> member function). Object references in C, on the other hand, map to opaque pointers (of type void *), and operations are mapped to C functions, each of which require an opaque object reference as the first parameter. Language mappings also specify how applications use ORB facilities and how server applications implement servants [PHA-98].

OMG IDL language mappings exist for several programming languages. OMG has standardized language mappings for C, C++, Smalltalk, COBOL, Java, and Ada. Other language mappings exist as well. For example, mappings have also been independently

defined for languages such as Eiffel, Modula 3, Perl, Tcl, Objective-C, and Python- but at this time they have not been standardized by the OMG [MHE-99].

IDL language mappings are critical for application development. They provide concrete realizations of the abstract concepts and models supplied by CORBA. A complete and intuitive language mapping makes it straightforward to develop CORBA applications in that language; conversely, a poor, incomplete, or ineffective language mapping seriously hampers CORBA application development. Official OMG language mapping specifications therefore undergo periodic revision and improvement to ensure their effectiveness [MHE-99].

The existence of multiple OMG IDL language mappings means that developers can implement different portions of a distributed system in different languages. For example, a developer might write a high-throughput server application in C++ for efficiency and write its clients as Java applets so that they can be downloaded via the Web. The language independence of CORBA is key to its value as an integration technology for heterogeneous systems.

2.7 Object Adapters

An object adapter is the primary way that an object implementation accesses services provided by the ORB. Services provided by the ORB through an Object Adapter include: generation and interpretation of object references, method invocation, security of

interactions, object and implementation activation and deactivation, mapping object references to implementations, and registration of implementations. The wide range of object granularities, lifetimes, policies, implementation styles, and other properties make it difficult for the ORB Core to provide a single interface that is convenient and efficient for all objects. Thus, through Object Adapters, it is possible for the ORB to target particular groups of object implementations that have similar requirements with interfaces tailored to them [PSA-98].

In CORBA, object adapters serve as the glue between servants and the ORB. An object adapter is an object that adapts the interface of one object to a different interface expected by a caller. In other words, an object adapter is an interposed object that uses delegation to allow a caller to invoke requests on an object without knowing the object's true interface.

CORBA object adapters fulfill three key requirements.

1. They create object references, which allow clients to address objects.
2. They ensure that each target object is incarnated by a servant.
3. They take requests dispatched by a server-side ORB and further direct them to the servants incarnating each of the target objects.

Without object adapters, the ORB would have to directly provide these features in addition to all its other responsibilities. As a result, it would have a very complex

interface that would be difficult for the OMG to manage, and the number of possible servant implementation styles would be limited.

In Java, servants are instances of Java objects. They are typically defined by deriving from skeleton classes produced by compiling IDL interface definitions. To implement operations, we override functions of the skeleton base class. We register these Java servants with the object adapter to allow it to dispatch requests to the servants when clients invoke requests on the objects incarnated by those servants.

Until version 2.1, CORBA contained specifications only for the Basic Object Adapter (BOA) [JSE-01]. The BOA was the original CORBA object adapter, and its designers felt that it would serve for the majority of applications, with other object adapters filling only niche roles. However, CORBA did not evolve as expected because of the following problems with BOA specification.

- The BOA specification did not account for the fact that, because of their need to support servants, object adapters tend to be language-specific. Because CORBA originally provided only a C language mapping, the BOA was written to support only C servants. Later attempts to make it support C++ servants proved to be difficult. In general, an object adapter that provides solid support for servants in one programming language is not likely to also provide adequate support for servants written in a different language. This is because of differences in implementation style and usage of those servants.

- A number of critical features were missing from the BOA specification. Certain interfaces were not defined and there were no servant registration operations. Even those operations that were specified contained many ambiguities. ORB vendors developed their own proprietary solutions to fill the gaps, resulting in poor server application portability between different ORB implementations.

The Portability Enhancement RFP issued by the OMG in 1995 to address these issues contained a seven-page listing of problems with the BOA specification.

CORBA version 2.2 introduced the Portable Object Adapter (POA) to replace the BOA. Because the POA addresses the full gamut of interactions between CORBA objects and programming language servants while maintaining application portability, the quality of the POA specification is vastly superior to that of the BOA. As a result, the BOA specification has been removed from CORBA.

2.7.1 Implementation of Object Request Broker

In the architecture, the ORB is not required to be implemented as a single component, but rather it is defined by its interfaces. Any ORB implementation that provides the appropriate interface is acceptable [MHE-99]. The interface is organized into three categories:

1. Operations that are the same for all ORB implementations
2. Operations that are specific to particular types of objects
3. Operations that are specific to particular styles of object implementations

Figure 2.1: Common Object Request Broker Architecture

Different ORBs may make quite different implementation choices, and, together with the IDL compilers, repositories, and various Object Adapters, provide a set of services to clients and implementations of objects that have different properties and qualities. There may be multiple ORB implementations (also described as multiple ORBs) which have different representations for object references and different means of performing invocations. It may be possible for a client to simultaneously have access to two object references managed by different ORB implementations. When two ORBs are intended to work together, those ORBs must be able to distinguish their object references. It is not the responsibility of the client to do so. The ORB Core is that part of the ORB that provides the basic representation of objects and communication of requests. CORBA is designed

to support different object mechanisms, and it does so by structuring the ORB with components above the ORB Core, which provide interfaces that can mask the differences between ORB Cores.

2.7.2 Reference Acquisition

Object references are the only way for a client to reach a target object. A client cannot communicate unless it holds an object reference. How, then, does a client obtain references, because the client must have at least one reference to start with? References have to be published by the servers in one of the following ways:

- Return a reference as the result of an operation (as the return value or as an inout or out parameter)
- Advertise a reference in some well-known service, such as Naming or Trading service
- Transmit an object reference by converting it to a string and writing it into a file.
- Transmit an object reference by some email or Web page

2.8 Proxies

When a reference is received by a client, the client-side run time instantiates a proxy object in the client's address space. A proxy is an instance that supplies the client an interface to the target object. The interface on the proxy is the same as the interface on the remote object; when the client invokes an operation on the proxy, the proxy sends a

corresponding message to the remote servant. In other words, the proxy delegates requests to the corresponding remote servant and acts as a local ambassador for the remote object, as shown in Figure 2.2 (a).

The language mapping does not change if client and server are collocated in the same address space. In particular, no changes to the source code are necessary in either client or server if we decide to link the server into the client. See Figure 2.2 (b).

If client and server are collocated, the client's request is still transparently forwarded by the proxy to the correct servant; in this way we preserve the location transparency of CORBA [MHE-99].

In both the remote and the collocated scenarios, the proxy delegates operation invocations made by the client to the servant. In the remote scenario, the proxy sends the marshaled request over the network, whereas in the collocated scenario, the request is dispatched through function calls.

2.9 General CORBA Application Development

To develop a CORBA application the following steps need to be followed. Based on the requirements of the application, we define the interfaces of the objects that we will have using Interface Definition Language (IDL).

1) We compile the IDL definitions into stubs and skeletons using the *IDL compiler* provided by the ORB vendor.

2) We declare and implement servant classes that can incarnate our CORBA objects.

3) We write a server main program that instantiates the CORBA classes and initializes the ORB during runtime.

4) We compile and link our server implementation files with the generated stubs and

(a)

Client and Server

(b)

Figure 2.2 Proxy represents the actual Server in CORBA environment

(a) Proxy to collocated object

(b) Local proxy to remote object

skeleton to create our server executable

- 5) Similarly we write, compile and link our client code together with the client stubs generated in step 2.**

Naturally, these steps vary somewhat depending on the nature of the application. For example, sometimes the server already exists, and we need write only the client. In that case, we would perform only those steps related to developing clients.

CHAPTER 3

Characterization of Middleware

3.1 Introduction

The development of applications in distributed systems has fundamentally been driven by two strong desires. The first one is the desire to use a remotely located component intelligence that is easily accessible due to the omnipresence of the distributed systems, and the development of modular re-usable component technology. The second desire is to create new applications by integrating legacy systems, databases and components in very short time frames, instead of re-inventing the wheel. Middleware provides interoperability in such a heterogeneous environment consisting of diverse components as well as a set of generic services. It allows distributed component technology to be developed and integrated into new applications [WRU-00]. The software industry has provided a wide variety of middleware technologies. Much of what has been available is proprietary to specific vendor, thereby locking a developer into its solution. However, this has been changing through the efforts of groups such as the Object Management Group (OMG) that have provided a forum for the creation of middleware standards.

Distributed computing systems consist of application processes executing on multiple machines. Middleware glues the various components of this distributed computing system together. It allows application processes to transparently collaborate across processes and networks despite different policies, operating systems, programming languages, machine data formats, and networking protocols [OMG-98].

Middleware specifies the standard interfaces between the interacting nodes, and provides the standard transport mechanism for the flow of information. It acts as a broker that helps perform the requested tasks by the distributed nodes. Middleware essentially provides an improved technology for the developer to build distributed applications. It provides the ability to focus on the semantics of the integration, rather than the mechanics [WRU-00].

3.2 Middleware Requirements

To achieve its goals, Middleware has certain requirements, which must be met to provide a stable, practical and viable solution. The requirements can be summarized as follows:

3.2.1 Scalability and Load Balancing

As the number of users in a distributed system increases, limited system resources are consumed and performance bottlenecks result. To relieve the bottlenecks, resources might be duplicated or routing controls might be provided that direct traffic to the least used

system resources. Ensuring that system resources are used in a balanced manner is essential for an efficient system. Middleware scalability depends on the efficient utilization of critical physical resources in variable load situations, and should be an important consideration in designing a system [PHO-94] [SDH-94].

3.2.2 Synchronization and Co-ordination

Distributed environments require synchronization and coordination services for the concurrent updates of distributed resources such as files and databases and other scenarios where shared resources are concurrently accessed. Proper control protocols or some type of standardized message signaling is essential to achieve this goal.

3.2.3 Endpoint Binding Transparency

To establish a connection or deliver a message, the transport layer requires network information. Distribution transparency requires that the middleware layer abstract the mapping of application layer software components from the physical network endpoints. References or binding handles provide a mechanism for encapsulating a physical network endpoint with a system-level name [LKE-75].

3.2.4 Ease of Use

Middleware should shield application developers from low-level network programming details. It should create and manage the semantics of the interaction between a client and server [TBR-95].

3.3 Heterogeneity of a System

3.3.1 Concept of Domains in a System

A domain is a collection of components that are unified because they share specific characteristics. Typically, these characteristics are related to some aspect of system operations, such as networking, security, or transactions. They can be oriented around programming considerations such as languages, references, or representations. However, the definition of domains has no bounds. They can be oriented around

Figure 3.1: The concept of CORBA domains [WRU-00]

applications, functionality, organization, products, or any logical inclusion that is required by a system [WRU-00]. Domains can be temporal or geographical in nature.

In the CORBA world a domain has a distinct scope. Within the scope of the domain, the ORB transparently handles interoperability. This is, because common characteristics and rules exist for each domain, usually defined by a vendor. Interoperability is required between different domains with different characteristics or rules via bridging. Domain differences can be administrative, related to such areas such as naming or security, or related to technological aspects, naming or syntax. A common and practical example is the case of a heterogeneous environment having multiple ORBs. The system can be divided into conceptual domains if an organization wants to integrate two applications that are built on two different ORBs. It is also likely that a single vendor's ORB might allow for multiple domains, especially of different administrative types. For example, a vendor might provide security services. If an organization uses the same vendor's ORB for developing applications, some with the security services and others without, then each application can be viewed as a domain. It is possible that ORBs from several vendors may interoperate natively and be used within a single domain

Bridging is a concept whereby a translation is made from the internal communication and information structures to those of either the intended recipient or common transient format. Using bridging, the ORBs interoperate through a common communications mechanism that might differ from the communication mechanism used within the ORB.

The bridge acts as a translator between the two. There are two types of bridges: immediate and mediated. An immediate bridge translates directly into the internal form of the other ORB. This works only when the internal structure is well known in advance. A mediated bridge can be viewed as a translator in which relevant dialogue between the ORBs is performed by converting the native formats into a neutral standard. The translation is done at both ends of the communication channel. Therefore each ORB is insulated from knowing the specific implementations of another ORB.

IOP is the common language for broad-based, mediated bridging. That is, it is the communications mechanism used by ORBs to communicate with each other. When one ORB is using IOP as its internal communication mechanisms, then immediate bridging is easily realized.

This concept is shown in Figure 3.1. Three types of domain are shown. Nodes within a domain can interact with each other by using any proprietary protocol they like as long as they agree on it (i.e immediate bridging). But, for inter-domains communication, standard inter-ORB protocol, such as IOP, has to be used.

3.3.2 Analysis of a Typical System

Thus domains provide a view of a system as being organized into common spheres of concern. It is at the intersection of these domains that interoperability is most important to manage. Heterogeneity is an inherent characteristic of any distributed system. Software

developed for any distributed system must deal with problems such as the failure of some of the systems in the network, partitioning of the network, problems associated with resource contention and sharing, and security-related risks keeping heterogeneity in mind.

3.3.3 Characterization of Heterogeneity

Dissimilarities between nodes in a system can be characterized at three levels:

a) Hardware Level

Different nodes within a distributed system may use different hardware for processing. There are differences in how different processors manipulate data storage in memory, thus giving rise to the “big-endian” and “little-endian” machines. For example, Intel processors are little-endian, and the processors made by Sun are big-endian. A conversion has to take place from little-endian to big-endian or vice versa for the correct interpretation of variables when a Sun machine contacts an Intel machine. Thus, hardware characterizes a level of heterogeneity in a system.

b) Operating System Level

Operating Systems used on communicating nodes may define another level of heterogeneity. For example different OS may define a variable by a different number of bits e.g int type can be defined by 32 or 64 bits. For example, the Solaris operating system is a 64 bit system, whereas Linux running on an Intel processor is 32 bit system.

c) Programming Language Level

Programming language used on the operating nodes may define another level of heterogeneity. For example, there are many data structures available in C++ that are not available in Java and vice-versa. Similarly, there is no concept of multiple-inheritance or pointers in Java. So for Java and C++ to understand each other, these concepts have to be converted into a neutral format and then understood by each other. If these nodes are using the same Programming Language, they can operate with each other in their native paradigm without any need for a neutral platform.

3.3.4 Concept of Flyovers in a System with Limited Heterogeneity

CORBA is a solution that provides interoperability for these heterogeneous nodes in a distributed system. But it also introduces overheads. Most of the interacting nodes tend to be similar to one another in many systems. As noted in Section 3.3, if we can develop some performance enhancement strategies that can selectively by-pass the abstractions provided by CORBA, while dealing with two homogeneous nodes, we can exploit this limited heterogeneity to improve performance. The advantage obtained will depend on the number of nodes that are similar to another.

3.4 GIOP

3.4.1 Introduction

The CORBA specifications define the mechanisms to register and access distributed objects. CORBA 1.1 gave us the first guidance relating to ORB interoperability. It defined interoperability and the organization of multiple ORBs. It did not include low-level protocol requirements for the interoperation of CORBA ORB products from different vendors. The OMG recognized this problem and began examining alternative solutions.

The General Inter-ORB Protocol (GIOP) specification was the result of this effort.

The GIOP specification has three primary parts:

- Transfer syntax
- Message formats
- Transport layers assumptions.

3.4.2 GIOP's Transport Assumptions

GIOP makes the following assumptions about the underlying transport that is used to carry the messages [OMG-99] [OMG-01].

(a) Connection-Oriented Transport

A connection-oriented transport allows the originator of a message to open a connection by specifying the address of the receiver. After a connection is established, the transport returns a handle to the originator that identifies the connection.

(b) Reliability of Transport Protocol

GIOP does not have the facilities to check for packet ordering, corrupt packets, lost packets, or duplicate packets. These tasks are the responsibility of the transport layer. Usually a transport layer such as TCP supports this requirement. It is responsible for guaranteeing that a correct message travels from the client to the server and vice versa.

(c) Provision of “byte-stream abstraction” by Transport Layer

The transport does not impose limits on the size of a message and does not require or preserve message boundaries. In other words, the receiver views a connection as a continuous byte stream. Neither the receiver nor the sender needs to be concerned about issues such as message fragmentation and retransmission.

(d) Notification of a Connection Loss

When a disorderly connection loss occurs, such as a client crashes or the network goes down, the transport layer must give the server some reasonable notification of this event. There may be some issues that may be looked at first. For example, a server may be multithreaded so that it handles many client connections at once. The connections are limited in number due to the resource limitations, so each connection can be quite valuable. Sometimes a client that has obtained a connection with a server develops a problem and can no longer communicate with the server. The server should immediately drop the connection to free it up for another client. If there were no proper mechanism for

notification to the server that a connection no longer exists, the server would have to enforce a client polling or timeout mechanism.

3.4.3 The Common Data Representation

The transfer of information requires a common syntax. The formal mapping for data types is captured with Common Data Representation (CDR) definitions. GIOP is a vehicle for passing the data that has been defined by using CDR. It specifies eight messages that have enough functionality that they can be used to solve any distributed problem. As already noted, GIOP further specifies its requirements for a transport layer. It needs these supports from a transport layer so that the system operates correctly [OMG-01]. OMG defined CDR so that it supports the following features:

(a) Variable byte addressing:

The sender does not have to undertake byte swapping to ensure that the data is in the correct order for the receivers; the receiver is responsible for doing this. The sender does not have to know what kind of machine is on the receiving end of its messages. This is the receiver's responsibility to ensure that its byte ordering matches the sender's byte ordering. It will swap the bytes if its internal representation is different from the client that interacted with it.

(b) Data alignment:

All data is aligned on its natural boundaries within a GIOP message. CDR defines alignment policies only for primitive types, but all complex types can be broken down into the constituent simple types.

(c) Complete IDL mapping

The data types defined in CDR have a one-to-one relationship with the data types defined in IDL. It means that all data types of IDL can be directly defined in CDR. Prior to the IIOP standard there were proprietary Inter-ORB protocols developed by individual vendors that did not always have all data-types defined in IDL. So these data-types had to be defined indirectly in terms of other data types. Complete IDL mapping in CDR means simple and more efficient data representation.

3.4.4 Connecting Client and Server

The CORBA standard connects two distributed nodes in the client/ server paradigm. The transfer of data between these distributed nodes may be connection-oriented or connectionless. When the transmission of data occurs from one process to another without prior coordination and planning it is called connectionless. Connectionless data transfer is intended for exchanging single packets of information. In a connectionless paradigm we give up some reliability in exchange for performance. If a router detects an error in a packet (say, the checksum is not correct), it can discard the packet without any concern for a successful transmission.

When a network task requires lengthy exchanges of data, as in a typical client-server interaction in CORBA, establishing a connection is preferred, both for performance and reliability. There are three basic phases that occur in a connection [MHE-99]:

- 1) The connection is established.
- 2) The data is transferred.
- 3) The connection is terminated.

The transport layer provides a channel for the exchange of data between components in a system. Connection-oriented transports provide two actively interacting components with a bi-directional stream. The stream coordinates the sending and receiving of data. In addition, connection-oriented protocols define the roles of client and server in a particular session. The client is the node that initiates a session and requests an operation on the server node. The roles of client and server are relative to a particular session only and may change for the next session.

In order for a CORBA client to use resources on a remote object, a connection to that object must be established. To open a connection to an object, a client must first obtain the object's object reference. A naming or trading service provides this capability by storing object references, together with information describing them.

The CORBA specification does not restrict the vendor in the ways that it implements these specifications. Usually vendors implement the CORBA specifications as a set of

software libraries, executables, and interaction rules. The set of libraries, along with a daemon process running in the domain, constitutes the ORB, (although specific implementations might vary). The ORB daemon is responsible for finding the object implementation pointed to by an object reference and establishing a connection between the client process and the server process [OMG-01].

GIOP defines the agents in a connection as client and server, where the client initiates the connection and sends requests over it. The server uses this connection to reply to the client. A connection can be closed via an orderly shutdown or an abortive disconnect. The server does an orderly shutdown when it reliably sends a “CloseConnection” message. The client, too, may perform an orderly shutdown on a connection. In this case, there might be pending requests to one or more servers. When the client closes the connections, the server simply cancels any out-standing requests from the client. A server may shutdown a connection once it issues a “CloseConnection” message reliably. Any pending request by clients will not be processed. There are eight possible message types defined by GIOP. But “reply” and “request” are the two messages that can be viewed as the actual work-horses of this protocol. Approximately 95% of all CORBA transactions happening today consist of these two messages [MHE-99].

3.4.5 Marshalling/Unmarshalling

Marshalling is the process of converting the platform dependent data representation to a neutral format data representation (which is CDR for IIOP). Unmarshalling is the

opposite of it. There are overheads associated with each marshalling and unmarshalling operation. When a client wants to pass parameters to the server, these parameters must be converted into CDR (i.e. marshalling), and converted again at the server nodes back to the platform dependent format (i.e. unmarshalling) to decipher the sent parameters. Marshalling/unmarshalling often involves data-copying operations, and introduction of padding, which leads to overheads.

3.5 Specializing GIOP into IIOP

The CORBA specification defines the GIOP as its basic interoperability framework. GIOP is not a concrete protocol that can be used directly to communicate between ORBs. Instead, it describes how specific protocols can be created to fit within the GIOP framework. IIOP is one concrete realization of GIOP. It is the most popular implementation of GIOP which enables clients and object implementations to communicate over TCP/IP [WRU-00].

TCP/IP is a connection-oriented protocol whereby a receiver listens for a request from a sender at a specific address.. The address of the receiver is a combination of an IP host address and a TCP port number. Multiple receivers may exist on a single host, each with its own port address. The host address is used to locate the right computer on the network.

The port number is the address at which the particular application is found at the computer defined by the host address.

3.5.1 Connection Mechanism

The client initiates a request by finding the address for the receiver and requesting a connection. The receiver may accept or reject the connection request. An IIOP implementation requires that a TCP/IP connection to be established between the sender and receiver in order for them to communicate GIOP messages and CDR formatted information.

Once a connection is established, GIOP messages may be sent between the sender and receiver. These messages are used to represent interactions between a client and object implementation. Once the request is completed, the TCP/IP connection is closed.

Objects must be able to represent themselves as being accessible through IIOP. The representation is described using the interoperable object references (IOR). The IOR is a data structure that is visible and maintained by the ORB. It provides information on type and protocol support. The IIOP IOR allows an object to identify the version of IIOP supported, the IP host and TCP port numbers, and a unique key to identify the object by the receiving ORB as shown in Figure 3.2.

```

module IIOP {
    struct Version {
        octet major;
        octet minor;
    };
    struct ProfileBody_1_1 {
        Version iiop_version;
        string host;
        unsigned short port;
        sequence<octet> object_key;
    };
    struct ProfileBody_1_1 {
        Version iiop_version;
        string host;
        unsigned short port;
        sequence<octet> object_key;

        sequence <IOP::TaggedComponent> components;
    };
};

```

Figure 3.2: IIOP IOR Structure [WRU-00]

3.6 Details of GIOP Messages

There are eight types of GIOP messages. These are listed in Table 3.1 and shown in Figure 3.3, together with those who may issue them, their values, and in which version of GIOP they are available. They fall into two categories:

- Administrative
- Object Invocation

Table 3.1: GIOP Messages

Message Type	Issuer	Enum Value	GIOP Versions
Request	Client	0	1.0, 1.1
Reply	Server	1	1.0, 1.1
CancelRequest	Client	2	1.0, 1.1
LocateRequest	Client	3	1.0, 1.1
LocateReply	Server	4	1.0, 1.1
CloseConnection	Server	5	1.0, 1.1
MessageError	Both	6	1.0, 1.1
Fragment	Both	7	1.1

The administrative messages are:

LocateRequest and LocateReply, used to find objects so that they can use them.

CancelRequest and CloseConnection, used to handle requests that are taking too long to execute or that are no longer desired.

MessageError, used for error handling.

Object invocation messages are Request and Reply, which are used to request an operation on an object and to allow the object to reply.

GIOP Message Header

When a message is sent from a client to a server or vice-versa, information must be transmitted so that the receiver understands what the message is, for which object it is intended, and what to do with it. To this end, GIOP defines a message header that

contains such crucial information. This includes the protocol for the message, the version of the GIOP used and the message's types, byte ordering, and size.

Figure 3.4 shows the layout of the message header under the two versions of GIOP. The layout aligns all of the types naturally, so no padding bytes are needed. The header breaks down as follows:

The first 4 bytes are the magic number, which indicates the protocol. For GIOP messages, this contains the value GIOP. The next byte is the message type-Request, Reply and so

Figure 3.3: GIOP Messaging

on. The last 4 bytes are an unsigned long containing the size of the message.

```

module {
struct Version {
    octet major;
    octet minor;
};

#ifndef GIOP_1_1
//GIOP 1.0
enum MsgType_1_0 {
//Renamed from MsgType
    Request,    Reply,    CancelRequest,    LocateRequest,    LocateReply,    CloseConnection,
MessageError
};

#else
//GIOP1.0
struct MessageHeader_1_0 { //Renamed from MessageHeader
    char magic[4];

```

Figure 3.4 Definition of GIOP message Header [WRU-00]

The GIOP message header is sent on every GIOP message that flows to and from clients and servers. After each message header is another header that is specific to the message type. This specific message is indicated to the recipient by the primary GIOP message header. Figure 3.5 shows the GIOP header in detail. A discussion of the different parts of the messages, giving more details, is provided.

(a) Magic Value

The ORB uses the magic value to identify GIOP messages. This value always contains the value “GIOP” in all uppercase letters and encoded in ISO Latin-1.

(b) GIOP Version

There are currently two versions of the GIOP protocol, 1.0 and 1.1. The major GIOP version number of the CORBA core specification is 1, and the minor versions are 0 and 1. These values are encoded as octets in the `GIOP::Version` structure. The `GIOP_Version` member of the request message header contains the version number of the GIOP protocol that the message uses. The version number applies to the GIOP's CDR and message formats. The GIOP version number and IIOP version number have the same structure, but they are not equivalent.

(c) Byte Order

`Byte_order` is a member of the message header only in GIOP 1.0. It indicates the byte ordering used in subsequent elements of the message (including `message_size`). A value of `FALSE` (0) indicates big-endian byte ordering and `TRUE` (1) indicates little-endian byte ordering.

(d) Message Type

The `message_type`, as its name implies, indicates the type of the message. The actual value of the message type member is the corresponding enumeration value of type `MsgType_1_0` if GIOP 1.0 is used or `MsgType_1_1` if GIOP 1.1 is used. The member `message_type` is an enumeration, so its representation should be long. The OMG, however, chose to encode it differently. One byte instead of 4 can be used to represent all message types, so the OMG chose to represent it in this way in order to improve call efficiency.

(e) Message Size

The `message_size` member holds the number of octets in the message following the message header. The message size is the size of the message body, minus the 12-byte message header and including any alignment gaps. As already noted, the message size is encoded using the byte order specified in the byte order bit.

Anatomy of a Request Message

Of the eight messages that GIOP has, only reply and request messages are the actual work-horses, and are used 90% of times [DSC-98.1]. To discuss a third performance enhancement strategy, an explanation of these two messages is necessary. We will take one of the messages, the Request Message and will analyze it in detail. The other one, the Reply message is very similar.

Request Message

When a client invokes a CORBA operation, that invocation is encoded as a Request Message. For example we use the following IDL:

```
interface foo{  
    long run(in short bar);  
}
```

If a client invokes the `run` operation on an object reference to a `foo` instance, a Request message will be created and sent from the client to the `foo` instance. When the `foo` instance is finished with the `run` operation it will reply to the client with a Reply message.

Figure 3.5 Order of a Request Message Stream

There are three components in a Request message, encoded in the following order: GIOP message header; Request header; and Request body.

Request Header Structure Figure 3.5 shows the layout of the Request header, in the two versions of GIOP. The Request header contains information that specifies which operation is being invoked on which object. This information is held in the object key and

```
// GIOP 1.0
struct RequestHeader_1_0 {
    IOP::ServiceContextList service_context;
    unsigned long request_id;
    boolean response_expected;
    sequence <octet> object_key;
    string operation;
    Principal requesting_principal;
};

//GIOP 1.1
struct RequestHeader_1_1 {
```

Figure 3.6 GIOP Request Header [WRU-00]

operation field. The response-expected field indicates if the sender expects a response from the recipient. The service-context field passes any special information, such as the sender's user name. The requestID is used to match up the corresponding Reply message. The requesting principal is used for security purposes to indicate the client issuing the request. Version 1.1 of GIOP added a reserved field for future use.

Service Context

The service context contains object services data that are passed from the client to the server.

RequestID

When a client sends a Request or LocateRequest message, it usually expects a reply. Since a client might send multiple requests to a server and the server might reply to them in different orders (assuming a multithreaded client and server), a requestID is used to associate a particular request to a particular reply. This ID must be a value that prevents the possibility of a reply to a request being used for a wrong request. Because it is the client that uses requestIDs, it is responsible for generating them in a unique way. Specifically, the client must not reuse a requestID value during a connection if the previous request containing that ID was canceled and no reply was received.

Response Expected

Most CORBA clients issuing a request on a server expect a reply and indicate this by setting the response-selected flag to TRUE (1). In some cases, however, a client sends a

request and neither expects nor wants a reply from the server. If the client is invoking a one-way request or if the invocation was via the Dynamic Invocation Interface (DII), with the INV_NO_RESPONSE flag set, it might choose not to receive a response from the server. In both these cases the response-expected flag is set to FALSE (0). However, a oneway or DII call may also be done when the client expects a response. If the flag is set to TRUE (0) , the client can receive replies that indicate both system exceptions and LOCATION_FORWARD responses. When a server is executing a one-way operation and the response-expected flag is set to TRUE (1) it may send a reply to the client as soon as it receives the request, but before the operation completes. For example, a response may be an acknowledgement or a LOCATION_FORWARD response.

Reserved

This numeric field was added in GIOP 1.1. It is always set to 0 and is reserved for future expansion.

Object Key

In CORBA, a client invokes operations on an object contained in a server. Since servers may contain more than one object, the client's knowledge of the port number, hostname, and process ID of the server is not enough. It also must know the target object: i.e the object that is being invoked on. This object key or ID information is stored in the objectkey member of the Request header. The object key is a server-only value and has no meaning to and may not be modified by the client.

Operation

A client invokes an operation on an object. Thus, in addition to its identifying the target object in the request, it must indicate which operation on that target object it is trying to invoke. The mechanism for identifying the operation in the message header is to use an IDL identifier naming within the context of the interface, but not a fully qualified scoped name.

3.7 CORBA: Not an ideal Middleware

In spite of all the advantages and benefits that it provides, CORBA is not an ideal Middleware. There are certain deficiencies in the CORBA specifications which are described in [WRU-00] and are summarized below.

3.7.1 The reality of IIOP Interoperability

Typically a CORBA-based system constitutes a single ORB vendor product. As the system grows or needs to have more than one domain due to some other reason, the system evolves into a multi-ORB system. Interoperability between these multi-vendor ORBs is a major consideration. For example, if we use the ORB of one vendor and the naming service of another vendor, there are chances that we may run into some problems due to different methods of storing the object IOR in the naming service by different vendors.

Even though IIOP and GIOP specify a standard interoperability protocol, obstacles still exist to seamless interfacing between products from different vendors [WRU-00]. The act of establishing a connection between a client and a server creates an object reference on the client. As noted in the previous section, an object reference that can work between different vendors' ORBs is an IOR. It is a metadata about an object that describes its location and name. It is on this object reference that the client invokes all of the operations on the target object. The object reference then delegates the call to the real object, which is located somewhere on the other side of the connection. Different vendors may have slightly different strategies for handling location independence of IORs, which can affect interoperability. While it is possible for clients and servers from different vendors to connect and interoperate, using correctly formed IORs, other unsolved problems remain. Foremost among these are server location and initial bootstrapping of a connection [WRU-00]. These issues become more difficult to solve when clients and servers do not have any prior knowledge of each other.

3.7.2 CORBA Messaging

CORBA offers two modes of operation invocation: synchronous and deferred synchronous. During a synchronous operation, the caller is blocked until the operation is finished and a reply is returned. This type is the only invocation method available to statically compiled stubs. During a deferred synchronous operation the caller continues execution after the call is made, but continues polling for the operation's results. Deferred synchronous operations are available only by using the Dynamic Invocation Interface

(DII). There is a need for truly asynchronous communications [DSC-96] [DSC-97]. It is one of the deficiencies of the current CORBA specifications and, if included in future, will give developers more flexibility and power when developing applications.

3.7.3 Performance

With all the benefits that the CORBA offers, the overheads it adds may seriously limit its benefits, especially in performance-thirsty or real-time applications where performance of a system is crucial. OMG has compiled two different specs, “Minimal CORBA specs” and “Real-time CORBA specs” for such applications. “Minimal CORBA” specification provides an optimized and smaller version of CORBA at the cost of features. The Real-time CORBA specification stipulates the guidelines for the implementation of CORBA that can be used in real-time applications.

CHAPTER 4

Performance Enhancement Techniques

4.1 Overview

This thesis presents three performance enhancement techniques for CORBA-based middleware with limited heterogeneity. Two of the performance enhancement techniques presented in the thesis are appropriate for data intensive applications, while the third is effective when short messages are exchanged between the client and the server. This chapter presents the background, design and implementation of these performance enhancement techniques.

4.2 Methods Used in Performance Enhancement

The purpose of this research is to explore the possibilities of the enhancement of CORBA performance under certain system conditions. This requires access to the core ORB code, as well as provision of some plumbing into the ORB. In particular,

- We must have *accessibility to the ORB source code*
- We should be able to *customize Event Handling* to invoke special code.

To meet these requirements, the basic strategy used in this research was the use of an Open source ORB and Interceptors. An Open source ORB was used in order to have accessibility to the source code of the ORB, and the Interceptors were used in order to have the customized ORB event handling.

This section explains the methods employed for enhancing the performance of such an open source ORB. Some methods, such as the use of Interceptors for time-stamping ORB routines, were developed during this research and were extensively used throughout the experiments.

4.2.1 Using Interceptors for Customized Event Handling

Background

CORBA implementations have long had proprietary mechanisms that allowed users to insert their own code into the ORB's flow of execution. This code, which is sometimes known as "Filters", "Transformers," or "Interceptors," is called at particular stages during the processing of requests and may directly inspect and even manipulate requests. For example, it is possible to monitor the requests mediated by the ORB. In addition, most implementations also allow us to modify request arguments, redirect requests to different target objects, or encrypt and decrypt message buffers before and after transport.

Because this message filtering mechanism is extremely flexible and powerful, the OMG made an attempt to standardize Interceptors in CORBA 2.2 specification. The results of this first attempt, however, were not satisfactory. The specification proved to be quite

loose, and different interpretations by various vendors resulted in Interceptor implementations that were not interoperable with each other [GBR-01]. A second attempt at specifying Interceptors was carried out more carefully and resulted in a much more detailed document which precisely stipulated the specification of Interceptors in detail. These Interceptors were called “Portable Interceptors” because, due to precise specification of their implementation, different implementations are interoperable and it is possible to port the Interceptors from the ORB of one vendor to the ORB of another vendor [JSE-01].

Specification

Interceptors are designed to allow the insertion of user-defined code into the ORB’s flow of execution. Interceptors are implemented as objects of particular classes, which are called at specific points during the processing of operation invocations. To “insert” our own code into the invocation path, we define subclasses of Interceptor classes and register instances with the ORB [GBR-99]. This gives access to the request itself and provides a convenient and powerful way of observing and/ or modifying requests. On the other hand, it requires more detailed knowledge about ORB data structures than necessary for application level code. Also, it adds additional code into the normal CORBA transaction execution path, thus slowing down the overall performance. Thus, Interceptors should only be used for research, debugging purposes, or for satisfying some very particular needs when there is no other alternative available.

A typical CORBA transaction consists of calling a remote method at a server by a client. That remote method will, in general, return some parameter value. If we analyze a CORBA transaction, there are eight points where Interceptors are (potentially) given control, as defined in the CORBA specs [OMG-98]. On the client side, these are points 1 & 2 (see Figure 4.1), before the request is sent across the network, and points 7 & 8 after the result is received and before control is returned to the caller. The Interceptor has the access to the structured request object at point 1. At point 2, the message can be accessed in marshalled form in a buffer just before it is sent onto the network.

On the server side, intercepting requests can also be done at different levels (point 3,4 and 5,6). For every single one of these eight points, a different method is called on a different Interceptors object. Thus, Interceptors can be defined at two levels: at the request level or at the message level. *Request-level Interceptors* are given access to a request object representing the current request, and may access and modify this request before and after

Figure 4.1: Different Stages in a typical CORBA transaction

it is actually invoked. *Message-level Interceptors* have access to the actual message buffer (marshalled data) at the client node before it is send to the server. Each of these Interceptor types is briefly discussed.

(a) Request-Level Interceptors

The ORB calls Request Interceptors on the client and the server side. This is done for both outgoing and incoming requests and results. This corresponds to point 1,8, 4 and 5 in Figure 4.1. The main purpose of Request Interceptors is to manipulate service context information, but they can also carry out many other useful tasks. The ClientRequestInterceptor interface, as shown in Figure 4.2, provides signatures of the methods that correspond to the five operations that can be performed when the Request Level Interceptor is invoked.

The operation `send_request` is called for any ORB-mediated, outgoing request. The Interceptor may raise a `ForwardRequest` exception to signal that the ORB should retry the operation with a new object reference that is provided in the body of the exception. In this


```
interface ClientRequestInterceptor:Interceptor{
    void send_request (in ClientRequest ri) raises (ForwardRequest);
    void send_poll (in ClientRequestInfo in);
    void receive_reply ( in ClientRequestInfo ri);
    void receive_exception (in ClientRequestInfo ri) raises (ForwardRequest);
    void receive_other ( in ClientRequestInfo ri) raises (ForwardRequest);
};
```

Figure 4.2: ClientRequestInterceptor interface

case, no other Interceptor will see the request at this interception point. Rather, the `receive_other()` operation will be called to signal the `ForwardRequest` exception. If a system exception is raised, the `receive_exception( )` operation of any remaining Interceptors is called instead. The `send_poll()` operation is called when clients poll for the results of deferred synchronous invocations. For a normal operation, we will use the `send_request` operation to invoke any piece of code just before the marshalling of the parameters.

(b) Message-Level Interceptors

Sometimes we need to invoke certain code after the marshalling routine has converted the native type parameters into the CDR format at the client node, or before unmarshalling at the server node, i.e., before the CDR format has been converted back into the native data types. Under these situations, we can use the Message-level Interceptors. We can invoke

these type of Interceptors in response to both a send message or a receive message. In the case of a send message, the ORB code invokes each message-level Interceptor via the `send_message` operation (when sending a message) or the `receive_message` operation (when receiving a message). Both operations have a message as an argument. The Interceptor generally transforms the message and then invokes `send`. Send operations return control to the caller without waiting for the operation to finish. Having completed the `send_message` operation, the Interceptor can continue with its function or return [SEI-98].

These Interceptors can be used to access and modify the output stream buffer at the client side and the input stream buffer at the server side.

In general, both types of Interceptors discussed earlier provide a highly flexible means of adding portable ORB Services to a CORBA-compliant object system. The flexibility derives from the capacity of a binding between client and target object to be extended and specialized to reflect the mutual requirements of the client and the target. The portability derives from the definition of the Interceptor interface in OMG IDL [BMC-98].

4.3 Performance Enhancement Strategy 1: Preventing Native-Neutral Data Conversions

4.3.1 Background

The marshalling/ unmarshalling engines in the ORB perform data conversions between the native format used in the client/server implementation and the CDR format defined in

CORBA. If we know in advance that both the client and the server use the same native data format it may be possible to avoid this unnecessary conversion from native to CDR at the client machine, and from the CDR to the native at the server machine. The avoidance of such unnecessary data conversions can reduce the data marshalling and unmarshalling overheads. We can save both CPU cycles expended for data conversion as well as avoid expensive data-copying operations at both the client and the server nodes used in many COTs ORBs, thus improving system performance.

4.3.2 Implementation

This technique is implemented by changing the marshalling/unmarshalling routines in the ORB source code, which is then re-compiled to produce the modified ORB. The results obtained by using this technique are discussed in chapter 5.

4.4 Performance Enhancement Strategy-2: Removing Padding

4.4.1 Background

Use of padding is popular among different communication protocols for various reasons. Presentation layer protocols such as CDR in CORBA use padding for data alignment. CDR aligns primitive data types on specific byte boundaries that are natural for most machine architectures. Each primitive type must start at a particular byte boundary relative to the start of the CDR-encoded byte stream it appears in (See Table 4.1). For example, short data types are aligned on a 2-byte boundary, longs are aligned on a 4-byte boundary, and doubles are aligned on an 8-byte boundary. Unused bytes between two

consecutive data elements are called padding bytes. Encoding data according to these alignments uses additional communication bandwidth required for the transmission of padding bytes that do not have any semantic value.

The use of padding bytes in CDR encoded messages is explained with the help of an example

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

Table 4.1. CDR alignment of primitive fixed-length types [OMG-98].

Starting Bytes Boundary	IDL Data Types
Multiples of 1	char, octet, Boolean
Multiples of 2	short, unsigned short
Multiples of 4	long, unsigned long, float, enumerated types
Multiples of 8	long long, unsigned long long, double, long double

This structure contains two primitive data types; a character, which can occur anywhere in a byte stream, followed by a double value, which must be aligned on an 8-byte boundary. There are padding bytes between the two fields in the structure. If we are able to remove the padding bytes from the output buffer of the ORB at the client side and then transmit the data, there is a potential for a performance improvement in terms of a smaller message transmission time. Note that in order for the receiving ORB to correctly interpret the data, the padding bytes must be reintroduced at appropriate places in the input buffer

at the server side. Thus, the technique gives rise to additional client/server processing and data-copying overheads. It means that the strategy is useful in communication intensive systems in which the benefit obtained by the shorter message transmission time offsets the increased computation overheads introduced by data copying and additional client/server processing. Note that this performance enhancement strategy can be used even when the two nodes are dissimilar.

4.4.2 Implementation

The performance enhancement strategy is implemented with the use of Interceptors that can be used to interact with the output buffer at the client side, and with the input buffer at the server side. At the client side the padding bytes are removed from the output buffer; then they are re-inserted at the server side. This strategy is transparent to the client and the server application code, as well as to the ORB. The Interceptor code removes the padding bytes after the message is processed by the ORB; whereas, the ORB interface at the server side receives a message in CDR form, because padding bytes are reintroduced at its input buffer.

While removing the padding at the client side, the information about padding can be stored in an array, $p(x)$. The consecutive array elements indicate the offset in the buffer stream and the corresponding number of padding bytes used. For example, if $P(0) = 3$, $P(1) = 4$, $P(2) = 9$ and $P(3) = 7$, it means that there are 4 padding bytes starting from offset 3 and there are 7 padding bytes starting from offset 9. For the experiments

described in this paper we are assuming that the structure of the "padding array" is known *a priori* at the server side.

There is an inherent computation-communication trade-off present in this technique. By invoking the Interceptors at both the client and the server sides, we are increasing client and server processing overheads. To be effective, the saving in communication times obtained with using this strategy must offset the overheads. The number of padding bytes depends on the type of data passed as parameters and their "arrangement" within the parameter list. If the proportion of padding in the data is p and the saving in communication time is s , then it is clear that:

$$s \propto p$$

Thus, if the Interceptor overheads are small, the performance improvement is expected to be directly proportional to the proportion of padding bytes in the message.

4.5 Performance Enhancement Strategy-3: Compressing Headers

4.5.1 Background

The nodes in a CORBA domain interact with each other using GIOP protocol. GIOP consists of encapsulated headers followed by the payload, which is the marshaled form of the invoked function parameters. Whether the calling node is big-endian or little-endian, the encapsulated headers contain information about the GIOP version being used and

service context information etc. We do not need all this information if we are communicating within a homogeneous domain. We can exploit this information to improve the performance of the system. We can modify GIOP headers to contain only that information that we really need. The resulting protocol will no longer be GIOP, but this customized protocol may be used to build flyovers to communicate within the homogeneous domain in an optimized fashion. To explain this enhancement technique, a detailed explanation of GIOP is presented in the next sub-section.

Note that the benefit obtained from this performance enhancement technique is inversely proportional to the percentage of payload in the total GIOP message. For a particular message, the encapsulated headers have a fixed size. Payload consists of a marshaled form of function parameters. For a smaller payload, the percentage decrease in the total message size due to this performance enhancement technique will be greater, leading to a larger potential performance benefit.

4.5.2 Implementation

Consider the communication between two nodes that are similar to one another. If we consider the GIOP module (see Table 4.2), there are some fields that we may not need. The total length of GIOP header is 12 bytes. We may not need `GIOP_version`, saving us two bytes. We may not need `byte_order`, considering, for example, that we are operating on little-endian machines on both nodes. The magic number that starts the header is needed to delineate the header bytes. Instead of writing “GIOP” we can use “GI” only for delineating purposes, saving us two bytes. So we only need `message_type` and

message_size for the communication within a homogeneous domain. Thus, instead of sending 12 bytes we will be sending only 7 bytes. This reduced size will be significant in the case of small messages. The fields are summarized in Table 4.2 along with the number of bytes contained in them.

Similar modification can be done on the reply header, and only those fields that are really needed can be used.

Table 4.2: GIOP fields that are needed in a transaction between two similar nodes

GIOP Header Field	No of Bytes used	Needed in homogeneous domain?
magic [4]	4	Can be reduced to 2 bytes
GIOP_version	2	No
Byte_order	1	No
Message _ type	1	Yes
message_size	4	Yes

CHAPTER 5

Results of Experiments on the Performance Enhancement Techniques

CORBA is used to provide the necessary infrastructure for interoperability in a heterogeneous environment. But typical distributed environments are characterized by the presence of a considerable percentage of homogeneity. As discussed in the previous chapter, the similar nodes can communicate with each other using a proprietary way of communication. This motivates the research presented in this chapter: *intra-ORB performance enhancement techniques*. These techniques can be used to build fly-overs, which are customized low-overhead protocols of communication that can be used between two similar nodes. When a node is to start communicating with other nodes for the first time, the communication event procedure may consist of one-time discovery procedure for each node in the domain, in which the nodes discover the level of homogeneity of the other nodes in the domain by querying a proprietary flag. The TRUE return value means that the contacted node uses the same platform. The calling node may choose to maintain this information (contacted node's IP address and value of Flag) in a table for future reference. FALSE or no return value means the contacted node is dissimilar, and standard CORBA invocation is to be used. Making a decision on this

information, the calling node may choose to use an optimized way of communication or may use the standard IIOP protocol for communication. This research provides the performance enhancement techniques that can be used to build fly-overs. Exact specification of the fly-overs is beyond the scope of this research and can be considered as a future extension of this research, as discussed in Chapter 6.

Inherent Trade-off

Some of our performance improvement techniques give rise to a trade-off between computation and communication times. The use of these strategies requires knowledge of system architecture and performance. Once we know that there is sufficient homogeneity present in the system to take advantage of the performance enhancement techniques, there are some other parameters to consider. It is necessary to have an idea about the location of the performance bottleneck. Based on where the performance bottleneck is, a particular strategy can be chosen and applied. For example, if inter-node communication is the performance bottleneck, reducing the length of messages at the cost of introducing computational overheads may provide an overall gain in system performance. A careful consideration of these factors is necessary before using a particular performance enhancement technique. The results of the experiments on the three performance enhancement techniques are discussed in this chapter. Before presenting these results, brief descriptions of the measurement techniques used, and the experimental environment, are presented.

5.1 Methods Used for Performance Measurements

5.1.1 Using JNI

As already noted, in certain situations during this research, very precise time stamping, in the resolution of microseconds, was required. A typical situation was to measure end-to-end response time for a request. This was not possible without having direct interaction with the operating system. We used JNI technology to interface our Java code with OS system calls.

The JNI is the native programming interface for Java that is part of the JDK. By writing programs using the JNI, we can use the platform dependent features of a particular platform. JNI allows Java code that runs within a Java Virtual Machine (JVM) to operate with applications and libraries written in other languages, such as C, C++, and assembly. In addition, the *Invocation API* allows us to embed the Java Virtual Machine into our native applications.

Figure 5.1 illustrates how we can call native language functions from a Java application that can measure ORB processing time-stamps.

Programmers use JNI to write native methods that handle those situations when an application cannot be written entirely in the Java programming language. JNI has various applications and is often used in the following scenarios:

- The standard Java class library may not support the platform-dependent

features needed by our application.

- We may already have an application library written in another programming language and we wish to make it accessible to Java applications.

Figure 5.1: JNI provides an interface between Java and C worlds

- We may want to implement a small portion of time-critical code in a lower-level programming language, and then have our Java application call these functions.

Programming through the JNI framework lets us use native methods to undertake many operations. Native methods may be written explicitly to solve a problem that is best handled outside of the Java programming environment.

The JNI framework lets our native method utilize Java objects in the same way that Java code uses these objects. A native method can create Java objects, including arrays and strings, and then inspect and use these objects to perform its tasks. A native method can also inspect and use objects created by Java application code [KSE-98]. A native method can even update Java objects that it created or that were passed to it, and these updated objects are available to the Java application. Thus, both the native language side and the Java side of an application can create, update, and access Java objects and then share these objects between them.

JNI enables us to use the advantages of the Java programming language from a native method. In particular, we can catch and throw exceptions from the native method and have these exceptions handled in the Java application. Native methods can also obtain information about Java classes. By calling special JNI functions, native methods can load Java classes and obtain class information. Finally, native methods can use the JNI to perform runtime type checking [JSE-01].

5.1.2 Measuring Marshalling Time at the Client Using Interceptors

Let us see how we can use Interceptors to measure the marshalling and unmarshalling times.

Measuring the marshalling and unmarshalling times is important for this research. Marshalling time is the time taken by the ORB marshalling routine to convert the parameters from the native representation to CDR. Similarly, unmarshalling time is the time taken by the unmarshalling routine to convert back from CDR to native. Suppose we want to measure the marshalling time of the parameters at the client side during a typical CORBA invocation. If we look at Figure 4.1, this procedure translates into measuring the time between point 1 and point 2.

We will use a *request level Interceptor* that will time stamp the time at the point 1. Say this time is t_1 . Then we will use a *message level Interceptor* at the client that will time stamp the time at point 2. Say this time is t_2 . Then $(t_2 - t_1)$ will give the total marshalling time.

This technique was used in this research extensively for time stamping different processes throughout this research.

5.2 Experimental Environment

Throughout the experimental data-gathering phase, a “quiet” network exclusively reserved for the purpose of experiments was used. This made sure that results are not affected by any other applications (running in the background) that might have utilized the system resources and affected the experimental outcome. The network consists of

166MHz Pentium II PCs interconnected by a 100Mbs hub. The PC's run under Linux. The CORBA compliant middleware used in this investigation is Orbacus.

At the time when this research was started, there were many ORBs available in the market. But very few of them followed the open source philosophy. At that time, an *Interceptor* was the part of optional CORBA specs. So, it was one of those features that was left as an optional requirement for the vendor. Orbacus was the ORB that was open source, and implemented in Java with the *Interceptor* optional specs implementation [OCC-99]. Being an open source product, it was possible for us to have access to the source of ORB code, like IIOP header generation routines, parameter marshalling functions, padding insertion routines etc. It provided us with the capability to by-pass or change certain ORB processing routines.

To observe the performance of these performance enhancement techniques, a single client-single server setup was used. The client invoked a single method in the server machine that used parameters of different types. The method performed simple arithmetic operations on the parameters and returned the result. First original end-to-end time was measured for each case. Then the IIOP *send* message that the client sends from its node to the server's node was optimized, and end-to-end time was measured again.

5.3 Performance Enhancement Strategy 1:

We created six test cases corresponding to the six different data types. The results are summarized in Figure 5.2 and Table 5.1. Figure 5.2 presents the average distribution of various times in a typical CORBA transaction, computed from the six test cases. A simple servant routine was invoked by the client during each transaction. In each test case an argument list consisting of sixteen elements of a specific data type was used. The server routine sums up the parameters, increments the sum by 1 and returns the result. For Character type, the parameter value was cast to an Integer type.

Table 5.1 Improvement in marshalling/unmarshalling from performance enhancement

Technique-1.

	Parameter Type	Original Marshalling Time (microsec)	Modified Marshalling Time (microsec)	Original Unmarshalling Time (microsec)	Modified Unmarshalling Time (microsecond)	Improvement (Marshalling & Unmarshalling)
1	Octet	158	121	175	136	22.8 %
2	Boolean	229	135	258	145	42.5 %
3	Char	237	142	266	150	41.9 %
4	Short	355	158	369	165	55.4 %
5	Long	589	205	540	210	63.3 %
6	Double	605	225	669	238	63.7 %

(a)

(b)

Figure 5.2. Results of using Technique-1 (a) Response Time (b) Percentage Improvement

The marshalling times, unmarshalling times, and communication times are measured by inserting system calls for time measurement at various points in the Interceptor code, as discussed in previous chapter. Figure 5.1 presents the average distribution of transaction times. It shows that marshalling/unmarshalling routines, on an average, consume approximately 19% of the total end-to-end response time of the transaction. This indicates the potential for performance improvement by reducing the marshalling/unmarshalling overhead. Note that the response time is measured from the time a request is generated by a client to the time at which the response from the server arrives.

Table 5.1 shows the savings in marshalling/unmarshalling that accrue from the use of this performance enhancement technique. These results show that marshalling/unmarshalling overheads vary with the data types. These overheads depend upon two factors:

- Number of bytes to be marshalled.
- Complexity of the marshalling/unmarshalling routine that depends in the type of data being transferred.

Figure 5.1(b) presents the improvement in end-to-end response time for a transaction that occurs with the different argument types. The minimum savings (2.2%) is obtained for a data type of octet whereas the maximum savings (16.6%) is observed for a data type of double. The savings that occur with other data types are in between the minimum and maximum values. For the octet data type the complexity of marshalling/unmarshalling is the minimum. On the other hand, long and double require marshalling/unmarshalling of

more bytes and that leads to the execution of more CPU cycles. Note that the performance improvements presented in this section correspond to a no load situation when only one client request is processed at a time by the system.

5.4 Performance Enhancement Strategy-2:

5.4.1 Results of Experiments

Figure 5.3 (a) shows the experimental results obtained by varying the proportion of padding bytes in the argument list in the client request. The number of arguments in the list is fixed at 24. With 24 arguments it is possible to achieve a range of percentage of padding bytes (0-48%). Since the performance improvement depends on the proportion of padding bytes, a similar graph is expected from a higher number of arguments. It is observed that the savings in communication is directly proportional to the percentage of padding bytes in the message. The slope of this line depends on the characteristics of the communication system. As the bandwidth increases, lesser time is needed to pass additional padding bytes, and thus the slope of the graph decreases. The Interceptor overhead, which can be measured by calculating the execution timings of the additional code, are also shown in Figure 5.3 (b) . As padding in data is increased, the savings in communication time increase linearly. The Interceptor overheads consist of two parts: a fixed part that corresponds to the calling and execution of the Interceptor method, and a variable part that corresponds to the data copying operations. When padding is approximately 22%, savings and overheads are approximately equal to one another. We call this point the threshold value of padding. If the proportion of padding bytes is below this threshold value, the system is in zone A. If the proportion of padding is higher than

the threshold, the system is in zone B. Trying to use this strategy in zone A degrades system performance (see Figure 5.3(b)). For example, if the proportion of padding is 15%, the additional overhead is 360 microseconds, whereas the savings in communication time is only 280 microseconds and the overall system performance deteriorates. If the proportion of padding is 48%, then an overhead of 550 microseconds is introduced, while a saving of 920 microseconds is achieved in terms of communication time. This improves the overall system performance by 370 microseconds. Note that the slope of the dark line in Figure 5.3(b) depends on the speed of the network, whereas the slope of the dashed line depends on the processing power of the client/server nodes. Thus, the padding threshold that corresponds to the intersection of the lines is dependent on system characteristics. Before using this technique we should calculate the threshold padding value, and determine whether the system is operating in “Zone A” or “Zone B”. This will enable us to decide whether this strategy should be used for a given system. Figure 5.3(b) presents the savings obtained due to padding removal and the net savings. The experimental data shows that a performance benefit from this technique starts when padding in the data is around 25%.

5.5 Performance Enhancement Strategy-3: Compressing Headers

5.5.1 Experimental Results

Table 5.2 shows that the benefit obtained for the shorter message is more than that of longer message. For example, in Table 5.2, for one parameter, the percentage saving is

(a)

(b)

Figure 5.3 Performance Optimization Strategy 2
 (a) Benefits from removing padding (b) Saving from padding and net saving

almost 9%. But it drops to 2.5% for 32 parameter message. The reason is that this technique optimizes only the header of the message. There is no performance enhancement for the message payload. For the shorter messages, the percentage of header in the total message is more, so the overall percentage benefit will be more. That is what is reflected in this table. The performance improvement that accrues from this technique is expected to increase with an increase in IPC workload for the system.

Table 5.2: Experimental Results of performance enhancement technique 3

	<i>1 parameter</i>	<i>2 parameters</i>	<i>4 parameters</i>	<i>8 parameters</i>	<i>16 parameters</i>	<i>32 parameters</i>
NO OF BYTES						
<i>Actual Message</i>	31	32	34	38	46	62
<i>Modified Message</i>	18	19	21	25	33	49
TIME IN MICROSECONDS						
<i>Overhead added</i>	250	250	252	257	266	302
<i>Actual Message</i>	3113	3189	3233	3354	3591	4096
<i>Modified Message</i>	2885	2955	3005	3126	3400	3965
<i>%age Savings</i>	7.902	7.918	7.58	7.29	5.617	2.496

5.6 Observing Optimized Behavior with Varying Degrees of Homogeneity

The extent of homogeneity present in a system is expected to affect system performance. We have used a simple system with a single-client and four servers to gain insights into system behavior. The servers can be of the same type as the client (type-A) or of a different type (type B). Degree of homogeneity on such a system is defined as the proportion of the number of “type A” servers that are identical to the client present in the system. The effectiveness of performance enhancement techniques has a direct

relationship with the degree of homogeneity of a system for obvious reasons. To observe the behavior of optimized system in varying degree of homogeneity, experiments using the second performance enhancement technique are conducted. The behavior of other performance enhancement techniques with varying degrees of homogeneity is expected to have the same pattern. It is left as a future work as described in chapter 6.

The following methodology was used to explore this relationship.

To explore the usefulness of this technique, five computers were used. One was used as a client and the other four were used as servers. The code on all the four servers consisted of a function (*func1*) having 24 arguments. The arguments of the function had a padding of 48%. This corresponds to the scenario that gave rise to the largest performance benefit captured in Figure 5.5. *Func1* was a simple routine that adds up all the parameters, increments the result by one and returns the value. In the case of char type, the value was converted to type byte before adding up (by casting it). Client invoked *func1* one by one on each of the four servers. Since synchronous communication is used, the client remains blocked until the results of the method invoked arrive from the server. As soon as the server returns the result, next server is invoked and so on. End-to-End time is the total time in the cycle taken to invoke methods on all the four servers. First, the end-to-end time is measured without applying any performance enhancement strategy. Then we considered that all of the servers were of the same type (i.e. the degree of homogeneity 100%). Thus, we can apply a performance enhancement strategy on all four servers. This is shown as Case 1 in Figure 5.4. In the Case 2, Figure 5.4, the performance enhancement technique is applied to only one of the four servers. Thus, the homogeneity level is 25%

in this case. Similarly, the experiments are repeated for the homogeneity of 75% and 50%. For the homogeneity of 75%, the performance enhancement technique is applied on three of the four servers, which communicate with the fourth server through regular CORBA ORB. For the homogeneity of 50%, the performance enhancement technique is applied on two of the four servers.

Case 1:

Case 2:

Figure 5.4 Division of nodes into domains according to their homogeneity

The results obtained are shown in Table 5.3. In Table 5.3 the Original End-to-End Time is the time taken by the client to invoke this method on the four servers without applying any performance enhancement technique. This time will be the same for all the cases. As the degree of homogeneity drops, the benefit obtained by using the performance enhancement techniques also drops, as we are now able to optimize a lesser percentage of nodes. As the level of homogeneity drops below 50%, the benefit obtained drops below 5%. So level of homogeneity present in a system is an important consideration before deciding to apply the performance enhancement techniques.

Table 5.3: Effect of varying degree of Homogeneity on Performance Enhancement Technique 2

<i>Degree of Homogeneity</i>	<i>End to End time</i>		<i>Percentage Savings</i>
	<i>Original</i>	<i>Performance Enhanced</i>	
100	16472	14804	10.1
75	16472	15190	7.7
50	16472	15600	5.2
25	16472	16078	2.3

5.7 Observing Effects of Increased System Load

5.7.1 Observing the Performance Enhancement in a Real World Scenario

The performance enhancement techniques developed in this research were initially tested in a simple test environment. This test environment involved a single client-single server system. The client was responsible for invoking the remote method on the server only once and waiting for the results synchronously. The end-to-end response time was

measured at the client, and the benefit of the performance enhancement techniques applied was analyzed.

Once the performance enhancement techniques were developed and had proved their worth in this simple test environment, they were tested in more real world situations. The real world scenario chosen was a work under development in a project in a local semiconductor industry. It is a project of a distributed SDRAM silicon compiler with a Java front-end tool connected via Internet to a Web-server (with at least 100Gb of data-sheets and configuration data). The user uses the front-end tool to enter the parameters (such as number of banks, number of leaf-arrays, capacity and the foundry process), and then uses the GUI to draw the complete layout of a module using leaf-cells. The interaction between client and server is quite frequent and can be characterized by bursts of requests. Multiple bursts of requests model operations performed by multiple clients who are simultaneously using the compiler. This “burstiness” emerges from the fact that there are many constraints that have to be checked using the databases before the user can edit or move any object for that particular layout. A typical constraint may be the minimum distance between two cell boundaries, dictated by the foundry database for this particular process. So, in response to a single operation performed by the user, a batch of requests are typically generated. A similar scenario can be expected in other systems, such as a web-server, in which multiple requests to embedded files are generated when the client accesses a page containing the embedded fields.

To produce a scenario characterized by such a batch of requests, a burst generator was designed. After a fixed interval of time, it generates a burst in the client machine with x number of requests. In order to achieve some variability in the number of requests in a batch, it is assumed to be exponentially distributed with mean $1/u$ that can be varied. Different mean values simulate different complexities of work done in the client machine. For example, if one is finalizing the layout of a leaf-cell in a 8 Mb SDRAM, the mean value of requests will be more than that required for the design of 256KB SDRAM.

The user chooses the mean number of requests in a batch before running the experiment. The burst generator runs in the client machine as a separate thread, and after a fixed time invokes x number of requests as a burst. The requests join a First Come First Server (FCFS) queue as shown in the Figure 5.5. For each request produced by the burst generator, the client invoked a method in the server machine that returned the results. The client is blocked until the result from the server arrives. The mean number of requests in a burst can be changed to change the utilization of the system, and by observing the response time, we can analyze the system behavior [MHE-99]. For certain experiments (e.g. to determine the effect of varying heterogeneity in the system), the requirement was to produce a system with multiple servers. For that, a burst generator similar to the one explained earlier was used. It ran at the client machine on a separate thread and generated the bursts of requests in the client. The client was connected with n number of servers and for each request it communicated with them one by one in a loop. The request resulted in invocation of a method in the given server that returned a result back to the client. If t_1 is the time when a client initiates a request, and t_2 is the time when the last of the n servers

Figure 5.5: Event Flow in Single Server – Single Client System

responded back then $t_2 - t_1$ will give us the total response time of that particular request. We can change the mean number of requests in a burst to change the utilization of the system. All the servers and the client are running the same CORBA ORB Orbacus [OBJ-99].

5.7.2 Results of Experiment

The results described so far were obtained on a single client- single server environment. The performance of the performance enhancement techniques when a stream of requests arrive is described in this section. The second performance enhancement technique is used in this experiment. The percentage of padding bytes is held at 48% which is the maximum percentage padding used in Figure 5.3(b). The experimental environment and the method used for controlling system load is described in Section 5.2. By varying the burst size of the requests we can change the system utilization and observe its effects on system performance. For both the original and performance enhanced system we varied

the burst size from a small value to a value large enough to saturate the system. The corresponding request arrival rates are reported in Table 5.7. The results are shown in Figure 5.6, Table 5.4 and Table 5.5. It can be observed that by applying the second performance enhancement technique, the knee of the curve has shifted to the higher utilization (see in Figure 5.6). It means that application of the performance enhancement technique helps to keep the utilization at lower level for the same system conditions. Table 5.4 shows that the server can handle around 215 requests per second before saturation without applying any performance enhancement technique. But if we apply the second performance enhancement technique, the request rate that saturates the server is 225 requests per second.

Figure 5.6: Effect of System Load

Curve(a)- Original System

Curve (b)- Performance Enhanced System

***Table 5.4 Results of applying performance enhancement strategy 2:
Original Response Time***

Arrival Rate (L) is average rate of requests generated by the traffic generator per sec

Average Service Time = 4021 microsecond.

Average Service Rate = $1/0.004021=248.6944$ requests per sec

Request Arrival Rate	Utilization of System	Response Time (microsecond)
1	0.004	3998
10	0.4	4087
50	0.201	4456
100	0.402	12587
150	0.603	36548
200	0.804	90365
215	0.864	123000

***Table 5.5 Results of applying performance enhancement strategy 2:
Response Time after applying Performance Enhancement***

Average Service Time = 3845 microseconds.

Average Service Rate = 260.078 request per sec

Request Arrival Rate	Utilization of System	Response Time (microsecond)
1	0.0038	3801
10	0.0384	3995
50	0.192	4232
100	0.384	11960
150	0.576	35210
200	0.796	83036
225	0.865	108258

5.8 Discussion

Performance enhancement obtained from the techniques presented in this chapter depends upon certain system parameters. In particular, the following factors should be used to weigh the usefulness of these techniques. Measurements are to be made on the system to determine these parameters.

- Degree of homogeneity in the system under consideration
- Number of users and load
- Proportion of padding bytes
- Average length of messages
- Proportion of interprocess communication (IPC) in total workload

Different classes of applications are characterized by different average values for these parameters. For example, if the distributed system where CORBA is to be applied is planned to be used for semi-conductor CAD applications, most of the operations will involve the transmission of a large number of data items (such as netlist parameters). On the other hand, if the distributed system is to be used for a PBX call-server then most of the messages are short (as they are control messages). One of the most important factors for obtaining the performance benefit by deploying these techniques, is the degree of homogeneity in the system under consideration. Performance Enhancement by deploying these techniques is possible for communication between nodes that are similar in nature. As the percentage of similar nodes increases in a system, so does the performance. Another important factor to consider is the percentage of the IPC in the total workload.

All the techniques presented in this research enhance the performance of IPC component of the application. If an application is CPU bound with rare IPC, it means that use of these techniques will lead to a limited performance enhancement. A larger performance benefit is expected for communication bound systems that are characterized by a high IPC.

As discussed in the previous paragraph, a larger degree of homogeneity and a large proportion of IPC in the workload increase the benefits of all three techniques. As observed in Section 5.7, system load has a significant effect on the performance enhancement expected from the techniques. The impact of the other factors on the usefulness of each of the three techniques is briefly discussed.

There are some other factors that affect the usefulness of each technique differently. They are discussed as follows:

Technique 1:

As noted in Section 4.3, the first technique produces performance enhancement by bypassing the marshalling/ unmarshalling routines. A similar performance enhancement technique is used for Remote Procedure Calls (RPCs) on certain systems. The performance enhancement thus obtained is dependent on the time spent by the marshalling/ unmarshalling engines to marshal a particular type of data. For example, the marshalling/ unmarshalling time of data type double is more than the marshalling/ unmarshalling time of short data type. Marshalling/ unmarshalling time also depends on

the number of parameters used. As the number of parameters is increased, so the total time to marshal them increases the total savings obtained by bypassing these marshalling/unmarshalling routines. The performance enhancement achieved by this technique is also dependent on the number of users. As the number of users increases, the system utilization increases and more contention occurs on the system. The performance enhancement obtained is expected to increase as we are bypassing these marshalling/unmarshalling routines that utilize the server resources.

Technique 2:

As already noted before, the second performance enhancement technique gives rise to a trade-off between savings in communication bandwidth and an increase in client/server processing time. It decreases the size of messages at the cost of client/server processing time. The deployment of this technique requires a careful analysis of the system conditions. With a message consisting of a large number of padding bytes, a larger communication bandwidth is expected to be consumed. However, if we are not careful about analyzing the system conditions, we can, in fact, considerably degrade the performance instead of improving it. The benefit obtained using this technique is directly proportional to the number of padding bytes present in the data. We should consider using this technique only if we are expecting to have data intensive operations with parameters having a mix of various types of data. The usefulness of this technique also depends on the number of users that are using it. As the number of users increases, the system utilization will increase. With the application of this performance enhancement technique the system saturation occurs at a higher load in comparison to the original system.

Technique 3:

The last performance enhancement technique compresses the header of the message shown in Table 5.2 which shows that the proportion of performance improvement is higher for short messages. This is because we are only applying the technique to the header of the message. If the header occupies a larger part of the total message, the effect of our enhancement technique will be more. However, the overall improvement in performance will increase on a system characterized by a high IPC workload in which a large number of messages is exchanged. As with previous techniques, an increase in system load that may be caused by an increase in the number of users is expected to lead to a higher performance improvement.

Most of the performance improvement reported in this thesis is for a single client and single server system. A higher performance benefit may be expected when the system load is increased and multiple clients compete for system resources. Note that even a small performance benefit may be useful for certain systems. For example, Nortel Network is known to spend \$55,000 in terms of manpower for achieving each 1% improvement in the capacity of their switches [BCA-99] [AGO-97].

CHAPTER 6

Conclusions

6.1 Contributions of this Research

This research focused on CORBA performance. It introduced three performance enhancement techniques that exploit limited heterogeneity in a system. The proposed name for the resulting solution (performance enhancement techniques with these run-time mechanisms) is 'flyover', as the resulting solution bypasses certain CORBA operations when two similar components communicate with one another. The scope of this research is limited to the development of these performance enhancement techniques, which can be used as a base, in the complete development of these flyovers. These techniques cannot therefore be used as a stand-alone solution that can be used to optimize a distributed system, and will need further work to mature into a viable product.

The major results from this research have been published in [IAH-01] and [SMA-00.2]

6.2 Performance Enhancement Techniques: Conclusions

The numeric values of the performance metrics reported in the thesis are dependent on the middleware Orbacus, as well as on the experimental environment used. We expect the

insights gained from this research to be useful in other environments as well. A summary of the performance enhancement techniques are presented in this chapter.

6.2.1 Performance Enhancement Technique 1

This technique should be used for data centric operations as it tries to optimize the way parameter data is passed in a CORBA transaction. As discussed in the previous chapter the performance enhancement obtained is by avoidance of marshalling/ unmarshalling of parameters and buffer copying when the interacting nodes are similar to one another. The benefit obtained will depend on the system conditions and the type of parameters used. For example, for octet type the benefit is small, as it does not undergo any marshalling/ unmarshalling. But for double type we obtained the maximum benefit 17%.

We need a mechanism that can inform the interacting nodes at run-time, whether they are of the same type or not. Development of such a mechanism can form an important direction for future research.

6.2.2 Performance Enhancement Technique 2

Among the performance enhancement techniques developed, this is the only performance enhancement technique which introduces significant processing overheads. So the benefit obtained by using it should be carefully weighed to ensure that it is much more than the overheads introduced. If used improperly, it is possible to degrade the system's performance instead of improving it.

If the padding present in the data is below a threshold value then the benefit obtained from removal of padding is less than the overheads produced due to additional code that is executed. Using this technique in this situation will in fact degrade the performance instead of improving it.

As discussed in Chapter 5, Performance Enhancement Technique 2 optimizes system performance by removing the padding present in the parameter data before the client sends it to the server. This technique assumes the presence of a data array that conveys the information of how many bytes have been removed from the client machine and at which offsets. This information is vital as the server will use it to recreate the padded parameter stream. A mechanism needs to be developed that can transfer this information from the client to the server.

6.2.3 Performance Enhancement Technique 3

The third performance enhancement technique is concerned with the compression of the IIOP header. The performance improvement obtained from this technique is significant only if the IIOP message is short. The performance enhancement technique is useful only if the system satisfies the following two conditions.

- 1- The client and server should be similar to one another.
- 2- The message should be short enough so that the potential benefit obtained is significant.

6.2.4 Future Research

Most of the performance results reported in this thesis are obtained with a single client invoking a method on a single server. As shown in Section 5.7, a higher performance benefit can be expected at a higher system load when multiple client requests compete for system resources. More experiments with all the performance enhancement techniques and different system architectures would provide more insight into the performance benefits that could accrue from the deployment of these strategies. This thesis has focused on performance enhancement techniques. Building a flyover system based on these techniques forms an important direction for future research.

References

- [AGO-96.1] A.Gokhale and D.Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks" in *Proceedings of SIGCOMM '96*, Stanford, California, August 1996. ACM, pp. 306-317.
- [AGO-96.2] A.Gokhale and D.Schmidt, "The 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.
- [AGO-97] A.Gokhale and D. Schmidt, "Evaluating Latency and Scalability of CORBA Over High-Speed ATM Networks" in *Proceedings of the International Conference on Distributed Computing Systems*, Baltimore, Maryland, May 1997. pp. 44-50.
- [BCA-99] B. Carroll, Nortel Networks, Personal Communication, September, 1999.
- [BMC-98] B. McCarty and L. Cassady-Dorion, *Java Distributed Objects*, Sams Publishing, 1998.
- [BSY-99] BEA Systems et al. "CORBA Component Model Joint Revised Submission", *OMG Document orbos/99-07-01 3ed.*, Object Management Group, July 1999.
- [DLE-99] D.Levine, S. Flores-Gaitan and D.C Schmidt "Measuring OS Support for Real-time CORBA ORBs" in *Proceedings of 4th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS'99)*, Santa Barbara, California, January, 1999.
- [DSC-95] D.Schmidt and S.Vinoski, "Comparing Alternative Client-side Distributed Programming Techniques", *SIGS C++ Report*, May 1995, pp. 56-60.
- [DSC-96] D.Schmidt and S.Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded Servers", *SIGS C++ Report*, February 1996, pp. 100-105.
- [DSC-97] D.Schmidt, A.Gokhale, T.Harrison and G.Parulkar, "A High-performance Endsystem Architecture for Real-time CORBA," *IEEE Communications Magazine*, February, 1997, Vol. 14, No. 2, pp. 156-160.

- [DSC-98.1] D.Schmidt, "Evaluating Architectures for Multithreaded Object Request Brokers", *Communications of the ACM*, October, 1998 , Vol. 41, No. 10, pp. 54-60.
- [DSC-98.2] D.Schmidt, D.Levine and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, April 1998, Vol. 21, pp. 294-324.
- [EGA-95] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [ESH-00] E.-K Shen, "Adaptive Middleware Systems", *M.Eng. Thesis*, School of Computer Science, Carleton University, Ottawa, Canada, May 2000.
- [GBR-99] G. Brose , *JacORB Programming Guide*, 1999, <http://www.inf.fu-berlin.de/~brose/jacorb>.
- [GBR-01] G. Brose, A. Vogel and K. Duddy, *Java Programming with CORBA*, Third Edition, OMG Press, 2001.
- [GPA-95] G. Parrington. "A Stub Generation System for C++", *Computing Systems*, Vol. 8, No. 2, Spring 1995, pp. 135-170.
- [GVA-96] G. Varghese. "Algorithmic Techniques for Efficient Protocol Implementations" in *SIGCOMM '96 Tutorial*, Stanford, California, August 1996, pp.130-144.
- [IAB-98.1] I.Abdul-Fatah and S.Majumdar, "Performance Comparison of Architectures for Client-Server Interactions in CORBA" in *Proceedings IEEE 18th International Conference on Distributed Computing Systems (ICDCS'98)*, Amsterdam, May 1998, pp. 170-174.
- [IAB-98.2] I.Abdul-Fatah "Performance of Client-Server based architectures " *Master of Engineering Thesis*, Carleton University, May 1997.
- [IAH-01] I. Ahmad and S. Majumdar, "Achieving High Performance in CORBA-Based Systems with Limited Heterogeneity" in *Proceedings of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing(ISORC 2001)*, Magdeburg, Germany, May 2-4 2001, pp. 350-359.

- [KMO-94] K. Modeklev, E. Klovning, and O. Kure, "TCP/IP Behavior in a High-Speed Local ATM Network Environment" in *Proceedings of the 19th Conference on Local Computer Networks*, Minneapolis, MN, October 1994, pp. 176-185.
- [LKE-75] L. Kleinrock, *Queuing Systems*, Wiley-Interscience, 1975.
- [MHE-99] M. Hennig, Vinoski, *Advanced Computer Programming with C++*, Addison-Wesley, 1999.
- [MST-95] M. Stal, "The Broker Architectural Framework", in *Proceedings Object-Oriented Programming Systems, Languages and Applications Conference (OOPSLA'95)*, Maitland, Florida, March 1995, pp. 114-120.
- [NWA-99] N. Wang and D. Schmidt, "Collocation Optimization for CORBA", *C++ Report*, September 1999.
- [NWA-00.1] N. Wang and D. Schmidt, "Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications", *Reflective Middleware Workshop at the IFIP/ACM Middleware 2000 conference (RM 2000)*, Palisades, New York, April 3-7, 2000, pp. 16-18.
- [NWA-00.2] N. Wang and D. Schmidt, "Optimizing the CORBA Component Model for High-performance and Real-time Applications" *Work-in-progress paper for the IFIP/ACM Middleware 2000 Conference*, Pallisades, New York, 2000.
- [OCC-99] Object Oriented Concepts, *ORBacus OnLine Manual*, For C++ and Java, 1999, www.ooc.com.
- [OMG-98] Object Management Group, *The Common Object Request Broker: Architecture and Specifications*, 2.2 ed, February 1998
- [OMG-99] Object Management Group, *Real-Time CORBA - Joint Revised Submission (Request for Proposal)*, OMG-TC Document Orbos/99-02-12, March 1999.
- [OMG-01] Object Management Group, *The Common Object Request Broker: Architecture and Specifications*, 2.4 ed, February 2001.
- [PHO-94] 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, 1994. IFIP, pp. 100-105.

- [PHA-98] P. Haggerty and K. Seetharman, "The Benefits of CORBA-Based Network Management", *Communications of the ACM*, October 98, Vol. 20, No. 9, pp. 73-79.
- [SDH-94] S. Dharnikota, K. Maly, and C. Overstreet. "Performance Evaluation of TCP (UDP)/IP over ATM networks". Department of Computer Science, *Technical Report CSTR-94-23*, Old Dominion University, September 1994.
- [SIE-98] J. Seigel, "OMG Overview: CORBA and the OMA in Enterprise Computing", *Communications of the ACM*, October 1998, pp. 37-43.
- [SMA-00.1] S. Majumdar and W. Wu, "CORBA Middleware: a Performance Perspective" Department of Systems and Computer Engineering, Carleton University.
- [SMA-00.2] S. Majumdar, I. Ahmad, E. Shen, I. Abdul-Fatah "Achieving High Performance Through Middleware Reconfiguration", *RM 2000, Workshop on Reflective Middleware*, New York, USA, April 2000.
- [SMI-00] S. Mishra and X. Liu, "Design, Implementation and Performance Evaluation of a High Performance CORBA Group Membership Protocol" in *Proceedings of the 7th International Conference on High Performance Computing*, Bangalore, India, December, 2000.
- [TBA-99] T. Barth, G. Flender and F. Thilo, "Load Distribution in a CORBA Environment" in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA '99)*, Edinburgh, UK, September 1999.
- [TBR-95] T. Braun and C. Diot. Protocol Implementation "Using Integrated Layer Processing." in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM) ACM*, Philadelphia, September 1995, pp. 200-205.
- [WRU-00] William Ruh, Thomas Herron, Paul Klinker, *IIOP Complete: Understanding CORBA And Middleware Interoperability*, Addison-Wesley, 2000.

[ZDN-01]

ZDNetAnchorDesk Editorial, "Changing trends in IT",
<http://www.zdnet.com/anchordesk/index.html>
23rd March, 2001.