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.

**Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0800**

UMI[®]

Performance of Adaptive Middleware Systems

by

E-Kai Shen

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements for the degree of
Master of Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario, Canada
May, 2000

© Copyright 2000, E-Kai Shen

**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.

0-612-52381-0

Canada

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

Performance of Adaptive Middleware Systems

submitted by E-Kai Shen,

**in partial fulfilment of the requirements for the degree of
Master of Computer Science**

for **E. Kranakis, Director
School of Computer Science**

S. Majumdar, Thesis Supervisor

**Carleton University
May 2000**

Abstract

Middleware such as CORBA-based products provides the communication infrastructure and inter-operability in a heterogeneous distributed environment. To achieve a high performance middleware software, the system architecture should be carefully considered. This thesis proposes an adaptive middleware architecture, which is called adaptive ORB, for CORBA-based systems. The agent of the adaptive ORB that maps an object name to the object reference has two modes of operations. In the handle-driven mode it returns a server's handle to the client which uses this handle to send the request to the server. In the forwarding mode the agent forwards the entire client request to the server which returns the result to the client. The adaptive ORB dynamically switches between these two modes depending on the current system load. Based on measurements made on a network of workstations and a synthetic workload, we observe that the adaptive ORB can produce a significant benefit in performance in comparison to a purely handle-driven or a purely forwarding ORB.

Acknowledgments

As a part time student, I would like to say it is really challenging to get this degree, particularly with thesis. I appreciate the strong support from my supervisor Professor, Shikharesh Majumdar, the informative suggestions from Istabrak N. Abdul-Fatah, and the friendship from the members of the RADS lab in Department of Systems and Computer Engineering.

Finally, I would like to give my special thanks to my wife, Shu-Chuan Tsou. Without her support and encouragement, I could have never gone this far.

Table of Content

Chapter 1 Introduction 1

1.1 Background	1
1.2 Motivation	2
1.3 Goals of this Research	3
1.4 Thesis Contributions	4
1.5 Thesis Outline	5

Chapter 2 CORBA Overview 6

2.1 Introduction	6
2.2 Structure of an Object Request Broker	9
2.2.1 ORB Core	9
2.2.2 ORB Interface	9
2.2.3 Client, Objref, and Object Implementation	10
2.2.4 OMG Interface Definition Language (IDL)	10
2.2.5 IDL Stub and Skeleton	11
2.2.6 Dynamic Invocation Interface (DII)	12
2.2.7 Dynamic Skeleton Interface (DSI)	12
2.2.8 Object Adaptor (OA)	13
2.2.9 GIOP/IIOP	14
2.2.10 Implementation Repository	15
2.2.11 Interface Repository	15
2.3 Performance Issues in CORBA	15
2.3.1 Presentation Layer	16

2.3.2 Memory Management	16
2.3.3 Demultiplexing	17
2.3.4 DII and DSI	19
2.3.5 IIOP	19
2.4 Real Time CORBA	20
2.4.1 Requirements for Real-Time CORBA	20
2.4.2 Real-Time CORBA Standard	22
2.5 Applications of CORBA	24

Chapter 3 The Adaptive ORB (A-ORB) 26

3.1 Introduction	26
3.1.1 The Handle-Driven ORB (H-ORB)	26
3.1.2 The Forwarding ORB (F-ORB)	28
3.1.3 Motivation for the Adaptive ORB	30
3.1.4 The Adaptive ORB	31
3.2 The implementation of the Adaptive ORB	33
3.2.1 Software Architecture of the A-ORB	34
3.2.2 Pseudo Codes of the A-ORB	39
3.2.2.1 Pseudo code for the A-client	39
3.2.2.2 Pseudo code for the A-agent	39
3.2.2.3 Pseudo code for the A-server	41
3.2.3 Execution Scenario of the A-ORB	42
3.2.3.1 The Client	42
3.2.3.2 The A-agent	44
3.2.3.3 The Server	47
3.3 Initialization and Request Path for the A-ORB	48
3.3.1 Activation Modes	48
3.3.2 Binding Operation	49

3.3.3 Initialization	52
3.3.4 The A-ORB Request Path	54
3.4 Design Issues and Problems Experienced	57
3.4.1 One-Way Operation v.s. Normal Operation	57
3.4.2 Event Control in Orbix	57
3.4.3 Multi-tasking vs. Multi-threading	59
3.4.4 Message Dispatching vs. Queue Polling	60
3.4.5 Persistent Servers vs. Non-Persistent Servers	61
3.4.6 Problems Related to Orbix or Solaris	61

Chapter 4 Instrumentation and Performance Measurement

63

4.1 The Workload and Configuration files	63
4.1.1 Workload	63
4.1.2 Configuration Files	65
4.2 Computation of Performance Matrices	67
4.2.1 Definition of Performance Matrices	67
4.2.2 Measurement of the Elapsed and CPU Time	71
4.3 Data Collection	74
4.3.1 The Control Process	74
4.3.2 The Measurement Steps	75
4.4 Verifying Data	77
4.5 Experimental Environment	78

Chapter 5 Results of Experiments

80

5.1 Effect of Server Demands	80
5.2 Effect of Message Size	92
5.3 Effect of Inter-Node Delay	98
5.3.1 Equidistant Nodes	99
5.3.2 Non-Equidistant Nodes	105
5.3.2.1 Client closer to the Agent	105
5.3.2.2 Server closer to the Agent	110
5.3.2.3 Client closer to the Server	112
5.3.2.4 Summary of Three Cases for Non-Equidistant Nodes	115
5.4 Systems with the Multi-Threaded Agents	115
5.5 The Best Threshold Value vs. Load Balancing	124

Chapter 6 Conclusions 130

6.1 Summary	130
6.1.1 The Impact of Server Demand	130
6.1.2 The Impact of Message Size	131
6.1.3 The Impact of Inter-Node Delay	131
6.1.4 Multi-Threading and Load Balancing	132
6.2 Conclusion	133
6.3 Further Research	134

Reference 136

List of Figures

Figure 2-1: Object Model of OMA	7
Figure 2-2: The structure and components of CORBA	8
Figure 2-3: The general CORBA application code development paradigm under the support of the CORBA protocol compiler	11
Figure 2-4: Layered CORBA Request Demultiplexing	17
Figure 3-1: High level interaction of the H-ORB Architecture	27
Figure 3-2: High level interaction of the F-ORB Architecture	29
Figure 3-3: High level interaction of the A-ORB Architecture	32
Figure 3-4: The processes used in this implementation of the A-ORB	34
Figure 3-5: IDL source code of the client, agent, and server	36
Figure 3-6: The interactions inside the A-agent process	37
Figure 3-7: Automatic launch of an orbix server process	50
Figure 3-8: The Orbix server locator mechanism	51
Figure 3-9: The Request Sequence Path of this A-ORB Implementation	56
Figure 4-1: Activities in the Server Process	69
Figure 4-2: Eight filter points	71
Figure 4-3: The example of getting CPU time by using ioctl()	73

Figure 4-4: The Control Process	74
Figure 5-1: Throughput of the A-ORB with Sa/Sb=10/15 ms, D=0 ms, L=150 bytes	83
Figure 5-2: Response Time of the A-ORB with Sa/Sb=10/15 ms, D=0 ms, L=150 bytes	83
Figure 5-3: Thread and CPU Utilizations of the A-ORB Agent with Sa/Sb=10/15 ms, D=0 ms, L=150 bytes, QL=1	84
Figure 5-4: Thread and CPU Utilizations of the A-ORB Agent with Sa/Sb=10/15 ms, D=0 ms, L=150 bytes, QL=3	84
Figure 5-5: Thread and CPU Utilizations of the A-ORB Agent with Sa/Sb=10/15 ms, D=0 ms, L=150 bytes, QL=24	85
Figure 5-6: Throughput of the A-ORB with Sa/Sb=50/75 ms, D=0 ms, L=150 bytes	86
Figure 5-7: Thread and CPU Utilizations of the A-ORB Agent with Sa/Sb=50/75 ms, D=0 ms, L=150 bytes, QL=3	87
Figure 5-8: Thread and CPU Utilizations of the A-ORB Agent with Sa/Sb=50/75 ms, D=0 ms, L=150 bytes, QL=24	87
Figure 5-9: Thread and CPU Utilizations of the A-ORB Agent with Sa/Sb=50/75 ms, D=0 ms, L=150 bytes, QL=0	88

Figure 5-10: Throughput of the A-ORB with Sa/Sb=250/375 ms, D=0 ms, L=150 bytes	89
Figure 5-11: Process and CPU Utilizations of the A-ORB servers with Sa/Sb=250/375 ms, D=0 ms, L=150 bytes, QL=0	90
Figure 5-12: Process and CPU Utilizations of the A-ORB servers with Sa/Sb=250/375 ms, D=0 ms, L=150 bytes, QL=24	90
Figure 5-13: Thread and CPU Utilizations of the A-ORB Agent with Sa/Sb=250/375 ms, D=0 ms, L=150 bytes, QL=0	91
Figure 5-14: Thread and CPU Utilizations of the A-ORB Agent with Sa/Sb=250/375 ms, D=0 ms, L=150 bytes, QL=24	91
Figure 5-15: Throughput of the A-ORB with L=4800 bytes, Sa/Sb=10/15 ms, D=0 ms	95
Figure 5-16: Throughput of the A-ORB with L=9600 bytes, Sa/Sb=10/15 ms, D=0 ms	95
Figure 5-17: Throughput of the A-ORB with L=19200 bytes, Sa/Sb=10/15 ms, D=0 ms	96
Figure 5-18: Thread and CPU Utilizations of the A-ORB Agent with L=4800 bytes, Sa/Sb=10/15 ms, D=0 ms, QL=3	97
Figure 5-19: Thread and CPU Utilizations of the A-ORB Agent with	

L=9600 bytes, Sa/Sb=10/15 ms, D=0 ms, QL=3	97
Figure 5-20: Thread and CPU Utilizations of the A-ORB Agent with	
L=19200 bytes, Sa/Sb=10/15 ms, D=0 ms, QL=3	98
Figure 5-21: Throughput of the A-ORB with D=50/50/50 ms, L=150	
bytes, Sa/Sb=10/15 ms	102
Figure 5-22: Throughput of the A-ORB with D=125/125/125 ms,	
L=150 bytes, Sa/Sb=10/15 ms	102
Figure 5-23: Throughput of the A-ORB with D=250/250/250 ms,	
L=150 bytes, Sa/Sb=10/15 ms	103
Figure 5-24: Thread and CPU Utilizations of the A-ORB with	
D=50/50/50 ms, L=150 bytes, Sa/Sb=10/15 ms, QL=1	104
Figure 5-25: Thread and CPU Utilizations of the A-ORB with	
D=50/50/50 ms, L=150 bytes, Sa/Sb=10/15 ms, QL=3	104
Figure 5-26: The relative distance when the client is closer to the agent	106
Figure 5-27: Throughput of the A-ORB with D=50/250/250 ms,	
L=150 bytes, Sa/Sb=10/15 ms	107
Figure 5-28: Thread and CPU Utilizations of the A-ORB Agent with	
D=50/250/250 ms, L=150 bytes, Sa/Sb=10/15 ms, QL=0	108
Figure 5-29: Thread and CPU Utilizations of the A-ORB Agent with	

D=50/250/250 ms, L=150 bytes, Sa/Sb=10/15 ms, QL=24	109
Figure 5-30: Thread and CPU Utilizations of the A-ORB Agent with	
D=50/250/250 ms, L=150 bytes, Sa/Sb=10/15 ms, QL=1	109
Figure 5-31: Thread and CPU Utilizations of the A-ORB Agent with	
D=50/250/250 ms, L=150 bytes, Sa/Sb=10/15 ms, QL=3	110
Figure 5-32: The relative distance when the server is closer to the agent	
Figure 5-33: Throughput of the A-ORB with D=250/50/250 ms,	
L=150 bytes, Sa/Sb=10/15 ms	112
Figure 5-34: The relative distance when the client is closer to the server	
Figure 5-35: Throughput of the A-ORB with D=250/250/50 ms,	
L=150 bytes, Sa/Sb=10/15 ms	114
Figure 5-36: Throughput of the A-ORB with D=125/125/125 ms,	
L=4800 bytes, Sa/Sb=10/15 ms, Level of Thread Cloning=1	117
Figure 5-37: Throughput of the A-ORB with D=125/125/125 ms,	
L=4800 bytes, Sa/Sb=10/15 ms, Level of Thread Cloning=2	117
Figure 5-38: Throughput of the A-ORB with D=125/125/125 ms,	
L=4800 bytes, Sa/Sb=10/15 ms, Levels of Thread Cloning=3	118
Figure 5-39: Thread and CPU Utilizations of the A-ORB Agent	
with D=125/125/125 ms, L=4800 bytes, Sa/Sb=10/15 ms,	

Level of Thread Cloning=3	118
Figure 5-40: Process and CPU Utilizations of the A-ORB servers	
with D=125/125/125 ms, L=4800 bytes, Sa/Sb=10/15 ms,	
Levels of Thread Cloning=3	119
Figure 5-41: Case 1 (Time-out:1 ms): Throughput of the A-ORB with	
D=125/125/125 ms, L=4800 bytes, Sa/Sb=10/15 ms, QL=24,	
Levels of Thread Cloning=1~4	122
Figure 5-42: Case 1 (Time-out:1 ms): Throughput of the A-ORB with	
D=125/125/125 ms, L=4800 bytes, Sa/Sb=10/15 ms, QL=24,	
Levels of Thread Cloning=5~8	122
Figure 5-43: Case 2 (Time-out:100 ms): Throughput of the A-ORB with	
D=125/125/125 ms, L=4800 bytes, Sa/Sb=10/15 ms, QL=24,	
Levels of Thread Cloning=1~4	123
Figure 5-44: Case 2 (Time-out:100 ms): Throughput of the A-ORB with	
D=125/125/125 ms, L=4800 bytes, Sa/Sb=10/15 ms, QL=24,	
Levels of Thread Cloning=5~8	123

List of Tables

Table 1: Levels for the Workload Factors	65
Table 2: Parameters of Experiments on Server Demands	80
Table 3: Effect of S_a/S_b on R at $N=1$ ($D=0$ ms, $L=150$ bytes)	81
Table 4: Effect of S_a/S_b on X at $N=24$ ($D=0$ ms, $L=150$ bytes)	92
Table 5: Parameters of Experiments on Message Sizes	93
Table 6: Effect of L on R at $N=1$ ($D=0$ ms, $S_a/S_b=10/15$)	93
Table 7: Effect of L on X at $N=24$ ($D=0$ ms, $S_a/S_b=10/15$)	94
Table 8: Parameters of Experiments on Equidistant Nodes	99
Table 9: Effect of $D_{ca}/D_{as}/D_{cs}$ on R at $N=1$ ($L=150$ bytes, $S_a/S_b=10/15$).....	100
Table 10: Effect of $D_{ca}/D_{as}/D_{cs}$ on X at $N=24$ ($L=150$ bytes, $S_a/S_b=10/15$)..	100
Table 11: Parameters of Experiments on Non-Equidistant Nodes	105
Table 12: Parameters of Experiments on Multi-Threading	116
Table 13: Parameters of more Experiments on Multi-Threading	121
Table 14: Throughput with $N=24$, $Q_L=24$, $L=4800$ bytes, $S_a/S_b=10/15$ ms, and $D=125$ ms	121
Table 15: Parameters Set 1 of Experiments on Threshold Values	125
Table 16: Throughput with $N=8$, $L=150$ bytes, $S_a/S_b=10/15$ ms, and	

D=50 ms.....	125
Table 17: Parameters Set 2 of Experiments on Threshold Values	126
Table 18: Throughput with $N=8$, $L=4800$ bytes, $S_a/S_b=10/15$ ms, and	
D=0 ms	126

Chapter 1 Introduction

1.1 Background

As the Internet, World-Wide Web, and E-Commerce have been changing people's lives, decentralized and distributed systems have become more widespread and important in the computer industry. Also, the ability to run distributed applications on various platforms is crucial for maintaining a graceful upgrade strategy which keeps the legacy systems and makes use of more powerful and cheaper hardware and software provided by the new technology. This leads to a heterogeneous environment.

Developing distributed applications is complex and challenging to most developers. When computers and networks get faster and cheaper, communication software gets slower, buggier, and more expensive. Moreover, inter-operability is crucial in a distributed system consisting of diverse components. An effective middleware is a good solution to the above problems since it can act as a glue to hold the distributed system together and achieve high performance. By using middleware, it is possible to have two applications, written in different programming languages and implemented on top of two different operating systems, to interact with each other.

Many network programming mechanisms (e.g. BSD sockets) and reusable component libraries (e.g. Sun RPC) lack the features of type-safe, portable, reentrant, and extensible interfaces. Object-oriented technology can provide distributed computing with many

advantages (such as encapsulation, data abstraction, inheritance, portability, extensibility) as it does for non-distributed computing.

The main goal of **Distributed Object Computing (DOC)** is to make developers program distributed applications using familiar techniques, such as method calls on objects, since it integrates OO concepts into a distributed system. It promotes reusability of software and provides high performance by distributing application functionality over multiple computing nodes. When the provisions of DOC are realized and standardized, the long-awaited “plug and play” software components and “Software ICs” can become feasible [28].

The **Common Object Request Broker Architecture (CORBA)** is an open, well known standard for middleware architectures. Proposed by the Object Management Group (OMG), CORBA is a platform for Distributed Object Computing. It provides an object-oriented, client-server architecture in which clients can receive service from servers that may run on diverse platforms or be written in different languages.

1.2 Motivation

This thesis is concerned with the performance of the CORBA-based middleware system which provides the communication infrastructure and inter-operability in a heterogeneous distributed environment. A middleware system makes it easier and faster to get software to market or into production. Although Commercial-Off-The-Shelf (COTS) middleware products provide inter-operability, most of them incur a performance penalty. Achieving high system performance is important in the context of a number of distributed systems that

include telecommunication systems and various embedded applications. ORBs with poor performance will force developers of such applications to choose a low level communication facility such as socket (byte stream) in order to satisfy the performance requirement even though it will slow down the process of development. To achieve a high performance middleware software, the system architecture should be carefully considered and investigated.

Abdul-Fatah's thesis [2] investigates the relative performance of three client-middleware-server interaction architectures: Handle Driven ORB (H-ORB), Forwarding ORB (F-ORB), and the Process Planner (P-ORB). Based on the result of experiments, it suggests the Adaptive ORB (A-ORB) which combines the strengths of the H-ORB (see Section 3.1.1 for a description) and the F-ORB (see Section 3.1.2 for a description). This thesis implements a performance prototype of an A-ORB using Orbix, a commercially available CORBA compliant ORB and one of the most widely used ORBs software in the industry. The software is executed on a network of Sun workstations running Solaris 2.6.

1.3 Goals of this Research

The main goal of this thesis is to design and implement a performance prototype of an A-ORB using Orbix-MT, measure its performance and analyze the measurement result. The interaction architecture of the A-ORB is briefly described as follows: a client sends a request to the a-agent. The agent does a name to handle mapping for the server and may return the handle of the server back to the client or forward the request to a selected server depending on the system load. If the client receives the server's handle from the agent, it

will send the request to the server by using the returned handle. This is the typical behavior of H-ORB (see Section 3.1.1). If the agent chooses to forward the request to a server, then it behaves like the F-ORB (see Section 3.1.2). The server will send the reply to the client no matter where the request comes from. The performance of the A-ORB, which adapts its behavior to the current system load, is measured under different workload conditions.

This research is motivated by the following questions:

- a. Is the performance of the A-ORB superior to that of the purely handle-driven ORB (H-ORB) and the purely forwarding ORB (F-ORB)?
- b. What is the impact of message size which is transmitted between the client and server on the performance of A-ORB?
- c. What is the effect of inter-node delay on the system performance?
- d. How does the distance between the client, agent, and server affect the performance of A-ORB?
- e. Where is the performance bottleneck of the A-ORB?
- f. If multi-threading technique is applied in the A-ORB agent, how does it improve the system performance?
- g. How can we choose the threshold value to achieve the best performance?

1.4 Thesis Contributions

The main contributions of this thesis are:

- i) The design and implementation of a performance prototype of an A-ORB based on the commonly used ORB, Orbix.
- ii) An investigation of the performance of the Adaptive ORB which, to the best of our

knowledge, is one of the first works in adaptive middleware systems.

- iii) The experimental observation that a significant benefit in performance can be achieved by using a hybrid ORB architecture that changes its behavior and adapts to changes in system workload conditions.
- iv) An analysis of the impact of multi-threading on performance of the A-ORB.
- v) The demonstration of the possibility of devising an adaptive ORB, even without access to the source code of the middleware product. Higher performance can be expected if changes to the ORB source code are possible.

A paper describing some of the research reported in this thesis is accepted for publication in ACM's Principles of Distributed Computing Conference (PODC 2000) [29].

1.5 Thesis Outline

This thesis is divided into six chapters. Chapter 2 provides an introduction of CORBA and describes its components. It also presents the current performance issues in CORBA and related works for real time CORBA systems. Chapter 3 introduces the background of A-ORB and describes the implementation details. Chapter 4 describes what the work load factors are and how the measurement is performed on the system. Chapter 5 presents and discusses the result of the measurement. Chapter 6 provides the conclusion and suggested further research issues.

Chapter 2 CORBA Overview

2.1 Introduction

The Common Object Request Broker Architecture (CORBA) is a standard written by the members of OMG, a non-profit organization and the world's largest information technology consortium [21]. The OMG was founded in 1989 with the purpose of promoting the theory and practice of object technology in distributed computing systems. The CORBA specification was adopted in 1991.

Object Management Architecture (OMA) is a high-level vision of a complete distributed environment. It is composed of an Object Model and a Reference Model. The Object Model defines how objects distributed across a heterogeneous environment can be described, while the Reference Model characterizes interactions between those objects. The design of CORBA is based on the Object Model of the OMA. As shown in Figure 2-1, the core of the OMA is the Object Request Broker (ORB).

CORBA can be illustrated in different aspects [27][24][3]. It is:

- 1) An open standard for **Distributed Object Computing (DOC)**: CORBA defines a set of components that allow client applications to invoke operations with arguments on object implementations.

Figure 2-1: Object Model of OMA

- 2) A **middleware** which is used for interaction between two remote nodes: As a middleware, CORBA resides between the client and server and simplifies application development by providing a uniform view of the heterogeneous environment and offering a higher level integration than traditional untyped TCP bytestreams.
- 3) A layer of abstraction that hides the differences in a heterogeneous environment: CORBA layers over the network and OS in a heterogeneous environment by automating common network programming tasks such as

object location, object activation, parameter marshalling/unmarshalling, fault recovery and security.

- 4) A marriage between **Objected-Oriented Technology** and **Distributed System**: In the 1980s, the Remote Procedure Call (RPC) was introduced as an example of making certain features of a distributed system transparent to most programmers. Unfortunately, RPC is still a procedural approach and does not have a strong commercial standard. As the object-oriented applications have gradually become one of the mainstreams of the new generation software, CORBA uses an objected-oriented, rather than a procedural, approach.

Figure 2-2: The structure and components of CORBA

CORBA provides a communication infrastructure for a heterogeneous, distributed collection of collaborating objects. It makes distributed application development comparable in difficulty to developing centralized programs. It also provides an infrastructure to integrate application components into a distributed system.

2.2 Structure of an Object Request Broker

The structure of CORBA ORB and its components are shown in Figure 2-2 [25]. They are described individually in the following sections.

2.2.1 ORB Core

The ORB Core is the communication infrastructure which accepts requests from clients, activates the objects if they are not running, delivers requests to objects, and returns responses to clients. The key feature of the ORB is its transparency in facilitating client-object communication. The ORB hides the object location, object implementation, and object execution state. The client and server could exist in the same process, different processes or different hosts across the network. To get the service from an object, the client does not have to know where the object resides, how the object is implemented, in which programming language it was written, or whether the object is activated.

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 mask the differences between ORB Cores [20].

2.2.2 ORB Interface

The ORB Interface provides services which are the same for all ORBs. It decouples

applications from the implementation details of the ORB. The ORB Interface does not depend on the object's interface or object adaptor. There are a few operations that are common across all objects (such as converting object references to strings and back) and useful to both clients and implementations of objects, while most of the functionality of the ORB is provided through the object adaptor, stub, skeleton, or dynamic invocation.

2.2.3 Client, Objref, and Object Implementation

To access an object and invoke operations on the object, a client has to obtain the object reference (objref as shown in Figure 2-2) for that object. An object implementation provides the semantics of the object, usually by defining the data for the object instance and the code for the object's methods. An object reference is the information needed to specify an object within an ORB. Both clients and object implementations have an opaque notion of object references according to the language mapping, and are thus insulated from the actual representation of them. A client knows only the logical structure of the object according to its interface and experiences the behavior of the object through invocations. A client has no knowledge about the implementation of the object. An object implementation does not depend on the ORB or on how the client invokes the object.

2.2.4 OMG Interface Definition Language (IDL)

The OMG Interface Definition Language (OMG IDL) is used to specify the operations and types that the object supports and thus defines the requests that can be made on the object. It is an object contract language instead of being a complete programming language. From the IDL definitions it is possible to map CORBA objects into particular programming

languages. Language mapping includes definition of the language-specific data types and procedure interfaces to access objects through the ORB.

The use of IDL promotes language independence, location transparency, and modularity. The IDL compiler translates IDL into a target language which could be C, C++, Smalltalk, JAVA, and so on.

Figure 2-3: The general CORBA application code development paradigm under the support of the CORBA protocol compiler

2.2.5 IDL Stub and Skeleton

These two components serve as the glue between the client and server application respectively, and the ORB. As shown for the example C++ application in Figure 2-3 [4], the IDL compiler generates the stub for the client and the skeleton for the object implementation according to the IDL definitions. A stub is a mechanism that effectively

creates and issues requests on behalf of a client, while a skeleton is a mechanism that delivers requests to the CORBA object implementation. The stub will be compiled by the local compiler and linked into the client executable code. The skeleton will be compiled by another compiler (if the client and server reside on different hosts or are written in different programming languages) and be linked into the server executable code.

Dispatching through the stub and the skeleton is also called Static Invocation since each is built into the application code, and the interface offered by the server should be known at compile time. The stub is known as the Static Invocation Interface (SII). The parameters of operations are marshalled by the stub and unmarshalled by the skeleton.

2.2.6 Dynamic Invocation Interface (DII)

The Dynamic Invocation Interface (DII) allows a client application to invoke requests on any object without having compile-time knowledge of the object's interface. Instead of 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 information (operation, type of parameters) that a client must specify for DII may be obtained from an Interface Repository or other run-time source.

2.2.7 Dynamic Skeleton Interface (DSI)

Just as the DII allows clients to invoke requests without having access to static stubs, the Dynamic Skeleton Interface (DSI) allows servers to be written without having skeletons for the invoked object compiled statically into the program. Instead of being accessed

through a skeleton that is operation-specific, an object's implementation is reached through an interface that provides access to the operation name and parameters in a manner similar to the client's DII.

Dynamic skeletons may be invoked both through client stubs and through the dynamic invocation interface. Either style of client request should provide identical results. The main reason for DSI's introduction in CORBA 2.0 is to support the implementation of gateways between ORBs utilizing different communications protocols [32].

2.2.8 Object Adaptor (OA)

The Object Adaptor (OA) serves as the glue between CORBA object implementations and the ORB itself. An object adaptor is the primary way that an object implementation accesses the services provided by the ORB. These services include the following [32]:

- Object registration
- Object reference generation and interpretation
- Object activation and deactivation
- Request demultiplexing
- Method invocation
- Security of interactions

CORBA provides the Basic Object Adaptor (BOA), which is the first defined and the most commonly used object adaptor. Examples of other types of object adaptors can be found in [24].

2.2.9 GIOP/IIOP

The General Inter-ORB Protocol (GIOP) specifies transfer syntax and a standard set of message formats for ORB inter-operation over any connection-oriented transport [24][20].

The Internet Inter-ORB Protocol (IIOP) determines how the GIOP is built over a TCP/IP transport. As a full solution for the heterogeneous environment, CORBA is required to be inter-operable so GIOP/IIOP was introduced and defined at CORBA 2.0. They provide protocol-level ORB inter-operability in a general and low-cost manner. The GIOP specification consists of the following elements:

- **Common Data Representation (CDR) definition:** the transfer syntax that maps OMG IDL types from the native end system format to a bi-canonical (able to support both little-endian and big-endian) format used to transfer data over networks.
- **GIOP Message Formats:** definitions of seven types for sending requests, receiving replies, locating objects, and managing channels.
- **GIOP Transport Assumptions:** types of transport protocols that can carry GIOP messages.

By using IIOP, ORBs are able to communicate with other ORBs that publish their locations in the inter-operable object reference (IOR) format. The general ORB inter-operability architecture introduced in CORBA 2.0 provides the direct ORB-to-ORB inter-operability and bridge-based inter-operability. Direct inter-operability is possible when two ORBs reside in the same domain and understand the same object reference. If two ORBs

reside in different domains, they need bridge-base inter-operability to communicate with each other.

2.2.10 Implementation Repository

The Implementation Repository contains information which allows the ORB to locate and activate object implementations. It is also a common place to store additional information related to implementations of ORB objects.

2.2.11 Interface Repository

The Interface Repository allows the OMG IDL information of objects to be accessed and written by a program at run-time. It is also a common place to store additional information related to interfaces to ORB objects.

2.3 Performance Issues in CORBA

Considerable overhead is found in conventional implementations of communication middleware when they are used for performance-sensitive applications over high-speed networks. As gigabit networks become widespread, inefficient middleware will force programmers of performance sensitive applications to use lower-level mechanisms to achieve the necessary transfer rates. Therefore, eliminating the sources of overhead to make CORBA succeed is very important to mission-critical applications. General applications can benefit from the improvement of performance as well.

The existing performance issues in CORBA implementations are described in the following sections.

2.3.1 Presentation Layer

The responsibility of the presentation layer includes:

- **Marshalling:** transforming typed data objects from higher-level representations to lower-level representations. This is done by the stub in CORBA.
- **Unmarshalling (or demarshalling):** transforming typed data objects from lower-level representations to higher-level representations. This is done by the skeleton in CORBA.

This layer converts application-level data into a portable format that masks byte order, alignment, and word length differences. The presentation layer is indicated as the major bottleneck in high-performance communication subsystems because of excessive data copying and conversion (marshalling/unmarshalling) overhead [8].

2.3.2 Memory Management

Conventional ORB end systems suffer from excessive dynamic memory management and data copying overhead [27]. Excessive copying increases memory and bus overhead for applications. For instance, the I/O subsystem and the ORB core each normally allocate a memory buffer for each incoming request. The I/O subsystem typically copies its buffer to the buffer allocated by the ORB core. The IIOP unmarshalling code also allocates memory to store the decoded request parameters. Moreover, IDL skeleton would allocate memory to contain those parameters. As a result, the memory allocation/deallocation operations and the excessive data copying among multiple layers in an ORB end system cause significant overhead [26].

In general, dynamic memory management is problematic for real-time systems since heap fragmentation can cause unpredictable results for different message sizes and different workloads. Discussions are provided in [26] and [27]. Although this problem may occur in many systems, it is not present in the adaptive middleware system discussed in this thesis.

2.3.3 Demultiplexing

An object method is uniquely identified by an object key and an operation name. The Object Adaptor is the component in the CORBA architecture which associates an object with the ORB, demultiplexes incoming requests to the object, and dispatches the appropriate operation of that object. The demultiplexing strategy used by an ORB can impact performance significantly. Conventional ORB demultiplexes the incoming requests to appropriate operation of the object using the steps shown in Figure 2-4 [7].

- Steps 1 and 2: The OS protocol stack demultiplexes the incoming client request through the data, networks, and transport layers up to the boundary of kernel and ORB Core.
- Steps 3, 4, 5: The ORB Core locates the appropriate Object Adaptor, the object, and the IDL skeleton of the target operation by using the addressing information in the client's object key.
- Step 6: The IDL skeleton locates the appropriate operation, unmarshals the request into operation parameters, and performs the upcall to the operation.

Figure 2-4: Layered CORBA Request Demultiplexing

Layer demultiplexing increases latency by increasing the number of times that internal tables must be searched. This becomes a serious performance problem when a large number of operations appear in an IDL interface or a large number of servants are managed by an ORB. Moreover, layered demultiplexing may cause priority inversions since object-level QoS is not available to the lowest level device drivers and protocol stacks in the I/O subsystem of an ORB endsystem [7]. For example, since a dispatcher of requests inside the ORB is not priority based, a request associated with a lower QoS operation may be dispatched before a request associated with a higher QoS operation.

Conventional ORBs have significant demultiplexing overhead. Demultiplexing requests in Orbix could take up to 17% of the total server processing time [6]. It is suggested that hashing or de-layered demultiplexing strategies can be used rather than the linear search which is known to be expensive and non-scalable.

2.3.4 DII and DSI

In comparison with Static Invocation Interface (SII), Dynamic Invocation Interface provides a more flexible way to construct and invoke a CORBA request at run time by querying the Interface Repository which is also a CORBA object. The creation and invocation of each DII request could require several actual remote invocations so such a request is several times more costly than an equivalent static invocation.

It has been shown that the CORBA implementations spend a significant amount of time creating a request, populating it with parameters and marshalling the data [10]. In particular, CORBA implementations of the DII and DSI perform much worse than the SII [8], whereas the SII does not perform well relative to programming with a lower-level communication mechanism such as TCP/IP.

2.3.5 IIOP

The IIOP uses CDR transfer syntax to map IDL data types into a portable network format. Some measurement has been done on the communication performance between heterogeneous ORBs. The results show that some implementations of IIOP protocols have yielded relatively poor performance over high-speed networks due to the excessive marshalling/unmarshalling overhead, data copying, and high-levels of function call overhead [9]. Moreover, the IIOP requests are transferred by using TCP/IP in which flow control and congestion control algorithms may cause some performance issues under real time systems.

2.4 Real Time CORBA

Predictability and performance are two main elements to be satisfied in real-time systems. Hard real-time systems, such as regulators for nuclear reactors, require a guarantee that the time constraints will be met. Failure to meet time constraints will cause a system failure. Soft real-time systems, such as telecommunication switches, allow a deadline to be missed and the system service to deteriorate.

Conventional ORBs are not suited for performance sensitive, real-time applications since they lack QoS specification APIs, QoS enforcement, real-time programming features, and performance optimizations [27]. To meet the characteristics of real-time systems, the development of a standard for Real-Time CORBA (RT CORBA) is proceeding [26]. The main goal of RT CORBA is to deal with the expression and enforcement of real-time constraints on end-to-end execution in a CORBA system to secure predictability.

2.4.1 Requirements for Real-Time CORBA

To illustrate why conventional ORBs cannot satisfy the characteristics of real-time applications, the requirements for RT CORBA were addressed in [12][33][27][14]. The main requirements are described as follows.

- a. **Global Timer:** All clocks on nodes in an ORB should be synchronized within a limited amount of skew, so a global timer is required.
- b. **Bounded Message Delay:** The maximum latency of message transfer must be fixed to ensure a worst-case message delay. The information also needs to be available so that end-to-end timing can be established.
- c. **Global Priority:** To utilize limited resources and secure predictability, priorities

should be established in RT CORBA. Various Operating Systems may provide different priority schemes so a global priority should be provided by RT CORBA to allow applications to make prioritized invocations in the same way between nodes with different priority schemes. Priority mappings should also be provided to map a global priority into the native priority and vice versa.

- d. **Priority Queueing and Scheduling:** A higher priority task should be scheduled before a lower-priority task. Real-Time CORBA requires that all CORBA-level software modules use priority based queueing. Forcing Priorities at all points in the end-to-end path is necessary, especially for the hard real-time systems.
- e. **Priority Inheritance:** When a lower priority request blocks the execution of a higher priority task request for an indefinite period, priority inversion, an undesired situation for real-time systems, will occur. Therefore, all components used in the underlying CORBA environment that synchronize tasks by blocking one task for another should implement and use priority inheritance.
- f. **Real-Time Event:** The standard CORBA Event Service specification lacks some important features required by real-time applications. These features include real-time dispatching and scheduling which can provide real-time guarantees and also include periodic rate-based event processing and efficient event filtering and correlation mechanisms.
- g. **Support for Multi-threading:** Multi-threading is used extensively in real-time systems. It is desirable to have multi-thread support and be thread-safe. Real-time scheduling and priority-based strategy should be available at the thread level.
- h. **QoS Specification APIs:** Real-time CORBA should support end-to-end QoS requirements. To achieve this, APIs that specify the end-to-end QoS should be provided. QoS APIs should permit clients to indicate the priorities of their requests and inform an ORB how frequently to execute operations that have periodic deadlines. This could be done by combining extended IDL and ORB QoS

interfaces.

- i. **QoS Enforcement:** Without network and OS support for QoS enforcement, real-time ORBs cannot deliver end-to-end guarantees to applications. ORBs must be able to schedule resources to allow high priority client requests to run to completion and avoid unbounded priority inversion. This scheduling should support preemptive dispatching to allow a higher priority task to be served by a CPU as soon as it becomes ready to run.

2.4.2 Real-Time CORBA Standard

As mentioned above, Real-Time CORBA is underway and a Joint Revised Submission for Real-Time CORBA [22] is available on the OMG web page. It is co-submitted by companies such as Nortel, Lucent, Iona, Sun, and HP. The main features in this specification are:

- **Real-Time CORBA Manager:** An interface which handles operations associated with the configuration of the Real-Time CORBA and deals with the creation and destruction of Real-Time CORBA IDL interfaces.
- **Real-Time CORBA Priority and Priority Mapping:** To overcome the heterogeneity of different Operating System-provided priority schemes and to allow applications to make prioritized invocations in the same way between nodes with different priority schemes, the Real-Time CORBA priority, which is a universal and platform independent priority scheme, is defined. A priority in the Real-Time CORBA can be mapped into a local priority which is native to a particular Operating System.

- **Thread Scheduling:** Multiple thread programming becomes more and more important in real-time systems since a thread consumes much less system resources and takes less overhead in comparison to heavy weight processes. For instance, no address space switching overhead will occur when CPU serves another thread of the same process. Real-Time CORBA uses threads as a schedulable entity.
- **Threadpools:** Real-Time CORBA uses a Threadpool to manage thread execution on the server-side. Features provided by Threadpool include preallocation of threads, partitioning of threads, bounding of thread usage, and buffering of additional requests.
- **Real-Time CORBA Mutex and Priority Inheritance:** A Real-Time CORBA implementation must provide the Mutex mechanism that implements some form of priority inheritance protocol and allows a consistent priority inheritance scheme to be delivered across the whole system.
- **Non-Multiplexed Connections:** A client in Real-Time CORBA can obtain a private transport connection to a server. This connection will not be shared or demultiplexed with other client-server object connections. As mentioned earlier, the layered multiplexing of conventional ORBs causes significant costs in performance.

- **Invocation Timeouts:** Real-Time CORBA applications can set a timeout on an invocation to support the requirement of Bounded Message Delay. It bounds the time that the client is blocked waiting for a reply from the server, so this feature enhances the predictability of ORB systems.

In addition to the above, the specification of Real-Time CORBA also defines Priority Banded Connections, Client and Server Protocol Configuration to extend and enhance the real-time features of CORBA system.

The work described in this thesis is based on a general CORBA compliant ORB. When RT CORBA specifications are standardized and real-time ORBs such as TAO [27] become more prevalent, adaptation of the work presented in this thesis to RT CORBA may be fruitful.

2.5 Applications of CORBA

CORBA and DCOM (Distributed Components Object Model) are two of the most important and popular middleware technologies. Although DCOM, which was developed by Microsoft, is strongly supported in Windows platforms, CORBA is the most widely used middleware in the UNIX world [18]. As strongly recommended for distributed applications, CORBA is used in many areas such as telecommunication, health care, electronic commerce, transportation, banking, and aerospace [21]. A network management system (NMS) which benefits from using CORBA in satisfying new system requirements is described briefly in [11].

Due to the increasing complexity of networks, a number of requirements for network

management systems have become more important. These requirements include providing standard interfaces for information sharing among management systems, having extensibility for handling change quickly, and providing a means to manage large networks. Since NMS has to deal with the distributed and heterogeneous environment, it can benefit from using CORBA significantly. The advantages of using CORBA for NMS include:

- **Providing an open/standard interface:** With standard interfaces in CORBA, the management system does not need to know vendor-specific information and can handle network resource management in a common way.
- **Creating an extensible application framework:** Network management architecture becomes more capable of handling new features and changing with networking technology without requiring significant redesign and coding because of the object-oriented technology supported by CORBA.
- **Scalable solutions:** CORBA provides a framework for distributed computing with heterogeneous components. The ability to add components implemented with various technologies enhances the scalability of the system.
- **Programming language independence:** Since CORBA is language-independent and provides several language mappings, network management systems can have servers implemented in C++ with better performance whereas their clients or user interfaces are written in Java which is portable, simple, and flexible.

Chapter 3 *The Adaptive ORB (A-ORB)*

3.1 Introduction

Three different architectures called the *Handle-Driven ORB (H-ORB)*, the *Forwarding ORB (F-ORB)*, and the *Process Planner ORB (P-ORB)*, were described and implemented in [2]. An Adaptive ORB (A-ORB) which combines the strengths of H-ORB and F-ORB is developed. The A-ORB is expected to have better performance than a pure H-ORB or a pure F-ORB. The H-ORB, F-ORB, and A-ORB will be described in the following sections. Since the P-ORB is not in the scope of this thesis, it will not be discussed.

3.1.1 The Handle-Driven ORB (H-ORB)

A widely used ORB architecture called the handle-driven ORB (H-ORB) is presented in Figure 3-1. In H-ORB, a client performs a binding operation when it requires a service. When an agent receives the binding request, it locates an available server, which is capable of performing the operation, and returns the handle of the server back to the client. The agent will activate the server (object implementation) if it is not already active. The handle (object reference) returned by the agent contains all information required to interact with the selected server. Therefore, the client can use this handle to send a request which invokes a remote method on the object implementation. The object implementation deals with this request and sends a reply back to the client.

Figure 3-1: High level interaction of the H-ORB Architecture

The client does not have to know where the server is because the agent, as a component of ORB, provides *location transparent* service to the client. The interaction between the client, agent, and server is shown in Figure 3-1. In fact, the client and object implementation could exist in the same process, different processes on the same host, or on different hosts connected by the network.

3.1.2 The Forwarding ORB (F-ORB)

Another ORB architecture called the Forwarding ORB (F-ORB) forwards the request from the client to a selected server. As shown in Figure 3-2, the client sends the entire service request to the F-ORB agent, termed F-agent. The F-agent locates the server, activates it if it is required, and then forwards the request to the selected server. To enable the server to send the reply directly to the client, either the name of the client making the request or its stringified object reference (objref) is embedded in the message. If the first approach, in which the name of the client is embedded, is used, the server can obtain the handle (objref) from the agent and store it locally at the beginning. For a subsequent request, the server only has to compare the name of the client with the locally stored handles to get the objref which is used in order to send the result back to the client. If the second approach that uses an embedded stringified objref is used, the server only has to unstringify the stringified objref and use the unstringified objref to send the reply directly back to the client.

Figure 3-2: High level interaction of the F-ORB Architecture

3.1.3 Motivation for the Adaptive ORB

The total number of messages per server interaction in F-ORB is 3, which is lower than the total number of messages in the H-ORB which is 4. Therefore, the F-ORB results in a smaller number of message transfers in comparison with the H-ORB. However, the marshalling/unmarshalling cost is minimized in the H-ORB since the marshalling/unmarshalling only happens while translating the typed data in the remote method invocation. Since the entire request is not sent, the binding operation at the agent leads to a smaller marshalling/unmarshalling overhead in comparison to the F-ORB.

A potential problem with the F-ORB is the possibility that the F-agent becomes a performance bottleneck at high system load. A performance bottleneck is the hardware device or software process which limits the growth of the system throughput when the number of clients increases. The bottleneck device is the device that saturates first in response to a growth in the workload. Examples of hardware bottlenecks in the conventional time sharing computer system include the CPU or disk. Software bottlenecks occur on client-server systems because of the synchronous nature of the send-wait-reply mechanism used in the client-server interaction. After a process sends a message, it waits until the message is received from the server. Thus it may utilize 100% of the time in terms of executing and waiting for the message to be received, although the CPU on which it executes is under utilized [19].

According to the result of performance measurements reported in [2], the F-ORB behaves better than the H-ORB when the number of clients is small. When the number of clients increases, the H-ORB exhibits a much better performance than the F-ORB. At low

system load, the F-ORB wins over the H-ORB because of the lower number of messages per method invocation. The F-ORB agent incurs more overhead than the simpler H-ORB agent. As a result, a software bottleneck is found at the F-agent at high system load.

Since the F-ORB performs better at the low load condition and the H-ORB demonstrates superior throughput at medium and high load conditions, a new ORB architecture which combines the strength and functionality of both the F-ORB and H-ORB was proposed. This new ORB architecture is called Adaptive ORB (A-ORB) since it can dynamically switch from the F-ORB to the H-ORB mode and vice versa depending on the workload condition.

3.1.4 The Adaptive ORB

The A-ORB is an architecture in which the agent (broker) switches its behavior based on the workload conditions. The agent in the A-ORB is called the A-agent. The high level interaction of the A-ORB is shown in Figure 3-3. In the A-ORB, a client sends an `a-bind()` request which includes the entire service request to the A-agent. The A-agent locates the server and activates it if it is needed. The A-agent checks the number of waiting client requests to decide its own behavior. If the number of waiting client requests is less than a pre-defined threshold, then the A-agent forwards the request to a selected server. This behavior is exactly the same as the F-ORB. In another case, if the number of waiting requests is equal to or greater than the threshold, the A-agent will return the handle of the sought server to the client, which in turn sends the request to the server directly by using

Figure 3-3: High level interaction of the A-ORB Architecture

the returned handle. This is the typical behavior of the H-ORB. In either case, the server performs the service and replies back to the originating client. To let the server send a reply to the client directly without going through the A-agent, the name or object reference of the originating client is embedded inside the request message sent to the server.

3.2 The implementation of the Adaptive ORB

In this research, the A-ORB is implemented on the Orbix-MT (Multi-Threaded version), which is commercially available and one of the most popular ORBs [23][3]. Since the source code is not available, some additional work has been performed around this implementation. For example, a client in A-ORB should send `a-bind()` to the A-agent instead of asking the Orbix agent (referred to as the default agent) for a handle of the server. Ideally, the default agent and the A-agent should be integrated together and can be contacted via the special ORB function call, `a-bind()`. Without the availability of source code, this intent could not be achieved. In the A-ORB implemented here, the A-agent is implemented as a separate process and the client still has to contact the default agent to get the handle of the A-agent and use this handle to send request to the A-agent.

Actually, there is not any process called “agent” in Orbix; the functionality of the default agent is provided by the Orbix Daemon and the Implementation Repository. Before being able to be invoked, the server (object implementation) should be registered in the Implementation Repository. In the system described in this thesis, an Orbix Daemon process exists on each node. Its responsibility includes receiving the commands to register objects in the Implementation Repository, locating the objects by searching the

Implementation Repository, and invoking the objects if it is not already active. The implementation of the A-ORB is described in detail in the following sections.

3.2.1 Software Architecture of the A-ORB

Figure 3-4: The processes used in this implementation of the A-ORB

The processes and messages (operations) transferred in the A-ORB system are shown in Figure 3-4. The parameters and data types are listed in Figure 3-5. The client sends the binding request, `a_bind()`, which includes the server name and the service needed. The decision of forwarding the request to the server or of returning the handle of the server to the client is made by the agent. If the decision is the former, then the agent will call the client's operation, `handleFromAgent()`, and pass the stringified object reference (handle) to the client. In this case, the client will call the server's method, `request()`, to send the request

to the server by using the returned handle. If the decision is the latter, the agent will call the server's method, `request()`, and relay the request to a selected server. After the server finishes the request, it will call the client's method, `replyFromServer()`, and send the reply back to the client which originated the request. In addition to the above operations, the other operations, including `startTimeFromCP()` and `termCmdFromCP()` shown in Figure 3-5, are used by the control process which is described in detail in Section 4.3.1.

The number of client processes is used to control the system load and could be as high as 24 in our experimental environment. Although we provide two classes of servers, these server processes actually use the same code. Each client cycle is comprised of two requests. The client sends the first request to server A. After receiving the reply from server A, the client sends the second request to server B and waits for its reply. For the purpose of load balancing, each server class contains two server processes (e.g. Server A1 and A2). The policy to select a specific server process from a class of server is based on the round-robin algorithm.

The most important component in the A-ORB is the A-agent. There is only one agent process but it contains three types of threads (see Figure 3-6) which are described.

- **Main thread:** it is the listener thread which is always waiting for a request from one of the clients. Whenever a request arrives, the listener thread checks the number of enqueued client requests in the forwarding queue. It compares the number of the waiting client requests with a threshold value and then makes the decision whether the agent should forward the request to the server or return the server's handle back to the client.

```
interface aClientSupport { // IDL operations of the A-client
void handleFromAgent(in string objRefServer);
void replyFromServer(in short result);
void startTimeFromCP(in long startTime, in short numOfClients);
void termCmdFromCP();
};

interface aAgentSupport { // IDL operations of the A-agent
void a_bind(in string serverName, in string objRefClient,
            in short reqServTime, in short delayClientAgent,
            in short delayAgentServer, in short delayClientServer,
            in short clientIndex, in string message);
void startTimeFromCP(in long startTime, in short numOfClients);
void termCmdFromCP();
};

interface aServerSupport { // IDL operations of the A-server
void request(in string objRefClient, in short reqServTime,
            in short delayClientServer, in short clientIndex, in string message);
void startTimeFromCP(in long startTime, in short numOfClients);
void termCmdFromCP();
};
```

Figure 3-5: IDL source code of the client, agent, and server

Figure 3-6: The interactions inside the A-agent process

- **Forwarding thread:** if the listener thread decides to forward the request, it will put the request into the forwarding queue which is accessible by both the listener thread and the forwarding thread. The forwarding thread periodically polls the forwarding queue. It will get one entry (request) from the forwarding queue and forward the request to the server if the forwarding queue is not empty.
- **Returning thread:** if the decision is made to return the handle of the server back to the client, the listener thread will copy the request into the returning queue which is accessible by both the listener thread and the returning thread. The returning thread checks the returning queue periodically. If there is at least one entry available in the returning queue, the returning thread will get the request information, retrieve the stringified objref of the client from the request, unstringify it, and send the handle of the server back to the client by using the unstringified objref of the originating client.

In this implementation of A-ORB, there is only one listener thread. However, there can be more than one returning thread and forwarding thread depending on the A-agent's configuration file. Multi-threading can enhance the performance significantly, especially for I/O bound systems. This can be verified in Section 5.4. Separating the requests based on their types and storing them in separate forwarding and returning queues is done to improve system efficiency. Inserting the requests into a single queue will increase the complexity of the polling algorithm and may result in increased contention at the common queue.

3.2.2 Pseudo Codes of the A-ORB

3.2.2.1 Pseudo code for the A-client

The pseudo code for the synthetic clients is presented briefly.

1. Register itself as an Orbix server and make it available to other Orbix clients (A-server and A-agent)
2. Read items from the client's configuration file including:
 - msgSize: message size
 - reqServTime: the request service time (CPU time)
 - nodeDelay: inter-node communication delay
 - numOfCycles: number of client cycles
 - titleOfCfg: the title of the experiment
3. Get the objref for itself and stringify it
4. Get an objref for the A-agent.
5. while(1)
 - 5.1. if any event arrives
 - The called operation saves parameters and returns immediately
 - endif
 - 5.2. if (event type == START_TIME)
 - send the first client request to the agent and request for a service from server A
 - endif
 - 5.3. if (event type == HANDLE)
 - send the request to a server by using the returned handle from the agent
 - endif
 - 5.4. if (event type == REPLY)
 - 5.4.1. if the reply comes from server A, then send the second request for the server B to the agent
 - 5.4.2. if the reply comes from the server B, then save the result in the local repository and start another client cycle
 - 5.4.3. if it is the reply for the last request, then calculate all results, write them into log file, and send the average values to the control process
 - endif
 - 5.5. if (event type == TERM)
 - read out all pending events and terminate this process
 - endif
- endwhile

3.2.2.2 Pseudo code for the A-agent

The pseudo code for the synthetic agent is presented briefly.

The Listening Thread (Main Thread):

1. Register the agent as an Orbix server and make it available to other Orbix clients (A-client and A-server)
2. Read items from the agent's configuration including:
 - queueLength: Threshold value for the forwarding queue length
 - numOfReturnThreads: number of the returning threads
 - numOfForwardThreads: number of the forwarding threads
3. Get the objrefs of server A1, A2, B1, B2 and store them in global variables
4. Initialize resources used by the forwarding/returning threads such as mutex locks
5. Invoke forwarding/returning threads
6. while(1)
 - 6.1. if any event arrives
 - The operation called by the client saves parameters and releases client immediately
 - endif
 - 6.2. if (event type == ABIND)
 - if (queue length of the forwarding queue >= QL) {QL is a threshold value define in Section 4.1.1}
 - put the request (a set of parameters) into the returning queue and increase its queue length by 1
 - else
 - put the request into the forwarding queue and increase its queue length by 1
 - endif
 - 6.3. if (event type == COMPUTE_STATS)
 - calculate thread and cpu utilization of the agent and send them to the control process.
 - endif
 - 6.4. if (event type == TERM)
 - read out all pending events and terminate this process
 - endif
- endwhile

Note: parameters for the a_bind() include the following:

- serverName: the server name requested by the client
- clntObjRefStr: stringified objref of the client which sent the request
- reqServTime: cpu time on the server requested by the client
- delayClientAgent: inter-node delay between the client and agent
- delayAgentServer: inter-node delay between the agent and server
- delayClientServer: inter-node delay between the client and server
- message: message with a specified size

The forwarding thread:

1. while(1)

```

1.1. if the forwarding queue is empty
    sleep  $t$  ms
else
    1.1.1. get the a-bind() parameters from the forwarding queue and decrease
           its queue length by 1
    1.1.2. select the server index (e.g. A1 or A2, B1 or B2) by the round-robin algorithm.
    1.1.3. sleep "delayAgentServer" ms
    1.1.4. forward the request to the selected server
endif
endwhile

```

The returning thread:

```

1. while(1)
    1.1. if the returning queue is empty
        sleep  $t$  ms
    else
        1.1.1. get the a-bind() parameters from the returning queue and decrease
               its queue length by 1
        1.1.2. select the server index (e.g. A1 or A2, B1 or B2) by the round-robin algorithm.
        1.1.3. unstringified the sending client's objref, clntObjRefStr
        1.1.4. sleep "delayClientAgent" ms
        1.1.5. return the handle of the selected server back to the originating client
    endif
endwhile

```

Note: The thread sleeps for t ms to avoid cpu exhaustion by infinite loops

3.2.2.3 Pseudo code for the A-server

The pseudo code for the synthetic servers is presented briefly.

```

1. Register the server as an Orbix server and make it available to other Orbix clients (
   A-client and A-agent)
2. while(1)
    5.1. if any event arrives
        The called operation saves parameters and returns immediately
    endif
    5.2. if event type == REQUEST
        5.2.1 run a while loop to consume a specified amount of the cpu time
        5.2.2 send the result back to the client
    endif
    5.3. if (event type == START_TIME)
        calculate process and cpu utilization of the server and send them to

```

```
        the control process
    endif
5.4. if (event type == TERM)
        read out all pending events and terminate this process
    endif
endwhile
```

3.2.3 Execution Scenario of the A-ORB

The execution sequence and interaction of the main components of this architecture are shown in Figure 3-3 and Figure 3-4. In this A-ORB implementation, the sender calls the operation provided by the receiver. Before sending any message, the sender sleeps to simulate the inter-node delay (see Section 4.1.1 for details). The description provided next describes the behavior and roles of the components of the A-ORB client/server architecture.

3.2.3.1 The Client

- 1) When the A-client is invoked, it registers itself as an Orbix server and makes itself available to other Orbix clients. In the CORBA architecture, clients exchanges messages with servers by calling the operations provided by servers. A client process could also be a server process. In Orbix, a server process calls `setServerName()` and `impl_is_ready()` to make itself available to other client processes. In A-ORB, the client expects to get the reply from a server so it must register itself to become accessible.
- 2) The A-client reads the A-client's configuration file to get the needed arguments to be passed to the sought server. The content of the client's configuration file will be described in Section 4.1.2.

- 3) Since the source code of Orbix is not available, some initialization for the client needs to be done. The initialization steps are described in Section 3.3.3.
- 4) Since CORBA supports the location transparency, a client does not need to know the location of the server. To get the service, the A-client must contact the A-agent in its domain to reach that server. The A-client sends an *"a-bind"* command which includes the server name, the service needed and other information (see Section 3.2.2.2) required by the A-agent or A-server. Since the A-client is not always expecting a reply from the A-agent, the operation called by the A-client returns right after the request is received by the A-agent. Therefore, the A-client becomes free to do any further activities. Actually, the A-client enters its main listening loop and waits for messages.
- 5) When the A-client receives a message, it terminates the listening status. When this occurs, two possible situations could happen. They are described below:
 - The call comes from the A-agent which decided to return the handle of an A-server back to the A-client. Ideally the A-client should be able to use this handle to invoke the remote server directly. Since we did not have access to Orbix source code, the A-agent itself is also a process instead of being a part of Orbix ORB. Therefore, the handle from the A-agent is a stringified object reference. The A-client has to call the Orbix function, `string_to_object()`, to unstringify it and convert it into the handle (objref). After the A-client uses the handle to send the request to the A-server, it waits until the reply from the A-server is available.

- The call is the reply from a server. This corresponds to the situation in which the forwarding mode of operation was used by the A-agent. The client sends two requests in a request cycle. The first request is for server A and the second one is for the server B. If the reply comes from the server A, the client will send the second request to the server B. If the reply is from the server B, it will calculate the response time for this cycle (see Section 4.2.1) and store the result locally. Then the client will initiate another request cycle.

3.2.3.2 The A-agent

The Main (Listener) Thread:

- 1) Similar to the A-client, the A-agent has to register to be an Orbix server and make itself available to other Orbix clients.
- 2) The main thread reads the A-agent's configuration file, which includes the threshold value of the queue length, the number of returning threads, and the number of forwarding threads. These items are described in details in Section 4.1.2.
- 3) Some initialization has to be done before the main thread can process the binding operation from a A-client. It is described in Section 3.3.3. After the initialization, the main thread enters the main event loop and waits for incoming messages.
- 4) When an a-bind request arrives from an A-client, the main thread has to make a decision if it should return the handle of the A-server back to the A-client or

forward the request to the A-server. If the number of enqueued client requests in the forwarding queue is equal to or greater than the threshold value for the queue length which is read from the configuration file, then the A-agent decides to return the handle of the server. This is a typical behavior that A-ORB uses to reduce the effect of the software bottleneck. Whenever the A-agent decides to return the handle, the main thread puts the request into the returning queue and increases its queue length by one. If the number of enqueued client requests is less than the threshold value, the A-agent decides to forward the request to the server. Therefore, the main thread stores the request into the forwarding queue and increases its queue length by one.

The Returning Thread:

- 1) The returning thread is created by the main thread at the initialization stage of the A-agent. After it is activated, the returning thread enters a while loop until it is dead. The activities inside the while loop are described as follows.
- 2) The returning thread checks the returning queue length. If the queue is empty, the returning thread will sleep up to t ms to wait for a request entry in the returning queue. Sleep() is not thread-safe. Due to this problem found in Solaris multi-threaded environment (see Section 3.4.6), this “sleep” mechanism is implemented by using the condition variable with time-out option, cond_timedwait() [31].
- 3) Whenever at least one request entry is put into the returning queue by the main

thread, the returning thread gets the request entry from the returning queue which includes the server class, the stringified objref of the originating client, and the inter-node delay between the agent and the client (see Section 3.2.2.2). Each class of A-servers consists of two server processes (see Figure 3-9). The returning thread selects a server of the specified server class by using a round-robin strategy. For example, there are two server instances for server class A which are server A1 and server A2. If server A1 is selected for this client request, server A2 will be chosen for the next request. The handle (stringified objref) of the selected server is obtained from the local repository and then is sent to the A-client by unstringifying the stringified objref of the A-client in the request entry. The returning thread sleeps to simulate the inter-node delay between the agent and the client. Simulating the inter-node delay in this fashion allows us to vary the inter-node delay and study its effect on performance. A more detailed discussion is provided in Section 4.1.

The Forwarding Thread:

- 1) The forwarding thread is also created by the main thread during system initialization. Its main responsibility is to forward the client request to the server. It periodically checks the forwarding queue inside a while loop until it is dead.
- 2) In the same way as the returning thread, the forwarding thread checks the forwarding queue length. If the forwarding queue is empty, then it sleeps and waits until at least a request entry is put into the forwarding queue by the main

thread.

- 3) Whenever the forwarding queue is not empty, the forwarding thread retrieves the information from the forwarding queue and chooses a server instance from the specified server class by using a round-robin strategy. Before the forwarding thread forwards the client request to the selected A-server, it sleeps for a predetermined amount of time to simulate the inter-node delay between the A-agent and A-server.

3.2.3.3 The Server

- 1) When the A-server is activated, it registers itself as an Orbix server and makes it available to other Orbix clients. The A-server is an Orbix server since it provides the operation, `request()`, to be called by either the A-client or the A-agent. However, the A-server is also an Orbix client since it calls the operation, `replyFromServer()`, which is provided by the A-client.
- 2) The server enters its main event loop. When a request arrives directly from the A-client or is forwarded from the A-agent, the skeleton intercepts the request, unmarshals the data, and then calls the sought method in the object implementation. The A-server does not know who the sender is since it provides the same operation, `request()`, to be called by either the A-client or the A-agent.
- 3) Depending on the value of the request server demand in the client request, the A-server will consume a specified amount of CPU time. Upon the completion

of the request processing, a reply is sent to the A-client by calling `replyFromServer()`. The stringified objref of the originating A-client is embedded in the request, so the A-server has to unstringify the objref and use it to send the reply. Before sending the reply, the A-server sleeps for a given amount of time to simulate the inter-node delay between the A-server and the A-client.

3.3 Initialization and Request Path for the A-ORB

3.3.1 Activation Modes

To understand how the initialization steps of the A-ORB is implemented in Orbix, it is necessary to explain how the Orbix server is activated. CORBA provides three different modes to launch servers and control how servers are implemented as processes by the underlying operating system [24]. These activation modes are described:

- i) **Shared activation mode:** This is the most commonly used activation mode and it is the default one. All of the objects with the same server name on a given host are handled by the same process on that host.
- ii) **Unshared activation mode:** In this mode, individual objects of a server are registered with the Implementation Repository. Only one active object of a given implementation can be managed in one server process.
- iii) **Per-method-call activation mode:** Individual operation names are registered with the Implementation Repository. A process is created to handle each operation call, and that process is destroyed once the operation has been completed.

In Orbix, a server process could be activated by the Orbix daemon automatically or by the user command manually. The latter is called a *persistent* server which is activated outside BOA (Basic Object Adaptor). The former is called a *non-persistent* server. Both types of servers must register themselves with the Orbix daemon and have records in the Implementation Repository, where the Orbix daemon will browse the information of the server before activating it. In the current implementation of A-ORB, the shared activation mode is chosen. All processes which act as Orbix servers are activated by the Orbix daemon (i.e. they are non-persistent servers).

3.3.2 Binding Operation

Although Orbix provides the Naming Service which can resolve a name within a given naming context and bind a name to an object reference, the easiest way to invoke a non-persistent server and get the object reference of the server in Orbix is to use the `_bind()` operation. The `_bind()` function is a static member function automatically generated by the IDL compiler for each interface [3]. While calling `_bind()`, the Orbix client can choose either specifying or not specifying the host. These two approaches are briefly described.

In the first approach, the host name for the server is specified by the client. To allow an Orbix daemon to manage the server processes, Orbix provides the Implementation Repository which maintains a mapping from a server's name to the file name of the executable code that implements this server. Therefore, the server code must be registered in the Implementation Repository before the server is invoked. Figure 3-7 shows how an Orbix server is activated automatically by the Orbix daemon.

Figure 3-7: Automatic launch of an orbix server process

The steps in Figure 3-7 are described:

- 1) A client makes its initial operation call to an object located in a server.
- 2) The Orbix daemon in the server host reads the detailed information about the server from the Implementation Repository, including the server launch command.
- 3) If the required server process is not activated yet, the Orbix daemon executes the server launch command.

The second approach to launch the server process does not need the client to specify the server host name. In this case, the client does not have to know the location of the server. In Orbix, the location transparency CORBA functionality is supported by the entity called the *locator*. The Orbix server locator allows users to associate a set of hosts with a server

name using the configuration file [15]. As shown in the system example presented in Figure 3-8, the locator in the client host knows how many server hosts associate to the specified server name by looking at the locator configuration file. In our environment, there is only one locator configuration file which resides in an NFS (Network File System) mounted directory and is accessible to each client host. The same situation applies to the Implementation Repository for all server hosts.

Figure 3-8: The Orbix server locator mechanism

The scenario of events that occur in the locator is described in Figure 3-8.

- 1) A client binds to an object in a server without specifying the host on which the

server runs. When the client binds to the object, the initial operation is implicitly made by Orbix.

- 2) Since the locator is implemented in the Orbix daemon, the daemon processes must exist on both the client and server host. The server locator determines the name of the host on which the server runs by reading its configuration files. If the server can be run on more than one host, the locator will choose one of them at random (not round-robin).
- 3) Orbix binds with the object at the host specified by the server locator.
- 4) The Orbix daemon at the server host reads the information from the Implementation Repository.
- 5) If the required server is not activated yet, the Orbix daemon executes the server launch command.

3.3.3 Initialization

The initialization of the A-client, A-agent, and A-server are described.

Client Initialization:

- The A-agent and the A-server have to know which client originates the request and how to communicate with it since they may have to send the handle or the reply to the A-client. Therefore, the A-client calls `_bind()` to get the objref of itself, stringifies the objref, and stores it locally.

- The A-client calls `_bind()` to get the objref of the A-agent. Ideally the A-client should only have to call `a_bind()` to get the server's handle without having to ask the default agent (Orbix daemon and Implementation Repository) for a handle to an A-agent because both agents should be integrated into one process. Because we cannot access the Orbix source code, the objref of the A-agent needs to be obtained before calling `a_bind()`.

Agent Initialization:

- The A-agent calls `_bind()` to implicitly invoke the A-server. There are four server processes in this A-ORB implementation. They are server A1, A2, B1, and B2. The A-agent stores both the objref and stringified objref (handle) in a global structure which is accessible by the forwarding thread and returning thread. The objref of the A-server will be used when the forwarding thread forwards the request to a selected A-server. The stringified objref will be used when the returning thread has to return an A-server's handle back to the A-client.
- The A-agent (main thread) initializes all resources (mutex, condition variable, forwarding queue, and returning queue) used by the forwarding thread and returning thread. Then the main thread creates a specified number (included in the A-agent's configuration file) of forwarding threads and returning threads.

Server Initialization:

- Since the A-server can always unstringify the stringified objref of the originating A-client from the client request, it is not necessary to have any initialization for the A-server.

All processes in the A-ORB are all implicitly activated at the initial stage.

3.3.4 The A-ORB Request Path

The different steps required in the completion of a client request are shown in Figure 3-9. The bold arrows represent the operations with simulated inter-node delays. In this case, the sender always sleeps a predetermined amount of time before sending a message. The dashed lines represent messages to or from the default agent. There is no simulated delays for this type of messages. The functionality of the default agent is provided by the Implementation Repository and the Orbix Daemon so it is shown individually in the figure. The Orbix Daemon must exist on each A-client, A-agent, and A-server host in our environment. However, it does not need to exist on a client host if the client process does not utilize the locator to look up a server since the locator functionality is provided by the Orbix Daemon. The A-agent can relay the request to the sought server of either class or return the handle of the server to the calling client. These request path steps are described:

- **Steps 1-2:** The client registers itself with the default agent in order to be accessible by the other processes. It also acquires its own objref, which will be attached in the following request in order to receive the server's handle from the A-agent or the reply from the server. It also gets the handle of the A-agent in order to send a `bind()` command.

- **Step 3-4:** The A-agent obtains the objref of all servers from the default agent and stores them into the local repository.
- **Step 5:** The client sends a-bind() command which includes its own objref in the request. In our experiments, this request is destined for a *class A* server.
- **Step 6:** The A-agent obtains a handle for a server of *class A* from the local repository. Let *A1* be the server which is chosen by the round-robin algorithm. This dynamic binding is required to achieve load balancing. The A-agent uses the obtained handle to forward the request to the *A1* server. This corresponds to the situation in which the A-ORB is operating in the forwarding mode.
- **Step 7:** The *A1* server unstringifies the client's objref in the request and sends the result to the originating client.
- **Step 8:** The client sends its request destined for a *class B* server.
- **Step 9:** The A-agent checks the workload conditions (the number of enqueued requests) and decides to return the server's handle to the client. It chose the *B2* server by the round-robin algorithm.
- **Step 10-11:** The client sends the request to the chosen server, *B2*, and embeds its own objref in the request. The B2 server sleeps prior to sending the reply to simulate the inter-node delay. The reception of the reply at the client side indicates the end of the client cycle.

Note that step 1-4 are executed only during system initialization and the sender sleeps to simulate the inter-node delay before sending in step 5-11.

Figure 3-9: The Request Sequence Path of this A-ORB Implementation

3.4 Design Issues and Problems Experienced

3.4.1 One-Way Operation v.s. Normal Operation

In the A-ORB, the A-agent can either forward the request to a server or return a server's handle to the client, so the client cannot always expect a message returned from the A-agent after it sends a `a_bind()` command to the A-agent. Two types of operations could be used to send a message from a client to a server in Orbix [3]:

- 1) **Normal operation:** The normal operation is a two-way and reliable operation. The caller is blocked until the operation is finished and returned by the server. An IDL example of the normal operation is: `"short a_bind()"`
- 2) **One-way operation:** The one-way operation is faster but less reliable in comparison to the normal operation because the caller does not expect any response from the receiver. The caller will be unblocked as soon as the message is sent out of the caller's domain. There is no guarantee that the message would reach the receiver and the caller may not know if a failure happens. A one-way operation cannot take any **out** or **inout** (the values of parameters will not be modified in this operation). It can only have a **void** return type. An IDL example of the one-way operation is: `"oneway void a_bind()"`.

3.4.2 Event Control in Orbix

In general Orbix applications, the client binds with an object of the server, invokes the object's operation, and then waits for the reply from the server. After finishing the

registration and initialization, the Orbix server has to call one of the following functions to be able to handle the incoming events (requests). These Orbix functions can retrieve an event from the Orbix internal queue and dispatch it to the requested operation which actually provides the implementation and service. These functions are described briefly:

- 1) **impl_is_ready()**: An Orbix server calls this function to indicate that it has completed the initialization and is ready to receive operation requests on its object. This function has to be called by each Orbix server process. It blocks the server until an event occurs, handles the event, and re-blocks the server to await another event. That means all incoming events are handled implicitly in this function until the server process is killed or timed-out. If zero is assigned to the time-out value, the server will return immediately. In this case, one of the following functions has to be called to handle events explicitly.
- 2) **processEvents()**: This function is similar to **impl_is_ready()** because it processes any number of events until it times out. However, a timeout of 0 blocks **processEvents()** until all pending events are processed whereas a timeout of 0 causes **impl_is_ready()** to return without processing any events.
- 3) **processNextEvent()**: This function blocks a server until an event occurs. It handles the event, and returns immediately. The function, **processEvents()**, has the same effect as calling **processNextEvent()** repeatedly until it times out.

In the A-ORB, we need to have more control on event handling since the A-agent could either forward a request to the server or return a server's handle back to the client depending

on the workload condition. All event types are listed in Section 3.2.2. Each A-ORB process (A-client, A-agent, or A-server) is an Orbix client process and an Orbix server process simultaneously. The event control steps in the A-ORB processes are described as follows:

- 1) The process calls `impl_is_ready()` with zero timeout in order to be unblocked immediately.
- 2) The process calls `processNextEvent()` to check each incoming event and dispatch the event to the requested operation.
- 3) The operation assigns a specific event type (e.g. `TYPE_ABIND`) to an event variable.
- 4) After the operation returns, the process checks the event type and decides the appropriate response. For example, the client will send a request to the server if the event type indicates that the server's handle is received. In another situation, it may send another `a-bind()` request to the A-agent or terminate the request cycle if the event type indicates that a reply has been received from the server.

By implementing the above steps, we can have a more flexible control on the events.

3.4.3 Multi-tasking vs. Multi-threading

To improve the performance and reduce the software bottleneck, agent cloning is considered in the A-ORB. In the Solaris platform, both multi-tasking and multi-threading are possible. The multi-threading is chosen to implement the application in the A-agent because of the following reasons:

- 1) The process context switching creates much more overhead than that in a multi-threaded process.
- 2) The code complexity of the A-ORB can be reduced since the global structure is accessible by all threads in a process and the IPC mechanism such as shared memory is not required.
- 3) The performance of the A-ORB can be improved because the characteristics of the A-ORB applications are good for multi-threading. The main thread, returning thread and forwarding thread are all I/O bound applications (wait before receiving or sleep before sending), which are more suitable for multi-threading in comparison to the CPU bound applications [31].

However, programming in the multi-threaded environment and keeping applications thread-safe are challenging, although both Solaris and Orbix support multi-threading.

3.4.4 Message Dispatching vs. Queue Polling

Both messages and shared variables can be used for communication among system components. Message passing is typically used for communication among different processes. In this A-ORB implementation, the main thread, returning thread, and forwarding thread are all in the same process so they can share all resources including the global variables such as the returning queue and the forwarding queue. The returning thread and forwarding thread periodically poll the returning queue and forwarding queue respectively. They sleep without consuming any CPU time except polling queues. This method of communication significantly reduces the complexity of the A-agent.

3.4.5 Persistent Servers vs. Non-Persistent Servers

Persistent servers are invoked manually, whereas non-persistent servers are invoked automatically. Different from [2], this A-ORB is implemented to use non-persistent server. The A-agent is invoked by the first client and all servers are invoked by the A-agent. The approach of non-persistent servers makes it possible for the automatic invocation and termination of the A-ORB processes. A control process (see Chapter 4) is designed to invoke, synchronize, and terminate all tasks.

3.4.6 Problems Related to Orbix or Solaris

One of the reasons for choosing Orbix to implement the A-ORB is because Orbix is a mature and commercialized ORB. Many experienced problems and solutions are documented in IONA's web site. Without this kind of documentation, the following problems, for example, would take a prohibitive amount of time to solve.

- **The usage of the locator:** The concept of the locator was introduced in Section 3.3.2. To use the locator mechanism, the path of the locator file (Orbix.hosts) needs to be specified by adding an entry of `IT_CONFIG_PATH` in `Orbix.cfg`.
- **Orbix daemon problem:** Occasionally, the server dies with the message: "Purported size was: 1347373383 Max allowed is 8388608". The solution for this problem is to force the Orbix daemon to always use IIOP protocol by defining "`TT_DAEMON_PROTOCOL=IIOP`" in `Orbix.cfg`.

- **Lack of file descriptors:** The default number of file descriptors for each process in Solaris is only 64. The way to create I/O devices in Unix is similar to the way to open a file. A connection between the client and server also consumes one file descriptor. The A-agent has to build at least one connection to each client and server. When the number of clients increases, the A-agent could get some exceptions due to the lack of file descriptors. Therefore the number of file descriptors is increased to be 256 in the A-ORB environment.
- **usleep() problem in the multi-thread environment:** To simulate the inter-node delay, the sender process sleeps for a specified interval of time before sending the message. Two system calls, `sleep()` and `usleep()`, with different resolutions are available. In the frequently asked questions listed on the web site of IONA (the producer of Orbix) [23], it is described that calling `sleep()/usleep()` in the main thread of an Orbix-MT application may cause the process to hang. As mentioned in [31], `sleep()` is not a thread safe function. To overcome this problem, the timeout mechanism in the condition variable is used to provide the same functionality as `usleep()` without causing any problem in the multi-threaded environment. Also, using the condition variable and the mutex lock [31] in the A-ORB can achieve synchronizing the access to the returning queue and the forwarding queue among all active threads.

Chapter 4 *Instrumentation and Performance Measurement*

4.1 The Workload and Configuration files

4.1.1 Workload

The selection of the workload is based on specifying the characteristics of the system as well as those of the applications executing on it that have a significant effect on the system performance. The workload factors are described as follow:

- 1) **Request Service Times S_a and S_b** in msec: S_a is the request server time for the server class A. S_b is the request service time for the server class B. This is the total CPU demand that each client request needs at each class of servers. It only represents the actual processing time and does not include other overheads such as marshalling/unmarshalling or binding. It is desirable to know whether loading the servers of both classes has an effect on the performance of A-ORB.
- 2) **Message Size (L)** in bytes: This is the length of message the client sends per request to the server or attaches in the a-bind() command to the A-agent. The message size is L. The message is a byte stream, consisting of the same character of size L, that is included in the request.

- 3) **Inter-Node Communication Delay (Dca, Das, Dcs)** in msec: Dca is the inter-node delay between the client and agent. Das is the inter-node delay between the agent and server. Dcs is the inter-node delay between the client and server. In the distributed environment, the client nodes may be several nodes away from the agent and server nodes. Since a store and forward mechanism is normally used at intermediate nodes in the path between the client, agent, and server, the delay is expected to increase with the number of intermediate nodes. It is important to investigate how much impact on the system performance of A-ORB the delay in the message delivery has.
- 4) **Number of Clients (N)**: This is the total number of active clients during the life of each experiment in the measurement. It is important to investigate how the system performance is affected by the increased number of clients.
- 5) **The Threshold of Queue Length (QL)**: This is the load threshold level set for the system. If the number of clients waiting in the forwarding queue is equal to or greater than this threshold, the incoming request will be put into the returning queue and the handle of the desired server will be returned to the originating client. It is important to know how the QL value affects the result of the performance measurement.
- 6) **Level of Thread Cloning**: Since both Solaris and Orbix support multi-threading, utilizing the multi-threading mechanism is expected to improve the system performance, especially for the I/O bound applications. Both the returning thread and forwarding thread can be cloned whereas only one main

thread is required. Determining the impact the thread-cloning has on the performance of A-ORB warrants investigation.

By using different combinations of S_a , S_b , L , D_{ca} , D_{as} , and D_{cs} , it is possible to know the impact on both I/O (communication) bound and CPU (computation) bound systems. In a CPU bound system, the server is heavily loaded whereas the message sizes and inter-node delays are small. In an I/O bound system, the message size is large and/or the inter-node delay is long, but the service times for the client requests are small.

Table 1: Levels for the Workload Factors

Factors	Levels
<i>N</i>	<i>1, 2, 4, 8, 16, 24</i>
<i>S_a/S_b (msec)</i>	<i>10/15, 50/75, 250/375</i>
<i>L (bytes)</i>	<i>4800, 9600, 19200</i>
<i>D_{ca}/D_{as}/D_{cs}</i>	<i>50/50/50, 125/125/125, 250/250/250, 50/250/250, 250/50/250, 250/250/50</i>
<i>QL (client)</i>	<i>0, 1, 3, 24</i>
<i>Level of Thread Cloning</i>	<i>1, 2, 3</i>

The different levels of the factors used in the experiments are listed in Table 1. The number and choice of the levels for each factor reflect a compromise between representativeness and an acceptable run time for the experiments.

4.1.2 Configuration Files

Since so many combinations of factors listed in Table 1 are possible, it is more convenient to have each set of workload factors stored in a different configuration file. In

this A-ORB implementation, two kinds of configuration files are used.

The client's configuration file

This configuration file defines the values of most workload factors and is read by the client during the initialization period. The content of the client's configuration file is described next. A detailed description of the first three attributes was presented in Section 4.1.1.

- 1) **Message Size**
- 2) **Request Service Times**
- 3) **Inter-Node Delay**
- 4) **Number of Cycles:** This value controls the total number of request cycles for each client. Each cycle contains two requests. The first request is for the server class A and the second one is for the server class B. The number of cycles should be large enough such that the system remains in steady state even at high load conditions. The number of cycles is 200 in our experiments.
- 5) **Title of the Experiment:** To make the automation of collecting the experiment result possible, a title of the experiment is required. When the measurement is to be conducted, a batch file is executed and all results (see Section 4.2.1) of the measurement are stored in log files, the location (path) of which is based on the title of the experiment. For example, the results of an experiment will be stored in the directory of msg4800D0_QL01 if the title of this experiment is msg4800D0 and threshold value is 1 (QL=1).

When the client prepares the parameters for the `a-bind()` command, it generates a byte stream according to the message size specified in this configuration file, attaches the values of request service times and inter-node delays in the request, and then sleeps a number of milliseconds (Dca) before sending the request.

The agent's configuration file

This configuration file defines threshold levels that affect the decision made by the A-agent.(returning or forwarding). This file is read by the A-agent at its initial phase. The entries in this file are described:

- 1) **The queue length (QL):** The detailed description of this entry can be seen in Section 4.1.1.
- 2) **The Number of Returning Threads:** This entry controls how many returning threads would be created in the A-agent.
- 3) **The Number of Forwarding Threads:** This entry decides the number of forwarding threads in the A-agent.

4.2 Computation of Performance Matrices

4.2.1 Definition of Performance Matrices

To investigate the impact of different workload factors on the system performance, the following performance matrices have to be collected and compared.

- 1) **Mean Client Response Time (R) in seconds:** The client response time is the elapsed time from the moment the client initiates its request cycle to the time

when the final reply arrives from the second server. As mentioned in Section 4.2.1, each request cycle consists of two requests. The first request is for the class A server and the second request is for class B server. The mean client response time of the system is the mean of the average response times of all active clients. The mean client response time is a measure of the latency of the system.

2) ***Overall Mean System Throughput (X) in client request cycles per second:***

This is the average total system throughput which is obtained by the summation of all client's throughputs. The throughput is the way to measure the capacity of the system. The throughput and response time are related to the Little's Law, which is described in Section 4.4.

3) ***Process and CPU Utilizations of the Servers (US):*** These utilizations are measured to gain more insight into system performance. The process utilization (U_s^P) and CPU utilization (U_s^{CPU}) are useful in identifying software and hardware bottlenecks. The general definitions of these two utilizations are listed as follow:

Process Utilization:

= Elapsed time of processing request / Total elapsed time including idle time.

CPU Utilization:

= CPU time of processing request / Total elapsed time including idle time.

Figure 4-1: Activities in the Server Process

As shown in Figure 4-1, there are six main activities in the server process of the A-ORB.

A: get from stream and unmarshal

B: request processing

C: unstringify client's objref

D: sleep (simulated inter-node delay)

E: marshall, put on stream, and send

F: Listen for the incoming requests

Note that the server process is idle and waiting for an incoming request in activity F so it does not consume any CPU time. The utilizations of the server process are computed by using the following equations:

$$U_S^P = \frac{A_{WALL} + B_{WALL} + C_{WALL} + D_{WALL} + E_{WALL}}{A_{WALL} + B_{WALL} + C_{WALL} + D_{WALL} + E_{WALL} + F_{WALL}}$$

$$U_S^{CPU} = \frac{A_{CPU} + B_{CPU} + C_{CPU} + D_{CPU} + E_{CPU}}{A_{WALL} + B_{WALL} + C_{WALL} + D_{WALL} + E_{WALL} + F_{WALL}}$$

U_S^{CPU} : CPU utilization of server S

A_{CPU} : CPU time consumed in activity A

B_{CPU} : CPU time consumed in activity B

C_{CPU} : CPU time consumed in activity C

D_{CPU} : CPU time consumed in activity D (CPU time consumed in executing the system call)

E_{CPU} : CPU time consumed in activity E

U_S^P : Process utilization of server S

A_{WALL} : Total time consumed in activity A

B_{WALL} : Total time consumed in activity B

C_{WALL} : Total time consumed in activity C

D_{WALL} : Total time consumed in activity D

E_{WALL} : Total time consumed in activity E

F_{WALL} : Total time consumed in activity F

- 4) **Thread and CPU Utilization of the Agent (UA):** These utilizations are measured in order to get insight into the performance of the A-agent. There are

three kinds of threads (main thread, returning thread, and forwarding thread), so each of them has its own thread utilization. If there is more than one returning thread or forwarding thread, the thread utilization is the mean value of utilizations from a given number of threads. The way of calculating thread/CPU utilization of the agent is similar to that of calculating process/CPU utilization of the server.

4.2.2 Measurement of the Elapsed and CPU Time

Figure 4-2: Eight filter points

Since the ORB hides the network characteristics from the clients and servers, it seems to be difficult to measure the time taken in marshalling and unmarshalling when the source

code is not accessible. Fortunately, Orbix provides **filters** which allow a programmer to specify additional code to be executed before or after the normal code of an operation or attribute. There are up to eight points where some code can be added. They are shown in Figure 4-2. The segment A and E, defined in Section 4.2.1, are for unmarshalling and marshalling respectively. We call `gethrtime()` at the following points:

- Between points 3 (In request pre marshal) and 4 (In request post marshal) to know how much time the server spends in unmarshalling an incoming request.
- Between points 1 (Out request pre marshal) and 2 (Out request post marshal) to know the time spent in marshalling an outgoing request¹ (reply for the A-server).

The following system calls are used to measure the elapsed time or CPU time:

- 1) **gettimeofday() [30]:** This function gets the system's notion of the current time which is expressed in elapsed seconds and microseconds since 00:00 Universal Coordinated Time, January 1, 1970. This function is used to measure the elapsed time in the A-client and implement the timer to control the idle time of each thread in the A-agent.
- 2) **gethrtime() [30]:** This function returns the current high-resolution real time. Time is expressed as nanoseconds since some arbitrary time in the past. This hi-res timer is ideally suited to performance measurement tasks, where cheap, accurate interval timing is required. In this A-ORB implementation, this function is used to get the total time consumed in each segment (see Section

1. Because the message passing in this A-ORB is uni-directional, the reply from the A-server is actually a "request" according to the definitions for the filters.

4.2.1) either in the A-agent or in the A-server.

- 3) **clock()**: This function returns the amount of CPU time (in microseconds) used since the first call to **clock()** in the calling process. According to the data from the performance measurement, this function is not able to get a high resolution CPU time. The resolution provided by this function is observed to be 10 milliseconds.
- 4) **ioctl()**: This Solaris system call can be used with the **PIOCUSAGE** flag to obtain the CPU time consumed by the calling process. The time reported is the sum of the user and system times of the calling process. An example is shown in Figure 4-3 to demonstrate how to use **ioctl()** to get the CPU time in the Solaris system:

```
#include <unistd.h> // for ioctl()
#include <sys/procfs.h> // for prusage_t
.....
double getCpuTime()
{
    static char proc[20];
    static int fd;
    prusage_t prusage;
    .....
    // if the file is not open yet...
    sprintf(proc, "/proc/%d", getpid());
    fd = open(proc, O_RDONLY);
    .....
    ioctl(fd, PIOCUSAGE, &prusage);
    .....
}
```

Figure 4-3: The example of getting CPU time by using `ioctl()`

4.3 Data Collection

4.3.1 The Control Process

Performance measurement is a time-consuming activity since the same experiment has to be repeated many times to get reliable results. To make the measurement smooth and automate the experiment, a Control Process is designed to control the activities of the whole system. Since the A-clients, A-agent, and A-servers are all non-persistent servers in Orbix, they can be activated by other processes. The Control Process controls the start and the end of all experiments.

Figure 4-4: The Control Process

The relationship between the Control Process and other processes is shown in Figure 4-4. This figure is similar to Figure 3-4 but the Control Process is added. The dashed lines represent the messages between the Control Process and other processes. These messages are for the control (activation and termination) and data collection. The responsibility of the Control Process includes:

- 1) Creating the data log directory.
- 2) Creating a number of A-clients according to the configuration file.
- 3) Initiating the performance measurement in an experiment by sending a startup command.
- 4) Collecting the measurement results (see Section 4.2.1) from the A-clients, A-agent, and A-servers.
- 5) Using those results to get the mean values for the whole system and store them into its data log file.
- 6) Terminating the experiment.

4.3.2 The Measurement Steps

The different steps followed during each experiment are described:

- 1) The Control Process creates a directory to contain data log files. The directory name will be sent to the A-clients and A-servers later. The log files generated by these processes will be stored in this directory.
- 2) The Control Process reads configuration files. Based on the number of A-

clients specified in the file, it binds and activates all clients. The first A-client will invoke the A-agent which will activate A-servers.

- 3) After all processes have been created, the Control Process sends commands to all A-clients almost at the same time. This action initiates the experiment so all A-clients begin to send requests to the A-agent.
- 4) Whenever the A-client finishes one request cycle, it calculates the response time and throughput, stores these results in the local memory, and then starts immediately another identical cycle without any think time. In the meantime, the A-agent continuously deals with the incoming requests, collects the thread/CPU utilization, and store these records locally. The A-server also measures the process/CPU utilization and puts them into the local memory.
- 5) The number of total request cycles is specified in the A-client's configuration file. After all A-clients send all requests and receive all replies from A-servers, they calculate the mean response time and throughput individually and write these results into their data log files. Finally they send these results to the Control Process to indicate the completion.
- 6) After receiving results from all A-clients, the Control Process sends a command to the A-agent. The A-agent then starts to calculate the mean thread utilization and CPU utilization. After the calculation, it sends these results to the Control Process.
- 7) The Control Process starts to send commands to all A-servers after it gets the

results from the A-agent. The A-servers realize no more requests are coming, so they start to calculate the process utilization and CPU utilization individually. They send those data to the Control Process.

- 8) At this point, the Control Process has collected all data from all other processes. It can start to calculate the *Mean Client Response Time, Overall Mean System Throughput, Process and CPU Utilization of the Servers, and Thread and CPU Utilization of the Agent*. After writing the final result into the data log file, the Control Process sends commands to terminate all other processes and exits.

4.4 Verifying Data

Although the isolated measurement network could generate more stable results, the experimental results still need to be verified. Each experiment consists in repeating client cycles a large number of times with a specific combination of workload factors. The experiment with the given set of workload parameters is also repeated and statistics are gathered to compute confidence intervals for performance measures. In most cases, we achieved an interval that is less than ± 5 % of the mean at a confidence level of 95% for the performance measure of interest. The **t-Distribution** is used to calculate the confidence interval which is explained in [17].

The final data from the performance measurement should also satisfy the *Little's law* and *Utilization law*. Little's law is one of the most commonly used theorems in queueing theory. It allows us to relate the number of active clients with the mean response time as

follows:

$$N = X * R$$

where,

N: The total number of active clients

X: The overall mean system throughput

R: Mean client response time

Utilization law is another method to verify the measured data. This law applies to this experiment by using the following equation:

$$U = X * D$$

where,

U: The CPU utilization in some server process

X: Throughput of that server in number of requests per second

D: Mean CPU time of that server in service time (second) per request.

A number of validating runs were made. In all of these runs, the data satisfied both Utilization Law and Little's Law.

4.5 Experimental Environment

Experiments were performed by executing the A-ORB system on a network of seven Sun Sparc-2 workstations running under the Solaris 2.6 operating system (see Figure 3-9). These nodes are connected by a 10 Mbps Ethernet LAN which is isolated from other nodes in the same lab. In other words, the experimental result would show high stability since little

unexpected data traffic can go into the measurement network. This condition is very useful and important for the performance measurement. Without this condition, all experiments have to be proceeded during the night time (presumably, there is not much extra traffic or process running on the shared network at that period) without any guarantee that the measured data could remain similar and stable. Any extra task can potentially affect the result.

Among these seven workstations, four are used for running four server processes (two instances of Server A and two instances of Server B). The A-agent runs on a separate workstation, whereas the other two workstations are used for running the client processes.

The results are directly useful for embedded applications that run the agent and the servers on separate nodes. Running the agent on a separate node is important when the workload is such that the agent is highly utilized. In systems in which a node is shared with other applications the performance of the CORBA system may depend on the device demands made by the other applications.

Chapter 5 Results of Experiments

Several sets of experiments were run to investigate the effect of the workload factors which are described in Section 4.1.1. A factor-at-a-time approach is chosen for each experiment. That means one of the factors was varied while the others were held constant. The results of each experiment are presented as a set of graphs that capture the relationship between one of the interested performance measures (plotted along the Y axis) and the number of clients (plotted along the X axis) with all the other fixed parameters stated in the figure caption. The discussion for the results is also presented in relative sections.

5.1 Effect of Server Demands

Table 2: Parameters of Experiments on Server Demands

Factors	Levels
<i>N</i>	1, 2, 4, 8, 16, 24
<i>Sa/Sb (msec)</i>	10/15, 50/75, 250/375
<i>L (bytes)</i>	150
<i>Dca/Das/Dcs (msec)</i>	0/0/0
<i>QL (client)</i>	0, 1, 3, 24
Level of Thread Cloning	1

The client requests may have various computation demands at the server nodes. This set of experiments is designed to investigate the relationship between server demands and the performance of the A-ORB. Table 2 presents the various parameters used, with the

variable parameters in italic fonts. The detailed discussion of the results of the experiments is presented.

Low Load Conditions:

As explained in Section 4.1.2, a $QL=0$ corresponds to a purely handle driven architecture since a handle is returned for each request to the originating client. Because $N \leq 24$, $QL=24$ implies that the client request is always forwarded to the server. As seen in Table 3, when $N=1$, all cases exhibit an increase in the latency (Mean Client Response Time defined in Section 4.2.1) with the increase in the server demands. However, the system with $QL=0$ has the longest response time since it requires four messages for interacting with any one server, whereas only three messages are required for a forwarding operation. Returning a handle also results in an extra inter-node delay.

Table 3: Effect of S_a/S_b on R at $N=1$ ($D=0$ ms, $L=150$ bytes)

S_a, S_b (msec)	R (sec) $QL=0$	R (sec) $QL=1$	R (sec) $QL=3$	R (sec) $QL=24$
<i>10, 15</i>	<i>0.2447</i>	<i>0.1552</i>	<i>0.1552</i>	<i>0.1544</i>
<i>50, 75</i>	<i>0.3447</i>	<i>0.2539</i>	<i>0.2517</i>	<i>0.2518</i>
<i>250, 375</i>	<i>0.8413</i>	<i>0.7513</i>	<i>0.7510</i>	<i>0.7509</i>

Medium to High Load Conditions:

System performances achieved at higher system loads for different S_a/S_b combinations are presented separately.

Server A: 10 ms, Server B: 15 ms

The mean response time and the corresponding overall mean system throughput are

measured for different number of clients. As shown in Figure 5-1, when N goes beyond 4, the throughput achieved with $QL=24$ exceeds the throughput achieved with $QL=1$ and $QL=3$. The system with $QL=24$ performs the best because pure forwarding takes the least time on message passing while the overhead of the whole system is relatively low. The server execution times are small so the expected waiting time in the forwarding queue is small in comparison to the additional delay incurred in returning a handle to the client.

In general, the results for the throughput between different QL values correspond with the results for the response time. For example, the system with $QL=0$ demonstrates the worst throughput (see Figure 5-1) so the response time for $QL=0$ is the longest as expected (see Figure 5-2). Note that throughput is related to response time by Little's law (see Section 4.4). Thus additional figures related to response time will not be presented later.

Figure 5-3, Figure 5-4, and Figure 5-5 show the Cpu Utilization and Thread Utilizations of the A-agent, which includes the main thread, the returning thread, and the forwarding thread (see definitions in Section 4.2.1) for $QL=1$, 3, and 24 respectively. Throughputs achieved with $QL=1$, $QL=3$, and $QL=24$ tend to flatten off at $N \geq 4$ (see Figure 5-1) because of the formation of performance bottlenecks. The thread utilization of the returning thread or the forwarding thread is close to 1 at higher values of N . Since a synchronous mechanism is used for the inter-process communication, a forwarding or returning thread waits until the transmitted message is received. With an increase in N , this waiting time increases so the returning or forwarding thread becomes saturated. Lower utilizations were observed at the servers. This indicates that the A-agent is becoming a software bottleneck.

Figure 5-1: Throughput of the A-ORB with $S_a/S_b=10/15$ ms, $D=0$ ms, $L=150$ bytes.

Figure 5-2: Response Time of the A-ORB with $S_a/S_b=10/15$ ms, $D=0$ ms, $L=150$ bytes.

Figure 5-3: Thread and CPU Utilizations of the A-ORB Agent with $S_a/S_b=10/15$ ms, $D=0$ ms, $L=150$ bytes, $QL=1$.

T_M = Thread Util. of Main Thread, T_R = Thread Util. of Returning Thread, T_F = Thread Util. of Forwarding Thread, CPU_G = Cpu Util. of the Agent

Figure 5-4: Thread and CPU Utilizations of the A-ORB Agent with $S_a/S_b=10/15$ ms, $D=0$ ms, $L=150$ bytes, $QL=3$.

Figure 5-5: Thread and CPU Utilizations of the A-ORB Agent with $S_a/S_b=10/15$ ms, $D=0$ ms, $L=150$ bytes, $QL=24$.

T_M = Thread Util. of Main Thread, T_R = Thread Util. of Returning Thread, T_F = Thread Util. of Forwarding Thread, CPU_G = Cpu Util. of the Agent

Server A: 50 ms, Server B: 75 ms

As shown in Figure 5-6, the A-ORB with $QL=1$ or $QL=3$ outperforms the purely handle-driven ($QL=0$) and forwarding ($QL=24$) ORBs. Inside the A-agent, the forwarding thread forwards the client's requests and waits until the chosen server finishes its current cycle and is ready to receive the message. Therefore, all requests have to wait inside the forwarding queue before being sent, although they may not be forwarded to the same server. When the server execution time increases, the queueing delay in this queue becomes more serious and causes an inferior throughput. The purely handle driven ORB performs the worst for the same reason mentioned earlier. The A-ORB, which dynamically changes its behavior based on the workload conditions, achieves the best performance.

Figure 5-6: Throughput of the A-ORB with $S_a/S_b=50/75$ ms, $D=0$ ms, $L=150$ bytes.

By comparing Figure 5-7 and Figure 5-8, we can see the reason why $QL=3$ (and $QL=1$) performs better than $QL=24$ (and $QL=0$). The forwarding threads at $QL=3$ and at $QL=24$ have almost the same utilization either at low load or high load conditions. Both of them reach saturation around $N=8$ when queueing delays experienced by requests becomes long. However, the returning thread with $QL=3$ has a lower utilization in comparison to the forwarding thread and requests experience a lower delay in the returning queue. As a result the system with $QL=24$ that always uses the forwarding operation demonstrates an inferior performance. As shown in Figure 5-9, the returning thread of $QL=0$ saturates when $N \geq 4$ and makes the A-agent reach the software bottleneck (see Section 3.1.3) since it does not have the ability of forwarding. The A-ORB, which effectively switches between using the forwarding and returning threads, demonstrates its ability to improve system performance.

Figure 5-7: Thread and CPU Utilizations of the A-ORB Agent with $S_a/S_b=50/75$ ms, $D=0$ ms, $L=150$ bytes, $QL=3$.

T_M = Thread Util. of Main Thread, T_R = Thread Util. of Returning Thread, T_F = Thread Util. of Forwarding Thread, CPU_G = Cpu Util. of the Agent

Figure 5-8: Thread and CPU Utilizations of the A-ORB Agent with $S_a/S_b=50/75$ ms, $D=0$ ms, $L=150$ bytes, $QL=24$.

Figure 5-9: Thread and CPU Utilizations of the A-ORB Agent with $S_a/S_b=50/75$ ms, $D=0$ ms, $L=150$ bytes, $QL=0$.

T_M = Thread Util. of Main Thread, T_R = Thread Util. of Returning Thread, T_F = Thread Util. of Forwarding Thread, CPU_G = Cpu Util. of the Agent

Server A: 250 ms, Server B: 375 ms

As shown in Figure 5-10, almost the same throughput is achieved with the A-ORB regardless of the QL values. The bottleneck in this situation is the server. It is desirable to know why the policy adopted at the A-agent for coordination has little effect on the performance of this case and why this set of experiments generates such a different result from the previous two experiments ($S_a/S_b=10/15$ and $50/75$). For the two lower sets of server demands, either the forwarding thread or the returning thread is the performance bottleneck. We know this by observing the thread utilization in the A-agent which is close to 1 at higher values of N (see Figure 5-4 and Figure 5-7). The utilizations of the servers are observed to be much smaller in comparison. The performance bottleneck shifts to the

servers for $S_a/S_b=250/375$ since the server utilization is close to 1 as shown in Figure 5-11 and Figure 5-12. However, the thread utilization of the purely forwarding ORB ($QL=24$) is close to 1 at high loads (see Figure 5-14) whereas the thread utilization of the purely handle-driven ORB is less than 0.6 (see Figure 5-13). This is because the forwarding thread has to wait until the server completes its current cycle. Therefore, the forwarding thread also saturates when the server reaches its bottleneck. When the server is the performance bottleneck, the QL values have very little impact on the overall system performance. This bottleneck shifting is also the reason why this set of experiments behaves differently from the others.

Figure 5-10: Throughput of the A-ORB with $S_a/S_b=250/375$ ms, $D=0$ ms, $L=150$ bytes.

Figure 5-11: Process and CPU Utilizations of the A-ORB servers with $S_a/S_b=250/375$ ms, $D=0$ ms, $L=150$ bytes, $QL=0$.

Figure 5-12: Process and CPU Utilizations of the A-ORB servers with $S_a/S_b=250/375$ ms, $D=0$ ms, $L=150$ bytes, $QL=24$.

Figure 5-13: Thread and CPU Utilizations of the A-ORB Agent with $S_a/S_b=250/375$ ms, $D=0$ ms, $L=150$ bytes, $QL=0$.

Figure 5-14: Thread and CPU Utilizations of the A-ORB Agent with $S_a/S_b=250/375$ ms, $D=0$ ms, $L=150$ bytes, $QL=24$.

However, we can still find a slight difference by observing Figure 5-10 and Figure 5-12. As seen in Table 4, when $N=24$, $QL=24$ performs the worst among all QL values because the forwarding thread saturates in conjunction with the server (see Figure 5-14 and Figure 5-12). Since a synchronous mechanism is used for inter-process communication in this A-ORB, a forwarding or returning thread waits until the transmitted message is received. In the pure forwarding case ($QL=24$), all client requests are queued in the forwarding queue before getting served, so the long queueing delay makes its performance slightly worse than the others at high load conditions.

Table 4: Effect of S_a/S_b on X at $N=24$ ($D=0$ ms, $L=150$ bytes)

S_a, S_b (msec)	X (Client Requests/sec $QL=0$)	X (Client Requests/sec $QL=1$)	X (Client Requests/sec $QL=3$)	X (Client Requests/sec $QL=24$)
<i>10, 15</i>	<i>7.1848</i>	<i>12.5027</i>	<i>12.7693</i>	<i>13.6477</i>
<i>50, 75</i>	<i>7.1592</i>	<i>10.9544</i>	<i>11.0672</i>	<i>9.9976</i>
<i>250, 375</i>	<i>4.3595</i>	<i>4.4096</i>	<i>4.4424</i>	<i>4.1300</i>

5.2 Effect of Message Size

The size of the messages corresponding to the client request and the results returned by the server are expected to vary with the type of the request and the nature of the service provided. This set of experiments is designed to investigate the relative performance of the A-ORB with different QL values when the message size L increases. As stated in Section 4.1.1, L is the number of bytes in the message corresponding to both the client request and the result returned. Table 5 presents the various parameters used in this set of experiments, with the variable parameters in italic font.

Table 5: Parameters of Experiments on Message Sizes

Factors	Levels
<i>N</i>	1, 2, 4, 8, 16, 24
Sa/Sb (msec)	10/15
<i>L</i> (bytes)	4800, 9600, 19200
Dca/Das/Dcs (msec)	0/0/0
<i>QL</i> (client)	0, 1, 3, 24
Level of Thread-Cloning	1

Low Load Conditions:**Table 6: Effect of *L* on *R* at *N*=1 (*D*=0 ms, Sa/Sb=10/15)**

<i>L</i> (bytes)	<i>R</i> (sec) QL=0	<i>R</i> (sec) QL=1	<i>R</i> (sec) QL=3	<i>R</i> (sec) QL=24
4800	0.2794	0.1914	0.1913	0.1916
9600	0.3181	0.2686	0.2673	0.2693
19200	0.3969	0.3493	0.3471	0.3608

As shown in Table 6, the latency at low load conditions (*N*=1) increases as the message size increases. Since there is no queueing delay, the difference in latency obtained at *N*=1 is determined primarily by the number of messages exchanged per client cycle. The pure handle driven approach (QL=0) requires four messages for interacting with servers, whereas the forwarding approach only needs three messages. Therefore, the system with QL=0 performs the worst in this situation. However, the performance gap between QL=0 and the other QL values is reduced when the message size increases as shown in Table 6.

Because the fourth message sent from the agent to the client includes only the server's handle, the additional delay with QL=0 has lesser impact at higher values of L. As L increases, the marshalling/unmarshalling times for the messages start dominating system performance and the A-ORBs with four different QL values perform comparably.

Medium and High Load Conditions:

Table 7: Effect of L on X at N=24 (D=0 ms, Sa/Sb=10/15)

L (bytes)	X (Client Requests/sec) QL=0	X (Client Requests/sec) QL=1	X (Client Requests/sec) QL=3	X (Client Requests/sec) QL=24
4800	6.5422	10.3493	10.6191	10.3581
9600	5.5721	8.2879	8.6566	8.5044
19200	4.5579	6.3599	6.3600	5.6342

As shown in Table 7, Figure 5-15, Figure 5-16 and Figure 5-17, the pure handle-driven (QL=0) approach generates the worst result due to the reasons mentioned earlier in this section. The other three QL values produce comparable performances for L=4800 and L=9600. However, the performance of the purely forwarding ORB (QL=24) is inferior to those of the adaptive ORB (QL=1 and QL=3) when L=19200. The reason is: when the message size becomes large, the transmission delay and marshalling/unmarshalling time also become more significant and thus cause a much longer queueing delay in the forwarding queue.

Figure 5-15: Throughput of the A-ORB with $L=4800$ bytes, $S_a/S_b=10/15$ ms, $D=0$ ms.

Figure 5-16: Throughput of the A-ORB with $L=9600$ bytes, $S_a/S_b=10/15$ ms, $D=0$ ms.

Figure 5-17: Throughput of the A-ORB with $L=19200$ bytes, $S_a/S_b=10/15$ ms, $D=0$ ms.

Both the pure forwarding and handle-driven approaches suffer from significant queueing delay at the high load conditions and large message sizes. The A-ORB (QL=1 or QL=3) demonstrates a superior performance in all cases since it can dynamically switch its behavior from forwarding to handle-driven. In the cases shown in Figure 5-18, Figure 5-19 and Figure 5-20, the forwarding thread saturates with the small number of clients ($N=4$), but the returning thread demonstrates a lower utilization and using a returning thread to send the handle back to the client seems to improve the overall performance in this situation.

Figure 5-18: Thread and CPU Utilizations of the A-ORB Agent with $L=4800$ bytes, $S_a/S_b=10/15$ ms, $D=0$ ms, $Q_L=3$.

Figure 5-19: Thread and CPU Utilizations of the A-ORB Agent with $L=9600$ bytes, $S_a/S_b=10/15$ ms, $D=0$ ms, $Q_L=3$.

Figure 5-20: Thread and CPU Utilizations of the A-ORB Agent with $L=19200$ bytes, $S_a/S_b=10/15$ ms, $D=0$ ms, $QL=3$.

5.3 Effect of Inter-Node Delay

The nodes for the A-clients, A-agent and A-servers could be located several nodes away from each other. Each intermediate node performs store and forwarding for a message and can introduce delays in the client cycle. The performance of the A-ORB with different QL values is to be investigated when the inter-node delay increases. The distance between nodes is measured by the inter-node delay. It is also desirable to find out the impact of the inter-node delay on system performance. Therefore the performance of both equidistant nodes and non-equidistant nodes are analyzed.

- **Equidistant Node:** The inter-node delay is the same between any two nodes (client, agent, and server) in the system.

- **Non-Equidistant Node:** The inter-node delay between any two nodes (client, agent, and server) can be different from each other.

5.3.1 Equidistant Nodes

Table 8: Parameters of Experiments on Equidistant Nodes

Factors	Levels
<i>N</i>	<i>1, 2, 4, 8, 16, 24</i>
<i>Sa/Sb (msec)</i>	<i>10/15</i>
<i>L (bytes)</i>	<i>150</i>
<i>Dca/Das/Dcs (msec)</i>	<i>50/50/50, 125/125/125, 250/250/250</i>
<i>QL (client)</i>	<i>0, 1, 3, 24</i>
Level of Thread Cloning	1

The parameters used for the experiment with equidistant nodes are shown in Table 8.

Low Load Conditions:

The latencies achieved with different QL values at N=1 are presented in Table 9. As captured in Table 9, a pure handle driven approach (QL=0) exhibits the longest latency and the worst performance at low load conditions whereas the purely forwarding approach (QL=24) and the adaptive ORB (QL=1 and QL=3) have comparable performance. Since the pure handle driven operation needs to pass one more message than the others, this causes more delay and latency in the case of QL=0. For all cases, the latency increases as the inter-node delay (or the distance between nodes) increases.

Table 9: Effect of Dca/Das/Dcs on R at N=1 (L=150 bytes, Sa/Sb=10/15)

Dca/Das/Dcs (msec)	R (sec) QL=0	R (sec) QL=1	R (sec) QL=3	R (sec) QL=24
<i>50/50/50</i>	<i>0.7099</i>	<i>0.4976</i>	<i>0.4969</i>	<i>0.4903</i>
<i>125/125/125</i>	<i>1.2958</i>	<i>0.9556</i>	<i>0.9552</i>	<i>0.9558</i>
<i>250/250/250</i>	<i>2.3043</i>	<i>1.6987</i>	<i>1.6991</i>	<i>1.6981</i>

Medium and High Load Conditions:

The importance of the adaptive ORB, which dynamically changes the mode of operation between a handle-driven and a forwarding mode, is captured in Table 10. As shown in Figure 5-21, Figure 5-22 and Figure 5-23, the adaptive ORB (QL=1 or QL=3) shows a substantially better performance in comparison to the pure handle-driven (QL=0) and the pure forwarding (QL=24) ORB at medium and high load conditions ($N \geq 4$). The throughput of QL=1 or QL=3 is almost twice of that of QL=0 or QL=24 (see Table 10).

Table 10: Effect of Dca/Das/Dcs on X at N=24 (L=150 bytes, Sa/Sb=10/15)

Dca/Das/ Dcs (msec)	X (Client Requests/sec) QL=0	X (Client Requests/sec) QL=1	X (Client Requests/sec) QL=3	X (Client Requests/sec) QL=24
<i>50/50/50</i>	<i>4.2095</i>	<i>8.6317</i>	<i>8.632</i>	<i>5.6030</i>
<i>125/125/125</i>	<i>2.7249</i>	<i>5.2605</i>	<i>5.2366</i>	<i>3.0352</i>
<i>250/250/250</i>	<i>1.6168</i>	<i>3.1119</i>	<i>3.0947</i>	<i>1.7179</i>

In general, the system performance of a handle returning and a forwarding operation depends on the inter-node delay, message size, and the number of requests (queue length)

in the forwarding queue and returning queue. The queueing delay increases with an increase in the number of enqueued message in the forwarding queue and returning queue. The forwarding operation is preferable when the queue length is small since its path has the shortest delay. As the number of clients increases and a longer queueing delay in the forwarding queue is encountered, the ability of returning a handle becomes more important in order to achieve a better performance. For the given experimental parameters, the pure forwarding and pure handle-driven approaches demonstrate substantial queueing delays and the adaptive ORB (QL=1 and QL=3) has superior performance.

Although the relative performances for different QL values remain similar in D=50, D=125 and D=250, we can find one interesting point by comparing Figure 5-21, Figure 5-22 and Figure 5-23. At a higher values of D, the pure handle-driven approach demonstrates a comparable performance with the pure forwarding ORB. The inter-node delay becomes the dominant parameter and has the strongest influence on performance when D=250. The additional message transfer delay incurred by a handle-driven ORB seems to be comparable to the queueing delay in forwarding queue by a message. It should be noted that the next request in the forwarding queue is sent and received by a selected server only when that server is released from the previous request. The server has to compute and send the result to the originating client before it is released. However, when a request from the returning queue is sent, it is immediately received by the client which is not executing currently but is waiting for a response to its service request. This is a possible reason why the handle-driven ORB has a similar performance to the forwarding ORB.

Figure 5-21: Throughput of the A-ORB with $D=50/50/50$ ms, $L=150$ bytes, $S_a/S_b=10/15$ ms.

Figure 5-22: Throughput of the A-ORB with $D=125/125/125$ ms, $L=150$ bytes, $S_a/S_b=10/15$ ms.

Figure 5-23: Throughput of the A-ORB with $D=250/250/250$ ms, $L=150$ bytes, $S_a/S_b=10/15$ ms.

Another point, which is observed in Figure 5-21, Figure 5-22, and Figure 5-23, is that the system with $QL=1$ performs better than the system with $QL=3$ when $N=2-16$, but both of them perform almost the same for the other values of N . The reason for that is indicated in Figure 5-24 and Figure 5-25. The returning thread for $QL=1$ starts to contribute to the throughput when $N=2$ and saturates when $N \geq 8$. The returning thread for $QL=3$ is utilized to process incoming requests when $N \geq 4$. Using only the heavily utilized forwarding thread seems to lead to an inferior performance. The returning threads for both $QL=1$ and $QL=3$ saturate when $N \geq 16$ so they have the similar throughput at the high load conditions.

Figure 5-24: Thread and CPU Utilizations of the A-ORB with $D=50/50/50$ ms, $L=150$ bytes, $S_a/S_b=10/15$ ms, $Q_L=1$.

Figure 5-25: Thread and CPU Utilizations of the A-ORB with $D=50/50/50$ ms, $L=150$ bytes, $S_a/S_b=10/15$ ms, $Q_L=3$.

5.3.2 Non-Equidistant Nodes

The parameters used for non-equidistant nodes experiments are shown in Table 11. The inter-node delay between any two nodes could be different from each other.

Table 11: Parameters of Experiments on Non-Equidistant Nodes

Factors	Levels
<i>N</i>	1, 2, 4, 8, 16, 24
<i>Sa/Sb (msec)</i>	10/15
<i>L (bytes)</i>	150
<i>Dca/Das/Dcs (msec)</i>	50/250/250, 250/50/250, 250/250/50
<i>QL (client)</i>	0, 1, 3, 24
Level of Thread Cloning	1

5.3.2.1 Client closer to the Agent

In this case, the client is much closer to the agent as shown in Figure 5-26. The inter-node delay is used to simulate the distance between two nodes. The messages passed among A-client, A-agent, and A-server are indicated by orders (1 to 4) and types ("F" means the forwarding message and "H" means the returning message).

Low Load Conditions:

The path length for the forwarding operation ($1+2F+3F$) is 550 ms, whereas the path length for the handle-driven operation ($1+2H+3H+4H$) is 600 ms. As shown in Figure 5-27, the throughputs for all QL values are comparable at low load conditions ($N=1$) since the lengths of these two paths do not have much difference.

Figure 5-26: The relative distance when the client is closer to the agent

Medium and High Load Conditions:

At higher loads, the pure forwarding ORB (QL=24) performs the worst, whereas the pure handle-driven ORB (QL=0) has better performance (see Figure 5-27). The lower inter-node delay in returning the handle to the client which is closer to the agent node gives rise to more than 100% improvement in throughput for the purely handle-driven ORB in comparison to the purely forwarding ORB when $N \geq 8$. The agent of the purely forwarding ORB saturates when $N=4$ (see Figure 5-29), whereas the agent of the purely handle-driven ORB saturates when $N=8$ (see Figure 5-28). This is because the longer inter-node delay between the agent and the server causes serious queueing delay in the forwarding operation at high load conditions. As a result the purely forwarding ORB demonstrates an inferior performance.

Figure 5-27: Throughput of the A-ORB with $D=50/250/250$ ms, $L=150$ bytes, $S_a/S_b=10/15$ ms.

The best performance is achieved by using the adaptive ORB ($QL=1$ or $QL=3$) (see Figure 5-27). When $N \geq 16$, $QL=1$ has comparable performance with $QL=3$. However, $QL=3$ does not perform as well as $QL=1$ when the number of clients (N) is greater than 2 and less than 16.

Some observations from Figure 5-29, Figure 5-30 and Figure 5-31 are summarized:

- The forwarding thread utilizations of $QL=1$, $QL=3$, $QL=24$ are approximately the same for any given value of N . This thread is saturated when $N=4$.
- The returning thread becomes very important since it is the only contributor to the system performance when the forwarding thread saturates.

- In the case of a purely forwarding ORB ($QL=24$), the returning thread is always in the idle loop, so the agent saturates after the forwarding thread saturates when $N \geq 4$. Therefore, the purely forwarding ORB performs the worst.
- In the case of $QL=3$, the returning thread cannot process the incoming request until $N=4$, so its throughput is the same as the purely forwarding ORB when $(1 \leq N \leq 4)$. The returning thread saturates when $N \geq 16$.
- In the case of $QL=1$, the returning thread can start to deal with the request when $N \geq 2$ so it performs better than the case of $QL=3$ when $(2 \leq N \leq 16)$. However, it also saturates at $N \geq 16$ and has the same throughput as $QL=3$ at high load conditions.

Figure 5-28: Thread and CPU Utilizations of the A-ORB Agent with $D=50/250/250$ ms, $L=150$ bytes, $S_a/S_b=10/15$ ms, $QL=0$.

Figure 5-29: Thread and CPU Utilizations of the A-ORB Agent with $D=50/250/250$ ms, $L=150$ bytes, $S_a/S_b=10/15$ ms, $QL=24$.

Figure 5-30: Thread and CPU Utilizations of the A-ORB Agent with $D=50/250/250$ ms, $L=150$ bytes, $S_a/S_b=10/15$ ms, $QL=1$.

Figure 5-31: Thread and CPU Utilizations of the A-ORB Agent with $D=50/250/250$ ms, $L=150$ bytes, $S_a/S_b=10/15$ ms, $QL=3$.

The adaptive ORB ($QL=1$ or $QL=3$) can offer superior performance than the purely forwarding ORB and purely handle-driven ORB because the cooperation between returning thread and forwarding thread can effectively reduce the software bottleneck. It is encouraging to see over 25% increase in throughput is achieved at $N=24$ by using $QL=1$ or 3 in comparison to the purely handle driven ORB (see Figure 5-27).

5.3.2.2 Server closer to the Agent

In this case, the server is much closer to the agent as shown in Figure 5-32. The messages passed among A-client, A-agent, and A-server are indicated by orders (1 to 4) and types ("F" means the forwarding message and "H" means the returning message).

Low Load Conditions:

The total forwarding path length ($1+2F+3F$) is 550 ms, whereas the total length of

handle-driven path ($1+2H+3H+4H$) is 1000 ms. The latter is almost twice the length of the former. As shown in Figure 5-33, the purely handle-driven ORB performs worse than the others at low load conditions. This is because one extra message is exchanged for the returning operation in comparison to the forwarding operation and there is significant difference in the path length (550 ms vs. 1000 ms).

Figure 5-32: The relative distance when the server is closer to the agent
Medium and High Load Conditions:

At higher loads, the purely forwarding ORB ($QL=24$) performs much better than the purely handle-driven ORB ($QL=0$). This is because the inter-node delay between the agent and the server, which affects the queueing delay in the forwarding queue, is much less than that between the agent and the client, which affects queueing delay in the returning queue. As a result, almost twice the throughput is achieved with the purely forwarding ORB.

As shown in Figure 5-33, the adaptive ORB ($QL=1$ or $QL=3$) exhibits a higher performance than the purely forwarding ORB. This also proves the importance of using a

hybrid middleware. There is still some difference between the throughputs of $QL=1$ and $QL=3$. When $4 \leq N \leq 16$, the A-ORB with $QL=3$ performs better than the A-ORB with $QL=1$. This is because there are more client requests forwarded to the server in the case of $QL=3$ and the forwarding path is much shorter than the returning path.

Figure 5-33: Throughput of the A-ORB with $D=250/50/250$ ms, $L=150$ bytes, $S_a/S_b=10/15$ ms.

5.3.2.3 Client closer to the Server

In this case, the client is much closer to the server as shown in Figure 5-34. The messages passed among A-client, A-agent, and A-server are indicated by orders (1 to 4) and types ("F" means the forwarding message and "H" means the returning message).

Low Load Conditions:

The contribution of the inter-node delays to the client response times is 550ms

$(1+2F+3F)$ for the forwarding operation and $600\text{ ms } (1+2H+3H+4H)$ for the returning operation. At low load conditions, all cases of different QL values perform similarly because there is not much difference on total inter-node delay between the returning operation and the forwarding operation (see Figure 5-35).

**Figure 5-34: The relative distance when the client is closer to the server
Medium and High Load Conditions:**

At higher loads, both the purely forwarding ORB ($QL=24$) and the purely handle-driven ORB ($QL=0$) perform comparably with one another (see Figure 5-35). The inter-node delay between the client and the agent dominates the queueing delay of the returning operation whereas the inter-node delay between the agent and server dominates the queueing delay of the forwarding operation. The agent has the same inter-node delay (distance) with the client and the server, so $QL=0$ and $QL=24$ have a comparable performance. The adaptive ORB ($QL=1$ or 3) demonstrates a much better performance

(approximately a 100% increase in throughput) than the purely forwarding ORB and purely handle-driven ORB.

Figure 5-35: Throughput of the A-ORB with $D=250/250/50$ ms, $L=150$ bytes, $Sa/Sb=10/15$ ms.

It is interesting to compare this case with the case in Section 5.3.2.1. Both of them have the same inter-node delays for the forwarding operation (550 ms) and the returning operation (600 ms). Although they generate the same results at low load conditions, significant difference on system performance can be found in Figure 5-27 and Figure 5-35 at high load conditions because:

- the total inter-node delay (distance) is the dominant factor of the system performance at low load conditions
- the inter-node delays of agent-client (for returning) and agent-server (for forwarding) are dominant factors of performance at high load conditions

5.3.2.4 Summary of Three Cases for Non-Equidistant Nodes

The relative performances of a purely forwarding ORB and a purely handle-driven ORB depends on the relative inter-node delay among nodes running the client, server, and agent. When the agent is closer to the server in comparison to the client, the purely forwarding ORB demonstrates a superior performance (see Figure 5-33) whereas the purely handle-driven ORB performs better when the agent is closer to the client (see Figure 5-27). The purely forwarding ORB and the purely handle-driven ORB have comparable performances when the distance between agent and client is the same as the distance between agent and server (see Figure 5-35). However, the adaptive ORB demonstrates the best performance irrespective of the relative inter-node delays.

5.4 Systems with Multi-Threaded Agents

A description of the internals of the A-agent was provided in Figure 3-6. For investigating the impact of multi-threading we have increased the number of forwarding and returning threads. Note that each forwarding/returning thread shares the forwarding/returning queue protected by a semaphore. In order to keep the number of experiments at a reasonable level, the experiments are run with a single set of L, D, Sa/Sb values. As shown in Table 12, this set of experiments is designed to understand the impact of the multi-threaded agent on the performance of the A-ORB. To alleviate the performance problems with the software bottlenecks encountered in the previous experiments, cloning of the returning thread and the forwarding thread is used. With a set of thread clones, one thread can process an outstanding message when the others are busy waiting for the recipients to

receive messages sent.

Table 12: Parameters of Experiments on Multi-Threading

Factors	Levels
<i>N</i>	<i>1, 2, 4, 8, 16, 24</i>
<i>Sa/Sb (msec)</i>	<i>10/15</i>
<i>L (bytes)</i>	<i>4800</i>
<i>Dca/Das/Dcs (msec)</i>	<i>125/125/125</i>
<i>QL (client)</i>	<i>0, 1, 3, 24</i>
<i>Level of Thread Cloning:</i> <i>Main (Listening) Thread</i> <i>Forwarding Thread</i> <i>Returning Thread</i>	<i>1</i> <i>1, 2, 3</i> <i>1, 2, 3</i>

As shown in Figure 5-36, Figure 5-37, and Figure 5-38, the throughput increases dramatically when the degree of thread cloning increases. In these cases, the A-ORB with $QL=1$ or $QL=3$ performs better than the purely forwarding ORB and purely handle-driven ORB. However, the gaps between system throughputs provided by the A-ORBs and the non A-ORBs shrink as there are more returning threads and forwarding threads in the agent which can effectively decrease the impact of the software bottleneck on the system performance. The enhancement of system performance can be observed. The purely handle-driven ORB ($QL=0$) saturates at $N=4, 8$, and 16 when the number of returning threads are $1, 2$, and 3 respectively. This is because the A-agent reaches a utilization of 1.0 at higher values of N (see Figure 5-39). The utilization of the A-servers are below 0.6 (see Figure 5-40).

Figure 5-36: Throughput of the A-ORB with $D=125/125/125$ ms, $L=4800$ bytes, $S_a/S_b=10/15$ ms, Level of Thread Cloning=1.

Figure 5-37: Throughput of the A-ORB with $D=125/125/125$ ms, $L=4800$ bytes, $S_a/S_b=10/15$ ms, Level of Thread Cloning=2.

Figure 5-38: Throughput of the A-ORB with $D=125/125/125$ ms, $L=4800$ bytes, $S_a/S_b=10/15$ ms, Levels of Thread Cloning=3.

Figure 5-39: Thread and CPU Utilizations of the A-ORB Agent with $D=125/125/125$ ms, $L=4800$ bytes, $S_a/S_b=10/15$ ms, $QL=0$, Level of Thread Cloning=3.

Figure 5-40: Process and CPU Utilizations of the A-ORB servers with $D=125/125/125$ ms, $L=4800$ bytes, $S_a/S_b=10/15$ ms, $QL=0$, Levels of Thread Cloning=3.

To investigate the performance bottleneck of the multi-threaded agent, we undertook more experiments. The parameters for Case 1 shown in Table 13 are used. Up to eight forwarding threads are created in the case of the purely forwarding ORB. As shown in Figure 5-41, throughput increases significantly when the number of the forwarding threads increase. However, the throughput starts decreasing when the number of forwarding threads is increased beyond 5 (see Figure 5-42). This result is unexpected because the throughput should remain the same when the bottleneck is reached, no matter how many threads are cloned.

It is desirable to understand why the system performance sometimes drops when the number of threads increases. Another experiment (Case 2) with the same parameters as in the previous case is therefore proposed. The only difference between these two cases is the

time-out limit for each thread. The time-out limit in Case 1 is one millisecond whereas that in Case 2 is one hundred milliseconds. The forwarding thread checks the forwarding queue and obtains one client request from the queue if it is not empty. If the forwarding queue is empty, the forwarding thread will sleep until the time-out. The same holds true for the returning thread. Figure 5-41 is comparable with Figure 5-43 but there is significant difference between Figure 5-42 and Figure 5-44. As shown in Figure 5-44, the experiments of Case 2 with the bigger time-out limit demonstrate comparable performance when the number of forwarding thread increases from 5 to 8. The throughput values at high load for both cases are also shown in Table 14. For a given level of thread cloning, the throughput achieved in Case 2 is better than that in Case 1. Thus by reducing the queue polling overhead we are able to achieve a higher performance. Sleep time for the forwarding and returning thread is a tuning parameter that is dependent on the level of thread cloning used.

As shown in Figure 5-44, increasing the level of thread cloning beyond 5 does not produce any additional performance benefit. Both the server process and agent CPU demonstrate high utilizations at the higher levels of multi-threading. For example, for a level of thread cloning=5 and $N=24$, the process utilization of the purely forwarding ORB ($QL=24$) for Server A and B are observed to be 0.81 and 0.84 respectively whereas the agent CPU utilization is 0.75. Thus increasing the level of thread cloning beyond 5 does not produce any significant improvement in performance. Since the agent CPU can become a performance bottleneck in a system using multi-threaded agent, it is very important to avoid unnecessary CPU usage.

Table 13: Parameters of more Experiments on Multi-Threading

Factors	Levels
<i>N</i>	<i>1, 2, 4, 8, 16, 24</i>
<i>Sa/Sb (msec)</i>	<i>10/15</i>
<i>L (bytes)</i>	<i>4800</i>
<i>Dca/Das/Dcs (msec)</i>	<i>125/125/125</i>
<i>QL (client)</i>	<i>24</i>
<i>Level of Thread Cloning:</i> <i>Main (Listening) Thread</i> <i>Forwarding Thread</i> <i>Returning Thread</i>	<i>1</i> <i>1, 2, 3, 4, 5, 6, 7, 8</i> <i>1, 2, 3, 4, 5, 6, 7, 8</i>

Table 14: Throughput with N=24, QL=24, L=4800 bytes, Sa/Sb=10/15 ms, and D=125 ms

Level of Thread-Cloning	Throughput of Case 1	Throughput of Case 2
<i>1</i>	<i>2.8816</i>	<i>2.9461</i>
<i>2</i>	<i>5.1806</i>	<i>5.3729</i>
<i>3</i>	<i>7.4093</i>	<i>7.6725</i>
<i>4</i>	<i>8.9959</i>	<i>9.4027</i>
<i>5</i>	<i>9.3099</i>	<i>9.5632</i>
<i>6</i>	<i>8.4399</i>	<i>9.6324</i>
<i>7</i>	<i>6.8548</i>	<i>9.6796</i>
<i>8</i>	<i>5.8772</i>	<i>9.6738</i>

Figure 5-41: Case 1 (Time-out: 1 ms): Throughput of the A-ORB with D=125/125/125 ms, L=4800 bytes, Sa/Sb=10/15 ms, QL=24, Levels of Thread Cloning=1~4.

Figure 5-42: Case 1 (Time-out: 1 ms): Throughput of the A-ORB with D=125/125/125 ms, L=4800 bytes, Sa/Sb=10/15 ms, QL=24, Levels of Thread Cloning=5~8.

Figure 5-43: Case 2 (Time-out:100 ms): Throughput of the A-ORB with D=125/125/125 ms, L=4800 bytes, Sa/Sb=10/15 ms, QL=24, Levels of Thread Cloning=1~4.

Figure 5-44: Case 2 (Time-out:100 ms): Throughput of the A-ORB with D=125/125/125 ms, L=4800 bytes, Sa/Sb=10/15 ms, QL=24, Levels of Thread-Cloning=5~8.

5.5 The Best Threshold Value vs. Load Balancing

This section presents a discussion on the choice of threshold values and introduces a load balancing strategy that does not depend on preset threshold values. Issues related to the choice of appropriate thresholds are discussed first.

In most of the previous experiments, the A-ORB has a better performance than the purely forwarding ORB and the purely handle-driven ORB. The ability to dynamically switch the mode of operation according to system load, has a significant impact on system performance. How to choose an appropriate threshold value to achieve the best performance needs investigation. As shown in Table 15, a set of threshold values are investigated for $N=8$ with the other parameters fixed. The results are presented in Table 16. The throughput increases as QL is increased from 0 to 1. In general, further increase in QL decreases the system throughput, indicating that the small threshold value such as $QL=1$ or $QL=2$ is the best choice. Another set of parameters is shown in Table 17 and the results of experiments are presented in Table 18. A system with $QL=0$ performs the worst whereas the performances produced by the other QL values are within 20% of one another.

The main difference between experiment Set 1 and Set 2 is with and without the inter-node delay respectively. The inter-node delay is one of the main reasons which cause saturation at the high load conditions. Saturation of the forwarding thread or the returning thread gives rise to a software bottleneck and is detrimental for system performance. The appropriate QL value needs to be determined. Analytical or simulation models of the system can be used to determine the appropriate QL for a given system and workloads.

Table 15: Parameters Set 1 of Experiments on Threshold Values

Factors	Levels
N	8
Sa/Sb (msec)	10/15
L (bytes)	150
Dca/Das/Dcs (msec)	50/50/50
QL (client)	0, 1, 2, 3, 4, 5, 6, 7, 8, 24
Level of Thread Cloning	1

Table 16: Throughput with N=8, L=150 bytes, Sa/Sb=10/15 ms, and D=50 ms

Threshold Value	Throughput
QL=0	4.4997
QL=1	8.8298
QL=2	8.9266
QL=3	8.0328
QL=4	7.1868
QL=5	6.2472
QL=6	5.8592
QL=7	5.8399
QL=8	5.8460
QL=24	5.8502
Load Balancing	8.9802

Table 17: Parameters Set 2 of Experiments on Threshold Values

Factors	Levels
N	8
Sa/Sb (msec)	10/15
L (bytes)	4800
Dca/Das/Dcs (msec)	0/0/0
QL (client)	0, 1, 2, 3, 4, 5, 6, 7, 8, 24
Level of Thread Cloning	1

Table 18: Throughput with N=8, L=4800 bytes, Sa/Sb=10/15 ms, and D=0 ms

Threshold Value	Throughput
QL=0	7.0702
QL=1	11.0637
QL=2	11.2807
QL=3	11.1908
QL=4	11.1091
QL=5	11.0629
QL=6	10.9174
QL=7	10.9324
QL=8	11.2227
QL=24	11.2186
Load Balancing	11.6080

The effective ways to alleviate software bottlenecks include multi-threading and load-balancing. The multi-threading technique has been presented in Section 5.4. The load-

balance technique is a heuristic that estimates the queuing delays in the forwarding queue as well as the returning queue and inserts the request into the queue with the shorter estimated delay. A high level description of the approach is provided:

- 1) Check the number of waiting requests (queue length) in the forwarding queue and returning queue.
- 2) Estimate the queuing delays (queue length multiplied by inter-node delay) which a new request will encounter in the forwarding queue and returning queue.
- 3) Compute the time consumed in the forwarding path (queuing delay plus the inter-node delay from the server to client) and returning path (queuing delay plus double of the inter-node delay between client and server) from the A-agent.
- 4) Put the request into the forwarding queue if the time consumed in the forwarding path is less than that in the returning path. Otherwise, choose the returning queue.

In summary the load balancing A-agent works as follows:

if (Forwarding Path Delay) < (Returning Path Delay)

 Put the request into the forwarding queue

else

 Put the request into the returning queue

Let

W: delay (includes queueing delay) from Agent to Client

X: delay (includes queueing delay) from Agent to Server

Y: delay from Client to Server

Z: delay from Server to Client

Q_f: number of requests in the forwarding queue

Q_r: number of requests in the returning queue

D: inter-node delay

In this heuristic we assume that the message transmission time is close to zero and we ignore the queueing delay at the server. The inter-node delay is assumed to be the same for any pair of nodes. Then the delays in the forwarding and returning queues can be estimated as follows:

$$\text{Forwarding Queue Delay} = X + Z$$

$$\text{Returning Queue Delay} = W + Y + Z$$

The expression (Forwarding Path Delay < Returning Path Delay) translates to:

$$(X + Z) < (W + Y + Z)$$

$$\Rightarrow X < W + Y$$

$$\Rightarrow Q_f * D < Q_r * D + D$$

$$\Rightarrow Q_f * D < (Q_r + 1) * D$$

$$\Rightarrow Q_f < Q_r + 1$$

Thus a request is inserted in the forwarding queue if $Q_f < Q_r + 1$; it is inserted in the returning queue otherwise. As shown in Table 16 and Table 18, the load-balancing agent demonstrates the best performance for the sets of parameters described in Table 15 and Table 17 respectively. An attractive feature of this technique is that we do not need a QL value. Continuing the investigation of the load balancing A-ORB with other parameter sets, especially with non-equidistant nodes, forms an important direction for further research.

Chapter 6 Conclusions

6.1 Summary

This thesis focused on the design, implementation, and analysis of the performance prototype of an A-ORB which is based on the Orbix middleware product. The performance of the A-ORB, which adapts its behavior to the workload conditions, is investigated and compared with the performance of a purely forwarding ORB and a purely handle-driven ORB. In general, a handle returning operation uses an additional message in comparison to a forwarding operation, whereas the forwarding operation requires a request to wait in a centralized forwarding queue. A summary of the results for each set of experiments is described.

6.1.1 The Impact of Server Demand

The impact of server demand is described and discussed in Section 5.1. The throughputs achieved with all QL values deteriorate as S_a and S_b increase. This is due to an increase in the queueing delay at the server. At low load conditions (small number of clients), a forwarding operation in the cases of $QL=24$, $QL=1$, and $QL=3$ produces a better performance than a handle-driven operation since only three messages are required for a forwarding operation. For a high load condition, the purely handle-driven approach performs the worst. The purely forwarding ORB performs the best with small server demands (10/15 ms), whereas the adaptive ORB performs superior to others with medium

server demands (50/75 ms). However, the cases with different QL values have comparable performance in a computation bound system in which the servers are characterized by heavy CPU execution times. Since the performance bottleneck appears in the servers (CPU saturation), the architecture of the agent seems to have very little impact on the performance.

6.1.2 The Impact of Message Size

A detailed description of the impact of message size on system performance is presented in Section 5.2. The throughputs for all QL values deteriorate as the message size increases. Due to one more message exchanged per client cycle, the purely handle-driven ORB performs the worst at low load conditions. For experiments with small message sizes, the purely forwarding ORB has a comparable performance with the adaptive ORB for both low and high load conditions. This is observed to be true for $L=4800$ and 9600 bytes. However, the performance of the adaptive ORB is significantly better than that of a purely forwarding ORB for a larger message size ($L=19200$ bytes). This is because the big message size causes longer transmission time which results in a longer queueing delay in the purely forwarding ORB. The performance of the purely handle-driven ORB is observed to be inferior to that of the purely forwarding ORB at high load.

6.1.3 The Impact of Inter-Node Delay

The effect of inter-node delay on performance is analyzed in Section 5.3. The throughput decreases when the inter-node delay increases. The experiments on inter-node delay demonstrate the importance of dynamically changing the mode of operation between

a handle-driven and a forwarding mode. The adaptive ORB performs better than the purely handle-driven ORB and the purely forwarding ORB in all cases of equidistant nodes (inter-node delay: 50 ms, 125 ms, 250 ms).

In the cases of non-equidistant nodes, the adaptive ORB demonstrates the best performance. However, the relative performances of the purely handle-driven ORB and the purely forwarding ORB depend on the inter-node delays among nodes running the client, the agent, and the server. When the agent is closer to the client in comparison to the server, the purely handle-driven ORB performs better than the purely forwarding ORB. This is because a lesser queueing delay is encountered in a handle-driven operation than in a forwarding operation. On the other hand, the purely forwarding ORB demonstrates a superior performance than the purely handle-driven ORB due to less queueing delay when the agent is closer to the server in comparison to the client. When the client is closer to the server in comparison to the agent which keeps the same distance to both the client and the server, the purely forwarding ORB and purely handle-driven ORB perform comparably due to similar amounts of queueing delay in the returning queue and forwarding queue.

6.1.4 Multi-Threading and Load Balancing

The impact of multi-threading and load balancing is considered in Section 5.4 and Section 5.5 respectively. They have proved to be effective ways to improve system performance. With a multi-threaded agent, the throughput can be improved up to 82% as shown in Case 2 of Table 14. Since multi-threading can alleviate software bottlenecks, the performance of all cases with different QL values are expected to improve when the agent

is the performance bottleneck. If a reasonable polling frequency is specified for the forwarding and the returning queues (Case 2 in Table 14), the throughput should increase as the number of forwarding threads and returning threads increase until the CPU capacity is reached. For example, when there are more than five returning threads and forwarding threads respectively, the throughput of the purely forwarding ORB with $QL=24$ cannot be increased much because the CPU of the agent node has saturated (see Section 5.4).

We found that an effective alternative to achieve high performance is to keep the load in the forwarding queue and returning queue balanced. The load-balancing technique described in Section 5.5 can reduce the queueing delay and balance the load in the forwarding queue and the returning queue. This technique does not need a predetermined QL value.

6.2 Conclusions

The adaptive ORB is observed to be superior to the purely forwarding ORB and purely handle-driven ORB in most situations. Situations in which the purely forwarding and the purely handle-driven ORBs rank first, the difference in A-ORB performance from that of the highest performer is observed to be small. Based on the current system load, the A-ORB can dynamically change its behavior and either returns a server's handle back to the client or forwards the client's request to the selected server.

At low load conditions, the total latency per client cycle, including the inter-node delay and transmission time, decides the superiority of performance whereas the queueing delay dominates system performance at high load conditions. In most cases, the performance

bottleneck is a software process and it occurs at the agent due to long queueing delays.

However, a hardware bottleneck (CPU) appears in:

- 1) the server with high server demands and with no inter-node delay (see Section 6.1.1).
- 2) the agent with many cloned threads which exceed the CPU capacity (see Section 5.4).

As mentioned earlier, the multi-threading technique is an effective way to improve the system performance when software bottlenecks exist on the system.

This thesis demonstrates the advantages of an adaptive middleware system and provides a number of insights into the relationship between different workload parameters and performance. This thesis also shows that it is possible to modify the interaction architecture on the middleware without having access to the source code of a commercial product. A higher performance benefit can be expected from the A-ORB when the source code of the agent is accessible.

6.3 Further Research

Although this thesis has been completed, there are some suggestions for further research or improvement:

- It is an important direction for further research to change the design of the agent in a COTS open source middleware product by changing its source code and thus achieving a complete adaptive middleware system.

- As Real-Time CORBA becomes more prevalent in distributed applications which are performance-sensitive, it is worthwhile to investigate an adaptive architecture in the context of real-time ORBs.
- Finding the best way to achieve load balance (see Section 5.5) requires further investigation of system performance under different workload conditions (e.g. systems with large messages) and on systems with non-equidistant nodes.
- We have used only a single criterion in the fixed threshold-based strategy employed by the A-agent. Whether or not using other information such as the inter-node delays in choosing between a forwarding operation and a handle returning operation can improve performance is an interesting question.
- We have used a polling based approach in the A-agent for dispatching requests from the forwarding and returning queues. Another alternative is to use a condition variable on which a forwarding/returning thread will block if the corresponding queue is empty. The blocked thread will wake up only when an item becomes available in this queue. Whether this alternative design improves system capacity warrants investigation.

References

- [1] Istabrak N. Abdul-Fatah, Shikharesh Majumdar, "Performance Comparison of Architectures for Client-Server Interactions in CORBA", Proceedings of IEEE 18th International Conference on Distributed Computing Systems (ICDCS'98), Amsterdam, May 1998, pp.2-11.
- [2] Istabrak N. Abdul-Fatah, "Performance of CORBA-Based Client-Server Architectures", M.Eng. thesis, Carleton University, Ottawa, Canada, April 1997.
- [3] Sean Baker, "CORBA Distributed Objects, Using Orbix", Addison Wesley, 1997.
- [4] Chih-Ping Chu, Chi-Jen Tzeng, "On the Code Development Paradigm of RPC and CORBA applications", Computer Communications, Vol. 21, No. 3, March 1998, pp.267-278.
- [5] David Diskin, Sherrie Chubin, "Predicting CORBA Performance Through Prototyping", Defense Information Systems Agency Joint Interoperability & Engineering Organization Center for Computer Systems Engineering.
- [6] Aniruddha Gokhale and Douglas C. Schmidt, "Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks", Proceedings of the IEEE International Conference on Distributed Computing Systems, Baltimore, May 1997, pp.401-410.
- [7] Aniruddha Gokhale and Douglas C. Schmidt, "Evaluating the Performance of Demultiplexing Strategies for Real-Time CORBA", Proceedings of IEEE GLOBE-

- COM '97 Conference, Phoenix, November 1997, pp.1729-1734.
- [8] Aniruddha Gokhale and Douglas C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks", Proceedings of ACM SIGCOMM '96 Conference, Stanford, August 1996, pp.306-317.
- [9] Aniruddha Gokhale and Douglas C. Schmidt, "Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance", Proceedings of the 31st Annual Hawaii International Conference on System Sciences, January 1998, pp.376-385.
- [10] Aniruddha Gokhale, Douglas C. Schmidt, "The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks", Proceedings of IEEE GLOBECOM '96 Conference, London, November 1996, pp.50-56.
- [11] Paul Haggerty and Krishnan Seetharaman, "The Benefits of CORBA-Based Network Management", Communications of the ACM, Vol. 41, No. 10, October 1998, pp.73-79.
- [12] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt, "The Design and Performance of a Real-Time CORBA Event Service", Proceedings of the 1997 OOPSLA Conference (ACM SIGPLAN), Atlanta, October 1997, pp.184-200.
- [13] Sungjune Hong, Youngjae Kimm, Minsoon Kweon, Dugki Min, and Sunyoung Han, "Object-Oriented Real-Time CORBA Naming Service on Distributed Environment", Proceedings of the Twelfth International Conference on Information Networking (ICOIN-12), 1998, pp.637-640.

-
- [14] Armond D. Inselberg, Peter Krupp, "Establishing the Requirements for Real-Time CORBA", Proceedings of MILCOM 97 Conference, Vol. 2, 1997, pp.768-772.
 - [15] Iona Technologies PLC, "Orbix C++ Administrator Guide", 1999.
 - [16] Raj Jain, "The Art of Computer Systems Performance Analysis, Techniques for Experimental Design, Measurement, Simulation, and Modeling", Wiley Professional Computing, 1991.
 - [17] Julien Maisonneuve, "Configuring Realtime CORBA Systems for Telecom Applications", Proceedings of ECOOP '97 Workshops (Object-Oriented Technology), Jyvaskyla, June 1997, pp.468-471.
 - [18] MITRE Corporation, "Recommendations for Using DCE, DCOM, and CORBA Middleware", White Paper in OMG home page (<http://www.omg.com>), April 1998, 58 pages.
 - [19] John E. Neilson, C. Murray Woodside, Dorina C. Petriu, and Shikharesh Majumdar, "Software Bottlenecking in Client-Server Systems and Rendezvous Networks", IEEE Transactions on Software Engineering, Vol. 21, No. 9, September 1995, pp.776-782.
 - [20] Object Management Group, "The Common Object Request Broker Architecture and Specification", V2.2, February 1998.
 - [21] Object Management Group, "OMG home page" (CORBA Standard and related information), <http://www.omg.com>.
 - [22] Object Management Group, "Real-Time CORBA - Joint Revised Submission",

OMG-TC Document Orbos/98-12-10, December 1998, 50 pages.

- [23] Orbix-Iona, "Iona Home Page", <http://www.ionas.com>.
- [24] Alan Pope, "The CORBA Reference Guide, Understanding the Common Object Request Broker Architecture", Addison Wesley, 1997.
- [25] Douglas C. Schmidt, "Overview of CORBA", <http://www.cs.wustl.edu/~schmidt/corba-overview.html>.
- [26] Douglas C. Schmidt, Aniruddha S. Gokhale, Timothy H. Harrison, and Guru Parulkar, "A High-Performance End System Architecture for Real-Time CORBA", IEEE Communications Magazines, February 1997, pp.72-77.
- [27] Douglas C. Schmidt, David L. Levine, Sumedh Mungee, "The Design of the TAO Real-Time Object Request Broker", Computer Communications, Vol. 21, No. 4, April 1998, pp.294-324.
- [28] Douglas C. Schmidt, Steve Vinoski, "Introduction to Distributed Object Computing", C++ report, 1995, 4 pages (obtained from <http://www.cs.wustl.edu/~schmidt>).
- [29] E-Kai Shen, Shikharesh Majumdar, Istabak Abdul-Fatah, "High Performance Adaptive Middleware for CORBA-Based Systems", in Proceedings of ACM's Principles of Distributed Computing Conference (PODC), July 2000 (to appear).
- [30] Sun Microsystems, "Solaris 2.6 Reference Manual-AnswerBook".
- [31] Sun Microsystems, "SunOS 5.2 Guide to Multi-Thread Programming", May 1993.
- [32] Steve Vinoski, "CORBA: Integrating Diverse Applications within Distributed Het-

erogeneous Environments", IEEE Communications Magazines, February 1997, pp.46-55.

- [33] Victor Fay Wolfe, Lisa Cingiser DiPippo, Roman Ginis, Michael Squadrito, Steven Wohlever, Igor Zykh, Russel Johnston, "Real-time CORBA", Proceedings of the Third Real-Time Technology and Applications Symposium, Montreal Canada, June 1997, pp.148-157.