Application Level Guidelines for Building High Performance CORBA-Based System

by

Weili Tao

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

Master of Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario, Canada

May 10, 2002

©Copyright 2002, Weili Tao

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.

0-612-79759-7

Canada

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

**Application Level Guidelines for Building High Performance
CORBA-Based System**

Submitted by Weili Tao,

In partial fulfillment of the requirements for the degree of
Master of Computer Science

Director , School of Computer Science

Professor Shikharesh Majumdar

Thesis Supervisor

Carleton University

May 10, 2002

Abstract

Next-generation distributed software must be interoperable, flexible and reusable. These requirements motivate the use of Object-Oriented (OO) middleware like the Common Object Request Broker Architecture (CORBA). CORBA is also intended to simplify the task of developing distributed applications. Although it is well-suited for conventional RPC-style applications, several limitations become evident when CORBA is used for a broader range of performance-sensitive applications running in heterogeneous environments over high-speed networks.

This thesis provides five performance enhancement design guidelines for high performance CORBA-Based systems. First, it studies different connection setup models in a CORBA environment and provides recommendations for minimizing the connection set up time. Second, it illustrates the performance impact of different ways of sending complex data structure and suggests guidelines for performance enhancement. Third, it evaluates the demultiplexing strategy used in a popular product called Orbix. Fourth, it provides a performance comparison between two different models of packing methods into object. Fifth it provides performance centric guidelines for packing objects into servers. The thesis also presents the study of load balancing issues in the application layer and compares the performance of different balancing strategies. Based on a commercial middleware product (Orbix) and measurements made on a network of workstations, this research provides insights into the design of high performance CORBA-Based applications.

Acknowledgments

Firstly, I would like to express very special thanks to my supervisor Professor Shikharesh Majumdar for his invaluable guidance and support. This thesis would not have been possible without his patience and encouragement.

I would also like to thank Brian Carroll and Mark Christopher of Nortel Networks for their support during the preparation of this thesis.

Finally, I wish to thank Mr. Naren Mehta for his advice as the system manager of the lab. I also acknowledge the support from our RADS lab group members – Richard Wu and Ping Li.

Table of Contents

<u>Chapter 1 Introduction</u>	1
1.1 Background	1
1.2 Motivations for the Research	2
1.3 Goals of this Research	3
1.4 Contributions of the Research	4
1.5 Thesis Outline	5
<u>Chapter 2 CORBA Overview</u>	6
2.1 Introduction	6
2.2 Structure of an Object Request Broker	8
2.3 CORBA Services	12
2.4 Benefits of CORBA-Based Applications	14
2.5 Related Work on CORBA Performance	16
<u>Chapter 3 Comparison of CORBA ORBs</u>	18
3.1 Introduction	18
3.2 Comparison of CORBA ORBs	19
3.2.1 CORBA Evaluation Criteria Template	19
3.2.1.1 Communication Extensions	22
3.2.1.2 Multi-threading Extensions	23
3.2.1.3 Management Extensions and Performance	24
3.2.1.4 Scalability	25

3.2.1.5 Robustness	25
3.2.2 ORB Comparison Matrix	26
3.2.2.1 ORB Platform Matrix	26
3.2.2.2 ORB Core Feature Matrix	26
3.2.2.3 ORB Services Support Matrix and Miscellanies	28
3.3 CORBA Benchmark Results	29
3.3.1 The Impact of Creation Order of Objects	29
3.3.2 Invocation Time for the First and Second Call on an Object	30
3.3.3 One-way and Two-way calls	30
3.3.4 Throughput with Different Types of Parameters	31
3.3.5 Data Type 'any'	32
3.3.6 Throughput Achieved with Different Char/String Aggregations	33
<u>Chapter 4 Guidelines for Performance Enhancement</u>	35
4.1 Introduction	35
4.2 Experimental Environment	35
4.3 Connection Setup Latency	36
4.3.1 Using _bind() Function	37
4.3.2 Connection Setup	37
4.3.3 Motivation for Different Connection Setup Model	39
4.3.3.1 Connection to an Active Server	39
4.3.3.2 Connection to a Collocated Server	40

4.3.4	Three Different Connection Setup Modes	41
4.3.4.1	Experiments	41
4.3.4.2	Factors	41
4.3.4.3	Performance Metrics	42
4.3.5	Results of Experiment	42
4.3.6	Design Guidelines	44
4.4	Passing Complex Data Structures	45
4.4.1	Motivation	45
4.4.2	Case Studies	48
4.4.2.1	Case A	48
4.4.2.2	Case B	49
4.4.3	Workload	51
4.4.4	Results of Experiment	51
4.4.4.1	Case A	52
4.4.4.2	Case B	54
4.4.5	Design Guidelines	57
4.5	Demultiplex Strategy in OrbixMT_2.3c	58
4.5.1	ORB Demultiplexing	58
4.5.2	Performance of Demultiplex Strategy Used in OrbixMT_2.3c	59
4.5.3	Results of Experiment	60
4.5.4	Design Guideline	62
4.6	Packing of Methods in Objects	62

4.6.1	Model Study	62
4.6.2	Comparison of the Two Models	64
4.6.3	Results of Experiments	64
4.6.4	Design Guideline	66
4.7	Packing of Objects	66
4.7.1	Four Different Packing Modes	66
4.7.2	Performance Comparison for the Four Different Modes	70
4.7.3	Results of Experiment	71
4.7.4	Design Guideline	76
Chapter 5	Guidelines for Load Balancing	77
<hr/>		
5.1	Introduction	77
5.2	Load Balancing Techniques	78
5.3	Strategies for Application-layer Load Balancing for CORBA-Based System	79
5.3.1	Nun-balanced Strategy	80
5.3.2	Static Symmetric Balancing Strategy	81
5.3.3	Half Dynamic Self Round Robin Strategy	82
5.3.4	Dynamic Self Round Robin Strategy	83
5.3.5	Dynamic Central Round Robin Strategy	84
5.4	The Implementation of the Five Strategies	85
5.4.1	Hardware / Software Platform	85
5.4.2	The Load Balancing System	86

5.5 Instrumentation and Performance Measurement	89
5.5.1 Workload	89
5.6 Results of Experiments	91
5.6.1 Simple System with Four Clients and Two Servers	91
5.6.2 More Complex System	95
 Chapter 6 Conclusions	 101
6.1 Summary	101
6.2 Guidelines for Application Design and Implementation	101
6.2.1 Connection Setup	101
6.2.2 Marshalling and Demarshalling of Complex Data Structure	102
6.2.3 Demultiplex latency with OrbixMT_2.3c	103
6.2.4 Packing of Methods	103
6.2.5 Packing of Objects	104
6.3 Load Balancing Strategies	104
6.4 Further Research	105
 References	 107
 Appendix A	
Supplementary Results for Higher Values of Service Times in a Simple System	 113

Appendix B

Supplementary Results with Exponentially Distributed Service Times and Think Times in a Simple System	116
--	------------

Appendix C

Supplementary Results for Higher Values of Service Times in a Complex System	123
---	------------

Appendix D

Supplementary Results with Exponentially Distributed Service Times and Think Times in a Complex System	126
---	------------

List of Figures

Figure 2-1:	CORBA object management architecture: Marketing Reference Model	7
Figure 2-2:	Components in the CORBA Reference Model	9
Figure 4-1:	How to automatically launch an Orbix server	38
Figure 4-2:	The connection set up latency for the five cases in three different connection setup modes	43
Figure 4-3:	Marshalling and demarshalling in Orbix	47
Figure 4-4:	Comparison of the three options in Case A	52
Figure 4-5:	Comparison of result for the four options in Case B	55
Figure 4-6:	Padding in a stream	57
Figure 4-7:	Layered CORBA request demultiplexing	59
Figure 4-8:	IDL definition with 500 methods	60
Figure 4-9:	Response time of invocation of different number of methods	61
Figure 4-10:	Response time of two different design models	65
Figure 4-11:	Mode I - Shared Object Mode	67
Figure 4-12:	Mode II - Multiple Objects Mode	68
Figure 4-13:	Mode III - Multiple Process Mode	69
Figure 4-14:	Mode IV - Multiple Hosts Mode	69
Figure 4-15:	Results of Experiment 1	72
Figure 4-16:	Results of Experiment 2	73
Figure 4-17:	Results of Experiment 3	74
Figure 5-1:	Strategy 1 - Non-balanced strategy	81
Figure 5-2:	Strategy 2 - Static symmetric balancing strategy	82
Figure 5-3:	Strategy 3 - Half Dynamic self round robin strategy	83
Figure 5-4:	Strategy 4 - Dynamic self round robin strategy	84
Figure 5-5:	Strategy 5 - Dynamic central round robin strategy	85
Figure 5-6:	Network environment for the more complex system	87
Figure 5-7:	Performance of load balancing strategy with ST=10ms, TT=500ms, NC=4, NS=2	93
Figure 5-8:	Performance of load balancing strategy with ST=20ms, TT=500ms, NC=4, NS=2	93
Figure 5-9:	Comparison of five strategies for various service times ST=10,0,40,60,80,120, TT=500ms, NC=4, NS=2	94
Figure 5-10:	Performance of load balancing strategy with ST=10ms, TT=500ms, NC=8, NS=4	98
Figure 5-11:	Performance of load balancing strategy with ST=20ms, TT=500ms, NC=8, NS=4	98
Figure 5-12:	Comparison of five strategies for various service times ST=10,0,40,60,80,120, TT=500ms, NC=8, NS=4	99
Figure A-1	Performance of load balancing strategy with ST=40ms, TT=500ms, NC=4, NS=2	114

Figure A-2	Performance of load balancing strategy with ST=60ms, TT=500ms, NC=4, NS=2	114
Figure A-3	Performance of load balancing strategy with ST=80ms, TT=500ms, NC=4, NS=2	115
Figure A-4	Performance of load balancing strategy with ST=120ms, TT=500ms, NC=4, NS=2	115
Figure B-1	Performance of load balancing strategy with ST=10ms (Exponential), TT=500ms, NC=4, NS=2	117
Figure B-2	Performance of load balancing strategy with ST=20ms (Exponential), TT=500ms, NC=4, NS=2	117
Figure B-3	Performance of load balancing strategy with ST=40ms (Exponential), TT=500ms, NC=4, NS=2	118
Figure B-4	Performance of load balancing strategy with ST=60ms (Exponential), TT=500ms, NC=4, NS=2	118
Figure B-5	Performance of load balancing strategy with ST=80ms (Exponential), TT=500ms, NC=4, NS=2	119
Figure B-6	Performance of load balancing strategy with ST=120ms (Exponential), TT=500ms, NC=4, NS=2	119
Figure B-7	Performance of load balancing strategy with ST=10ms, TT=500ms (Exponential), NC=4, NS=2	120
Figure B-8	Performance of load balancing strategy with ST=20ms, TT=500ms (Exponential), NC=4, NS=2	120
Figure B-9	Performance of load balancing strategy with ST=40ms, TT=500ms (Exponential), NC=4, NS=2	121
Figure B-10	Performance of load balancing strategy with ST=60ms, TT=500ms (Exponential), NC=4, NS=2	121
Figure B-11	Performance of load balancing strategy with ST=80ms, TT=500ms (Exponential), NC=4, NS=2	122
Figure B-12	Performance of load balancing strategy with ST=120ms, TT=500ms (Exponential), NC=4, NS=2	122
Figure C-1	Performance of load balancing strategy with ST=40ms, TT=500ms, NC=8, NS=4	124
Figure C-2	Performance of load balancing strategy with ST=60ms, TT=500ms, NC=8, NS=4	124
Figure C-3	Performance of load balancing strategy with ST=80ms, TT=500ms, NC=8, NS=4	125
Figure C-4	Performance of load balancing strategy with ST=120ms, TT=500ms, NC=8, NS=4	125
Figure D-1	Performance of load balancing strategy with ST=10ms (Exponential), TT=500ms, NC=8, NS=4	127
Figure D-2	Performance of load balancing strategy with ST=20ms (Exponential), TT=500ms, NC=8, NS=4	127
Figure D-3	Performance of load balancing strategy with ST=40ms (Exponential), TT=500ms, NC=8, NS=4	128
Figure D-4	Performance of load balancing strategy with ST=60ms (Exponential), TT=500ms, NC=8, NS=4	128

Figure D-5	Performance of load balancing strategy with ST=80ms (Exponential), TT=500ms, NC=8, NS=4	129
Figure D-6	Performance of load balancing strategy with ST=120ms (Exponential), TT=500ms, NC=8, NS=4	129
Figure D-7	Performance of load balancing strategy with ST=10ms, TT=500ms (Exponential), NC=8, NS=4	130
Figure D-8	Performance of load balancing strategy with ST=20ms, TT=500ms (Exponential), NC=8, NS=4	130
Figure D-9	Performance of load balancing strategy with ST=40ms, TT=500ms (Exponential), NC=8, NS=4	131
Figure D-10	Performance of load balancing strategy with ST=60ms, TT=500ms (Exponential), NC=8, NS=4	131
Figure D-11	Performance of load balancing strategy with ST=80ms, TT=500ms (Exponential), NC=8, NS=4	132
Figure D-12	Performance of load balancing strategy with ST=120ms, TT=500ms (Exponential), NC=8, NS=4	132

List of Tables

Table 1	ORB platform matrix	26
Table 2	ORB core feature matrix	27
Table 3	Object services matrix	28
Table 4	ORB Miscellanies	29
Table 5	Levels for experiment factors	42
Table 6	Connection set up latency	43
Table 7	Response time (μ s) achieved with the three options in Case A	52
Table 8	Response time (ms) achieved with the four different in Case B	54
Table 9	CDR alignment of primitive fixed-length types	56
Table 10	Performance demultiplexing of strategy in OrbixMT_2.3c	61
Table 11	Response time (μ s) for two different design models	65
Table 12	Levels for experiment factors	71
Table 13	Response time (ms) for four modes in Experiment 1 ST=0.03ms, NC=5,8,10,15	72
Table 14	Response time for four modes in Experiment 2 ST=10ms, NC=10	73
Table 15	Response time for four different in Experiment 3 ST=20ms, NC=10	74
Table 16	Levels for experiment factors	91

Glossary

BOA	Basic Object Adapter
CDR	Common Data Representation
CORBA	Common Object Request Broker Architecture
DCE	Distributed Computing Environment
DII	Dynamic Invocation Interface
DOC	Distributed Object Computing
DSI	Dynamic Skeleton Interface
ESIOP	Environment-Specific Inter-ORB Protocol
GIOP	General Inter-ORB Protocol
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
OA	Object Adapter
OMA	Object Management Architecture
OMG	Object Management Group
OO	Object Oriented
OOP	Object-Oriented Programming
ORB	Object Request Broker
POA	Portable Object Adapter
SII	Static Invocation Interface
SSL	Secure Socket Layer
TCP	Transmission Control Protocol

List of Terms

CL	Collocation of client and server
CSD	Client-Server Distribution
NC	Number of Clients
NS	Number of Servers
S	Message size
SA	Server Activity
ST	Service time for the server class
TT	Client Think Time

Chapter 1 Introduction

1.1 Background

The degree of improvement that has occurred in computer technology in the past half century is truly staggering and totally unprecedented in other industries. Two advances, in particular, should be mentioned here. The first is the development of powerful microprocessors, and the second is the invention of high-speed computer networks. The result of these developments is that it is now not only feasible but also convenient to put together computing systems composed of a large number of CPUs connected by a high-speed network. These systems are usually called distributed systems, in contrast to the previous centralized systems consisting of a single CPU [TAN-96]. Compared with centralized systems, distributed systems have many advantages. Potentially, a distributed system has a much better price/performance ratio than a single large centralized system. Another reason for building a distributed system is that some applications are inherently distributed. A third potential advantage of a distributed system over a centralized system is higher reliability; by distributing the workload over many machines, a single chip failure will bring down at most one machine, while leaving the rest intact. Also, connecting a group of isolated computers into a distributed system achieves an enhanced person-to-person communication. Finally, a distributed system potentially offers more flexibility in comparison to giving each user an isolated personal computer.

However, because of the limitations of the available conventional design tools the implementation of the distributed system is a major task. The Distributed Object Computing (DOC) approach is one of the major computing trends of this decade since it combines the advantages of distributed processing with those of Object-Oriented

technology such as encapsulation and polymorphism that offer benefits of design flexibility and software reuse. During the implementation of DOC system, however, an important system attribute, heterogeneity, must be considered.

Common Object Request Broker Architecture (CORBA) is a specification for the emerging DOC technology that is proving to be one of the most important innovations in the recent history of information systems. It addresses the two most prominent problems faced in the software industry today: 1) the difficulty of developing client-server applications 2) the difficulty of rapidly integrating legacy systems, off-the-shelf applications with new components. CORBA has provided abstractions for distributed object-oriented programming that has allowed developers to seamlessly integrate diverse applications into heterogeneous distributed systems. The core of any CORBA-based system, the Object Request Broker (ORB), hides the low-level details of platform-specific networking interfaces, allowing developers to focus on solving the problems specific to their application domains, rather than having to build their own distributed computing infrastructures [VIN-98].

1.2 Motivations for the Research

In recent years, the growth of the Internet and a substantial progress in Internet related technologies are remarkable. With the aim of easing the design of distributed applications, OMG defined a distributed object computing middleware – CORBA which allows client objects to access remote services in heterogeneous settings. CORBA is intended to support the production of flexible and reusable distributed services and applications. It enhances application flexibility and portability by automating many

common development tasks such as object location, parameter marshaling and object activation. In general, CORBA improves upon conventional procedural RPC middleware. However, research papers [SCH-96.2, SCH-96.11, SCH-98.1] show that the performance of conventional CORBA middleware implementations is relatively poor compared to lower-level implementations using sockets and C/C++, since the ORBs incur a significant amount of data copying, marshaling, demarshaling, and demultiplexing overhead. During the development of many CORBA-based distributed applications it is important to plan the design to ensure it meets the performance expectations and needs of its users. This thesis was inspired by the “XA-Core Capacity Analysis, Projection, and Tools” group of Nortel Networks, that was interested in gaining knowledge of the performance of the available ORB products as well as developing a set of performance centric guidelines for design/implementation of CORBA-based telecommunication applications. The thesis summarizes performance comparison surveys of different ORB products reported in the literature and investigates design/implementation issues that are concerned with the enhancement of application performance.

1.3 Goals of this Research

The primary goal of this thesis is to provide a set of design/implementation guidelines to the designers of CORBA-based systems. All the experiments used in the investigations reported in this thesis are based on the commercially available CORBA compliant ORB software called OrbixMT_2.3c [ION-97]. A number of issues that are concerned with the performance of CORBA-based applications are discussed.

This thesis is motivated by the following questions:

- a. What is the latency associated with the setting up of a connection between a client and its server? How can the connection set up latency be minimized?
- b. How does the ways of passing complex arguments to a method that are likely to impact the marshalling/demmarshalling overhead affects system performance? Is there a method for reducing such overhead and attaining high performance?
- c. How does the method dispatching strategy adopted in OrbixMT_2.3c affect system performance?
- d. How to pack multiple methods to get better performance?
- e. What is an efficient way to package objects in a multi-client multi-server environment?
- f. What is the impact of different load balancing strategies on system performance, and how sensitive is each strategy to the increase in number of clients and servers?

A summary of the comparison among different ORBs available in the literature is also reported.

1.4 Contributions of the Research

Most of the existing work on CORBA performance has focused on building fast ORBs. This thesis is concerned with performance optimization techniques at the application level. The contribution of this thesis includes a set of guidelines that can be used during application design and implementation. The guidelines are concerned with a number of different issues: connection setup latency, parameter passing, placement of methods into servers, packing of multiple methods into objects and packing of objects into servers. We have also devised and investigated a number of load balancing strategies

at the application level. Based on a synthetic workload, a network of workstations, and the OrbixMT_2.3c middleware we have investigated the impact of a number of performance optimization techniques that concern these issues. The performance improvement is observed to vary with the technique: from a few percent to an order of magnitude. Insights gained into system behavior and performance are reported. Some of the observations are intuitive, but our experimental results help the designer to quantify the performance impact and therefore understand the importance of the performance optimization in the context of a given application. The other observations are counter-intuitive and are therefore important for system design and implementation. The results of this research are likely to be most useful in the context of general CORBA-based systems. A part of this research is published in [TAO-02].

1.5 Thesis Outline

This thesis is organized as follows. Chapter 2 provides the overview of CORBA and the advantages of using CORBA as the middleware in a distributed system. A summary of comparison of different commercial ORBs that is available in the literature is presented in Chapter 3. Chapter 4 describes five guidelines for the CORBA application design/implementation. A detailed investigation of different load balancing strategies in a multi-client multi-server environment is provided in Chapter 5. In Chapter 6, we present our conclusions and recommendations for future research.

Chapter 2 CORBA Overview

2.1 Introduction

CORBA is an open standard that provides infrastructure to support the construction of distributed systems. The mission of CORBA is to define interfaces for interoperable software using an object-oriented technology. CORBA helps solve compatibility issues that arise when objects are written in different languages and run on different platforms, and is an effective solution to build a heterogeneous system of distributed objects.

The OMA Reference Model

Founded in 1989, the Object Management Group (OMG) is an international consortium for the software industry. The OMG was formed to stimulate the use of object technology in software engineering, and to develop the models and standard interfaces for the development and application of distributed environments using object technology. OMG does not produce software itself, but only publishes specifications that any interested vendor can use to develop software [SIE-98].

In 1990, OMG published the Object Management Architecture Guide (OMA), followed by Revision 2.0 in 1992. This guide laid the foundation for the definition of interfaces that different components use to communicate; it also addressed the need for standardized services and facilities in object-oriented environments. OMA introduced an unequivocal terminology for OOP languages, systems, databases and object libraries. Furthermore, it defined an abstract framework for OO systems, a collection of goals for techniques and architectures used for building such systems and a reference model for distributed OO systems.

Figure 2-1: CORBA object management architecture: Marketing Reference Model
(from [SIE-98])

Figure 2-1 is an overview of OMG's object management architecture (OMA), the industry standard reference model for object technology. The OMA Guide identifies four categories of software interfaces for standardization: the CORBA Object Request Broker (ORB), the CORBA Services, the CORBA Facilities, and the CORBA Domains. Each of these briefly described.

ORB – ORB is the central component of CORBA, through which all objects communicate. It encompasses all communication infrastructures necessary to identify and locate objects, to handle connection management and to deliver data across distributed heterogeneous computing environments.

CORBA Services – Provide basic functionality that almost any object might need. CORBA Services are generic. They are totally independent of any application domain. A number of CORBA services are described in Section 2.3.

CORBA Domains – It comprises various market areas, such as financial services, telecommunication applications and healthcare. The number of CORBA domains is potentially large, and represents areas for future growth of OMG standards activities.

CORBA Facilities – They are higher-level horizontal interface standards focused on application interoperability, and are horizontally applicable across domains. They include a collection of common-purpose objects, such as reusable components for the user interface, exception handling and printer output [SIE-98].

2.2 Structure of an Object Request Broker

Within a CORBA environment, objects residing in different processes, perhaps on different hosts and written in different languages, can communicate by sending requests to each other. In order to facilitate this, object interfaces are made available to other objects. These interfaces specify what kind of services the objects can deliver; this is enough for the client to be able to use those objects.

The implementation language of the server object may differ from the language the clients are written in. The interface is specified in a language-independent manner and interface stubs can be generated for the languages that a particular ORB product supports. Also, many ORB products run on a variety of platforms, including most UNIX flavors, and Windows NT. With the inter-ORB interoperability specified in CORBA 2.0 it is possible for applications using different ORBs to inter-operate.

Figure 2-2: Components in the CORBA Reference Model (from [SCH-98.10])

The conceptual architecture of a CORBA-based system is shown in Figure 2-2. Each component in the CORBA reference model is outlined here.

Client: The client program performs application tasks by obtaining object references and invoking methods. The server can be remote or collocated relative to the client. Ideally, accessing a remote server should be as simple as calling a method on a local object. Clients issue requests either through client stubs or through the Dynamic Invocation Interface that are described later in this section.

Object Implementation: This component implements the methods defined by an OMG IDL interface. It is identified by its object reference, which uniquely identifies the object in a server process.

ORB Interface: It provides standard operations that: 1) initialize and shut down the ORB 2) convert object references to strings and back, and 3) create argument lists for requests made through Dynamic Invocation Interface (DII) described later.

Interface Definition Language (IDL): Object interfaces are written in a programming language-independent specification language dubbed IDL. On the client side, an IDL compiler converts an IDL specification into client stubs. On the server side, the IDL compiler generates server skeletons. The advantage of using IDL is that interfaces can be easily specified.

IDL Stubs and Skeletons: They serve as a “glue” between the client, the object implementation and the ORB. Stubs provide a strongly-typed, static invocation interface (SII) that marshals application data available in the client’s native language format into a common packet-level representation called Common Data Representation (CDR). It also demarshals the results returned from the server. Conversely, skeletons demarshal the packet-level representation back into data represented in the native language for the server and marshal the results to be sent back to the client.

Dynamic Invocation Interface (DII): It allows clients to generate requests at run time. This flexibility is useful when an application has no compile-time knowledge of the interface it will access at run time. The DII also allows clients to make deferred synchronous calls, which de-couple the request and response portions of two-way operations to avoid blocking the client until the servant responds. In contrast, SII stubs

support only two-way, that is request/response, and one-way, that is, request-only operations.

Dynamic Skeleton Interface (DSI): It is the server's analogue to the client's DII. It provides a run time binding mechanism for servers and allows an ORB to deliver requests to a server that has no compile-time knowledge of the IDL interface it is implementing. Clients making requests need not know whether the server ORB uses static skeletons or dynamic skeletons. Likewise, servers need not know whether clients use the DII or SII to invoke requests.

Object Adapter (OA): It associates an object implementation with an ORB, demultiplexes incoming requests to the object implementation, and dispatches the appropriate operation up-call on that object. There are two types of object adapters that are briefly described.

Basic Object Adapter (BOA): It is an interface intended to be widely available and to support a wide variety of common object implementations. It includes convenient interface for generating object references, registering implementations, activating implementations and authenticating request etc.

Portable Object Adapter (POA): It is designed to allow programmers to construct object implementations that are portable between different ORB products.

Interface Repository: It is the component of the ORB that provides persistent storage of interface definitions – it manages and provides access to a collection of object definitions specified in IDL [OMG-00].

Implementation Repository: It contains information that allows the ORB to locate and activate object implementations. Most of the information in the Implementation

Repository is specific to an ORB or operating environment. In addition, it provides a common location to store information associated with object implementation, such as administrative control, resource allocation, and security.

ORB Core: It is responsible for delivering the request to the object implementation and returning a response, if any, to the client. For a server executing remotely, a CORBA-compliant ORB Core communicates via the Internet Inter-ORB Protocol (IIOP), which is a version of the General Inter-ORB Protocol (GIOP) that runs atop the TCP transport protocol. An ORB Core is typically implemented as a run-time library linked into client and server applications. In addition to general Inter-ORB Protocols, there can be Environment-Specific Inter-ORB Protocols (ESIOPs) [OMG-95]; the DCE Common Inter-ORB Protocol (DCE-CIOP) [OMG-95] is an example of such a protocol [CHA-98].

2.3 CORBA Services

The OMG has defined a set of standard services, known as CORBA Services, which can be provided as options above the core CORBA system. The CORBA Services define some of the most commonly required assistance that application programmers need for their systems, and without them, many application programmers would have to re-invent similar services and then use valuable time implementing them. Here, we discuss the definition of a number of popular CORBA services [VIN-98].

Naming Service – CORBA defines a service by which a human-readable name can be bound to a CORBA object. The purpose of this service is to provide applications with means to locate objects in a network, and can be used to obtain initial access to server objects.

Event Service – This service defines generic interfaces for passing event information among event sources and event consumers. Two roles are defined for objects by the event service: the supplier role (produce event data) and the consumer role (process event data). Event data are communicated between suppliers and consumers by issuing standard CORBA request. Event communication is initiated by two approaches: the push model and the pull model. Event channels are standard CORBA objects that allow multiple suppliers to communicate with multiple consumers asynchronously [OMG-00]. An event supplier puts events in an event channel and consumers waiting for a particular event receive a notification on an event occurrence. This service is used in the notification service in telecom applications for example [PAV-99].

Trading Service – This service provides applications with means to locate a service that meets specific criteria. With the help of Trading Service, servers can describe themselves using a set of properties. Clients can then use the Trading Service to locate a server with specific properties.

Transaction Service – This service provides applications with a support for ACID (Atomic, Consistent, Isolated, Durable) transactions. With the help of the transaction service, clients can encapsulate individual operations within a transaction.

Persistence Service – This service equips servers with means to provide persistency for objects. Relying on the persistence service, servers can provide objects whose lifetime is not limited by the server execution time.

Lifecycle Service – This service provides applications with basic operations related to object life cycle. The lifecycle operations include creating, deleting, copying and moving of objects.

Security Service – This service defines a general security framework that can be implemented in many ways, including the use of access control lists or capabilities.

Time Service – This service provides a set of interfaces that provide applications with the current time, and allow time and time interval values to be manipulated.

Property Service – This supports the notion of a property set, and any object can be associated with an object of this type.

Concurrency Control Service – This service provides a number of IDL interfaces to a locking facility that an object can use to control concurrent access.

Query Service – This service defines a simple collection type and a framework for making queries over the objects in such a collection.

The CORBA services are not of equal importance, nor of equal quality. The Naming Service, for example, is a simple, clear specification that is easily implemented on a standard CORBA system and then used in many applications. Some services have a narrower focus and are more complex. The Transaction Service, for example, is quite sophisticated and difficult to implement.

2.4 Benefits of CORBA-Based Applications

Here we discuss some advantages of using CORBA in the design and implementation of distributed application [HAG-98].

Creating an extensible application framework: A continuous change in distributed systems is a source of significant problem. As a result it inevitably forces changes in software. As CORBA is a standard based on an object model, it provides a good means

for tackling this problem. By using CORBA and object-oriented programming techniques, software can be enhanced and maintained easily.

Scalable solutions: High scalability is one of the most important considerations during the design of a network-based system. It means the system is able to handle large numbers of objects without hindering performance. CORBA enables systems to handle thousands of objects because objects are distributed among multiple servers; the location of an object in the system is completely transparent and objects interact with each other in the same manner regardless of whether they are local or remote.

Providing an open/standard interface: In order to design a complex distributed system successfully, there must be support for the exchange of information using standardized interfaces. One of the chief concerns from the customers is that the interfaces are compatible with their own system's architecture, operating system, and programming language. CORBA offers strong support for this, because it is operating system and programming language neutral.

De-coupling user interfaces: De-coupling is essential in OO design; it allows changes in one object but does not require changes in others. The use of object-oriented programming techniques has helped to provide isolation between the user interface and the application code. By having a protocol-defined interface, CORBA forces the separation of an object interface and implementation from its use. This means it de-couples the user interface code completely from the object implementation.

Programming language independence: In a heterogeneous system, different applications can be written in different languages and may use different operating systems. CORBA is programming-language independent; this allows different pieces of

the system to be written in different languages, and run on top of different operating systems.

2.5 Related Work on CORBA Performance

Research on CORBA performance has taken a number of different directions. Due to space limitation only a limited set of work is described in this section. A more complete survey is available in [MAJ-00].

A number of papers have captured the performance limitations of several existing CORBA-based middleware products. The limitations of the original CORBA standard in providing real time QoS, and in handling partial failures are described in [MAF-97]. Using load balancing techniques at the middleware can lead to an improvement in system performance [SCH-01.3]. The integrating of load balancing with name service is proposed in [BAR-99]. CORBA has been used in the context of various applications. The performance of a group membership protocol based on CORBA is discussed in [MIS-00]. A performance comparison of low level C++ wrappers for sockets outperformed two implementations of CORBA: Orbix and ORBeline [SCH-96.11]. The latency and scalability problems in conventional middleware products are discussed further in [SCH-96.8] whereas an investigation of the latency and scalability of two commercial products Orbix-2.1 and Visibroker 2.0 on systems employing high speed ATM networks is presented in [SCH-97.5]. The authors observe high latency for these CORBA implementations that also give rise to poor system scalability. A performance comparison of a number of products with a multithreaded architecture, CORBAplus, HP Orb Plus, miniCOOL, MT-Orbix, TAO, and Visibroker is available in [SCH-98.10]. Achieving

high performance by using innovative client-middleware-server interaction architectures is described in [ABD-02] whereas performance optimized ORBs that exploit limited heterogeneity in systems are described in [AHM-01, WU-01]. Real time ORBs that need to provide low latency as well as predictability have started receiving attention from researchers (see [LI-00] for example).

Most of the previous works on CORBA performance enhancements have focused on the construction of low overhead fast ORBs. This thesis focuses on enhancement of system performance through appropriate performance optimization techniques applied during application design and implementation.

Chapter 3 Comparison of CORBA ORBs

3.1 Introduction

This chapter is motivated by the requirements of “XA-Core Capacity Analysis, Projection, and Tools” group of Nortel Networks, that has supported our research. A summary of the comparison among different ORB products available in the literature is presented in this chapter. This literature survey is expected to be beneficial to the designers in choosing an appropriate ORB for their CORBA-based telecommunication systems. Application level guidelines for improving the performance of applications are addressed in the following chapters.

Since its first inception in 1991, CORBA – both the standard (whose current version is CORBA/IIOP 3.0) and the implementations - have had to continually evolve in order to remain viable as a basis for distributed applications [VIN-98]. Right now, major CORBA vendors offer a range of C++ and Java ORBs that differ in many significant aspects. For a customer who is not intimate with the many OMG standards, it is becoming increasingly difficult to assess the practical impact of the differences between various ORB implementations.

This chapter describes a set of criteria that would be used to evaluate different ORB implementations [CHA-98]. The aspects of the ORB functionality covered in the criteria are from the application developer’s point of view (this includes functionality specified in the OMG standards, together with useful proprietary extensions). The criteria are used to evaluate the following ORB products (ORBIX from IONA, VISIBROKER from INPRISE, OMNIORB from AT&T, and TAO born of research at Washington

University). For performance comparison purposes, a suite of benchmarks results are summarized [VIR-99].

This set of criteria is very useful in general for comparing different ORB products. A subset of these has been used in Section 3.2.2 that reports on the results published in [PUD-01].

3.2 Comparison of CORBA ORBs

3.2.1 CORBA Evaluation Criteria Template

The criteria template described in this section provides a model for the users to thoroughly compare the characteristics of ORBs from different vendors [CHA-99]. There are four parts of this template: Platform Support, Object Request Broker, Object Services and Miscellanies. The details of this template are presented.

EVALUATION CRITERIA TEMPLATE [CHA-98]

❖ Platform Support

❖ Object Request Broker

➤ Standardized Functionality (defined in Section 2.2)

- ✓ Interface Definition Language
- ✓ Remote Invocation
 - BOA
 - POA
- ✓ Dynamic Type Manipulation
 - Dynamic Invocation Interface
 - Dynamic Skeleton Interface
- ✓ Repositories
 - Interface Repositories

- Implementation Repositories
 - ✓ Interoperability
- Non-Standard Extensions
 - ✓ Communication Extensions (defined in Section 3.2.1.1)
 - Locating Objects
 - Binding to Objects
 - Instantiating Implementations
 - Persistent References
 - Automatic Activation
 - Automatic Reconnection
 - Invocation Extensions
 - Customizability
 - ✓ Multi-threading Extensions (defined in Section 3.2.1.2)
 - Support multi-threading on Client and Server side
 - Flexible threading strategies
 - ✓ Management Extensions (defined in Section 3.2.1.3)
 - Installation, Configuration
 - Administrations tools
- Scalability (defined in Section 3.2.1.4)
 - ✓ Performance with large numbers of objects
 - ✓ Performance with large number of clients
 - ✓ Resource consumption per object
- Robustness (defined in Section 3.2.1.5)
 - ✓ Reliable Servers
 - ✓ Network Failures
 - Mechanism to detect loss of communications – rebinding
 - ✓ Network Congestion
 - Limitation - load balancing on binding
 - ✓ No memory leaks

❖ Object Services (defined in Section 2.3)

- Naming Service
- Trading Service
- Event Service
- Transaction Service
- Concurrency Control Service
- Security Service
- Persistent State Service
- Property Service
- Query Service
- Time Service
- Lifecycle Service

❖ Miscellanies

- Requirements
 - ✓ Supported Platform
 - ✓ Compilers
 - ✓ Resource Consumption
- Development
 - ✓ Quality of Document
 - ✓ Quality of Example Code
 - ✓ Debugging Features
 - ✓ Web site for information sharing
 - ✓ Integration with IDEs such as Visual Café
- Vendor
 - ✓ Expertise
 - ✓ Pricing Policy

The standardized functionality parts are all CORBA implementations defined by OMG, which we have introduced in Chapter 2. Here we will give the explanation on non-standard extensions.

3.2.1.1 Communication Extensions

Locating objects – One of the very basic features required by ORB clients is the ability to locate an object given a set of criteria. In the standardized OMA architecture, this feature is provided by the Trading Service, but since this service was standardized relatively late, many vendors have introduced proprietary solutions in the meantime.

Binding to Objects – The standard way to bind to an object is to get a bootstrap reference to the Naming or Trading Service and to use that reference to obtain the reference or handle for the desired object. In addition to these functions, the ORB vendors often introduce proprietary binding mechanisms.

Instantiating Implementations – The instantiation of a CORBA object on the server side of an application consists of: creating the actual implementation object; creating the unique object reference; registering the implementation object with the reference. The mechanism to do this was described poorly in the original CORBA standard (this is corrected in CORBA 2.2) and thus its implementations differ in the individual ORBs [CHA-98]. The same is true for the object deletion mechanism.

Persistent References – An important feature of the ORB is the support for persistent object references. These are necessary when implementing persistent objects and are usually tied to the server activation mechanism. Again, the original CORBA standard was

not detailed enough in this area (corrected in CORBA 2.2) and thus most ORBs offer proprietary solutions.

Automatic Activation – An important part of the ORB functionality, used especially with persistent servers and servers that are to provide a higher degree of reliability, is the ability to launch a server on demand. This feature is generally implied by the existence of the Implementation Repository.

Automatic Reconnection – This is related to the automatic activation mechanism is a mechanism that allows a client to automatically reconnect to a server in case of a communication failure.

Invocation Extensions – Extensions to the standard invocation mechanisms (e.g. asynchronous invocations and callbacks) are provided by most ORBs as well.

Customizability – It is also important to have the possibility to customize the ORB behavior (e.g. intercepting and filtering of messages, piggybacking of data onto messages and event hooks).

3.2.1.2 Multi-threading Extensions

Multi-threading Servers

The most common case of a multi-threading extension is a support for multi-threaded server implementations. Several basic strategies for server implementation are available in the literature [SCH-98.10]:

- **Single thread:** all incoming requests are serialized and processed by a single thread.

- Thread per request: For every incoming request a new thread is created. After processing a request, the thread is terminated.
- Pool of threads: During the start of the server, a pool of threads is created to handle incoming requests. Each thread waits for an incoming request, handles it and loops back to repeat this sequence. This strategy allows for parallel processing, while limiting the server's use of resources.
- Thread per client: A separate thread is created for each client process that connects to a server.
- Thread per object: A separate thread is created to handle requests for one server object only. This can be useful in real-time system, where the thread associated with one object has higher priority than of the others.

Clearly, it is desirable that the ORB supports at least some of the strategies listed above. It is also important that the ORB is flexible enough to allow other strategies to implemented.

Multi-threaded Clients

Apart from employing multi-threaded servers, there is a strong benefit from employing multi-threaded clients. For example the client can use non-blocking calls. A special thread is created to make a remote synchronous operation. The client can continue processing without waiting for the result of the operation.

3.2.1.3 Management Extensions and Performance

In most cases, the installation of an ORB running on a single computer is simple to configure and manage. However, with the growing number of network hosts running

ORB clients and servers the management becomes complicated and a need for a sophisticated configuration and management tool arises. As there are no OMG standards for ORB configuration and management, the configuration and management tools are proprietary and offer functions specific to a given ORB.

3.2.1.4 Scalability

A special section is dedicated to the scalability of the ORB. Three benchmarks are described to compare the scalability of different ORBs. The first one measures the dependence of the invocation time and memory consumption on the number of proxies and servants handled by the ORB. The second one is used to measure the dependence of the invocation time on the number of clients handled by a server, and the last one evaluates the behavior of the ORB when overloaded by one-way requests.

3.2.1.5 Robustness

Here, the support for building a reliable server is discussed. Limits of the ORB implementation, such as the maximum number of servants and the maximum size of requests are discussed.

In this section we have discussed four groups of criteria that can be used to evaluate different ORB implementations. These are – Platform Support, ORB Features, Object Services and Miscellanies. In the following subsections we will compare Orbix 2000, Visibroker for C++3.0, OmniORB 4.0 and TAO 1.2a (beta) using this template. These four ORBs are chosen because of their distinct characterizations. We know that Orbix by

IONA Technologies and Visibroker by Inprise Corp are two popular ORBs used in many commercial and telecommunication applications. OmniORB is an open source code that is free to its users. TAO focuses on the real time application.

3.2.2 ORB Comparison Matrix

Four comparison matrixes are presented in this section to illustrate evaluation criteria we discussed before – Platform support, ORB core features, Object service support and Miscellanies.

3.2.2.1 ORB Platform Matrix

CORBA is introduced to provide interoperability as well as transparent location of servers in a distributed and heterogeneous client-server environment [ABD-97]. This heterogeneity may arise because of the use of different programming languages, different operating systems, or different types of networks [BAK-97]. Table 1 lists the platforms supported by each ORB [PUD-01].

Table 1: ORB platform matrix

ORBs \ OS	Solaris	HPUX	AIX	DEC	Linux	NT	W2000	Other
Orbix 2000	Y	Y	Y	Y	Y	Y	Y	Y
Visibroker for C++3.0	Y	Y	Y	Y	Y	Y	Y	Y
OmniORB 4.0	Y	Y	Y	Y	Y	Y	Y	-
TAO 1.2a (beta)	Y	Y	Y	Y	Y	Y	Y	-

3.2.2.2 ORB Core Feature Matrix

In this section, the ORB's compliance to OMG standards is evaluated. As described in the evaluation criteria template, the following aspects are important for the ORB. These are interface definition language bindings, interoperability, remote invocation (BOA and POA), dynamic type manipulation (Dynamic Invocation Interface, Dynamic Skeleton Interface) and Repositories. Table 2 discusses these core features in the context of different products.

Table 2: ORB core feature matrix (from [PUD-01])

		Orbix 2000	Visibroker for C++3.0	OmniORB 4.0	TAO 1.2a (beta)
Language Bindings	IDL	Y	Y	Y	Y
	C++	Y	Y	Y	Y
	C	N	N	N	N
	SmallTalk	N	Y	N	N
	Ada	N	N	N	N
	Java	Y	Y	-	N
	TCL	N	N	N	N
	Perl	N	N	N	N
	Cobol	Planned	N	N	N
Protocols	IIOP	Y	Y	Y	Y
Core	DII	Y	Y	Y	Y
	DSI	Y	Y	Y	Y
	IFR	Y	Y	Y	Y
	BOA	N	Y	N	Y
	POA	Y	Y	Y	Y

Orbix is one of the popular ORB, with a wide range of supported platforms and availability of a number of accompanying products. Visibroker is based on the code of

Orbeline, and is available for several C++ and Java platforms. It also offers a number of accompanying products such as object service. OmniORB is a lightweight CORBA implementation. TAO is a high-performance ORB endsystem for real-time CORBA [SCH-97.2].

3.2.2.3 ORB Services Support Matrix and Miscellanies

The CORBA Services are a set of enabling interfaces for globally applicable service capabilities [MOW-97]. Table 3 describes the services supported by the different ORB products whereas various miscellaneous features are addressed by Table 4.

Table 3: Object services matrix (from [PUD-01])

Services	Orbix 2000	Visibroker For C++3.0	OmniORB 4.0	TAO 1.2a (beta)
Naming	Y	Y	Y	Y
Lifecycle	-	N	-	Y
Event	Y	Y	Y	Y
Trading	Y	N	N	Y
Security	N	N	N	Y
Transaction	Y	N	N	N
Persistent State	Y	N	N	N
Concurrency	N	N	N	Y

Table 4: ORB Miscellanies (from [PUD-01])

Miscellanies	Orbix 2000	Visibroker For C++3.0	OmniORB 4.0	TAO 1.2a (beta)
Developer License Fees	Y	-	N	N
Runtime License Fees	Y	-	N	N
Source freely available	N	N	Y	Y
Product released under Open Source License	N	N	Y	Y

3.3 CORBA Benchmark Results

The CORBA Benchmarks project was undertaken by Virginia University to compare the performance of various kinds of ORBs, such as ORBIX, TAO and VISIBROKER [VIR-99]. In all the tests, the server was running on a Compaq Proliant 333MHz bi-Pentium II machine with 512Mb of RAM, and the client was running on a 266 MHz Pentium II with 64Mb of RAM. The middleware used are: Orbix_3.0, TAO0.2.46 and Visibroker C++3.3. The results are summarized in the following subsections.

3.3.1 The Impact of Creation Order of Objects

To test the impact of creation order of objects, first, 1000 instances of an object were created on the server side. The naming service is used by the client to get the object's reference. Then the client calls a method on the first created implementation 1000 times, and then calls a method on the last created 1000th instance the same number of times.

The results show that for Visibroker and TAO there are no big differences between the latencies achieved with the 1st created object and the 1000th object. In Visibroker, method invocation on the 1000th object takes 3% more time than that observed with the 1st object. TAO takes 2.5% more time than the 1st object. But in Orbix 3.0, the call on 1000th object takes 17% more time than that on 1st object. A possible explanation is that Orbix 3.0 uses a linear search as the demultiplexing strategy.

3.3.2 Invocation Time for the First and Second Call on an Object

In this test, a method which has no parameter and returns a void is invoked two times. The time to complete the first invocation is compared with the time to invoke the second.

The results show that, for Orbix and Visibroker, the first call takes more time than the second one. It takes 7 times more invocation time in Visibroker for the first call compared to the second one, and, for Orbix 2.3, the first call takes 44% more time than the second, whereas for Orbix 3.0, the first call needs 19% more invocation time than the second. This is because the proxy is constructed when the first method is called on the object. The proxy is reused in the subsequent calls. But in TAO, this phenomenon does not exist.

3.3.3 One-way and Two-way calls

There are two types of operations that can be used to send a message from a client to a server – a two-way send (also called a normal operation) and a one-way send.

Normally, the caller of an operation is blocked while the call is being processed by the target object until the operation request has been carried out and a reply is received. This is called a two-way send, which is a reliable operation.

In a one-way send, 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. However, there is no guarantee that the message would reach the receiver, and the caller may not know if a failure occurs. A one-way operation must specify a void return type and cannot have out or inout parameters, and it also cannot have a "raises" clause. The one-way operation is faster but less reliable in comparison with the normal operation.

To compare the performance of these two types of operations, a method without a parameter and returning void is used. As expected, the results demonstrate that a one-way call is much faster than two-way calls. With Visibroker, one-way calls are nine times faster than two-way calls. With Orbix 3.0, a one-way call is more than four times faster, whereas with TAO, a one-way call is eight times faster [VIR-99].

3.3.4 Throughput with Different Types of Parameters

To test the throughput achieved with different basic data types (octet, char, short, long, double, float), a set of methods are used, each one using one 'in' parameter of a given size for an array or a sequence of various basic data types. For this test, each method was called 1000 times. The size of the 'in' parameters was 1KBytes. The parameters passed to each method are described.

Array structures: octet[1024], char[1024], short[512], long[256], double[128], float[256].

Group 1b:	seq<any,1024>(octet), seq<any,1024>(char), seq<any,512>(short), seq<any,256>(long), seq<any,128>(double)		
Group 2a:	octet[1024], char[1024], short[512], long[256], double[128]		
Group 2b:	seq<octet,1024>,	seq<char,1024>,	seq<short,512>, seq<long,256>, seq<double,128>

The test results show that compared to other basic data types (long, char, short...), using 'any' for simple types is not slower. But using 'any' type is much slower than using standard types directly as a parameter when used with composed types such as arrays and sequences.

3.3.6 Throughput Achieved with Different char/string Aggregations

The purpose of this test is to compare the cost of sending various char/string based data structures. Two-way calls were used, in five different ways to send certain sizes of characters. The message sizes used for this test include 1KBytes, 2KBytes, 4KBytes, 8KBytes to 10Kbyte. The data sent includes types string, array char, sequence char, array string, and sequence string

The results show that the fastest way to send characters over the wire is to use arrays of chars or a sequence of chars. Sending arrays of strings and sequences of strings is much slower than the other three ways [VIR-99]. We have investigated system performance achieved when data structures composed of various data types are transferred between client and server (see Section 4.3).

Most of the issues discussed in this chapter are concerned with the middleware product used in a DOC system. The research presented in the following chapters

however, is concerned with application level issues and focuses on techniques to be used during system design and implementation for improving system performance.

Chapter 4 Guidelines for Performance Enhancement

4.1 Introduction

Compared with traditional low-level sockets, the performance of CORBA middleware implementations is relatively poor [SCH-96.2], since the ORBs incur a significant amount of data copying, marshaling, demarshaling, and demultiplexing overhead. So, careful design of a CORBA-based distributed application is required to achieve high performance. In this chapter, we describe five design issues for improving the performance of CORBA-based applications. The first issue is concerned with connection setup latency. The second issue investigates the impact of marshalling and demarshalling of complex data structure on performance. The third analyzes OrbixMT_2.3c demultiplexing strategy. The fourth looks at two design models. Model I uses one interface to take care of all methods, whereas Model II defines multiple interfaces each of which takes care of one method. The last issue is concerned with how to efficiently package the servants in a multi-client multi-server environment. A number of experiments are carried out to get insight into system performance. Design of experiments, descriptions of applications used in the experiments, and results of performance evaluation are presented in this chapter. Based on the results of experiments, a number of design guidelines and recommendations for achieving high performance are described.

4.2 Experimental Environment

The experiments used in the investigation of the issues were conducted on the “quiet network” system in the Real Time and Distributed System lab. The nodes within this

network are isolated from other nodes in the same lab and are dedicated for running the experiments presented in this thesis. All the nodes in this “quiet network” are Sun Sparc-2 workstations running under the Solaris 2.6 operating system. A 10Mbps Ethernet LAN is used to connect these nodes. The middleware running on the nodes is “OrbixMT_2.3c”. The client and server used in all the investigations are written in C++. Various latencies are investigated during the experiments. The latencies are measured in terms of response time that is defined next.

Response Time: is the difference between the time the result from the server arrives and the time at which the client performs the method invocation. Note that response time is related to system throughput by Little’s Law:

$\text{Throughput} = \text{mean response time} * \text{number of clients}$

Setting up a connection between a client and its server involves a number of steps that are described in Section 4.3.2. A `_bind()` function provided by the ORB is used for this connection set up. The connection set up latency is defined as follows.

Connection set up latency: is the difference in time at which the `_bind()` function returns and the time at which the function is invoked.

The time of occurrence of an event is captured by using the `gettimeofday()` solaris system call. The experiments described in this thesis are run long enough such that the confidence interval for the performance metrics is less than or equal to $\pm 5\%$ of the mean at a confidence level of 95%.

4.3 Connection Setup Latency

CORBA is designed to enhance distributed applications by automating a set of common networking tasks such as object registration, location, and activation [BAK97]. Each object in the system has a unique object reference; it includes such information as the name of the server in which the object is located and the host on which the server runs. There are two ways to obtain an object reference and locate the object in the system: using the Naming Service and binding. Naming Service allows a name to be mapped to an object handle or reference and allows that object to be subsequently found by resolving that name. Binding is the other way to find an object reference. Compared with the first approach, the binding function is simpler and more direct [ION-97]. In this section we will discuss Orbix `_bind()` function.

4.3.1 Using `_bind()` Function

Binding is used to find a particular object and create a proxy for it in the client's address space. When an operation is invoked on the proxy, Orbix transmits the request to the target object.

In the client side, when `_bind()` is called, the following parameters need to be supplied:

- `<object name>:<server name>`, the names the object and the server on which the object is running. The server name is the same as the one used by the server while it registers to the CORBA Implementation Repository.
- `<host name>`

4.3.2 Connection Setup

Figure 4-1: How to automatically launch an Orbix server

A typical scenario for a connection set up is presented in Figure 4-1.

1. A client makes its first bind call to an object located in a server.
2. The Orbix daemon reads the server details from the Implementation Repository, including the server launch command.
3. If the server is not already activated, the Orbix daemon executes the server launch command to activate the server.
4. The server handle is then returned to the client. (This handle is then used by the client to invoke a method in a server).

The connection set up latency is the sum of the times required to perform these four operations.

An Orbix client invokes IDL operations on a distributed object using normal C++ method invocations, as if the object were located in the client's address space. Orbix

converts these method invocations to a series of network messages and sends these messages to the server process that contains the target object. At the server side the ORB receives these messages that are then translated to method invocations on the target object.

The Orbix Daemon is a process that runs on each server host and implements several ORB components, and manages the server processes running in the system. The Implementation Repository is a component of Orbix that maintains information about objects available in the system. It saves each object's name and the file name for the executable codes associated with the object.

Before running the application on the client side, there are two mandatory operations that need to be performed:

- Running the Orbix Daemon on the server host.
- Registering the Server.

Server code must be registered with the Implementation Repository. Server registration has the advantage that it allows the Orbix daemon to launch a server that is not running when one of its objects is used. During the registration, two parameters are required; the first one is the server's name, the second parameter is the full name of the server executable file, including the absolute directory path.

4.3.3 Motivation for Different Connection Setup Models

4.3.3.1 Connecting to an Active Server

Section 4.3.2 describes the steps to get the connection setup between the client and the server. Before the connection setup, the server should make an object available to the

client through the Orbix function `CORBA::BOA::impl_is_ready(serverName, timeoutValue)`. This function takes two parameters: one is a server name and the other is a timeout value. The timeout value indicates the period of time the server is alive or how long the server is blocked on `impl_is_ready()` waiting for a request to arrive from a client. When the connection from the client is closed after it receives the response, the server will be still in an active mode waiting for the next request to arrive. We have investigated two kinds of connections by using the `_bind()` function. First one is to set up the connection to a non-active server. Two alternative ways can achieve this. In the first type a connection is set up with a non-active server; that is, set up the connection with the server after the timeout interval. The second type is to connect to an active server. To implement this, we just set up a connection while the server is alive. By setting up a connection to an active server, the time used for launching a server can be saved.

4.3.3.2 Connection to a Collocated Server

In some applications, we may find that, even if objects are not distributed, it is still useful to define a different interface for each object. Further, the designer can allocate parts of the application on the same node. Orbix supports the collocation function to make the client and server objects stay within the same address space. The collocation mechanism by-passes the Orbix runtime, and makes calls on the object very efficient. If the client and server are collocated, the bind call will enforce the client to look for the object only in the caller's address space. Then the subsequent calls on the object will use direct function calls instead of Orbix runtime, and thus make the calls very efficient.

4.3.4 Three Different Connection Setup Modes

The purpose of this experiment is to compare the response times for setting up a connection with OrbixMT_2.3c using various combinations of three different modes - connecting to a non-active server, connecting to an active server and connecting to a collocated server.

4.3.4.1 Experiments

Experiments are conducted to investigate five cases associated with the three modes.

Case I and II test mode one – setup a connection with a non-active server.

Case I: Client and server are allocated to two different address spaces. They are located on the same node. The server is non-active.

Case II: Client and server are located on two different nodes. The server is non-active.

Case III and IV test mode two – setup a connection with an active server.

Case III: Client and server are allocated to two different address spaces. They are located on the same node. The server is active.

Case IV: Client and server are located on two different nodes. The server is active.

Case V: Client and server are allocated in same address space.

4.3.4.2 Factors

The following factors are used to control the experiments (see Table 5). A particular combination of these factors gives rise to each of the five cases studied.

- 1) Server Activity (SA): This factor is controlled through the setting of the timeout value used in the function `impl_is_ready()` called by the server. The state of the server influences the connection setup steps. If the server is not active, Orbix will need to activate it automatically during the connection setup.
- 2) Client-Server Collocation (CL): The collocation of client and server is expected to impact latency. A collocation of client and server makes the following calls on objects more efficient because of the Orbix runtime is by-passed.
- 3) Client-Server Distribution (CSD): For the non-collocation modes, client and server are in two different address spaces. But they can be allocated on the same node or on two different nodes.

Table 5: Levels for experiment factors

Factors	Levels
SA (Server Activity)	A(active)/NA(non-active)
CL (Collocation of client and server)	C(collocated) / NC (non-collocated)
CSD (Client-Server Distribution)	SN(same node) / DN(different nodes)

4.3.4.3 Performance Metrics

Connection set up latency is measured in milliseconds. The total time consumed to setup the connection is used for the comparison of five different cases. The `gettimeofday()` system call is used to measure the elapsed time in the connection setup.

4.3.5 Results of Experiment

The different combination of the factors that produce the five different cases are describes in Table 6. The connection set up latency is included in the table and is also presented graphically in Figure 4-2.

Table 6: Connection set up latency

Factors	Mode I		Mode II		Mode III
	Case I	Case II	Case III	Case IV	Case V
SA	NA	NA	A	A	A
CSD	SN	DN	SN	DN	SN
CL	NC	NC	NC	NC	C
Latency (ms)	171.8	174.4	54.6	55.2	0.9

Figure 4-2: The connection set up latency for the five cases in three different connection setup modes

Note that Mode III requires that both client and server are located in the same active process. As a result this mode gives rise to Case V only and other combinations of SA and CSD are invalid.

The results of experiments captured in Figure 4-2 lead to the following observations:

- Comparing Mode I with Mode II, we find that the latency time for Mode II to finish the connection set up is approximately 32% of that for Mode I. Also according to the scenario we introduced before, an extra step in Mode I is needed to activate the non-active server. This shows that in the complete connection setup period, nearly two thirds of the time is used to activate the non-active server.
- Comparing the latency of Mode III with Mode I and II, we find that the time used to setup a connection in Mode III – the collocation mode is approximately 1.6% of that required for a non-collocated connection to an active server, only 0.5% of that required for a connection to a non-collocated non-active server.
- Client and server distributed in different nodes take less than two percent longer time in comparison to two processes allocated in the same node.

4.3.6 Design Guidelines

Based on the observations described at the end of the previous subsection the following recommendations are made.

- Server activation time can be substantial for LAN-based systems (telecommunication system and embedded system for example). At least

the popular servers that are used frequently should be kept alive so that it is possible to save the server activation time during connection setup.

- Allocating the client and server on the same node produces only a marginal improvement in latency (1.1% to 1.5% for the results presented in Table 6) in comparison to allocating them on separate nodes connected by a LAN. With a larger inter-node delay the performance improvement is expected to increase. For a communication bound system allocating client and server on the same node is preferable. Careful considerations are required for CPU bound systems characterized by servers with a heavy CPU demand.
- When allocated on the same node, collocating the client and server in the same address space can greatly save the connection setup time. Multiple orders of magnitude improvement in performance are observed in the data presented in Table 6.

4.4 Passing Complex Data Structures

4.4.1 Motivation

Passing significant volumes of data from the client to the server impact both the marshalling/demarshalling times as well as communication delays. Both of these factors contribute to the response time that is a performance measure of interest.

In high-performance communication systems, the presentation layer is a major performance bottleneck [HOS-97]; the data conversion overhead in this layer has often been identified as the performance problem. This layer transforms typed data objects

from high-level representations to lower-level representations (marshalling) and vice versa (demarshalling), and transforms application-level data into a portable format that masks byte order, alignment, and word length differences.

There are many techniques for reducing the cost of presentation layer conversions. For instance, consider the tradeoffs between using compiled versus interpreted code for presentation layer conversions. Compiled marshaling code is efficient, but requires excessive amounts of memory. In contrast, interpreted marshaling code is slower, but more compact. Philip Hoschka[HOS-97] 's research paper points out that in today's distributed application, marshalling may require executing several thousands of CPU instructions per message transfer, and the marshalling and demarshalling codes are often inefficient because they are overly general. He investigates a "fast path " marshalling and automatic optimization stage.

In CORBA, the marshalling and demarshalling engines perform data conversions between the native format used in the client/server implementation and the CDR format defined in CORBA; client-side stubs and server-side skeletons that are automatically generated by the Interface Definition Language (IDL) compiler are responsible for performing this transformation. Figure 4-3 illustrates the marshalling and demarshalling associated with the transmission and reception of an operation or attribute request in Orbix.

The CDR Encoder converts application data types from their native host representation into the CDR representation that transmits CORBA requests and replies over the network [OMG-00]. The CDR Decoder is the inverse of the Encoder. It converts request values and reply values from the incoming CDR stream into the native host

representation. In addition to the delays introduced due to marshalling/demarshalling there is a communication delay that is directly proportional to the number of bytes transferred.

Figure 4-3: Marshalling and demarshalling in Orbix (from [BAK-97])

In this section, we will not discuss how to optimize marshalling and demarshalling by modifying the stub/skeleton code. Instead, from the software designer's point of view, we will study how to transfer the arguments composed of complex data structures and discuss what is the best way to send them over the network. Optimization of system performance by modifying marshalling and demarshalling routines is discussed in [WU-01].

4.4.2 Case Studies

In many network applications, complex information is transmitted as the request argument and then communicated to other nodes. In this section we will discuss how to pass complex data types efficiently to avoid excessive overhead during marshaling/demarshling. Two cases will be presented in this section for the purpose of illustrating the impact of different ways of constructing the argument list on response time.

4.4.2.1 Case A

In this case we will focus on two complex data types with fixed size - the structure and the array. Separately, the structure and array consist of basic data types.

The situations to be discussed in this case are as follows:

Send two arrays: The first array is composed of a basic data type - char, and the second array is composed of double as the parameter. We can use three different combinations of parameter and methods to implement this.

First option – Case A.1

Use one method with one structure as the parameter:

Method1 (parameterA).

ParameterA is an array of a structure; within this structure, there are two components – a char and a double,

ParameterA:

```
struct CD [size] {  
    char c;
```

```
double d;  
};
```

Second option – Case A.2

Use one method with two arrays as parameters:

Method2 (parameterA, parameterB):

The first parameterA is an array of char and the second one is an array of double.

Parameter A: char[size] c;

Parameter B: double[size] d;

Third option –Case A.3

Use two methods, where each carries one parameter. We assume that the operation performed by the original method can be divided into two parts. One part is achieved by Method31 and the other by Method32.

Method31 (parameterA):

Method32 (parameterB):

Here parameterA and parameterB are exactly the same as in Case A.2.

4.4.2.2 Case B

In this case, we will study a parameter that is composed of basic data types as well as complex data types. Four options will be investigated.

First option – Case B.1:

Use one method with one nested structure as the parameter.

Method1 (parameterA) ;

ParameterA is a complex structure consisting of two complex data types – an array of char, a structure and one basic data type – a boolean.

The structure of Parameter A:

```
struct CSB {  
    char[size] c;  
    struct s {  
        int x;  
        int y;  
        int z;  
    }  
    boolean b;  
};
```

Second Option – Case B.2

We still use one method, but three data types as three parameters,

Method2 (parameterA, parameterB, parameterC);

Separately, the three parameters are the array of char, the structure and the boolean.

ParameterA : char[size] c;

ParameterB: struct s {
 int x;
 int y;
 int z;
};

ParameterC: boolean b;

Third option – Case B.3:

One step further, we split the structure as three basic data types. Now we get five parameters, but they are still carried by one method invocation.

Method3 (parameterA, parameterB, parameterC, parameterD, parameterE);

The definition of five parameters can be seen below:

Parameter A: char[size] c;

Parameter B: int x;

Parameter C: int y;

Parameter D: int z;

Parameter E: boolean b;

Fourth option – Case B.4

We use five methods; each one carrying one of the parameters described in option

III.

Method41 (parameterA);

Method42 (parameterB);

Method43 (parameterC);

Method44 (parameterD);

Method45 (parameterE);

4.4.3 Workload

The size of the message transmitted from the client was varied and the mean response times obtained in the different cases are compared.

Message size (S): This is the size of the array used in the method invoked by client. It is used to investigate how the message size affects the result of the performance measurements. The array sizes used in experiment are 1, 100, 1000 and 5000 elements.

4.4.4 Results of Experiment

4.4.4.1 Case A

Table 7: Response time (μ s) achieved with the three options in Case A

Array Size	1	100	1000	5000
Response time - CaseA.1	2,131	3,472	24,556	123,703
Response time - Case A.2	2,014	2,314	8,356	41,489
Response time - Case A.3	3,747	4,298	15,490	74,168
<i>Ratio – Case A.1 (%)</i>	<i>100%</i>	<i>100%</i>	<i>100%</i>	<i>100%</i>
<i>Ratio – Case A.2/A.1 (%)</i>	<i>94%</i>	<i>67%</i>	<i>34%</i>	<i>34%</i>
<i>Ratio – Case A.3/A.1 (%)</i>	<i>176%</i>	<i>124%</i>	<i>63%</i>	<i>60%</i>

Figure 4-4: Comparison of the three options in Case A

In order to compare the performance of the three options clearly, we choose Case A.1 as the base case. The metric ratio for a particular option is computed as the ratio of the response time achieved with the given option and that achieved with Case A.1.

Figure 4-4 displays the ratios for the three options. The following observations are made from the data presented.

- Compared with two other options, the performance of Case A.2 that separates the complex type of data parameters into several simple data types always displays the best performance. Also, with the increase in message size, the improvement achieved with Case A.2 increases. For the array sizes of 1,100,1000 and 5000, the performance improvements are 6%, 33%, 66% and 66% respectively.
- While the array size is relatively less than a thousand, Case A.1 produces a better performance than Case A.3. When array size is 1, Case A.3 leads to 76% more response time than Case A.1. With an increase in message size, the relative performance of Case A.3 improves. When the array size increases to 1000, Case A.3 will save 37% time compared with the base case, and with the array size to 5000, Case A.3 achieves a saving of 40% in comparison to Case A.1.
- Invoking two methods gives rise to a large overhead in comparison to invoking a simple method. When the size of the parameter list is small, the saving in marshalling/demarshalling in Case A.3 is smaller than the increase in method invocation overhead. However when the parameter list

is sufficiently long the increase in method invocation overhead is offset by the decrease in marshalling/demarshalling times.

4.4.4.2 Case B

Table 8: Response times (ms) achieved with the four different options in Case B

Array Size	1	100	1000	5000
Response time - Case B.1	370	384	419	690
Response time - Case B.2	362	375	502	560
Response time - Case B.3	360	363	388	511
Response time - Case B.4	976	980	1011	1321
<i>Ratio - Case B.1 (%)</i>	<i>100%</i>	<i>100%</i>	<i>100%</i>	<i>100%</i>
<i>Ratio - Case B.2/B.1 (%)</i>	<i>97.8%</i>	<i>97.7%</i>	<i>95.7%</i>	<i>81.2%</i>
<i>Ratio - Case B.3/B.1 (%)</i>	<i>97.3%</i>	<i>94.5%</i>	<i>92.7%</i>	<i>74.1%</i>
<i>Ratio - Case B.4/B.1 (%)</i>	<i>263.8%</i>	<i>255.2%</i>	<i>241.3%</i>	<i>191.4%</i>

Figure 4-5: Comparison of result for the four options in Case B

Figure 4-5 displays the ratios of response times associated with the sending different sizes of complex structures through four options: Case B.1 is used as the base case for computing response time ratios. The message size used were 1, 100, 1000, and 5000 elements. The observed response times are captured in Table 8. The results show that:

- Case B.3, which splits the complex data structure into five simple data types, outperforms the other three options, and with the increase in the message size, the improvement in performance becomes higher. Compared with the base option, the performance improvement increases from 2.7% to 25.9% as the array size changing from 1 to 5000 elements.
- Case B.2, which splits one complex parameter to three simpler parameters, obtains the second best performance; the response time improvement for this option ranges from 0.5% to 7% better than base option - Case B.1.

- Case B.4 always presents the worst performance, because four extra remote calls are used. In the worst case, when the array size is 1, it produces a response time that 1.6 times than that achieved with the base case. But with an increase in message size, its relative performance is improved.

In general, the CORBA implementations performed poorly when sending complex structure data types. This is because of excessive copying and marshaling/demarshaling overhead, many layers of virtual function calls, and excessive writes resulting from non-optimal fragmentation of request buffers [SCH-98.1]. Another potential problem with complex data type is the incorporation of padding bytes. CDR requires that different data types are aligned at particular bytes boundaries (see Table 9). Thus two consecutive arguments in an argument list may be separated by padding bytes. For example, the stream of data for a struct {char c; double d;} is shown in Figure 4-6. Note that since d must start at a specific byte boundary padding bytes are inserted between c and d. The padding bytes do not have any semantic value but consume network bandwidth. From the results of the experiments, we find that splitting the complex data types to simple data types will greatly improve performance.

Table 9: CDR alignment of primitive fixed-length types (from [AHM-01])

Starting Bytes Boundary	IDL Data Types
Multiples of 1	Char, octet, boolean
Multiples of 2	Short, unsigned short
Multiples of 4	Long , unsigned long, float, enumerated types
Multiples of 8	Long long, unsigned long long, double, long double

Figure 4-6: Padding in a stream (from [AHM-01])

4.4.5 Design Guidelines

The following guidelines are useful in reducing the marshaling and demarshaling overhead as well as the communication time, when a complex data structure is to be sent. They are based on the observations described at the end of the previous subsection.

- Splitting the complex parameter into basic data type improves system performance.
- In some cases, using several methods each carrying a simple parameter to replace one method with one complex parameter achieves a performance improvement.
- A comparison between Case A.1 and Case A.3 as well as between Case B.1 and B.4 provides an interesting insight. Sometimes, when multiple methods are invoked by the same client on the same server sending all the parameters as arguments of a single method can be beneficial. The name of all the methods to be executed need to be included in the parameter list for the first method.

Note that the first design guideline requires an application developer to use a parameter passing format that may not be the most natural way for specifying the argument list. However, it may be possible to develop a tool that will take the application

code and transform it automatically into a form in which a complex data structure is split into the basic data types. With such a tool the software designer will be able to use the complex argument in its most natural form and the code that is executed will be in a form that enhances system performance.

4.5 Demultiplex Strategy in OrbixMT_2.3c

4.5.1 ORB Demultiplexing

The demultiplexing strategy used by an ORB can impact system performance. As we know, a standard GIOP-compliant CORBA request contains two parts: one is the identity of its remote object implementation, the other is the name and arguments for the remote method. The remote object implementation is represented by an object reference, and the remote method is represented by a string. ORB endsystems must demultiplex and dispatch incoming client requests to the appropriate method of the target object (see Figure 4-7). Four different demultiplexing strategies that can be used are described in [SCH-97.11].

- Linear Search
- Perfect Hashing
- Dynamic Hashing
- De-layered Active Demultiplexing [SCH-97.2]

As an example we describe the linear search strategy. Discussions of the other three strategies are available from [SCH-97.11] and [SCH-97.2].

The linear search demultiplexing strategy is a two-step layered demultiplexing strategy. First, the Object Adapter uses the object key to linearly search through the

active object map to locate the right object and its skeleton. And then the Object Adapter uses the method name to linearly search the method table of the associated skeleton to locate the appropriate method and invoke an up-call on it. These steps are based on the assumption that each object map maintains a pointer to its associated skeleton and each skeleton maintains a table of operation defined by the IDL interface.

Figure 4-7: Layered CORBA request demultiplexing (from [SCH-97.5])

4.5.2 Performance of Demultiplexing Strategy Used in OrbixMT_2.3c

As mentioned in the previous section, there are four strategies that can be used to demultiplex by ORB. Since Orbix is not an open source code, we can not find which demultiplexing strategy is used from its source code. The purpose of this experiment is to investigate the impact of the demultiplexing overhead in Orbix on performance.

The demultiplexing overhead may increase with the number of methods defined in an interface. To measure the demultiplexing overhead, we define an interface with 500 methods (see Figure 4-8), and our experiments defined a range of methods (1st, 100th, 200th, 300th, 400th and 500th) to be invoked. Since we are focusing on the overhead of demultiplexing, no parameters are passed, thereby eliminating the overhead of presentation layer conversions. The called method returned immediately after invocation.

```
IDL definition

interface int1{

    method1 () {};
    method2 () {};
    method3() {};
    .....
    method100(){};
    .....
    method500 () {};

};
```

Figure 4-8: IDL definition with 500 methods

On the client side, the same object reference is used to invoke all the methods. The response time is measured for the invocation of the different methods.

4.5.3 Results of Experiment

The results of the experiment are captured in Table 10 and Figure 4-9.

Table 10: Performance of demultiplexing strategy in OrbixMT_2.3c

Method	Response Time (μ s)	Ratio
1 st method	16,406	100%
100 th method	16,742	102%
200 th method	17,210	105%
300 th method	17,537	107%
400 th method	18,019	110%
500 th method	18,352	112%

Figure 4-9: Response times of invocation of different number of methods

For comparison purposes, we set the response time to invoke the first method as the base value, and the ratio column presents the ratio of response time achieved with a particular method invocation with the base value. The results demonstrate that:

- The response time associated with the invocation of the 500th method overhead is twelve percent higher than that for the 1st method.
- The rate of increase is approximately two percent for every 100 additional methods in the interface.
- The black box test reveals that with the increase in method numbers, the response time is increased almost linearly. Note that these results are dependent on the ORB implementation and may change with the ORB product used.

4.5.4 Design Guideline

Based on the results of the experiment that show that the response time associated with a method invocation increases with the position of the method in the interface definition, the following design recommendation can be given to the software designer using OrbixMT_2.3c:

- If there are a large number of methods defined in one interface, more time critical or the most frequently used methods should be defined as early as possible in the IDL declaration.

4.6 Packing of Methods in Objects

4.6.1 Model Study

In this section we will discuss the packing of methods in an object. Consider a system in which there are a number of methods available in the server side to be remotely invoked by the client. Two different design models can be used. One uses one object for all these methods; this means that we define one interface and this interface encapsulates

all the available methods. The other model involves the design of multiple interfaces with one method defined in every interface. In these two models any number of methods and interfaces can be defined. In our model, we assume there are 50 methods available to the client and each method is independent and does not share any data with any other method. The two different design models that are at the two ends of a spectrum are briefly described.

Model I

In the first step, during the IDL definition, we design one interface; within the same interface 50 methods are defined. The server is responsible for creating one object and then making the server ready for the remote call from client. The client obtains the object reference first through the bind operation, and then uses the handle to remotely invoke any method taken care of by this object. The definition of IDL is as follows:

Object Implementation

```
Interface {  
    Method 1();  
    Method 2();  
    .....  
    Method 50();  
}
```

Model II

In the second model, we define fifty interfaces during the IDL definition; each interface encapsulates only one method. The server creates fifty objects, each object corresponding to one interface, and then makes the server ready for the client. Before the

client remotely invokes the method, it should use the bind function to get the specific object reference that takes care of this method. The IDL definition is as follows:

```
Object 1 Implementation      Interface 1{
                                Method1();
                                }

Object 2 Implementation      Interface 2{
                                Method2();
                                }

.....

Object 50 Implementation     Interface 50{
                                Method50();
                                }
```

4.6.2 Comparison of the Two Models

The purpose of this experiment is to compare two design models from the performance point of view. The response time associated with remote invocation of an available method is measured. In total we have fifty methods. The 1st, 10th, 20th, 30th, 40th, 50th methods in two different models are invoked.

4.6.3 Results of Experiments

For the comparison of two design models, we use the response time for the first model as the base value.

The key observations captured in Table 11 and Figure 4-10 are summarized:

- The performance of Model I is always superior to that of Model II. When invoking the 1st, 10th, 20th, 30th, 40th and 50th methods, Model II always takes 2.5% to 3.7% more in terms of response time than Model I.

Table 11: Response time (μs) for two different design models

	Model I Response Time (μs)	Model II Response Time (μs)	Comparison <i>Case II/Case I (%)</i>
1 st method	16,445	17,051	103.7%
10 th method	16,496	17,065	103.4%
20 th method	16,542	17,092	103.3%
30 th method	16,573	17,096	103.2%
40 th method	16,639	17,118	102.9%
50 th method	16,689	17,124	102.6%

Figure 4-10: Response times of two different design models

- With an increase in the number of methods, the difference between these two models is reduced. (from the highest 3.7% reduce to 2.6%)
- As expected this response time seems to increase with the position of the method in the IDL declaration.

4.6.4 Design Guideline

- As shown in Figure 4-10, Model I is better than Model II from the performance point of view. This may or may not be the case with other ORB products, since the results depend on the specific demultiplexing and dispatching strategy used in particular ORBs.

4.7 Packing of Objects

Many operating systems allow a process to create lightweight threads, with each thread having its own set of CPU registers and its own stack. In particular, each thread is independently scheduled by the operating system, so that it can run in parallel with the other threads in its process. An impact and benefit of multi-threading is that the latency of requests can be reduced. Both clients and server may benefit from multi-threading. In this section we will discuss how to efficiently pack the objects in an application consisting of multiple clients and servers. The different packing strategies use different process and threading models.

4.7.1 Four Different Packing Modes

Four different modes that implement four different packing strategies will be discussed as follows.

Mode I

Shared Object Mode: In this mode, one object on a given host is managed by one process on that specific host. In a multi-client application, all the clients will bind to the same object located in the same server (see Figure 4-11).

Figure 4-11: Mode I - Shared Object Mode

Mode II

Multiple Objects Mode: In this mode, a separate object is created for each client. All these objects are located on the same host and are in the same process (see Figure 4-12).

Figure 4-12: Mode II – Multiple Objects Modes

Mode III

Multiple Processes Mode: In this mode, each object has its own server that is registered with the implementation repository. Different server processes handle invocations from different clients. All these servers are still located in one host (see Figure 4-13).

Mode IV

Multiple Hosts Mode: The motivation for this mode is that if system resources are available, it may be better to serve the client requests on multiple hosts. This can increase parallelism in execution, because each object has its own process that can occupy a specific CPU. With this mode, each object has its own server and these servers are located on different hosts (see Figure 4-14).

Figure 4-13: Mode III - Multiple Processes Mode

Figure 4-14: Mode IV - Multiple Hosts Mode

4.7.2 Performance Comparison for the Four Different Modes

The threading model adopted in this experiment is one in which a thread is created automatically for each incoming operation request. Any number of such threads can be running concurrently on a server. So on the server side, each request is dispatched on a new thread; server uses the thread filter method to create threads and dispatch incoming requests on these new threads. The code for filter is supplied by Orbix[ION-97] and it should be viewed as code that extends the ORB, rather than as application level code.

Concurrency control is an important issue for the thread safety. In particular, if clients simultaneously request the server to invoke IDL operations on the same target object within the server, then that object will be subject to concurrent access. When we write the server program, we should take care of such concurrent access by using locking code. We have used mutex lock for this purpose.

Workload

- 1) Number of Clients (NC): In a multi-clients environment, this parameter is used to control the total number of active clients during each experiment. It is important to investigate how the overall system performance is affected by the number of clients
- 2) Service Time (ST): This is the service time for the server class; specifically, it is the invocation time for the server to process the method. It only represents the actual processing time and does not include other overheads such as marshalling/demarshalling etc. The service time for a server is fixed and does not vary from invocation to invocation.

Table 12 presents the levels for the different factors we used in our experiment. In Experiment 1, we have used a method that does one simple operation, increase a counter by one. The time required to finish this method invocation is 0.03ms, so we choose this smallest level used for service time. Meanwhile, with the fixed service time we vary the number of clients to reveal how performance is impacted by the increase of number of clients. In the two following experiments, we fix the number of clients, vary the service time through 10ms to 20ms for the purpose of studying how performance is impacted with the increase in service time. The specific value of ST is achieved by executing a while loop for a predetermined number of times in the method.

Table 12: Levels for experiment factors

Experiment #	Factors	Levels
1	NC	5, 8, 10, 15
	ST (ms)	0.03
2	NC	10
	ST(ms)	10
3	NC	10
	ST (ms)	20

4.7.3 Results of Experiment

Table 13: Response times (ms) for the four different modes in Experiment 1

ST = 0.03 ms, NC =5,8,10,15

Number of Clients	# 5	# 8	# 10	# 15
Mode I	100.0	152.8	190.0	276.9
Mode II	99.9	153.2	192.8	278.0
Mode III	96.8	155.0	219.4	433.1
Mode IV	71.7	125.4	169.4	257.6

Figure 4- 15: Results of Experiment 1

In Experiment 1, the service time for the object is fixed at a specific value while the number of clients is varied. For all values of N Mode IV provides the best performance (See Figure 4-15 and Table 13). Mode I and Mode II display comparable performance whereas Mode III's performance is the worst at larger number of clients. It is no surprise that Mode IV that uses multiple CPUs produces the best performance. A possible

explanation for the inferior performance of Mode III is the context switching overhead incurred by the multiple processes. In Mode I and Mode II multiple threads in the same process avoid the extra processes switching overhead.

In order to confirm our observation we have compared the performance of the four modes at NC=10 with ST=10 ms (Experiment 2) and ST=20ms (Experiment 3). A similar behavior is observed in each of these experiments (see Figure 4-16, Figure 4-17, as well as Table 14 and Table 15)

Table 14: Performance of the four modes in Experiment 2

ST =10 ms, NC =10

Number of Clients =10	Mode I	Mode II	Mode III	Mode IV
Response Time (ms)	291.5	291.8	319.8	220.6
Comparison (%)	--	+0.1%	+ 9.71 %	- 23.42 %

Figure 4-16: Results of Experiment 2

In Experiments 2 and Experiment 3, we use the response time of Mode I as the base value. The row called “comparison” in Table 14 and Table 15 presents the percentage increase or decrease in response time of the three other modes in comparison to Mode I.

A positive value means that this mode produces a higher response time in comparison to Mode I. Whereas, a negative value indicates this mode needs produces a lower response time than Mode I.

Table 15: Performance of the four modes in Experiment 3

ST =20 ms, NC =10

Number of Clients =10	Mode I	Mode II	Mode III	Mode IV
Response Time (ms)	371.8	367.0	391.8	256.0
Comparison (%)	--	- 1.29 %	+ 5.38 %	- 31.15 %

Figure 4-17: Results of Experiment 3

Summary of Observations

The performance of Mode IV is always superior to the three other modes because more CPU resources are used.

The following discussion on Modes I, Mode II and Mode III, is focused on how to efficiently pack the objects on the server side when the server runs on a single node. All

these objects are located in the same host, with one CPU available. The insights resulting from the experiments are summarized:

- Mode I and Mode II produce no significant difference in performance. Irrespective of service time changes from 0.03 ms to 10 ms and 20 ms, and the number of client changes from 5, 8, 10 to 15, the performance differences between these two modes lie within 1.5%. This implies that a single object is apt for handling multiple clients in a multi-threaded environment and using multiple objects does not give rise to any significant improvement in performance.
- Mode III always exhibits the worst performance. When service time is 0.03 ms, with the increase in the number of clients, performance of Mode III becomes increasingly worse in comparison to that of Mode I and Mode II. That is, when the number of clients is 5, the response time of Mode III is 1.0% higher than that of Mode I, but when number of clients increases to 10 and 15, the response time achieved with Mode III becomes 15% and 56% longer than that for Mode I respectively.

When we fixed the number of clients to 10 and increased the service time the relative performance of Mode III improved, although it is still the worst case. For example, when service time is 0.03 ms, Mode III produces a 15.5% higher response time in comparison to Mode I. However when service time increases to 10ms and 20ms, the response time for Mode III are 9.7% and 5.4% higher than that for Mode I.

- The inferior performance of Mode III indicates that even if different objects are invoked by the different clients packing the objects in the same server process may give rise to a significant improvement in performance.

4.7.4 Design Guideline

The summary of observations presented in the previous subsection leads to the following recommendation.

In order to get a better performance with an application with multiple clients and multiple server objects running on the same host, using one or multiple objects located in same process to handle all the requests from the clients is recommended. Locating objects in different processes on one host may degrade performance. As expected allocating each server process on a different CPU can give rise to a significant improvement in performance on a multiple CPU environment.

Chapter 5 Guidelines for Load Balancing

5.1 Introduction

Load balancing is a popular and cost effective technique for improving network server performance; it distributes client workload equitably among multiple replicated back-end servers to improve overall system responsiveness. With load balancing, hardware or software mechanisms determine which server will execute each client request. It helps to improve system scalability by ensuring that client application requests are distributed and processed equitably across a group of servers. It also helps to ensure that all resources are used efficiently [TAN-95].

In a distributed system, the mechanism of load balancing can be implemented in any or all of the following four layers: Network-based load balancing, OS-based load balancing, middleware-based load balancing and application-based load balancing. The first three load balancing categories and the advantages of middleware-based load balancing are discussed in [SCH-01]. The network-based load balancing mechanism is provided by IP routers and DNS (Domain Name Servers). Based on current load conditions, the DNS can assign a different IP address to each request dynamically. OS-based load balancing mechanism is provided through load sharing and process migration mechanism. A process from a highly loaded node is migrated to a lightly loaded node for executed. However, middleware-based load balancing is performed in middleware that tries to distribute requests in such a way that load is spread uniformly on replicated servers. In this chapter we will focus on application-based load balancing mechanism.

5.2 Load Balancing Techniques

Clients in a distributed system have the opportunity to choose the servers that service their requests. When multiple options are available in a system, it is in the best interest of the client application to choose the particular server that can most quickly service the request. A client application attempts to choose the service application with the least load. The question is, how does a client know how loaded a server is? There are a variety of techniques for making this decision. In general, we can classify policies of load balancing into two categories: non-adaptive and adaptive policies [SCH-01]. The typical non-adaptive policy includes simple round robin algorithm or a randomization algorithm. Randomization algorithm entails maintaining a list of all available server instances and then randomly choosing one server for serving a client request. The adaptive policies will utilize runtime information, such as the amount of idle CPU available on each back-end server, to select the replica that will handle a particular request. In this thesis, we will concentrate on the non-adaptive round robin policy. Adaptive policies are more complex and can incur additional overheads in obtaining information on system state. Whether or not the performance improvement produced offsets the overhead for the adaptive policies forms an important direction for future research.

Round robin entails maintaining a list of all available server instances and then sequentially using each server for subsequent client request. When the last option is used, the next client request is routed to the server at the beginning of the list. For example, with a server list (1...K...N), if the current request is routed to server K, the following request is handled by server K+1. After a request is routed to server N we go back to the

top of the list and start with server 1 again. This is not very sophisticated, but is simple to implement and can provide significant performance increases in some situations.

5.3 Strategies for Application-layer Load Balancing for CORBA-Based Systems

We have devised a number of load balancing strategies at the application level. Before discussing the strategies, we will introduce some terms that will be used in our multi-client multi-server environment. A client runs cyclically sending a request to a server in each cycle. Two kinds of clients are introduced. One is the busy client, which is continuously sending requests to the server without any think time between two consecutive remote invocations. A solid square is used to depict it. The other is the non-busy client, which allows some think time between two successive invocations. A hollow square is used to represent it. The interactions between a busy client and server are shown by solid lines, however, interactions from the non-busy client to a server are shown by dotted lines. Hollow circles are used to illustrate available servers. The idea of symmetric environment is used in the following discussion. A symmetric environment means that the number of busy clients and non-busy clients are the same. Two or multiples of two servers are ready to provide service to the clients. For the purpose of simulating a relatively simple and complex system, two sets of experiments are implemented in this chapter. In our first set of experiments four clients (two busy and two non-busy) and two servers are used to discuss the load balancing strategies. In order to investigate the load balancing strategies in the context of more complex systems, we have performed another experiment that uses double the number of clients and servers used in the first set. That is,

eight clients and four servers are used. We have compared five application layer load balancing strategies are discussed in the following subsection.

In the first three strategies, we discuss what needs to be done when we have a priori knowledge of which clients are busy and which are not busy. In contrast, the last two strategies may also be used when the designer has no idea whether a specific client belongs to a busy category or a non-busy one.

5.3.1 Non-balanced Strategy

For the consideration of clearly comparing different strategies, we design this case to establish a performance baseline, which is completely without load balancing. We make all the busy clients bind to the same server and all the non-busy clients bind to the other server (see Figure 5-1). In this strategy, the CORBA bind function is used to obtain the object reference; the client is bound to a replica once during the initial request binding.

In this case, the server that provides service to the busy client is heavily loaded whereas the other server that provides service only to the non-busy clients is lightly loaded. The result of this strategy is that the busy server is always keep busy, consequently the waiting time of the busy client will be increased. But on the other hand, most time of the non-busy server is always in the idle situation that means the waste of the resource. Consequently, the performance of the whole system will be poor.

Figure 5-1: Strategy 1 - Non-balanced strategy

5.3.2 Static Symmetric Balancing Strategy

Although round robin is not the most sophisticated strategy, it is beneficial in some situations. More intelligent systems will have a mechanism for judging the load of a particular set of services, and then will choose the one that is least busy. The class of the clients (busy or non-busy) discussed in this strategy is known a priori; we can balance the load in advance according to the prior known class of clients. This strategy is called static symmetric balancing (see Figure 5-2). This means, we symmetrically distribute half of the busy clients to one server, the other half to the other server. The same operation is repeated for the non-busy clients. The result is that the burden of each server is fairly distributed. As in the case of the first strategy, all clients just bind once to the server during the initial request by using the CORBA binding function.

In comparison to Strategy 1, this strategy is expected to improve the overall performance because it fairly balances the load. This benchmark strategy provides a baseline for the best performance.

Figure 5-2: Strategy 2 - Static symmetric balancing strategy

5.3.3 Half Dynamic Self Round Robin Strategy

Within the round robin approach, there is a choice of policies to implement in the multi-client multi-server environment. The three strategies described in the next two subsections will use the round robin technique to balance the load. The primary difference between these three strategies and the previous two is that the client choose a replica server each time a request is invoked, rather than just once during the initial request binding.

The first of the three round robin strategies described in this subsection is called half dynamic self round robin strategy (see Figure 5-3). Half means only the busy clients use a round robin strategy, but the non-busy clients still use static allocation to servers

performed in advance. The word self is used because each busy client undertakes server selection by itself. As Figure 5-3 shows, all busy clients firstly get all server's handle at the beginning and then locally use the round robin strategy to choose the server to use before each method invocation; the non-busy clients are bound to different servers in advance. This means it uses the CORBA bind operation once before the initial request.

Figure 5-3: Strategy 3 – Half Dynamic self round robin strategy

5.3.4 Dynamic Self Round Robin Strategy

One of the alternatives of the round robin technique used in a multi-client multi-server environment is the strategy we discussed in this subsection, strategy four which is called dynamic self round robin (see Figure 5-4). This strategy can be used in the situation in which we can not predict the load of each client. It means we can not identify that whether the client belongs to the category of busy one or non-busy one in advance. Under this circumstance, each client, no matter busy or non-busy, will use the round robin itself during each invocation to choose the server, and finally fulfil the load balance. Initially, all clients get all available server's object reference by CORBA binding

function. Then, during each invocation, the client selects the server by using a local round robin technique.

With this strategy, from each client's point of view, the load is fairly balanced, but from the whole system's point of view, it may not. The waiting time for the busy client may be longer than that of strategy two and three.

Figure 5-4: Strategy 4 – Dynamic self round robin strategy

5.3.5 Dynamic Central Round Robin Strategy

The other alternative – Strategy five, dynamic central round robin strategy (see Figure 5-5), is an adaptive request architecture for load balancing. This strategy introduces a dispatcher server that is used to receive all requests from clients and give back a server handle to the client. In this case, the dispatcher server plays the role of the load balancer. Its responsibility is to select an appropriate server replica in accordance with its load balancing policy and return the server handle to the client. This means each client should go to this particular server first to obtain the handle before each invocation

and then use this handle to invoke the assigned server. We have experimented with a round robin dispatching strategy in the thesis.

This strategy can help to avoid the excessive competition for one server, but on the other hand, it has the potential of degrading performance due to the extra remote invocation overhead required to obtain the handle.

Figure 5-5: Strategy 5 – Dynamic central round robin strategy

5.4 The Implementation of the Five Strategies

5.4.1 Hardware/Software Platform

We ran experiments using nine Sun Sparc-2 workstations connected by a 10Mbps Ethernet LAN. All the nodes are in a “quiet network” environment described in Section 4.2. The middleware running on the nodes is “OrbixMT_2.3c”. Sun C++ is used as the application programming language. Two sets of experiment are described in this chapter. The first experiment set uses four clients, two servers as well as a dispatcher server. The

second set of experiments investigates a more complex environment characterized by eight clients and four servers and a dispatcher server. Figure 5-6 describes the layout of the more complex system used in the second set of experiments.

Among the nine workstations illustrated in Figure 5-6, four are used for running eight client threads (two per workstation), one workstation is used by the dispatcher server, and the remaining four workstations are used for running four server processes.

5.4.2 The Load Balancing System

The system consists of a number of nodes as the main components. These nodes are interconnected via a 10Mbps Ethernet, and are described next.

The Client Nodes: Each node is a Sun Sparc-2 running under the Solaris 2.6 operating system. Each client process in one node runs two threads in order to simulate two clients; respectively the two clients: a busy client and a non-busy client. Each client runs cyclically and generates a request in each cycle. Whenever the busy client finishes one cycle, it immediately starts another. However, the non-busy client exhibits a think time between two consecutive cycles.

- **The Server Nodes:** This system has a total of four replicated server nodes. Each node is a Sun Sparc-2 workstation. Each server registers itself as an Orbix server and makes it available to other Orbix clients; it contains a single method that can be called by any client.
- **The Dispatcher Node:** One Sun Sparc-2 workstation is designated as the dispatcher. This node is used in one of the load balancing strategies. The process

Figure 5-6: Network environment for the more complex system

running on the nodes receives any client request, and returns the handle of the server selected by using a load balancing algorithm to the client.

- The implementations of the five strategies described in the previous section are explained in the context of the system with four clients and two servers. A similar technique is used in case of the more complex system.

With all the strategies each server registers itself as an Orbix server, and makes it available to the Orbix clients. When a request arrives, the server process executes a while loop including a statement that is the increasing of one; the execution time for this loop operation is set to a predetermined service time.

Strategy 1: Non-balanced

This strategy corresponds to the worst possible scenario in which the two busy clients are bound to server A and the non-busy clients to server B.

Strategy 2: Static symmetric balancing

In this strategy, one busy client and one non-busy client are bound to server A, and the other busy and non-busy client are bound to server B. This strategy performs static binding as in the case of the previous strategy but it tries to spread the load evenly between the two servers.

Strategy 3: Half dynamic self round robin strategy

One non-busy client is statically bound to server A and the other to server B. Each busy client runs a round robin strategy locally to select the server: server A is selected in the first cycle and server B is selected in the next cycle and so on.

Strategy 4: Dynamic self round robin strategy.

In this strategy, the busy or non-busy clients use all the available servers. Every client uses a local round robin strategy to select the server: server A is followed by server B that is followed by server A and so on.

Strategy 5: Dynamic central round robin strategy.

This strategy uses a centralized round robin strategy as opposed to the local round robin strategy used at the client. A bind request from each client is sent to a centralized dispatcher node that selects the server using a round robin strategy and returns the handle to the client. The client uses this handle to invoke the method at the server.

5.5 Instrumentation and Performance Measurement

5.5.1 Workload

The following workload parameters are expected to have a significant impact on performance. Note that two of the parameters, NC and ST were defined in Section 4.7.2. For the sake of completeness they are included in this description.

- 1) Number of Clients (NC): This is the total number of clients used in an experiment. It is important to investigate how the system performance is affected by the increased number of clients. In the first set of experiments there are totally four clients: two busy and two non-busy. However, in the second set of experiments that is a more complex system, there are totally eight clients with four busy ones and four non-busy ones.

- 2) Number of Servers (NS): This is the total number of replicated servers that are able to provide service to the clients during an experiment. It is also important to investigate how the system performance is affected by the increased number of servers.
- 3) Request Service Times (ST): This is the request service time for the server. It only represents the actual processing time in server side, and does not include other overheads such as marshalling/demmarshalling and traffic time. It is desirable to know whether loading the servers has an effect on the performance of the load balancing strategy. This parameter can be either a constant or a random variable with an exponential distribution with a given mean value.
- 4) Client Think Time (TT): This is the sleep time between two consecutive requests for a non-busy client. The ratio of client cycle time and think time can be used to describe the characteristic of the non-busy client. By increasing the ratio the client can be made busier. The think time is either a constant or a random variable with an exponential distribution and a given mean value.

The main performance metric of interest is system throughput that is defined next.

System Throughput: This is the sum of the throughputs for all the clients. The client throughput is the number of client cycles completed per second. The performance of the load balancing strategies is compared by using the system throughput which is defined next. The client cycle time for a busy client includes only the response time for the server request whereas an additional think time is included for a non-busy client.

The different levels of the factors used in the experiments are listed in Table 16.

Table 16: Levels for experiment factors

Factors	Levels
NC	4,8
NS	2, 4
ST (ms)	10,20,40,60,80,120
TT (ms)	500

5.6 Results of Experiments

The five load balancing strategies described in Section 5.3 were implemented. Experiments were run to study the effect of the workload factors on the performance of the load balancing strategies. A factor at a time approach was adopted for each experiment. One of the factors was varied while the others were held constant. The results of each experiment is presented as a set of graphs and tables that capture the throughput of the busy client, the non-busy client and the overall system throughput for different load balancing strategies. The impact of NC, NS and ST on the relative performance of the five strategies is described in the next sections.

5.6.1 Simple System with Four Clients and Two Servers

In this set of experiments the number of clients is fixed as four, which includes two busy clients and two non-busy clients respectively; the number of servers is fixed at two. One more extra dispatcher server is used in the fifth strategy. The purpose of this set of experiments is to investigate the impact of the increase of service time (ranged from

10ms, 20ms, 40ms, 60ms, 80ms to 120ms), on the performance of the five different load balancing strategies.

Figure 5-7 and Figure 5-8 illustrate the throughput of the whole system. The throughput of Strategy 1 is used as the base value for each set of experiments, and the comparison of the other four strategies with the base strategy reveals either an improvement (positive value) or a degradation (negative value) of system performance for the specific strategy. Note that performance of the system with $ST=10ms$ and $ST=20ms$ are presented in this chapter. The results for higher values of ST that display a similar trend are presented in Appendix A.

The results discussed in this chapter correspond to a fixed value of service time and think time. The results of experiments with an exponential distribution of think time and service time are presented in the Appendix B. The relative performance of the load balancing strategies for an exponentially distributed service time are similar to those achieved with fixed service times.

The main observations from the figures are summarized:

- Strategy 2 always gives the best performance; when service time is 10ms the improvement is nearly 50% (see Figure 5-9). However, with the increase in service time, the improvement is reduced, and we can find when service time is increased to 120ms the improvement is only 3.3% (see Figure 5-9).
- Strategy 3 and Strategy 4 display the second best performance when the service time is less than 80ms. Strategy 4 is slightly worse than Strategy 3, but the difference between these two is no larger than 2%. With the increase in service

**Figure 5-7: Performance of load balancing strategy with
ST=10ms, TT=500ms, NC=4, NS=2**

**Figure 5-8: Performance of load balancing strategies with
ST = 20ms, TT=500ms, NC=4, NS=2**

Figure 5-9: Comparison of five strategies for various service times

ST=10,20,40,60,80,120 ms, TT=500ms, NC=4, NS=2

time to more than 80ms, the performance of these strategies becomes slightly worse than the base strategy – Strategy 1.

- Though the central dispatcher is used to balance the load, Strategy 5 did not demonstrate good performance. This may be because of the extra remote invocation required to get the handle from the dispatcher. But one interesting point is that with the increase in service time, the situation is improved (see Figure 5-9). This provides the motivation for us to study this strategy in a more complex system.
- The load balancing strategies seem to differ significantly in performance for a range of low to medium values of ST. For very high values of ST both the servers are busy most of the time and the choice of load balancing strategy has a lesser impact on performance.

5.6.2 More Complex System

For the purpose of studying a more complex system, we increase the number of clients to eight, and the number of servers to four. This set of experiments is designed to investigate the relative performance of different strategies in an environment with more clients and servers. Figure 5-10 and Figure 5-11 illustrate the results of the experiments with the constant service time ($ST=10ms$ and $ST=20ms$) and think time. Results achieved with higher service times display a similar trend and are available in Appendix C. The results achieved with exponentially distributed think time and service time can be found in the Appendix D. The relative performance of the policies with an exponential distribution is similar to those achieved with a constant service time.

The observations made from the results presented in Figure 5-10, Figure 5-11 and Figure 5-12:

- As in the case of the simple system, the throughput of Strategy 1, which gives rise to a totally non-balanced system, is always the worst case.
- All the figures in this set of experiments illustrate that Strategy 2 is still the leading strategy that statically balances the load.
- Strategy 3 and Strategy 4 still obtain a better performance than Strategy 1. With the increase in service time, the performance improvement with respect to Strategy 1 is reduced (see Figure 5-12).
- The last, and also the most interesting, finding is that with the increase in the number of servers and clients, Strategy 5, which uses a central dispatcher demonstrates an improvement in system throughput in comparison to Strategy 1. The rationale behind such a behavior is explained briefly. The centralized strategy introduces the overhead of obtaining an object handle from the dispatcher. However, this strategy potentially distributes the load more uniformly across the server instances. For example, in one of the local self round robin strategies it is still possible for all the clients to select the same server instance at a given point in time resulting in queuing at one server which the other server(s) is (are) idle. In case of the centralized system, the dispatcher sends each consecutive request for a server to a different instance leading to a better load distribution. The additional overhead seems to offset the performance improvement due to better load balancing on the simple system. On the more complex system that can result in more congestion, benefit of using a better load distribution strategy exceeds the

additional overhead and Strategy 5 performs better than in case of the simple system.

As the results of our experiments show, application level load balancing strategies can increase the overall system throughput. In particular, Strategy 2 that statically balances load can greatly improve system throughput. Strategy 2 however needs accurate a priori knowledge of service demands of clients. Moreover client behavior is also expected to remain consistent with one class (busy and non-busy). In the absence of such a priori knowledge of system behavior on systems in which a client changes its behaviors from busy to non-busy vice versa, the dynamic strategies may be more useful. Also, the round robin technology, adopted by each client will lead to a better performance in comparison to the unbalanced strategy in most cases. The centralized dispatcher introduces an extra method invocation for each client to obtain the handle. For simpler systems with fewer numbers of clients and servers, the performance is poor and a local round robin strategy at each client may be preferable (see Figure 5-9). But, with an increase in the number of clients and servers, the additional overhead is offset by the performance improvement due to load balancing and the centralized global round robin strategy demonstrates its advantage by improving system performance.

Note that the conclusions derived are based on a limited set of experiments described in this chapter. A large set of experiments using various other workload needs to be carried out to develop a general set of conclusions. For example, scenarios in which busy clients have non-zero think times and servers are characterized by a high variability in service times need investigation. With a high service time variability and a non-zero think time and even with Strategy 2, it is possible for multiple large requests to contend for a

**Figure 5-10: Performance of load balancing strategies with
ST=10ms, TT=500ms, NC=8, NS=4**

**Figure 5-11: Performance of load balancing strategies with
ST=20ms, TT=500ms, NC=8, NS=4**

Figure 5-12: Comparison of five strategies for various service times

ST=10,20,40,60,80,120ms, TT=500ms, NC = 8, NS = 4

server while another server instance is idle. Experimentation with such workloads can produce interesting insight.

Chapter 6 Conclusions

6.1 Summary

This thesis focuses on studying design and implementation guidelines for enhancing the performance of CORBA-based distributed applications. In order to identify the effective strategies and the relative performance of different designs, various implementations and strategies were investigated under different workload parameters and environment conditions. The design of the experiments and the interpretation of the results are illustrated in Chapter 4 and Chapter 5. Chapter 3 summarizes the comparison of different ORB products together with some benchmark results reported in the literature.

6.2 Guidelines for Application Design and Implementation

Based on observations from a number of experiments performed under different workloads and system environments, a number of insights that are useful during application design and implementations for enhancing system performance are developed. A brief summary of these insights is presented.

6.2.1 Connection Setup

In Section 4.2, we discussed five different models in the context of setting up the connection between a client and its server. The different models used various combinations of server states (active/non-active), whether or not the server and client are allocated on the same node and whether or not client and server are colocated in the same address space. The different models were observed to give rise to different

connection setup latency. Based on this observation the following suggestions are made. Keeping a server active is observed to lead to a substantial saving in connection set up time. Each active server however consumes system resource. So if it is not possible to keep all the servers active the more frequently called servers should be kept active all the time. If possible a client and server should be collocated. An order of magnitude improvement in connection set up latency is observed when the client and server are collocated in the same address space. This result is counter-intuitive because normally the functionally different client and server are compiled into separate processes.

6.2.2 Passing Complex Data Structures

A detailed discussion of how to efficiently pass parameters consisting of complex data structures in a CORBA environment is presented in Section 4.3. Two cases are studied. In the first case an array of structures consisting two basic data types is investigated. In the second case, a structure composed of basic data types as well as a nested structure is considered. Different ways to pass these parameters to the server are discussed for each of these cases. Passing the data structure in its natural form does not produce the best performance. When we split the data structure into its component basic data types the highest performance is achieved. The performance improvement is primarily due to the saving in marshaling/demarshaling and communication bandwidth that accrues from the elimination of padding bytes in the data transmitted between the client and server. In some cases, using multiple methods, each using a single parameter instead of one method with one complex parameter is observed to give rise to a performance improvement.

Our experiments are conducted on a system using a 10Mbps Ethernet. On a faster network if the communication delay becomes very small compared to the server execution times, the usefulness of this technique may be reduced. However, on a communication bound system characterized by small server execution times and large data transfers we can expect a significant performance improvement from using this technique as is achieved on the slower network.

6.2.3 Demultiplexing latency with OrbixMT_2.3c

From the experiments discussed in Section 4.4 it is interesting to note that the demultiplexing latency is a function of the position of the operation in the interface definition. It is observed that with every 100 additional methods defined in the interface, the rate of the increase of the invocation time is approximately two percent for the ORB product used in this investigation. Thus, if there are a large number of methods defined in one interface, it is better to define the most frequently used method as early as possible in the IDL declaration. Note that this observation may not hold in all situations and may be dependent on the middleware product used.

6.2.4 Packing of Methods

In Section 4.5, we discuss two models. Model 1 uses one interface definition with multiple methods. Model 2 use multiple interface one per method. The results of experiments show that the performance of Model 1 is always superior to Model 2. Thus whenever possible, using one object to encapsulate all the methods is recommended. This

may be counter-intuitive for certain applications with a large number of methods of different types.

6.2.5 Packing of Objects

How to efficiently package objects in a multi-client environment is described in Section 4.6. Four modes are discussed. Mode I uses one object managed by one process on a specific host to handle all the requests from multi-clients. Mode II creates a separate object for each client, with all the objects located on the same host and in the same server process. In Mode III, different server processes (one per object) are created to handle invocations from different clients. The processes are on the same host. Mode IV distributes the server process in several hosts, which uses of more resources than used in the other three modes, and therefore achieves the best performance. The performance of Mode I and Mode II are similar. However Mode III always shows the worst performance due to more context switching overhead. Based on these observations the following recommendations are made to the system developer.

- As expected, if more hosts are available, making full use of them can lead to a higher performance.
- On a single host at the server side, using one or multiple objects located in the same server process for handling all the requests from multiple clients is expected to produce higher performance than a system which uses a separate process for every object.

6.3 Load Balancing Strategies

The effect of five load balancing strategies on performance is presented in Chapter 5. Our results reveal that the static symmetric load balancing policy always achieves the best performance. Exact a priori knowledge of client behavior is required however for the implementation of the policy. Moreover, such a strategy is useful only when the clients always remain in one class busy or non-busy. In other situations strategies such as the other three are required. In a relatively simple environment with four clients and two servers, the dynamic central round robin strategy produces an unexpected result: it demonstrates the worst performance even in comparison to a non-balanced system. This is because of the extra remote invocation required to obtain the server handle in this strategy. On the other hand the half dynamic self round robin strategy and dynamic self round robin strategies exhibit the second best performance. However, in a more complex environment, for example with eight clients and four servers, the dynamic central round robin strategy produces a system performance that is better than the two self round robin strategies. As pointed out in Section 5.6.2, the insights regarding the performance of the load balancing strategies are based on a limited set of experiments. More experimentation with various other workload types is required to develop a more general set of conclusions.

6.4 Future Research

The research described in this thesis leads to some future research or improvements that are presented here:

- Access control is an important issue in a CORBA network application; investigation is needed for techniques that reduce connection setup latency for Secure Socket Layer (SSL) connections (including verifying certificates).
- One interesting issue is to investigate whether the changing of the buffer size used by the CORBA middleware can improve response time for long messages.
- More design issues, such as the performance impact of the allocation of the CORBA Naming server and Trading server on nodes warrant investigation.
- We have implemented five different load balancing strategies in the application level. Another alternative is to implement them at a lower level through changing the ORB source code. Performance improvement through such lower level optimization techniques needs further investigation.
- Only non-adaptive policies are investigated in this thesis. As mentioned in Section 5.2 research on adaptive policies forms an interesting direction for future work.
- All the servers used in our experiment ran on the same platform. Extending the load balancing strategies to handle different types of servers, running on CPUs of different speeds for example, is worth of further research.

References

- [ABD-02] Abdul-Fatah and Shikharesh Majumdar, "Performance Comparison of Architectures for Client-Server Interactions in CORBA", *IEEE Trans on Parallel and Distributed Systems*, 2002, pp.111-127.
- [ABD-97] Istabrak N. Abdul-Fatah, "Performance of CORBA-Based Client-Server Architectures", M.Eng. Thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, May 1997.
- [AHM-01] Imran Admad, Shikharesh Majumdar, "Achieving High Performance in CORBA-Based Systems with Limited Heterogeneity", *Proc. IEEE International Symposium on Real Time Object Oriented Computing*, Magdeburg, Germany, April 2001, pp. 350-359.
- [BAK-97] Sean Baker, *CORBA Distributed Objects*, Addison Wesley, 1997.
- [BAR-99] T. Barth, G.Flender, B. Freisleben and F. Thilo, "Load Distribution in a CORBA Environment", *Proc. International Symposium on Distributed Objects and Applications (DOA '99)*, Edinburgh, UK, September 1999.
Available from: <http://dlib.computer.org/conferen/doa/0182/pdf/-01820158.pdf>
- [CHA-98] Distributed Systems Research Group, "CORBA Comparison Project", Charles University, Prague, Czech Republic, June 1998.
Available from: http://nenya.ms.mff.cuni.cz/publications/CORBA-report_0698.pdf

- [CHA-99] Distributed Systems Research Group, "CORBA Comparison Project, Project Extension Final Report", Charles University, Prague, Czech Republic, August 1999.
- Available from: http://nenya.ms.mff.cuni.cz/publications/CORBA-report_0899.pdf
- [HAG-98] 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.
- [HOS-97] Philipp Hoschka, "The Marshalling Problem", Ph.D. Thesis, University of Nice-Sophia Antipolis, France, 1997.
- Available from: <http://www-sop.inria.fr/personnel/hoschka>,
- [ION-97] IONA Technologies PLC, *Orbix Programmer's Guide, Orbix 2.3 – Distributed Object Technology*, October 1997.
- [ION-97A] IONA Technologies PLC, *Orbix Administration*, October 1997.
- [LI-00] B. Li and K. Nahrstedt, "QualProbes: Middleware QoS Profiling Services for Configuring Adaptive Applications", *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000)*, Pallisades, New York, April 2000, pp. 256-272.
- [MAF-97] Mafeis, D.C. Schmidt, "Constructing Reliable Distributed Communication Systems with CORBA", *IEEE Communications Magazine*, Vol. 35, No. 2, February 1997, pp. 72-78.

- [MAJ-00] S. Majumdar, W.K. Wu, "CORBA Middleware: a Performance Perspective" in *Contract Report for Department of National Defense*, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, October 2000.
- [MIS-00] S. Mishra and X. Liu, "Design, Implementation and Performance Evaluation of a High Performance CORBA Group Membership Protocol", *Proc. 7th International Conference on High Performance Computing*, Bangalore, India, December 2000, pp. 121-130.
- [MOW-97] Mowbray, Thomas J, *Inside CORBA – Distributed Object Standards and Applications*, Addison-Wesley, 1997.
- [OMG-00] Object Management Group, *Common Object Request Broker Architecture (CORBA) and Specification*, v2.4, October 2000.
Available from: <http://www.omg.org>
- [OMG-95] Object Management Group, *Common Object Request Broker Architecture (CORBA) and Specification*, Chapter 13, v2.0, July 1995.
Available from: <http://www.infosys.tuwien.ac.at/Research/Corba/OMG/-dceciop.htm>
- [PAV-99] Juan Pavon, "Building Telecommunications Management Applications with CORBA", *IEEE Communications Surveys*, Second Quarter 1999, pp. 1-15.
- [PUD-01] Arno Puder, "CORBA Product Profiles", August 2001.
Available from: <http://www.ap-c.org/corba/matrix/>

- [SCH-96.2] Douglas C. Schmidt, "Design Principles and Optimizations for High Performance ORBs", 1996.
- Available from: <http://www.cs.wustl.edu/~gokhale/Papers/oopsla97.ps.gz>
- [SCH-96.8] Douglas C. Schmidt, Aniruddha Gokhale, "Measuring the Performance of Communication Middleware on High-Speed Networks", *Proc. ACM SIGCOMM Conference*, Vol. 26, No. 4, California, USA, August 1996, pp. 306-317.
- [SCH-96.11] Douglas C. Schmidt, Aniruddha Gokhale, "The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks", *Proc. IEEE GLOBECOM '96 Conference*, Vol. 1, London, England, November 1996, pp. 50-56.
- [SCH-97.2] Douglas C. Schmidt, "A High-Performance Endsystem Architecture for Real-time CORBA", *IEEE Communications Magazine*, Vol. 14, No. 2, February 1997.
- Available from: <http://www.cs.wustl.edu/~schmidt/PDF/ORB-endsystem.pdf>
- [SCH-97.5] Douglas C. Schmidt and Aniruddha Gokhale, "Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks", *Proc. IEEE 17th International Conference on Distributed Systems (ICDCS'97)*, Baltimore, USA, May 1997.
- Available from: <http://www.cs.wustl.edu/~schmidt/PDF/ICDCS-97.pdf>
- [SCH-97.11] Douglas C. Schmidt and Aniruddha Gokhale, "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA", *Proc.*

IEEE GLOBECOM'97 Conference, Vol. 3, November 1997, pp. 1729-1734.

- [SCH-98.1] Douglas C. Schmidt, "Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance", *Proc. HICSS Conference '98*, Hawaii, USA, January 1998.

Available from: <http://www.cs.wustl.edu/~schmidt/HICS97.ps.gz>

- [SCH-98.10] Douglas C. Schmidt, "Evaluating architectures for Multithreaded Object Request Brokers", *Communications of the ACM*, Vol. 41, No. 10, October 1998, pp. 54-61.

- [SCH-01] Douglas C. Schmidt, Ossama Othman, Carlos O'Ryan, "Strategies for CORBA Middleware-Based Load Balancing", *Distributed Systems Online*, Vol. 2, No. 3, March 2001.

Available from: <http://dsonline.computer.org/0103/features/oth0103.htm>.

- [SHE-00] E-Kai Shen, "Performance of Adaptive Middleware for CORBA-Based Systems", M.C.S Thesis, Department of Computer Science, Carleton University, Ottawa, Canada, 2000.

- [SIE-98] Jon Siegel, "OMG Overview CORBA and the OMA in Enterprise Computing", *Communications of the ACM*, Vol. 41, No. 10, October 1998, pp. 37-43.

- [TAN-95] Andrew S. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, 1995.

- [TAN-96] Andrew S. Tanenbaum, *Computer Networks*, Third Edition, Prentice Hall Inc., 1996.

- [TAO-02] Weili Tao, Shikharesh Majumdar, "Application Level Performance Optimization for CORBA-Based Systems", *Proc. 3rd International Workshop on Software and Performance (WOSP 2002)*, Rome, Italy, July 2002 (accepted for publication).
- [VIN-98] Steve Vinoski, "New Features for CORBA 3.0", *Communications of the ACM*, Vol. 41, No. 10, October 1998, pp. 44-52.
- [VIR-99] Virginie University, "Benchmarks Results", 1997.
Available from: <http://www.beust.com/virginie/Benchmarks>
- [WU-01] Wai Keung Wu "Flyover –A Technique for Achieving High Performance CORBA-Based Systems ", M.A.Sc Thesis, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, August 2001.

***Appendix A: Supplementary Results for Higher Values of
Service Times in a Simple System***

Throughput	Strategy 1	Strategy 2	Strategy 3	Strategy 4	Strategy 5
Busy client	9.35	12.99	10.00	9.86	9.26
Non-Busy client	1.75	1.71	1.65	1.61	1.59
System	22.19	29.39	23.30	22.95	21.71
Improvement (%)	-	32.44%	5.03%	3.44%	-2.16%

**Figure A-1: Performance of load balancing strategies with
ST=40ms, TT=500ms, NC=4, NS=2**

Throughput	Strategy 1	Strategy 2	Strategy 3	Strategy 4	Strategy 5
TP1 (Busy client)	8.06	10.00	8.58	8.55	7.72
TP2 (Non-Busy client)	1.68	1.60	1.55	1.53	1.55
TP (System)	19.49	23.21	20.27	20.15	18.54
Improvement (%)	-	19.07%	4.02%	3.41%	-4.90%

**Figure A-2: Performance of load balancing strategies with
ST=60ms, TT=500ms, NC=4, NS=2**

Throughput	Strategy 1	Strategy 2	Strategy 3	Strategy 4	Strategy 5
Busy client	6.81	7.45	6.93	6.90	6.54
Non-Busy client	1.62	1.44	1.44	1.43	1.47
System	16.86	17.79	16.75	16.65	16.00
Improvement (%)	-	5.47%	-0.69%	-1.30%	-5.11%

**Figure A-3: Performance of load balancing strategies with
ST=80ms, TT=500ms, NC=4, NS=2**

	Strategy 1	Strategy 2	Strategy 3	Strategy 4	Strategy 5
TP1 (Busy client)	5.04	5.39	5.06	5.02	5.00
TP2 (Non-Busy client)	1.49	1.36	1.34	1.31	1.36
TP (System)	13.06	13.49	12.80	12.65	12.71
Improvement (%)	-	3.28%	-2.05%	-3.18%	-2.68%

**Figure A-4: Performance of load balancing strategies with
ST=120ms, TT=500ms, NC=4, NS=2**

***Appendix B: Supplementary Results with Exponentially
Distributed Service Times and Think Times in a
Simple System***

**Figure B-1: Performance of load balancing strategies with
ST=10ms (Exponential), TT=500ms, NC=4, NS=2**

**Figure B-2: Performance of load balancing strategies with
ST=20ms (Exponential), TT=500ms, NC=4, NS=2**

**Figure B-3: Performance of load balancing strategies with
ST=40ms (Exponential), TT=500ms, NC=4, NS=2**

**Figure B-4: Performance of load balancing strategies with
ST=60ms (Exponential), TT=500ms, NC=4, NS=2**

**Figure B-5: Performance of load balancing strategies with
ST=80ms (Exponential), TT=500ms, NC=4, NS=2**

**Figure B-6: Performance of load balancing strategies with
ST=120ms (Exponential), TT=500ms, NC=4, NS=2**

Throughput	Strategy1	Strategy2	Strategy3	Strategy4	Strategy5
Busy	16.69	25.88	21.72	21.58	13.88
Non-busy	1.88	1.86	1.85	1.84	1.74
System	37.13	55.48	47.14	46.84	31.23
Improvement (%)	-	49.41%	26.95%	26.14%	-15.89%

**Figure B-7: Performance of load balancing strategies with
ST=10ms, TT=500ms (Exponential), NC=4, NS=2**

Throughput	Strategy1	Strategy2	Strategy3	Strategy4	Strategy5
Busy	13.15	19.63	16.60	15.89	11.74
Non-busy	1.83	1.82	1.80	1.78	1.69
System	29.97	42.90	36.80	35.34	26.86
Improvement (%)	-	43.14%	22.79%	17.92%	-10.38%

**Figure B-8: Performance of load balancing strategies with
ST=20ms, TT=500ms(Exponential), NC=4, NS=2**

Throughput	Strategy1	Strategy2	Strategy3	Strategy4	Strategy5
Busy	9.38	12.98	9.99	9.88	9.29
Non-busy	1.75	1.71	1.65	1.61	1.59
System	22.26	29.38	23.28	22.98	21.76
Improvement (%)	-	31.99%	4.58%	3.23%	-2.25%

**Figure B-9: Performance of load balancing strategies with
ST=40ms, TT=500ms(Exponential), NC=4, NS=2**

Throughput	Strategy1	Strategy2	Strategy3	Strategy4	Strategy5
Busy	8.07	10.01	8.60	8.56	7.74
Non-busy	1.66	1.59	1.56	1.53	1.55
System	19.46	23.20	20.32	20.18	18.58
Improvement (%)	-	19.21%	4.41%	3.69%	-4.53%

**Figure B-10: Performance of load balancing strategies with
ST=60ms, TT=500ms(Exponential), NC=4, NS=2**

Throughput	Strategy1	Strategy2	Strategy3	Strategy4	Strategy5
Busy	6.79	7.47	6.94	6.88	6.55
Non-busy	1.62	1.44	1.44	1.43	1.47
System	16.82	17.82	16.76	16.62	16.04
Improvement (%)	-	5.95%	-0.36%	-1.19%	-4.64%

**Figure B-11: Performance of load balancing strategies with
ST=80ms, TT=500ms(Exponential), NC=4, NS=2**

Throughput	Strategy1	Strategy2	Strategy3	Strategy4	Strategy5
Busy	5.03	5.41	5.07	5.04	5.02
Non-busy	1.49	1.36	1.34	1.31	1.36
System	13.04	13.54	12.82	12.70	12.76
Improvement (%)	-	3.83%	-1.69%	-2.61%	-2.15%

**Figure B-12: Performance load balancing strategies with
ST=120ms, TT=500ms(Exponential), NC=4, NS=2**

***Appendix C: Supplementary Results for Higher Values of
Service Times in a Complex System***

Throughput	Strategy 1	Strategy 2	Strategy 3	Strategy 4	Strategy 5
Busy client	5.49	8.93	6.54	6.13	8.40
Non-Busy client	1.61	1.56	1.52	1.41	1.59
System	28.40	41.95	32.23	30.17	39.99
Improvement (%)	-	47.73%	13.50%	6.25%	40.83%

**Figure C-1: Performance of load balancing strategies with
ST=40ms, TT=500ms, NC=8, NS=4**

Throughput	Strategy 1	Strategy 2	Strategy 3	Strategy 4	Strategy 5
Busy client	5.00	7.41	5.92	5.56	6.76
Non-Busy client	1.59	1.45	1.45	1.35	1.49
System	26.36	35.42	29.48	27.62	32.99
Improvement (%)	-	34.37%	11.85%	4.78%	25.15%

**Figure C-2: Performance of load balancing strategies with
ST=60ms, TT=500ms, NC=8, NS=4**

Throughput	Strategy 1	Strategy 2	Strategy 3	Strategy 4	Strategy 5
Busy client	4.57	6.06	5.15	5.00	5.72
Non-Busy client	1.50	1.34	1.37	1.30	1.41
System	24.29	29.62	26.11	25.19	28.51
Improvement (%)	-	21.96%	7.49%	3.74%	17.41%

**Figure C-3: Performance of load balancing strategies with
ST=80ms, TT=500ms, NC=8, NS=4**

Throughput	Strategy 1	Strategy 2	Strategy 3	Strategy 4	Strategy 5
Busy client	3.13	4.31	3.64	3.52	4.22
Non-Busy client	1.47	1.22	1.15	1.16	1.28
System	18.40	22.09	19.14	18.72	22.00
Improvement (%)	-	19.86%	3.95%	1.55%	19.37%

**Figure C-4: Performance of load balancing strategies with
ST=120ms, TT=500ms, NC=8, NS=4**

***Appendix D: Supplementary Results with Exponentially
Distributed Service Times and Think Times in a
Complex System***

Figure D-1: Performance of load balancing strategies with ST=10ms (Exponential), TT=500ms, NC=8, NS=4

Figure D-2: Performance of load balancing strategies with ST=20ms (Exponential), TT=500ms, NC=8, NS=4

Throughput	Strategy1	Strategy2	Strategy3	Strategy4	Strategy5
Busy	5.50	9.00	6.54	6.20	8.59
Non-busy	1.61	1.58	1.52	1.43	1.57
System	28.44	42.32	32.24	30.52	40.66
Improvement (%)	-	48.80%	13.36%	7.31%	42.96%

**Figure D-3: Performance of load balancing strategies with
ST=40ms (Exponential), TT=500ms, NC=8, NS=4**

Throughput	Strategy1	Strategy2	Strategy3	Strategy4	Strategy5
Busy	5.25	7.60	6.05	5.82	6.88
Non-busy	1.68	1.45	1.44	1.37	1.48
System	27.71	36.21	29.96	28.77	33.45
Improvement (%)	-	30.65%	8.10%	3.84%	20.69%

**Figure D-4: Performance of load balancing strategies with
ST=60ms (Exponential), TT=500ms, NC=8, NS=4**

Throughput	Strategy1	Strategy2	Strategy3	Strategy4	Strategy5
Busy	4.68	6.25	5.46	5.23	5.92
Non-busy	1.61	1.33	1.36	1.29	1.40
System	25.17	30.33	27.28	26.09	29.26
Improvement (%)	-	20.51%	8.37%	3.64%	16.25%

**Figure D-5: Performance of load balancing strategies with
ST=80ms (Exponential), TT=500ms, NC=8, NS=4**

Throughput	Strategy1	Strategy2	Strategy3	Strategy4	Strategy5
Busy	3.27	4.46	3.78	3.69	4.46
Non-busy	1.50	1.20	1.14	1.17	1.28
System	19.08	22.65	19.68	19.44	22.97
Improvement (%)	-	18.71%	3.16%	1.89%	20.41%

**Figure D-6: Performance of load balancing strategies with
ST=120ms (Exponential), TT=500ms, NC=8, NS=4**

Throughput	Strategy1	Strategy2	Strategy3	Strategy4	Strategy5
Busy	9.52	17.54	12.78	11.88	12.70
Non-busy	1.88	1.82	1.79	1.72	1.74
System	45.60	77.46	58.29	54.39	57.74
Improvement (%)	-	69.87%	27.83%	19.28%	26.63%

Figure D-7: Performance of load balancing strategies with ST=10ms, TT=500ms (Exponential), TT=500ms, NC=8, NS=4

Throughput	Strategy1	Strategy2	Strategy3	Strategy4	Strategy5
Busy	7.70	13.61	9.71	9.11	11.22
Non-busy	1.83	1.72	1.69	1.60	1.69
System	38.14	61.31	45.59	42.83	51.64
Improvement (%)	-	60.76%	19.54%	12.30%	35.39%

Figure D-8: Performance of load balancing strategies with ST=20ms, TT=500ms(Exponential), TT=500ms, NC=8, NS=4

Throughput	Strategy1	Strategy2	Strategy3	Strategy4	Strategy5
Busy	5.43	8.89	6.58	6.17	8.52
Non-busy	1.73	1.54	1.51	1.40	1.56
System	28.66	41.71	32.35	30.30	40.33
Improvement (%)	-	45.55%	12.87%	5.72%	40.72%

**Figure D-9: Performance of load balancing strategies with
ST=40ms, TT=500ms(Exponential), TT=500ms, NC=8, NS=4**

Throughput	Strategy1	Strategy2	Strategy3	Strategy4	Strategy5
Busy	5.10	7.45	5.99	5.59	6.77
Non-busy	1.66	1.44	1.45	1.35	1.49
System	27.04	35.56	29.77	27.76	33.06
Improvement (%)	-	31.47%	10.09%	2.64%	22.24%

**Figure D-10: Performance of load balancing strategies with
ST=60ms, TT=500ms(Exponential), TT=500ms, NC=8, NS=4**

Throughput	Strategy1	Strategy2	Strategy3	Strategy4	Strategy5
Busy	4.60	6.21	5.23	5.16	5.82
Non-busy	1.59	1.34	1.37	1.31	1.41
System	24.78	30.21	26.41	25.89	28.92
Improvement (%)	-	21.93%	6.60%	4.48%	16.72%

Figure D-11: Performance of load balancing strategies with ST=80ms, TT=500ms(Exponential), TT=500ms, NC=8, NS=4

Throughput	Strategy1	Strategy2	Strategy3	Strategy4	Strategy5
Busy	3.31	4.39	3.75	3.66	4.33
Non-busy	1.48	1.22	1.23	1.19	1.27
System	19.16	22.44	19.93	19.39	22.42
Improvement (%)	-	17.10%	4.02%	1.19%	16.98%

Figure D-12: Performance load balancing strategies with ST=120ms, TT=500ms(Exponential), TT=500ms, NC=8, NS=4