INFORMATION TO USERS

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

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Performance Analysis of Client-Broker-Server Systems

by

Omotunde Adebayo

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

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6
June 3, 1997

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.

0-612-26984-1

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

Performance Analysis of Client-Broker-Server Systems

submitted by Omotunde Adebayo, B. Sc,
in partial fulfillment of the requirements for
the degree of Master of Engineering

Chair, Department of Systems and Computer Engineering

Thesis Co-supervisor

Thesis Co-supervisor

Carleton University
August 6, 1997

ABSTRACT

The importance of brokers in client-server systems is growing, in part, to accommodate open, heterogeneous system design and, in part, to facilitate fault tolerance and improved performance through load balancing. The added complexity brings forth a number of architectural design decisions affecting performance including: broker distribution, the threading model, and server replication. It is found that the performance characteristics of the forwarding and half-forwarding architectures depends on whether client requests are of a read or write type. Broker distribution is found to improve the performance of basic broker architectures for small object service times and/or large message sizes. The choice of a threading paradigm for the broker is also found to impact performance. Stateful object replication is found to improve performance in some areas of the parameter space and should only be used when the performance gain in having replicas outweighs the overhead in maintaining the replicas.

ACKNOWLEDGMENT

I would like to express my sincere thanks to my supervisors, Prof. J.E. Neilson and Prof D.C. Petriu, for their support, patience, encouragement, constructive criticisms and helpful suggestions which were essential in carrying out the research and writing this thesis.

The support I received from my fellow RADS colleagues is also highly appreciated. Last but definitely not the least, I would like to acknowledge the Telecommunication Research Institute of Ontario (TRIO) for funding this research.

I dedicate this thesis to my parents, Ronke, Nike, Ladi, Bukola and Patricia.

Table of Contents

Chapter 1	Introduction	1
1.1	Motivation for the Thesis	5
1.2	Methodology	6
1.3	Goals of the Thesis	7
1.4	Contributions	8
1.5	Outline of Thesis	9
Chapter 2	Background	11
2.1	Adaptive Communication Environment (ACE)	11
2.1.1	A Family of Design Patterns for Flexibly Configuring Network Services in Distributed Systems	12
2.1.2	ACE C++ Wrapper Components	13
2.1.2.1	Reactor: Event Demultiplexing and Service Dispatching	15
2.1.2.2	The Connector Class	16
2.1.2.3	The Acceptor Pattern	20
2.1.2.4	Router Pattern	25
2.1.2.5	IPC_SAP Class Category	25
2.1.2.6	Concurrency Class Category	25
2.2	The Common Object Request Broker Architecture	26
2.2.1	Client/Server/Broker Interactions	33
2.2.2	A Simple Client/Server Example Using CORBA	36
2.3	Threads	39
2.4	Software Modeling Methods	42
2.5	PARASOL	43
2.5.1	Hardware	43
2.5.2	Task Management	44
2.5.3	Inter-task Communication	44
2.5.4	Task Synchronization	45
2.5.5	Statistics Capturing	45
2.5.6	Structure of a Typical PARASOL model	45
2.5.7	System Modeling using PARASOL	46
2.6	Measurement Tools	47
2.6.1	DECALS	47
2.6.2	Getrusage	49
2.6.3	Displacement Technique	49
2.6.4	Netstat	49

Chapter 3 Client/Broker/Server models and Broker Designs using ACE 51

3.1	Broker models	51
3.2	A Group Communication Architecture in CBS systems	60
3.3	Broker Functionality	64
3.3.1	Broker Implementation	65
3.4	Server and Client Task Design and Implementation	66
3.5	Multi-threading Models in the Broker Design	67
3.5.1	Thread-per-Client Broker	68
3.5.2	Thread Pool Broker	70
3.5.3	Single-thread-Reactive Broker	74

Chapter 4 Measurements 79

4.1	Characterization of the CBS Components	79
4.1.1	Reactor Component	80
4.1.1.1	Reactor Characterization	81
4.1.2	Bus Sub -System	84
4.1.2.1	Bus Speed Measurements	85
4.1.2.2	Bus Speed Measurement Results	85
4.1.2.3	CPU measurement results for Send and Receive operations	88
4.1.2.4	Thread Pool Insert and Remove CPU times ..	88
4.1.2.5	Model Input Parameters	90
4.2	Forwarding Broker Performance Results	90

Chapter 5 CBS Modelling and Model Validation 95

5.1	Forwarding Broker PARASOL Model Evolution	95
5.1.1	Bus Modelling	96
5.1.2	Client Modelling	99
5.1.3	Server Modelling	100
5.1.4	Broker Modelling	101
5.2	PARASOL Model Prediction Results	101
5.2.1	TPC PARASOL Model Validation	102
5.2.2	DB_TPC PARASOL Validation	104
5.2.3	TPOOL PARASOL Validation	105
5.2.4	DB_TPOOL PARASOL Validation	107
5.2.5	TREAC PARASOL Validation	108
5.2.6	DB_TREAC PARASOL Validation	110

5.3	Simulator Model of other Broker Architectures	111
5.3.1	Half Forwarding Broker (distributed and non-distributed) Model	111
5.3.2	Handle Broker Model	112
5.3.3	Broker Model with object replication	112
 Chapter 6 Performance Results		113
6.1	Workload pattern	113
6.2	Confidence Intervals	115
6.3	Performance Impact of broker distribution on the forwarding broker	115
6.4	Performance comparison of threading models	121
6.4.1	Choosing an appropriate threading level for TPOOL	124
6.5	Performance comparison of the handle-driven, half- forwarding and forwarding broker model and the impact of broker distribution	126
6.5.1	Interactions involving only Read type requests	126
6.5.2	Interactions involving only Write Type Requests ..	132
6.5.3	Interactions involving Equal proportion of Read and Write Type requests	133
6.6	Performance impact of replicating stateful objects in CBS systems	135
6.7	Critical Message Size for NOR_BROKER and OR_BROKER	139
 Chapter 7 Conclusions and Future Work ..		150
 References		156

List of Figures

Figure 2-1 . . . Object interaction diagram for ACE Reactor	14
Figure 2-2 . . . The structure and participants in ACE Reactor framework	15
Figure 2-3 . . . Structure and participants in the ACE Connector Framework	17
Figure 2-4 . . . Object Interaction diagram for the ACE_Connector framework	20
Figure 2-5 . . . Structure and participants in the ACE Acceptor Framework	22
Figure 2-6 . . . Object Interaction diagram for the ACE_Acceptor framework	24
Figure 2-7 . . . Some Participants in the SOCK_SAP Class Category	26
Figure 2-8 . . . Components in CORBA distributed object computing	27
Figure 2-9 . . . The interaction between server and broker when a server binds its service.	34
Figure 2-10 . . Client, server, and broker interaction when a client issues a request to the broker	35
Figure 2-11 . . Types of threads in a typical operating environment	40
Figure 2-12 . . Processes in the DECALS system	48
Figure 3-1 . . . Client-Broker-Server interactions in a forwarding broker model	52
Figure 3-2 . . . Shows the interaction when a client makes a request in the distributed forwarding Broker Model.	54
Figure 3-3 . . . A server issues a bind operation in a distributed forward broker model.	55
Figure 3-4 . . . Client -Broker -Server interaction in a half forwarding broker model.	56
Figure 3-5 . . . Client-Server-Broker interaction in a distributed half-forwarding broker model.	57
Figure 3-6 . . . Client-Broker-Server interaction in a handle driven broker model.	58
Figure 3-7 . . . Client-Broker-Server interaction in a Distributed handle-driven broker model.	59
Figure 3-8 . . . Shows the interaction that takes place when a client makes a write request in the distributed forwarding broker model with replicated object implementations. The state is kept consistent through the use of group communication.	62
Figure 3-9 . . . Thread-per-Client Broker model using ACE Components	69
Figure 3-10 . . The Thread-per-Pool model using components provided by ACE.	71
Figure 3-11 . . The Single-Threaded Reactive Broker model using ACE components.	78
Figure 4-1 . . . CPU Times for relevant Reactor methods on workstation helicon	82
Figure 4-2 . . . CPU Times for relevant Reactor methods on workstation mira	82
Figure 4-3 . . . CPU Times for relevant Reactor methods on workstation alcor	83
Figure 4-4 . . . CPU Times for relevant Reactor methods on workstation sputnik	83
Figure 4-5 . . . CPU Times for relevant Reactor methods on workstation bellatrix	84
Figure 4-6 . . . CPU times for send and receive methods of ACE SOCK_Stream on workstation bellatrix	89
Figure 4-7 . . . CPU times for send and receive methods of pipes on workstation bellatrix	89
Figure 4-8 . . . Response time versus number of clients for CBS systems	

	employing TPC, TPOOL, TREAC, DB_TPC, DB_TPOOL and DB_TREAC broker. S1 = 11.5 msec, S2= 12.7 msec, RPMS = 100 bytes and RQMS = 63 bytes.	92
Figure 4-9 ...	System Throughput versus number of clients for CBS systems employing TPC, TPOOL, TREAC, DB_TPC, DB_TPOOL and DB_TREAC broker. S1 = 11.5 msec, S2= 12.7 msec, RPMS = 100 bytes and RQMS = 63 bytes	92
Figure 4-10 ..	System throughput versus number of clients for CBS systems employing TPC, TPOOL, TREAC, DB_TPC, DB_TPOOL and DB_TREAC broker. S1 = 13.39 msec, S2= 16.05 msec, RPMS = 10000 bytes and RQMS = 63 bytes.	94
Figure 4-11 ..	Client response time versus number of clients for CBS systems employing TPC, TPOOL, TREAC, DB_TPC, DB_TPOOL and DB_TREAC broker. S1 = 13.39 msec, S2= 16.05 msec, RPMS = 10000 bytes and RQMS = 63 bytes.	94
Figure 5-1 ...	CBS Model Construction Process	96
Figure 5-2 ...	Response time versus number of clients for TPC prototype and simulator. S1 = 11.5 msec, S2 = 12.7 msec, RPMS = 100 bytes, RQMS = 63 bytes.	103
Figure 5-3 ...	Response time versus number of clients for TPC prototype and simulator. S1 = 13.39 msec, S2 = 16.05 msec, RPMS = 10000 bytes, RQMS = 63 bytes	103
Figure 5-4 ...	Response time versus number of clients for the DB_TPC prototype and simulator. S1 = 11.5 msec, S2 = 12.7 msec, RPMS = 100 bytes, RQMS = 63 bytes	104
Figure 5-5 ...	Response time versus number of clients for the DB_TPC prototype and simulator. S1 = 13.39msec, S2 = 16.05 msec, RPMS = 10000 bytes, RQMS = 63 bytes.	105
Figure 5-6 ...	Response time versus number of clients for the TPOOL prototype and simulator. S1 = 11.5 msec, S2 = 12.7 msec, RPMS = 100 bytes, RQMS = 63 bytes.	106
Figure 5-7 ...	Response time versus number of clients for the TPOOL prototype and simulator. S1 = 13.39 msec, S2 = 16.05 msec, RPMS = 10000 bytes, RQMS = 63 bytes	106
Figure 5-8 ...	Response time versus number of clients for the DB_TPOOL prototype and simulator. S1 = 11.5msec, S2= 12.7, RPMS = 100 bytes, RQMS = 63 bytes	107
Figure 5-9 ...	Response time versus number of clients for the DB_TPOOL prototype and simulator. S1 = 13.39 msec, S2 = 16.05 msec, RPMS = 10000 bytes, RQMS = 63 bytes	108
Figure 5-10 ..	Response time versus number of clients for the TREAC prototype and simulator. S1 = 11.5 msec, S2 = 12.7 msec, RPMS = 100 bytes, RQMS = 63 bytes	109
Figure 5-11 ..	Response time versus number of clients for the TREAC prototype and simulator. S1 = 13.39 msec, S2 = 16.05 msec, RPMS =	

	10000 bytes, RQMS = 63 bytes	109
Figure 5-12 ..	Response time versus number of clients for the DB_TREAC prototype and simulator. S1 = 11.5 msec, S2 = 12.7 msec, RPMS = 100 bytes, RQMS = 63 bytes	110
Figure 5-13 ..	Response time versus number of clients for the DB_TREAC prototype and simulator. S1 = 13.39 msec, S2 = 16.05 msec, RPMS = 10000 bytes, RQMS = 63 bytes	111
Figure 6-1 ...	Throughput versus number of clients for CBS systems employing TPC, TPOOL, TREAC, DB_TPC, DB_TPOOL and DB_TREAC broker. S1 = 11.5 msec, S2 = 12.7 msec, RPMS = 100 bytes, TT = 1sec	116
Figure 6-2 ...	Response time versus number of clients for CBS systems employing TPC, TPOOL, TREAC, DB_TPC, DB_TPOOL and DB_TREAC broker. S1 = 11.5 msec, S2 = 12.7 msec, RPMS = 100 bytes, TT = 1sec	116
Figure 6-3 ...	Response time for DB_TPOOL and TPOOL. S1 = 1.15 msec, S2 = 1.27 msec, RPMS = 100 bytes, TT = 1sec.	118
Figure 6-4 ...	Utilization figures for the participants in TPOOL. S1 = 1.15 msec, S2 = 1.27 msec, RPMS = 100 bytes, TT = 1sec	118
Figure 6-5 ...	Utilization figures for the participants in DB_TPOOL. S1 = 1.15 msec, S2 = 1.27 msec, RPMS = 100 bytes, TT = 1sec	119
Figure 6-6 ...	Response time versus number of clients for CBS systems employing TPC, TPOOL, TREAC, DB_TPC, DB_TPOOL and DB_TREAC broker. S1 = 13.39 msec, S2 = 16.05 msec, RPMS = 10000 bytes, TT = 1sec	120
Figure 6-7 ...	Bus utilization versus number of clients for CBS systems employing TPC, TPOOL, TREAC, DB_TPC, DB_TPOOL and DB_TREAC broker. S1 = 13.39 msec, S2 = 16.05 msec, RPMS = 10000 bytes, TT = 1sec	120
Figure 6-8 ...	Response time for TPOOL, TREAC and TPC. S1 = 1.15 msec, S2 = 1.27 msec, RPMS = 100 bytes, TT = 1sec	122
Figure 6-9 ...	Response time for TPOOL, TREAC and TPC. S1 = 1.39 msec, S2 = 1.61 msec, RPMS = 10000 bytes, TT = 1sec	123
Figure 6-10 ..	Response time for TPOOL, TREAC and TPC. S1 = 11.5 msec, S2 = 12.7 msec, RPMS = 100 bytes, TT = 1sec	123
Figure 6-11 ..	Response time for TPOOL for S1 = 11.5 msec, S2 = 12.7 msec, RPMS = 100 bytes and TT = 1sec	125
Figure 6-12 ..	Response time for TPOOL for S1 = 13.39 msec, S2 = 16.05 msec, RPMS = 10000 bytes and TT = 1sec	125
Figure 6-13 ..	Response time for F_BROKER, HF_BROKER and HD_BROKER. S1 = 11.5 msec, S2 = 12.7 msec, RPMS = 100 bytes, RQMS = 100 bytes and TT = 1 sec	127
Figure 6-14 ..	Utilizations for F_BROKER, HF_BROKER and HD_BROKER. S1 = 11.5msec, S2 = 12.7msec, RPMS = 100 bytes, RQMS = 100 bytes and TT = 1 sec	127

Figure 6-15 .. Response time for F_BROKER, HF_BROKER and HD_BROKER. S1 = 11.5 msec, S2 = 12.7 msec, RPMS = 10000 bytes, RQMS = 100 bytes and TT = 1 sec	129
Figure 6-16 .. Response time for the distributed version of the architectures (i.e., DB_F_BROKER, DB_HF_BROKER and HD_BROKER). S1 = 11.5 msec, S2 = 12.7 msec, RPMS = 10000 bytes, RQMS = 100 bytes and TT = 1 sec	129
Figure 6-17 .. Response time for F_BROKER, HF_BROKER and HD_BROKER. S1 = 1.15 msec, S2 = 1.27 msec, RPMS = 100 bytes, RQMS = 100 bytes and TT = 1 sec	130
Figure 6-18 .. Utilizations for F_BROKER, HF_BROKER and HD_BROKER. S1 = 1.15 msec, S2 = 1.27 msec, RPMS = 100 bytes, RQMS = 100 bytes and TT = 1 sec	131
Figure 6-19 .. Response time for DB_F_BROKER, DB_HF_BROKER and HD_BROKER. S1 = 1.15 msec, S2 = 1.27 msec, RPMS = 100 bytes, RQMS = 100 bytes and TT = 1 sec	131
Figure 6-20 .. Response time for F_BROKER, HF_BROKER and HD_BROKER. S1 = 11.5 msec, S2 = 12.7 msec, RPMS = 100 bytes, RQMS = 10000 bytes and TT = 1 sec	132
Figure 6-21 .. Response time for DB_F_BROKER, DB_HF_BROKER and HD_BROKER. S1 = 11.5 msec, S2 = 12.7 msec, RPMS = 100 bytes, RQMS = 10000 bytes and TT = 1 sec	133
Figure 6-22 .. Response time for F_BROKER, HF_BROKER and HD_BROKER. S1 = 11.5 msec, S2 = 12.7 msec, RPMS = 10000 bytes, RQMS = 10000 bytes and TT = 1 sec	134
Figure 6-23 .. Response time for DB_F_BROKER, DB_HF_BROKER and HD_BROKER. S1 = 11.5 msec, S2 = 12.7 msec, RPMS = RQMS = 10000 bytes and TT = 1 sec	134
Figure 6-24 .. Read response time for SW = 50 msec, SR = 50 msec, rp = 0.75, MS = 2000 bytes and TT= 10 sec	136
Figure 6-25 .. Write response time for SW = 50 msec, SR = 50 msec, rp = 0.75, MS = 2000 bytes and TT= 10 sec	136
Figure 6-26 .. Utilizations for SW = 50 msec, SR = 50 msec, rp = 0.75, MS = 2000 bytes and TT= 10 sec	137
Figure 6-27 .. Read response time for SW = 50 msec, SR = 50 msec, rp = 0.25, MS = 2000 bytes and TT= 10 sec	138
Figure 6-28 .. Write response time for SW = 50 msec, SR = 50 msec, rp = 0.25, MS = 2000 bytes and TT= 10 sec	138
Figure 6-29 .. Utilization for SW = 50 msec, SR = 50 msec, rp = 0.25, MS = 2000 bytes and TT= 10 sec	139
Figure 6-30 .. Read response time for SW = 50 msec, SR = 50 msec, rp = 0.25, MS = 10000 bytes and TT= 10 sec	140
Figure 6-31 .. Write response time for SW = 50 msec, SR = 50 msec, rp = 0.25, MS = 10000 bytes and TT= 10 sec	140
Figure 6-32 .. Bus utilization for SW = 50 msec, SR = 50 msec, rp = 0.25,	

MS = 10000 bytes and TT= 10 sec	141
Figure 6-33 .. Critical message size for read requests. SR = 50 msec, SW = 50 msec, rp = 0.3, TT = 10 sec	146
Figure 6-34 .. Critical message size for read requests. SR = 50 msec, SW = 50 msec, rp = 0.5, TT = 10 sec	146
Figure 6-35 .. Critical message size for read requests. SR = 50 msec, SW = 50 msec, rp = 0.7, TT = 10 sec	147
Figure 6-36 .. Critical message size for write requests. SR = 50 msec, SW = 50 msec, rp = 0.3, TT = 10 sec	147
Figure 6-37 .. Critical message size for write requests. SR = 50 msec, SW = 50 msec, rp = 0.5, TT = 10 sec	148
Figure 6-38 .. Critical message size for write requests. SR = 50 msec, SW = 50 msec, rp = 0.7, TT = 10 sec	148

List of Tables

Table1 Reactor remove_handler() cpu times	84
Table2 Reactor register_handler() cpu times	84
Table3 Netstat report taken on receiver workstation before running experiment	86
Table4 Netstat report taken on receiver workstation while experiment was taking place	86
Table5 Remove and Insert methods CPU times on workstation bellatrix	88

Chapter 1

Introduction

Most of today's system and application software development tends to be very complex. Object-Oriented (OO) programming provides an attractive programming model to cope with such complex systems and software while at the same time providing the much needed properties such as maintainability, extensibility, and reusability [24]. This model is therefore very relevant in distributed systems, as they are generally very complex. OO technology provides distributed computing with many of the same benefits (such as encapsulation, reuse, portability, and extensibility) as it does for non-distributed computing. In fact, it is often more natural to use object-oriented techniques than it is for non-distributed computing [2]. This is due to the decentralized characteristic of distributed systems. In non-distributed applications, there is often a need to sacrifice abstraction and modularity for an increase in performance. For example, many programmers tend to use global variables or access fields in structures directly to avoid the overhead of passing parameters and calling functions, respectively. OO distributed programming is a generalization of the client-server model, in that objects encapsulate an internal state and make it accessible through a well-defined interface [10]. Client applications may import an interface, bind to a remote instance of it, and issue remote object invocations.

Building complex object oriented distributed software systems using a set of decoupled and interoperating components as opposed to a monolithic version of the same system results in superior software that is potentially more flexible, easier to maintain and readily changeable [8]. By partitioning functionality into independent components, soft-

ware systems built using this approach are potentially distributable and scalable.

In client/server (CS) systems, a client and server form two independent components. In this system, a client needing a service from a particular server sends a request to the appropriate server and blocks until the response to its request arrives. On receiving the client request, the server performs the requested service and returns the result to the client. For small static software systems, the CS computational model works fine since clients access only a small set of known services or servers such as a file server and printing service. However, as the system grows and evolves, individual clients need to be constantly updated to reflect routine enhancements, substitutions, relocation as well as the addition of servers and services. In addition, accessing servers and services becomes increasingly difficult in heterogeneous networks, since clients and servers have to, for example, be aware of the byte order of each others machine, and since servers may be relocated to another machine with a different byte order, the client and server application would have to be modified to reflect this simple change in system configuration.

One solution to this problem is to have the interaction between clients and servers in a software system further decoupled by introducing a broker component in-between a client and a server. A server can be considered as an object since it encapsulates some data structures and allows the manipulation of the encapsulated data through requests from client application, which are like method invocations. Servers register themselves with the broker and make their services available through well defined interfaces. Clients access services provided by the server by issuing a request to server via the broker component. The client does this by invoking the appropriate method of the server object just as it would any C++ object, which results in a request being sent to the broker, containing the

appropriate information such as the particular service or type of service needed [3]. The broker has the responsibility of locating the relevant server based on the information contained in the client request and forwarding the request to the appropriate server. The server upon receiving the request, does the necessary computation and returns the result back to the broker, which in turn relays this information to the requesting client. The software architecture described above is known as a client/broker/server (CBS) model. Client applications which are based on the CBS model can obtain services from servers that could be local to its machine or thousands of miles away on another machine or network without having to worry about what operating system the server uses or on what hardware platform it resides. Brokers therefore act as the glue between clients and servers that allow CS interactions to take place in a very flexible and interoperable way. Broker models that could be used in CBS systems are [9]:

- A forwarding broker that relays a client's request to the relevant server application, retrieves the response, and relays this reply back to the client.
- A handle driven broker returns a service handle back to the client, which contains all the information required to interact with the server for the given service (such as its name, the address of its network node and the request format).

The forwarding broker provides one point of control for clients and servers in CBS systems. If for whatever reason the connection between the server and broker or the broker and client is severed, this can be detected and restarted by the broker which is at the center of the communication path. As can be readily observed, a lot of network traffic is generated in this broker model, coupled with the fact that all CS interactions must pass through the broker. This can impact significantly the performance of the system. It can therefore be

said that CBS systems employing the forwarding broker model has its short-coming as far as performance is concerned. This broker model definitely needs looking into. In the handle driven broker model, the potential for performance degradation is reduced since messages between the client and server bypass the broker. However, the client takes on the added responsibility of detecting and recovering from failures. In addition, in the handle-driven broker mode, when a server crashes, a transient behavior occurs, since all clients that were currently using the server would almost simultaneously send a request to the broker for a new handle. So, designers have to make a choice between these two models when using the CBS software model.

The broker pattern of CBS systems is the model adopted by the Object Management Group (OMG) in its Object Management Architecture (OMA) [12]. OMG was formed in 1989 with a goal of creating standards and architectures that allowed for the interoperability and portability of OO applications. OMA defines at a high level of abstraction, the participants and infrastructure necessary for distributed OO computing. Towards this, the OMG group came up with the Common Object Request Broker Architecture (CORBA) specifications which describes the interfaces and services that must be provided by a CORBA compliant broker.

CORBA defines a set of components and interfaces that allow client applications to invoke operations on object implementations, whether they be remote or local to its machine. Thus, the complexity of developing distributed applications is reduced by making distribution transparent to the developer since the task of establishing passive and active connections, the marshalling and unmarshalling of parameters, together with the demultiplexing and dispatching are done automatically by CORBA. Hence, a developer

can concentrate more on the application problem without having to grapple with the low-level communication programming which are error prone, difficult to debug and time consuming to develop.

In this thesis, a number of CBS architectures are studied with the aim of evaluating and comparing their performance characteristics. The systems were built using the Adaptive Communication Environment (ACE) toolkit [14]. ACE provides a set of reusable C++ wrappers, class categories and frameworks that perform common communication tasks such as demultiplexing, event handler dispatching, connection establishment etc.

1.1 Motivation for the Thesis

Despite great strides in the development of high speed networks and computers, the design and implementation of flexible, scalable and efficient OO distributed software still poses a great challenge. Distributed software systems design and implementation based on the CBS software model is becoming the approach used for improving the flexibility, portability, and maintainability of such systems. CORBA which adopts the broker model, was introduced to promote the design and implementation of distributed software applications by automating common networking tasks such as object registration, location, and activation; request demultiplexing; parameter marshalling and demarshalling; and operation dispatching. Hence, understanding the performance implication of using this software model in software development is therefore of great importance. For example, in the forwarding broker model, as was mentioned earlier, excessive network traffic is generated coupled with the potential of a bottleneck at the broker, all of which could have a significant impact on the performance of these systems. These issues are addressed in the thesis.

Current efforts on CORBA research has been focused on its implementation and little attention has been paid to improving its performance. Research work on CORBA include that done by Schmidt [1]. Schmidt in his work, tries to pinpoint where the key sources of overhead exist in higher-level communication software such as CORBA. His findings indicate that the overheads in some CORBA implementations such as ORBIX and ORBELINE emanate from a variety of sources including (1) non- optimized presentation layer conversions, data copying, and memory management; (2) excessive control information carried in request messages; (3) inefficient and inflexible receiver-side demultiplexing and dispatching operations. Maffeis in his research on CORBA shows how objects in CORBA compliant broker systems can be replicated through the use of group communication [10][11]. In this thesis, we take this a step further by presenting some broker architectures and showing under what conditions does the introduction of group communication in object implementation replication improve the performance of such systems. Finally, we study various ways in which the broker component may be threaded and look at the relative performance of the different threading strategies used.

1.2 Methodology

A two-phase approach involving both prototype systems and discrete simulation models was chosen to carry out performance studies of CBS systems. The reasons were threefold. First, work done by Istabrak [39] revealed that ORBeline, the only CORBA product available in the department, cannot support more than 24 clients and becomes unstable for large number of clients. The use of this CORBA product for carrying out performance studies was thus ruled out. This led to the use of prototypes which were built

using ACE. Second, it was recognized that the size of the design parameter space was very large requiring, even for a relatively modest sized study, the use of efficient methodologies. This suggested an instrumented prototype approach. Third, to eliminate “ambient noise” in prototype performance measurements, it would be necessary to dedicate the experimental apparatus, i.e., the network of workstations to these experiments. This suggested the development of efficient simulators which would not require dedicated hardware. The input parameters into the simulation models were obtained by implementing some CBS systems using the ACE technology and running an instrumented version of the implementations with the help of a tool known as DECALS (Distributed Experiment Control and Logging System) [27], in a controlled environment to obtain the desired parameters.

Performance figures were then obtained from the actual live runs of the non-instrumented prototype implementations and used to validate the corresponding simulated versions. This also helped to validate indirectly components common to other prototypes, that were later to be built using the simulator, such as the bus sub-system, send and receive operations of the various participants in the CBS system as well as the cpu times consumed when certain operations took place. Other CBS systems were later simulated using the earlier obtained parameters.

1.3 Goals of the Thesis

The introduction of a broker into the client/server computational model brings about some performance overheads. One of the goal of this thesis is to examine architectural characteristics, such as server replication, threading model and broker distribution, that could

help improve the performance of CBS systems and/or reduce the overheads brought about by the introduction of the broker component. Another goal of the thesis is to better understand the performance implication of using the CBS model in software systems for various broker architectures. Towards these goals, the performance of a number of broker architectures and designs are investigated under various application parameters. Areas investigated in the thesis are summarized below.

- CBS system model validation
- Multi-threading of the broker component and its performance impact on CBS systems.
- Distribution of brokers across the network (i.e having a broker on each machine in the network where clients or servers of the CBS system exist)
- Relative performance of broker models and architectures under various parameters.
- Object replication through the use of group communication and its performance impact on CBS systems

1.4 Contributions

The list below summarizes the contributions and work that have been done in this thesis:

- Design and development of a number of CBS prototypes with the ACE technology
- Instrumentation of these CBS prototypes to obtain the parameters for the CBS simulation models that were to be built
- Performance measurements of these CBS prototypes for the validation of the

CBS simulation models

- The investigation of some architectural characteristics (i.e, threading model for brokers, broker distribution, and sever replications) that have an impact on the performance of CBS systems

1.5 Outline of Thesis

Chapter 2 provides some background information related to the thesis: This includes the Adaptive Communication Environment (ACE) toolkit, multi-threading, CORBA, tools (i.e, getrusage, netstat and DECALS) that were used in gathering performance measurements and in the CBS characterization, and the simulator PARASOL [25][26].

Chapter 3 presents various design aspects of various CBS prototypes using ACE. It discusses design issues like multi-threading and presents some other broker models and architectures that could be used in such systems.

Chapter 4 describes the experiments that were run: It also shows how the subset of the CBS systems, implemented using ACE, were characterized; how they were set up and run using DECALS and shows the performance measurements results of the system for various performance parameters such as number of clients in the system, and message size. Furthermore, it describes how the measurement of the effective speed of the bus was taken.

Chapter 5 presents the evolution of the PARASOL models of the CBS prototypes. It also discusses how the model prediction comes close to the performance results obtained from the actual live experiments described in chapter 4. This chapter also

describes other broker models that were built using PARASOL.

Chapter 6 discusses the performance results of the broker models and architectures described in chapter 5 for various parameters.

Chapter 7 concludes the study of work on the performance analysis of CBS systems and describes the findings and future work.

Chapter 2

Background

This chapter provides in greater detail some background information on ideas and concepts relevant to this thesis. The first section presents some information on the ACE toolkit which was used in building some of the CBS prototypes. The next section discusses the CORBA standards for distributed OO computing, which adopts the broker pattern in its architecture [12]. We present the components that make up its architecture and show how a simple client/server application might be built using this technology. Next, we present the basic ideas and concepts behind threads. Tools used in gathering performance measurements, and application parameters for use in the CBS models are described next. Finally, the simulator PARASOL used in modeling and extending the CBS systems built using ACE is described.

2.1 Adaptive Communication Environment (ACE)

ACE provides a set of reusable C++ wrappers, class categories and frameworks that perform common communication software tasks such as event demultiplexing, event handler dispatching, connection establishment, dynamic configuration of application services, and concurrency control across a range of operating systems platforms [4][14].

Rather than program directly with the low-level interfaces like sockets or TLI which are error prone, ACE provides software developers with a set of C++ wrappers that eliminates some of the tediousness and confusion that arise when using TLI, socket interfaces and other UNIX mechanisms. ACE represents a mid-way alternative for software

developers of distributed software between directly using low-level, efficient interfaces provided by different distributed operating systems or less efficient, high-level interfaces provided by communication frameworks like CORBA [12] or Distributed Computing Environment (DCE) [28]. ACE is a “light-weight” middleware compared to CORBA and DCE, providing a lower-level interface, but still improving programming simplicity, portability, and reusability of communication software.

The primary goal of ACE was to simplify the use of OS mechanisms that provide Interprocess Communication (IPC), communication port demultiplexing and concurrency. ACE implements a set of fundamental design patterns that simplify to some extent the development of concurrent event-driven communication software.

However, ACE shares some of the same drawbacks suffered by low-level communication mechanism as sockets. In particular, too much of the code required to program at this level is not directly related to the application. The programmer using ACE components still has to wrestle with the communications details of his/her application rather than focus on the application problem space. While ACE does eliminate some of the pitfalls that occur using low-level sockets interfaces like, a lot of attention is still given to correctly programming the communication infrastructure.

2.1.1 A Family of Design Patterns for Flexibly Configuring Network Services in Distributed Systems

The design and implementation of extensible communication software for complex distributed systems still poses a great challenge in today’s software developmental efforts. Design patterns represents recurring solutions to problems that arise when developing

software within a particular context [30]. Patterns capture the static and dynamic structure and collaboration among key participants in software designs. Design patterns are a promising technique for capturing and articulating proven techniques for developing extensible distributed software. Patterns expresses the structure and collaboration of participants in a software architecture at a level higher than the source code. Design patterns used in this thesis are the acceptor, connector, reactor and the router patterns. ACE implements these set of fundamental design patterns which are used in this thesis to simplify the development of concurrent event-driven CBS systems.

2.1.2 ACE C++ Wrapper Components

ACE C++ wrappers are grouped into class categories. The primary class categories supported by ACE are:

- Reactor - performs event demultiplexing and event handler dispatching by encapsulating OS event demultiplexing mechanisms (such as select, poll, and WaitForMultipleObjects [6][7]).
- IPC_SAP - which encapsulates local and/or remote IPC mechanisms such as sockets, TLI, and memory-mapped files [15];
- Connection - encapsulates active/passive connection establishment mechanisms [16];
- Concurrency - encapsulates and extends multi-threading and synchronization mechanisms [20]

Figure 2-1 Object interaction diagram for ACE Reactor

Figure 2-2 The structure and participants in ACE Reactor framework

2.1.2.1 Reactor: Event Demultiplexing and Service Dispatching

A common technique used in designing communication software (especially servers) is to have them wait and respond to events generated from multiple sources. An example of software that uses this technique is the UNIX `inetd` server, which uses an event loop that monitors a set of I/O communication ports using UNIX `select` call. Each communication port is associated with a particular application that provides some network services (such as the standard Internet `ftp`, `telnet`, `time-of-day`). When a request from a client arrives on one of these ports, the associated service handler is dispatched to perform the request (e.g., `inetd` forks and execs the relevant to handle the client request).

ACE Reactor [5] was implemented to help automate these common event processing activities (event demultiplexing and event handler dispatching) which reoccur over and over again. The Reactor encapsulates UNIX event demultiplexing mechanisms (select or poll) within a portable and extensible C++ wrapper. Figure 2-1 shows the object interaction for the Reactor framework. By using the Reactor, the need for complicated threading, synchronization, or locking within an application can often be eliminated. The Reactor decouples the demultiplexing and dispatching mechanisms (which are independent of an application and thus reusable) from application specific activities.

The structure of objects in the Reactor framework is illustrated in Figure 2-2. These objects perform demultiplexing of events (such as I/O events received on communication ports, and signal events) and dispatching the appropriate methods of pre-registered event handler(s) to process these events. As shown in Figure 2-2, all the event handler objects derive from the `Event_Handler` abstract base class. This class specifies an interface that is used by the Reactor to dispatch application specific methods in response to the arrival of events.

I/O handle-based events are dispatched via the `handle_input`, `handle_output`, and `handle_exceptions` methods. Subclasses of `Event_Handler` (such as `Orb_Handler` described in Chapter 3) extends the base class interface by defining additional methods and data member. In addition, virtual methods in the `Event_handler` interface are overridden in `Orb_Handler` class to implement broker specific functionality.

2.1.2.2 The Connector Class

The connector class in the ACE toolkit decouples active service initialization from pro-

Figure 2-3 Structure and participants in the ACE Connector Framework

cessing that take place after the initialization phase. The Connector class resolves the following forces that impact the design of connection-oriented communication software when using lower-level network programming interfaces (like sockets and TLI):

- The need to reuse active connection establishment code for each new service -
- The Connector pattern permits key characteristics of services, such as concurrency strategy, to be separated from the mechanisms used to establish the connections. This separation is very important since service characteristics change more frequently than connection establishment mechanisms. Hence, this helps

to decouple software components and increases software reusability. With this separation, the code used for connection establishment in one communication software can be reused in another, even if the service characteristics of the software are different.

- The need to enable flexible service concurrency policies - Once a connection is established, peer applications use the connection to exchange data to perform some type of service. A service can run in a single-thread, in multiple threads, or multiple processes, regardless of how the connection was established.
- The need to actively establish connections with a large number of peers efficiently - The Connector class can employ asynchrony to initiate and complete multiple connections in non-blocking mode. Figure 2-3 illustrates the structure and participants in the Connector class. As shown in the figure, the participants in this class category leverage off the ACE Reactor class by inheriting from its Event_Handler interface. Using the Reactor framework enables multiple connections to be actively established asynchronously within a single thread of control.

The Connector is a factory that assembles the resources necessary to create, connect, and activate a service handler. It is parameterized by a particular type of PEER_CONNECTOR and SVC_HANDLER. The PEER_CONNECTOR supplies the underlying transport mechanism (such as C++ wrappers for sockets or TLI) used to actively establish the connection asynchronously. The SVC_HANDLER specifies an abstract interface for defining a service that communicates with a connected Peer. The Connector connects this endpoint to its Peer when a connection is established successfully.

Moreover, a SVC_HANDLER is parametrized by a PEER_STREAM endpoint. Note that by inheriting from the Event_Handler, a SVC_HANDLER can register with a Reactor in order to demultiplex data within a single event-driven thread of control.

Parameterized types are used to decouple the Connector pattern's connection from the type of service and the type of connection mechanism. Developers produce Concrete Connectors by supplying arguments for these types. This enables the wholesale replacement of these types, without affecting the Connector pattern's connection establishment.

The collaboration among participants in the Connector pattern shown in Figure 2-4 are decided into three phases.

- Connection initiation phase - which actively connects one or more SVC_HANDLERS with their peers. Connections can either be initiated synchronously or asynchronously. The Connector determines the strategy for actively establishing connections.
- Service initialization phase - which activates a Svc_Handler by calling its open() method when the connection associated with it completes successfully. The open() method of the Svc_Handler performs service specific initialization.
- Service processing phase - which performs the application-specific service processing using the data exchanged between the Svc_Handler and its connected peer. Depending on the open method of Svc_Handler, this phase may employ the Reactor pattern.

Figure 2-4 illustrates the collaboration among participants in the Connector pattern using synchronous connection establishment. The forwarding broker uses the Connector pattern to simplify the task of connecting a large number of clients to the requested ser-

Figure 2-4 Object Interaction diagram for the ACE_Connector framework

vice. All connections are initiated asynchronously and the connections are then completed in parallel.

2.1.2.3 The Acceptor Pattern

The Acceptor pattern decouples passive service initialization from the tasks performed

once the service is initialized. The Acceptor class resolves the following forces that impact the design of connection-oriented communication software (particularly when using lower-level network programming interfaces like sockets and TLI).

- The need to reuse passive connection establishment code for each new service
 - The Acceptor pattern permits key characteristics of services (such as concurrency strategy or the data format) to evolve independently and transparently from the mechanisms used to establish the connections. Since the service characteristics change more frequently than connection establishment mechanisms this separation of concerns helps reduce software coupling and increases code reuse.
- The need to make connection establishment code portable across platforms that contain different network programming interfaces
 - Parameterizing the Acceptor's mechanisms for accepting connections and performing services helps to improve portability by allowing the wholesale replacement of these mechanisms. This makes the connection establishment code portable across platforms that contain different network programming interfaces.
- The need to enable flexible service concurrency policies
 - Once a connection is established, peer applications use the connection to exchange data to perform some type of service. A service can run in a single thread, in multiple threads, or multiple processes, regardless of how the connection was established. Figure 2-5 illustrates the structure and participants in the Acceptor pattern. This pattern leverages off the Reactor pattern to passively establish multiple connections within a single thread of control. It is parameterized by concrete types

Figure 2-5 Structure and participants in the ACE Acceptor Framework

that conform to interfaces of the formal template argument **SVC_HANDLER** (which performs a service in conjunction with a connected peer) and **PEER_ACCEPTOR** (which is the underlying mechanism used to passively establish the connection). The **Svc_Handler** shown in Figure 2-5 is a concrete type that defines the interface for an application specific service. It inherits from **Event_handler**, which allows it to be dispatched by the **Reactor** when connection events occur. In addition, **Svc_Handler** is parameterized by a **PEER_STREAM** endpoint. The **Acceptor** associates this endpoint with its **Peer**

when a connection is established successfully.

As with the Connector pattern, parameterized types are used to enhance portability since the Acceptor pattern's connection establishment strategy is independent of the type service and the type of IPC mechanism. Programmers supply concrete arguments for these types to produce a `Concrete_Acceptor`.

Figure 2-6 illustrates the collaboration among participants in the Acceptor pattern.

These collaboration are divided into three phases:

- Endpoint initialization - which creates a passive mode endpoint (encapsulated by `PERR_ACCEPTOR`) that is bound to a network address (such as IP address and port number). The passive-mode endpoint listens for connection requests from peers. This endpoint is registered with the Reactor, which then goes into an event loop waiting on that endpoint for connection requests to arrive from peers.
- Service activation phase - Since an Acceptor inherits from an `Event_Handler`, the Reactor can dispatch the Acceptor's `handle_event` method when connection events arrive. When connections arrive, the Reactor calls back to the Acceptor's `handle_event` method. This method performs the Acceptor's `Svc_handler` activation strategy. This strategy assembles the resources necessary to create a new `Concrete_Svc_Handler` object, accept the connection into the object, and activate the `Svc_Handler` by calling its `open` method.
- Service processing phase - once activated, the `Svc_Handler` processes incoming event messages using the Reactor pattern or some other form of concurrent event handling such as the Active Object pattern [38].

Figure 2-6 Object Interaction diagram for the ACE_Acceptor framework

The CBS prototype built in this thesis uses this pattern when requests come from a server to bind its services with the ORB. In this case, servers connect to the ORB, which uses the Acceptor pattern to decouple the activity of connecting passively from the actual binding process (such as storing the servers service name, interface and location).

2.1.2.4 Router Pattern

The Router pattern decouples multiple sources of input (i.e., from clients) from multiple sources of output (i.e., servers) to forward data correctly without having the broker blocking [29][35]. The Router pattern was employed in the single threaded broker design in this thesis and it uses the Reactor pattern in its implementation.

2.1.2.5 IPC_SAP Class Category

ACE provides a number of class categories that inherit from the base class IPC_SAP (InterProcess Communication Service Access Point). Among these class categories is the SOCK_SAP class category that provides applications with an OO interface to the Internet domain and UNIX domain protocol families. Compared to the Internet Inter-ORB protocol (IIOP) which requires a connection-oriented protocol (TCP/IP), the IPC_SAP can use either a connection-oriented protocol or a connectionless protocol like the Universal Datagram Protocol (UDP). Application using this C++ wrapper avoid some of the lower-level programming details associated with network programming. Figure 2-7 shows some of the classes involved with this class category. The SOCK_Stream component provide reliable, connection oriented byte streams. The SOCK_Acceptor class provides provide connection establishment functionality commonly used by server processes. The SOCK_Connector allows for active connection establishment functionality which is normally used by client processes.

2.1.2.6 Concurrency Class Category

ACE provides a set of C++ wrappers that encapsulates Solaris and POSIX Pthreads,

Figure 2-7 Some Participants in the SOCK_SAP Class Category

multi-threading and synchronizations mechanisms. These ACE wrappers provide an OO interface to these mechanisms and help automate the initialization of synchronization objects.

2.2 The Common Object Request Broker Architecture

CORBA is an open standard for distributed object computing [13][36]. The CORBA standard defines a set of components that allow client applications to invoke operations with arguments on object implementations that may be in the same process or remote on another machine thousands of miles away. Invoking operations on CORBA object implementations is like invoking operations on normal C++ objects, but unlike C++ objects, pointers to objects cannot be passed. CORBA defines a Core Object Model that isolates

Figure 2-8 Components in CORBA distributed object computing

the requestors of services (clients) from the providers of services (referred to object implementations in the CORBA world) by a well defined encapsulating interface. This means that clients are isolated from the implementation details of services such as data representation and executable code. The use of objects in CORBA naturally accommodates heterogeneity and autonomy: Heterogeneity since messages sent to objects depend only on their interfaces, autonomy because object implementations can change transparently, provided they maintain their interfaces [37].

In the traditional client/server model, when a client and a server need to interact, the communication aspect is handled by these two participants. This results in a software system that suffers from several dependencies and limitations. For example, the system becomes dependent on the communication mechanism used and clients need to know the

location of server(s). One solution to overcome these limitations would be to decouple clients and servers in their interactions and that is what the CORBA architecture framework attempts to do. The CORBA architecture uses the broker pattern which requires that servers registers themselves with a broker component and make their services available to clients through well defined interface methods. In CORBA, object interfaces are described in IDL (Interface Definition Language), a declarative language with syntax resembling that of C++. IDL provides basic data types (such as short, long, float, double, and boolean), constructed types (such as struct and discriminated unions), and templates types (such as sequence and string). These are used in operation declarations to define arguments types and return types. In return, operations are used in interface declarations to define the services provided by objects.

Clients invoke operations on the object implementations by sending requests in the form of messages via the broker component to the object implementation. The broker component is in charge of locating the relevant server implementation, forwarding the request to the server and returning the results and exceptions back to the client.

Figure 2-8 shows the primary components in the CORBA architecture. The function of each component is described below [1].

- **Object Implementation (or server):** This defines operations that implements a CORBA IDL interface. This is done by defining data for the object instance and code for the object's methods. Object implementation can be written using C, C++, Java, Smalltalk, or Ada. The term server is not meant to restrict implementation to situations in which method invocations cross address space or machine boundaries. A variety of objects can be supported, including separate

servers, libraries, a program per method, an encapsulated application etc.

- **Client:** This is the entity that issues requests to an object implementation. The client does this by forwarding the request(s) to the broker component which in turn locates and transmits the request to the object implementation. After execution at the object implementation, the broker return the results from the server back to the client. Clients do not need to know the locations of the servers it is issuing request to. This allows for the addition of new services or the relocation of existing servers to a new location, even at runtime. In the CORBA architecture, there is no restriction that servers cannot issue requests to other servers (i.e become a client) or that clients cannot themselves become servers even during runtime. This somewhat deviates from the traditional notion of a client/server system.
- **Object Request Broker (ORB):** The ORB is the intermediary between the client and the server that are communicating. It is the glue that allows the decoupled clients and servers to inter-operate. The ORB is responsible for receiving requests from the client component, locating an object implementation, transparently activating it if necessary, forwarding the client's request to the implementation and returning the results (if any) to the client. There is a wide variety of object implementations possible within the Common ORB Architecture[12]. These are:
 - (1) Client and Implementation - resident ORB: Here, an ORB is implemented in routines resident in the clients and object implementations. The stubs in the client (the layer between a client and the ORB) can either use a

location-transparent IPC mechanism or directly access a location service to establish communication with the implementations. Code linked with the implementation is responsible for setting up appropriate databases (implementation specific) for use by clients;

- (2) Server-based ORB: This type of implementation centralizes the management of the ORB by having clients and implementations communicate with one or more server ORB's whose job it is to route requests from clients to implementations. This ORB could be a normal program as far as the operating system is concerned, and normal IPC could be used to communicate with the ORB;

- (3) System-based ORB: To enhance the security, robustness and performance, the ORB could be provided as a basic service of the underlying operating system. Object references could be made un-forgable, reducing the expense of authentication on each request. Because the operating system could know the location and structure of clients and implementations, it would be feasible for a variety of optimizations to be implemented, for example, avoiding marshalling when both are on the same machine;

- (4) Library-based ORB: For objects that are lightweight and whose implementation can be shared, the implementation might actually be in a library. In this case, the stubs could be actual methods. This eliminates the need for marshalling if both the client and implementation are on the same machine. This type of implementation assumes that clients are not malicious. The broker component offers both clients and servers an Application Programming Inter-

face (API) that include operations for registering servers and invoking requests. When a request arrives for a server or object implementation, the broker passes the request directly to the server. If the server is currently inactive, the broker activates it. The results and exceptions (if any) from a service execution are transmitted by the broker to the client that originally issued the request.

- **CORBA stubs:** This is the layer between the clients and the broker component. This layer is necessary to provide transparency in that a non-local object appears to the client as a local one. It hides the marshalling of parameters and results, the creation and deletion of memory blocks, and the IPC mechanism when messages are sent back and forth between a client and the broker component. These stubs are automatically generated from the CORBA IDL interface by a CORBA IDL compiler. The use of the compiler eliminates the manual routine marshalling, reduces the potential for inconsistencies between client stubs and server skeletons and increases the chances for automated compiler optimization.
- **CORBA skeletons:** This is the equivalent to the client-side stub. The difference is that they are responsible for receiving requests, unpacking incoming messages, un-marshalling the parameters, and calling the appropriate service.
- **Dynamic Invocation Interface:** This interface allows a client application invoke operations on objects without having compile-time knowledge of their interfaces. An example is a graphical user interface (GUI) that allows a user to browse various drawable object's Interface Repositories and learn about the operations allowed on each object to be used. Unlike IDL stubs (which allow

only an RPC style communication), the DII also allows client application to make non-blocking deferred synchronous (separate send and receive operations) and one-way (send only) calls.

- **Dynamic Skeleton Interface (DSI):** This is server side analogue to the client-side DII. The DSI allows an ORB deliver requests to an object implementation that does not have compile-time knowledge of the type of object it is implementing.
- **Object Adapter:** This allows an object implementation access services provided by the ORB. CORBA allows a variety of Object Adapters. Services normally provided by the ORB through an Object Adapter include: object implementation activation and deactivation, registration and de-registration of object implementations, and method invocations. All object implementation inherit from the Object Adapter class. This enables the ORB invoke operations on object implementation since it is guaranteed some minimal interfaces.
- **Object Services:** There are various services specified by the OMG [36]. They are the Name Service, Event Service, Trader Services, Lifecycle Service, Relationship Service, Persistent Service, Externalization Service, Object Query Service, Object Properties Service, Object Transaction Service, Concurrency Service, Licensing Service, Security Service, Secure Time Service, Object Collection Service, and the Object Startup Service. Services that make the greatest contribution to the capabilities of CORBA based systems are described as follows: The Name Service associates a name with an object reference that contains the location of an object implementation; The Event Ser-

vice allows the communication of a change in state of a particular object to be communicated to other objects using this object state; The Trader service enable clients to dynamically discover new applications and get information about services. The trader service is like the telephone yellow pages; The Life-cycle Service provides standard signatures for object Create, Delete, Move, and Copy - fundamental operations for any object based system; The Relationship service allows objects to be grouped; The Security Services gives answers to questions like how to obtain end-to-end security in multi-vendor object networks; The Transaction Service, and the Concurrency Service that supports it with file and resource locking, provide CORBA based systems with object-oriented access to the mainstream of business accounting systems, OLTP (On-Line Transaction Processing).

2.2.1 Client/Server/Broker Interactions

Figure 2-9 shows the interactions that take place when a server registers itself with the broker. The broker upon start-up, enters its main event loop and awaits the arrival of requests from any party wishing to use its services. Assuming that the server has been started somehow, it executes some initialization code and registers with the broker. The broker upon reception of the server registration request, extracts the necessary information such as server location, object implementation interface and location etc. and stores it in a repository database. These repositories are used to locate and activate servers. The broker then sends the server an acknowledgment indicating that the request was successful. After receiving the acknowledgment from the broker, the server enters its main loop awaiting

Figure 2-9 The interaction between server and broker when a server binds its service.

requests from clients.

Figure 2-10 shows the interactions when a client sends a request to the broker. Upon start-up of the client application, and after some initialization, the client invokes a method on a remote object implementation. The client-side stub packs all the parameters in the method call together with other relevant information into a message and transmits this message to the broker. The broker receives this request and looks up the location of the relevant server in its repository and forwards the message to the corresponding server skeleton. The server skeleton then unpacks all the parameters and other information such as the method it is expected to call and then invokes the appropriate service. On completion of the required service, the results are returned to the server skeleton which packages the result and other relevant information into a message and passes it to the broker. The broker on receiving the response, forwards it to the client stub that then unpacks the message into appropriate data sets.

Figure 2-10 Client, server, and broker interaction when a client issues a request to the broker

2.2.2 A Simple Client/Server Example Using CORBA

In this section, a simple demonstration of how a client/server application may be developed is shown [13]. The server side of our CS example based on the IDL specification is shown below. The syntax used in this example are those of ORBeline, a C++ implementation of CORBA.

```
// IDL specification for server object.
interface ServerObject {
int get_age (in string persons_name);
};
```

Using the IDL compiler provided by an ORB implementation, four files are generated. A header file named `ServerObject_interface_client.hh` is created for the client side.

```
class ServerObject: public virtual CORBA::object{
public:
    static ServerObject *_bind(
        CORBA::Environment &_env,
        const char *object_name = NULL,
        const char *host_name = NULL,
        const CORBA::BindOptions *opt = NULL);
    virtual CORBA::Boolean get_age(
        const string &persons_name,
        CORBA::Environment &_env);
};
/* File ServerObject_interface_client.hh */
```

This file contains a C++ class declaration which is a mapping from the `ServerObject` IDL

interface. This is used by the client to invoke operations on the server side object implementation. The other client file generated by the compiler is `ServerObject_interface_client.cc` that contains definitions for the method defined in the `ServerObject_interface_client.hh` header file which is actually a stub definition that is responsible for marshalling the string data type containing the persons's name when a `get_age()` method is called by the client.

On the server side, the IDL compiler generates a header file `ServerObject_interface_server.hh` which contains a C++ class declaration corresponding to the `ServerObject` IDL definition called `ServerObject_impl`. This class contains a decla-

```
class ServerObject_impl : public virtual ServerObject{
....
public:
...
    // This method has to be implemented on the Server side.
    virtual CORBA::int get_age (
        const CORBA::string &persons_name) = 0;
    // Stub generated automatically by the IDL compiler.
    static void _get_age (
        void * obj,
        CORBA::MarshalStream &stream,
        CORBA::Environment & _env,
        CORBA::Principal *principal);
};/* File ServerObject_interface_server.hh*/
```

ration for the skeleton method `_get_age()` which is called by the server-side proxy to unpack parameters associated with a method invocation from the client side. In addition, this class contains declaration for a pure virtual function for the method `get_age()` which

should be overridden in the user-defined class derived through inheritance from `ServerObject_impl` and which provides the actual `get_age()` service to clients. The `ServerObject_interface_server.cc` file contains the code generated by the IDL compiler for the skeleton `_get_age()`. The following is how an implementation might derive from the `ServerObject_impl` class to obtain a concrete class, which we call, `ConcreteServerObject_impl` and override the pure virtual function `get_age()`.

```
class ConcreteServerObject : public ServerObject_impl{
...
public:
    // invoked by the CORBA skeleton.
    virtual int get_age (
        const char *persons_name){
        int age = name_db_->lookup_age (persons_name);
        return age;
    }
private:
    // This is a pointer to a database containing the name
    // of people and their ages.
    Name_Database *name_db_;
};
```

The server needs to instantiate the `ServerObject` class and inform the ORB through the call `impl_is_ready()` that it is ready to accept requests from clients. In its simplest form, the server `main()` function can be written as follows:

```
#include <ServerObject_server.hh>
main(){
    class ConcreteServerObject * server_object =
        new ConcreteServerObject;
    CORBA::BOA::impl_is_ready();
};
```

The client of the ServerObject receives the name of the person's name whose age is wanted as an argument. The Client then invokes the method `_bind()` to locate the appropriate server object implementation and also obtain an handle on it.

```
#include <iostream.h>
#include <ServerObject_interface_client.hh>
main (int argc, char **argv) {
    // invoke the method _bind to obtain a handle pointing
    // to the object.
    ServerObject * server_object = ServerObject::_bind();
    if (server_object == NULL){
        cout << "Bind Call failed in Client" << endl;
        exit(1);
    };
    // invoke method on object.
    int age = server_object->get_age(persons_name);
    cout << "Age of " << persons_name << " is " << age;
    exit(1);
};
```

The `_bind()` call is automatically generated by the IDL compiler. If no options are passed to the call, the ServerObject implementation is located and a connection is established to it. If the ServerObject Implementation cannot not be located for whatever reason, the `_bind` operation fails and returns NULL. If the bind call is successful, the client can now invoke on the Server Object by using the C++ object returned by `_bind`.

2.3 Threads

A process is a collection of resources that enable a program to execute. In traditional UNIX systems, each process contains only a single thread of control. A process can

be defined by the resources it uses (such as CPU time, memory, files, and I/O devices) and by the location at which it is executing. Many applications are difficult to build using single-threaded processes. An example of this is that of a single threaded file server where all other clients requests would have to wait while the server process retrieves a file from the disk on behalf of the current client. This can lead to a very poor response time (as seen by the clients) since disk operations are time consuming and no other processing can take place while the disk retrieval is going on. One solution is to have multiple threads in the same process. A thread, some times called a lightweight process (LWP), is an execution

Figure 2-11 Types of threads in a typical operating environment

path within a process that can execute concurrently with other threads within the process. Threads are called lightweight because the context switching associated with it is negligible when compared to that of separate processes. Modern operating systems like Windows NT, UNIX, and OS/2 support the execution of several threads in a single process. Threads are similar to processes in some ways. For example, threads can be in one of several states: ready, blocked, running, or terminated. Like processes, threads share the system resources like CPU, memory, etc. A thread within a process has its own stack and program counter. However, unlike processes, threads are not independent as they share the memory space of the process to which they belong. Multi-threading becomes very useful in cases where the machine on which a server process runs has multiple processors since cpu computations relating to several client requests can take place in parallel.

Threads are used in the development of the broker component of the CBS prototypes to avoid the problem of blocking at the broker that would occur while it awaits the reply from the server, if a single threaded broker was for example used to forward a request to the server on behalf of a client. In addition, they allow the use of both blocking send and receive calls, which helps in the design since multiple broker activities (such as forwarding, binding, etc.) can proceed independently.

There are basically two types of threads, namely kernel threads and user-level threads (see Figure 2-11). Most modern operating systems support kernel threads (such as Mach and OS/2 systems). User threads on the other hand, are implemented in user space through a set of user-level library calls rather than via system calls. Since user-level threads do not involve the kernel, switching can be done independently of the kernel and therefore, faster. Thus blocking a thread and switching to another thread can be done very

efficiently. User level threads do have their disadvantages. For example, if a user level thread blocks waiting for an I/O event to occur, the whole process in which the thread is executing blocks.

2.4 Software Modeling Methods

Software performance models, used to study the performance characteristics of software systems can be classified into two types: analytic models and simulation. Analytic models do not allow a detailed representation of a real system and hence are more abstract and need more interpretation of the results. In addition, relatively fewer systems can be effectively modelled accurately using the analytic method. Their advantage is the fact that they can be solved fast and at a relatively low cost. Examples of analytic models that can be used in predicting the performance of software systems are Queueing Network(QN) Model [31], Stochastic Rendezvous Network Model(SRVN) [32], Petri-Net(PN) Model [33].

Simulation is the imitation of the operation of a real-world system over time. It involves an artificial generation of the history of the real system, and the observation of that artificial history to draw inferences concerning the operating characteristic of the real system. Simulation allows a detailed model of a system to be built and hence performance predictions are very accurate. However, they do require a lot of computational power and are relatively slow. An example of a simulation tool that can be used for software performance prediction is PARASOL [25][26].

In this thesis, we will use simulation to predict and compare the performance of several CBS systems which might have comparable performance characteristics.

The PARASOL simulator is the tool chosen to model the CBS system. The reasons for this choice is that PARASOL is primarily suited to modeling distributed systems and it is readily available in the department.

2.5 PARASOL

PARASOL is a discrete event simulator developed by Prof J.E Neilson at the Computer Science department, Carleton University. PARASOL is well suited to the modeling of distributed computer systems. The PARASOL kernel is written in C which can be used by both C and C++ programmers in their modeling of systems. PARASOL also consists of C++ wrappers that facilitate the use of the same functionality by C++ programmers.

From a modeler's point of view, PARASOL can be thought of as a primitive distributed operating system that provides a number of services which are accessible for modeling software systems. PARASOL supplies calls that support task management, inter-task communication and task synchronization. PARASOL also provides calls that facilitates the collection and reporting of statistics of the system under study.

PARASOL contains component constructors that allow a systems's hardware and/or architecture to be captured in its language. These components are briefly presented in the following subsections:

2.5.1 Hardware

PARASOL allows a system's hardware such as buses, links, nodes and CPUs to be defined through various constructor calls. These calls are `ps_build_node`, `ps_build_link` and `ps_build_bus`. `ps_build_node` allows a user specify the number and speed of CPU's on a

particular node together with the scheduling discipline. The scheduling discipline for the ready to run queue of the CPU's are first-come-first-served (FCFS), head-of-the-line (HOL) and preempt/resume priority (PR) and round-robin (RR). Buses and links can be constructed through the calls `ps_build_bus` and `ps_build_link` respectively. The queuing discipline for PARASOL links and buses can be made FCFS, although random queueing is available for buses to approximate to some extent the CSMA/CD style of bus arbitration.

2.5.2 Task Management

System software tasks can be modelled directly in PARASOL. The user is responsible for supply the code to be executed by the task. User defined tasks are started through calls `ps_create`, `ps_create2`. After tasks have been created, they can be manipulated through calls such as `ps_sleep`, `ps_hold`, `ps_awaken` etc. Virtually any system component, even nodes and buses, can be modelled using PARASOL tasks.

2.5.3 Inter-task Communication

PARASOL inter-task communication is modelled after Mach's [23] port based communication. In other words, when a task is sent a message, the message is actually placed in a port and the task to which the message was directed has the responsibility of retrieving the message through one of the four message retrieval calls provided in PARASOL. The call to use depends on the type of port. All created tasks, by default, own one port known as the standard port. The number of ports that a task owns can be increased through a number of PARASOL calls. Seven system calls are available for sending messages between tasks

on the same node as well as tasks on different nodes connected through a link or bus.

2.5.4 Task Synchronization

PARASOL provides a number of task synchronization mechanism such as spin-locks and counting semaphores which can be used directly or can be used to construct a more sophisticated synchronization scheme.

2.5.5 Statistics Capturing

By default, PARASOL defined hardware device utilization statistic such as the CPU, bus, and link utilizations are captured automatically. This feature can be turned off by supplying an appropriate flag when these devices are being defined. PARASOL also allows for user defined statistics to be conveniently captured and stored. Three types of user defined statistics are allowed. One type, referred to as VARIABLE, is used when a user is interested in tracking some system variable over the duration of the simulation and measuring its average value (i.e., queue length, and CPU utilization). The second type, known as SAMPLE statistic, is useful in cases where a user wants to find sample means of some population observations over the course of the simulation. Examples of these are mean response time, and mean waiting time. The third type, known as RATE, is used when a user is interested in the rate of occurrence of some event. Examples of these are the rate at which a client completes a cycle (client throughput).

2.5.6 Structure of a Typical PARASOL model

Any PARASOL model consists of a driver that is responsible for running the model

according to the user's wish, an execution environment comprised of a set of hardware components that the user would have created through PARASOL constructors, and a set of user-defined tasks that run within the execution environment. The execution environment on which all PARASOL tasks run is constructed using processing nodes, one-way communication links and multi-way communication buses. In most cases, PARASOL hardware components are sufficient to model systems. However, should there be a need for more elaborate components, it is the user's responsibility to build upon those provided by PARASOL.

2.5.7 System Modeling using PARASOL

The first task in building a PARASOL model of any system is to identify the software elements and hardware devices that make up the system. Examples of system software elements to look for in this stage are threads, interrupt service routines, servers (i.e., file server), processes e.tc. Once these tasks have been identified, they can be easily mapped onto their PARASOL equivalent and even in some cases, there exist a one to one mapping. The hardware components of most system under study have a corresponding PARASOL equivalent (i.e., CPU's, bus, links and nodes).

Other system components that cannot be directly identified as software tasks or that have no PARASOL hardware equivalent can be mapped onto one or more PARASOL tasks. An example of this is the disk drive of a file server. One way to model the disk drive onto PARASOL tasks is to have two tasks. The first task, which will be referred to as the *disk server* task, accepts disk requests and passes it on to the second task, the *disk drive* task. The disk server task is also responsible for sending a reply to the disk request. The

second task models the actual disk service delay associated with satisfying disk requests and changing the head position. Choosing two tasks to model the disk drive makes it easy to develop the logic associated with these essentially asynchronous activities.

2.6 Measurement Tools

In order to perform measurements of the CBS prototypes, we used the measurement tool DECALS (Distributed Experiment Control and Logging System) to setup and run the experiment, the UNIX measurement function, *getrusage()* (for cpu consumption), and the UNIX command *netstat* (for bus speed measurement).

2.6.1 DECALS

DECALS [27] is a collection of libraries that enables a user to run and control the execution of a distributed system. An experiment implies the distribution of application processes across network nodes, their execution until completion (the meaning of completion is determined by the user) and the gathering of measurements of specific interest. DECALS starts of by reading in a set of configuration files, a list of experiments and one or more experiment description files. These files are written by the a user and they describe to DECALS what experiments are to be run and how their execution is to be controlled. It is up to the user to include in his/her application, code that captures measurements of interest. DECALS provides a number of interfaces that facilitate the logging of user defined information.

The DECALS system is comprised of a number of processes that are distributed over a network of workstation nodes (see Figure 2-12). There is one Experiment controller

process per experiment. This process is responsible for running and initializing all other processes. The set of processes called RPTD is used by the Experiment controller to invoke the remaining processes during initialization phase. The group of processes called Application Processes is the set of processes on which measurement and analysis is being performed. The application processes communicate with the Experiment Controller through a set of interfaces provided by the CEC library and with the Collector through a set of interfaces provided by the COP library. DECALS allows as many processes as the workstations can handle to be running on any node. The Collector process collects mea-

Figure 2-12 Processes in the DECALS system

surements from the application processes and stores them in a file. Only one Collector may be running on each node in the network which is used by an experiment.

2.6.2 Getrusage

UNIX provides a system call known as *getrusage()* that allows information about the resources utilized by the current process, or all of its terminated child to be known. In this thesis, the resources of interest is the total CPU used by the current process in both kernel space and user space.

2.6.3 Displacement Technique

Work done by Marc Courtois based on Woodside's displacement technique [34] reveal that the CPU consumption returned by *getrusage* represents only a part of the total CPU resources used on behalf of a process, when it sends or receives messages using the ACE stream objects which relies on the TCP/IP protocol families. This is because of the fact that when a user process uses a communication mechanism based on the TCP/IP protocol in exchanging information between a peer, only some of the cpu resource consumed in the kernel space (due to ISR activities in the kernel) is charged to the user process. Therefore, when *getrusage* is used in the user process, it reports only part of the cpu resource consumed by the user process. The displacement technique captures all the cpu resource consumption that is related to a user process's send and receive operations.

The send and receive cpu resource consumption, which are used as input parameter into the simulation models, were those reported by the displacement technique.

2.6.4 Netstat

Netstat is a UNIX command that reports the status of the network. If run with the *-i* option, it shows the number of packets sent and received at workstation node over a specified

period of time for TCP/IP packets. The output statistic of netstat can be used to measure the effective speed of the bus as seen by an application.

Chapter 3

Client/Broker/Server models and Broker Designs using ACE

As discussed in chapter one, the forwarding broker model suffers from excessive network traffic and has the potential for a bottleneck at the broker component. This chapter shows alternative models that help eliminate excessive network traffic and also reduce the potential for bottleneck at the broker.

Software developers in an attempt to improve performance of their system usually have to decide whether to use multi-threading or use a single thread of control. This chapter shows various implementation of the broker components using a single threaded model and some multi-threading models, namely thread-pool and thread-per client. In the single threaded broker design, a reactive technique was employed with the help of the ACE Reactor framework. The Reactive dispatching uses a single thread of control using non-blocking I/O. This chapter also shows how the broker component was built using components provided by ACE.

3.1 Broker models

A broker acts as an intermediary between client applications that need service and server applications that are willing to provide these service. Brokers relieve clients from having to maintain information on where and how to obtain particular services. All server applications wishing to allow clients use its service register themselves with the broker.

Figure 3-1 depicts the forwarding broker model. Here, a broker sends a request to

Figure 3-1 Client-Broker-Server interactions in a forwarding broker model

the relevant server application, retrieves the response from server, and relays this reply back to the client. This is the approach adopted by the OMG CORBA. This model has an advantage in that it naturally allows for load balancing. Furthermore, all failed transactions can be detected and restarted centrally. A complete client interaction with the server object requires a total of four messages. As explained in chapter one, this model can lead

to excessive network traffic in cases where clients do not reside on the same node as the broker component and the object implementation (server), since four network messages would be needed to complete a client interaction. At the other extreme case, also shown in Figure 3-1, we can have a scenario in which the client, broker and server all reside on the same workstation node. In this situation, a client does not need any network messages to complete an interaction (but still needs a total of four messages to complete an interaction). However, as the number of nodes in the network increases, majority of client/broker/server interactions would require four network messages, which could lead to a degradation in performance since networks are usually slow.

Another problem with this model is that all client-server interactions must pass through the broker component which creates a potential for a bottleneck at the broker. One way to solve both these problems is to distribute brokers across workstations nodes in the network that have either a client or server. The local broker component on each machine is responsible for forwarding a client's request to the appropriate server or object implementation. This model, which will be referred to as the *distributed forwarding* broker model, provides a double edge sword in solving both these problems. Firstly, network traffic is reduced in this model since a client's request to the broker is done through a local IPC mechanism as opposed to a network message of the non-distributed version where the broker may exist on another machine. As can be seen in Figure 3-2, the number of network messages is at most two no matter the locations of the clients, servers and brokers. Secondly, the potential for a bottleneck at the broker is reduced since we now have more than a single copy of the broker in the system.

Figure 3-3 shows what happens when a server advertises its services by issuing a

Figure 3-2 Shows the interaction when a client makes a request in the distributed forwarding Broker Model.

bind operation. The server sends a request to its local broker, which in turn sends this bind requests to the other brokers in the group to update their database. One disadvantage with the distributed broker model is that the time for servers to bind could increase depending on whether the broker that initially receives the bind request sends a copy of the bind information using unicasting, multicasting or broadcasting. This model takes advantage of the fact that binds in practice occur much less frequently than client requests.

Figure 3-3 A server issues a bind operation in a distributed forward broker model.

Another way to have this bind operation done, is to have the other brokers in the group discover that a new server has advertised its services (which will happen when a client local to the broker issues a request for service from this new server). This approach is more prone to failures of one or more of the brokers.

An alternative to reducing the excessive network traffic of forwarding broker model is to have the server return the reply directly to the client, rather than having the reply go through the broker as was the case in the forwarding broker model. This model, which will be referred to as the *half-forwarding* broker model, is depicted in Figure 3-4.

Figure 3-4 Client -Broker -Server interaction in a half forwarding broker model.

This model has one of the advantages of the forwarding broker model in that it allows for load balancing per request. As can be observed, a maximum of three network messages is needed to complete a client interaction with a server. The total number of messages needed to complete client interactions is also three which represents a significant reduction when compared to the number needed in a forwarding broker model, which requires a total of four messages. Furthermore, the potential for a bottleneck at the broker (compared to the forwarding broker model) is reduced, since the server reply bypasses the broker.

To further reduce the network traffic of the half-forwarding broker model, the bro-

Figure 3-5 Client-Server-Broker interaction in a distributed half-forwarding broker model.

ker component can be distributed across the workstations in the network that have client(s) and/or server(s). This model, which will be referred to as the *distributed half-forwarding broker model* is shown in Figure 3-5. As can be observed, the maximum number of network messages needed to complete a client's request is two. The total number of messages to complete a client request's (three), however remains the same as the non-distributed version.

The interactions that take place when a server issues a bind request follow the same pattern as that of the distributed-forwarding broker model.

Figure 3-6 Client-Broker-Server interaction in a handle driven broker model.

In the handle driven broker model, the broker returns a service handle back to the client, which contains all the information required to interact with the server for the given service (such as its name, the address of its network node and the request format). The client uses this handle to issue requests directly to the relevant server (object implementation) which in turn replies directly to the client. This interaction is depicted in Figure 3-6. This model has one main advantage when compared to the other broker models discussed so far in this chapter, which is that the broker component is bypassed in a client-server

Figure 3-7 Client-Broker-Server interaction in a Distributed handle-driven broker model.

interaction. Hence, the potential for a bottleneck at the broker is greatly reduced. However, the task of detecting and recovering from a fault is placed on the client applications. Furthermore, the model does not allow for load balancing.

Lastly, Figure 3-7 shows a distributed version of the handle-driven broker model. Here, a client can store handles to the object implementation(s) for later reuse and hence client/broker interaction take place only once. Hence, this model was not considered in the performance comparison done which is discussed in chapters 4 and 5, since it effectively

boils down to client/server model after the initial client/broker interaction. This assumes that the object implementation does not crash for whatever reason, which is the case in our performance comparison, and hence there is no need to obtain the handle from the broker again after the first time.

3.2 A Group Communication Architecture in CBS systems

One way that could potentially lead to improvement in performance of CBS systems is through the use of object replication. Here, multiple copies of an object implementation are distributed across workstations of the network on which there exist at least one client using it. The states of these objects are kept consistent through the use of server group communication. An operation that will cause the state of an object to be modified is called a write operation. Similarly, an operation that will not cause a modification of the object implementation is called a read operation. One advantage of replicating an object implementation on the relevant workstations is that reads can be done locally, thus avoiding any network traffic or delay, and therefore faster responses. Another advantage of replicating object implementation is that multiple reads accesses on an object can take place in parallel. The main disadvantage with replicating object implementation is the overhead of keeping the states consistent. Hence, for group communication to be effective as far as performance is concerned, the read/write ratio must be high enough.

Both the distributed forwarding and half-forwarding broker models accommodate object replication naturally since the local broker component could take on the additional task of deciding whether or not to send a client request to the local object implementation (in the case of a read type request) or to all object implementations (including the local

object implementation for write requests). They also take advantage of that the fact the all reads can be done locally (since both the broker and object implementation(s) reside on the same machine as the client application).

Figure 3-8 shows the interactions that take place when a client issues a write request to the server object via its local broker. Although not shown in the diagram, one of the local broker components in the broker group takes on the added responsibility of a sequencer that provides global ordering. The other brokers in the broker group obtain a sequence number from the broker acting as a sequencer, and append this number when it forwards a client's write type request to all replicas of the object. Whenever any of the object implementation replicas notices a gap in the sequence number as a result of a receiving a new client write request, it delays this update until the remaining write messages necessary to fill in this gap comes in. This scheme provides consistent time global ordering [24] for write requests. The processing of read requests (which takes place locally) are also delayed, if a gap is noticed in the write updates, until the write message necessary to fill in the gap arrives. This ensures ordering for write and read operations taking place locally on a node. The advantage of this is the performance gain obtained from doing reads locally.

Maeffis in [10] shows how the BOA of CORBA implementation might be modified to incorporate object replication through the use of group communication. The first object implementation creates a new object group through the use `BOA::create_group` and invokes the method `join()` of the newly created object implementation to join the group. Subsequent object implementations can join this newly created group by obtaining the group reference through the `BOA::resolve_group` method, and then invoking its `join()`

Figure 3-8 Shows the interaction that takes place when a client makes a write request in the distributed forwarding broker model with replicated object implementations. The state is kept consistent through the use of group communication.

method.

When there is a need for an increase in the degree of replication of an object implementation, the broker takes on the responsibility of creating and adding the new object to the group. It then obtains the internal state of an existing object implementation by invoking

```

class BOA {
public:
    // Standard BOA interface specified by CORBA
    .....
    // Interface allowing group communication to take place
    // transparently.
    static void create_group (Obj_ptr grp, Environment_ptr = 0);
    static void resolve_group (const String &grp_name, Obj_ptr &grp);
    static void bind_group (String &grp_name, const Obj_ptr &grp);
    void join (Obj_ptr group, Environment_ptr = 0);
    void leave (Object_ptr, Environment_ptr = 0;)
    static void destroy_group (Obj_ptr grp, Environment_ptr = 0);

    virtual void get_state (AnySeq& state, Environment_ptr);
    virtual void set_state (const AnySeq& state, Environment_ptr);

};

```

ing its `get_state` method and marshalling it into an CORBA object of type `AnySeq`. This state is then transferred to the newly created object implementation and unmarshalled followed by the invocation of the new object implementation's `set_state` method. It is the programmers responsibility of writing the application specific `get_state` and `set_state` methods. A simple example of how the `set_state` and `get_state` method [10] can be written is shown below.

```

// A simple example of how the get_state and set_state method might
// be written.
void ServerObject_impl::get_state (AnySeq& state){
    // Pack state of object into the state data object.
    state[0] <<= object_state_1;
    state[1] <<= object_state_2;
    state[2] <<= object_state_3;
};

void ServerObject_impl::set_state (AnySeq& state){
    // Unpack the received state of the object.
    state[0] >>= object_state_1;
    state[1] >>= object_state_2;
    state[2] >>= object_state_3;
};

```

```
// Server side implementation
// Instantiate an object implementation
Object_Implementation impl;

// declare a group reference
Group_Reference grp;

// Create an object group.
BOA::create_group(grp);

// Place implementation into group.
impl.join(grp);

// Install the group reference
BOA::bind_group (grp_name, grp)

// Client Side
// invoke the method _bind to obtain a handle pointing to the
object.
ServerObject * server_object = ServerObject::_bind();
if (server_object == NULL){
    cout << "Bind Call failed in Client" << endl;
    exit(1);
};

// invoke method on object.

int age = server_object->get_age(persons_name);

cout << "Age of " << persons_name << " is " << age;

exit(1);
```

3.3 Broker Functionality

The basic functionality provided by the broker implementation are described below:

- Bind - This allows an object implementation register its services with the broker. The broker uses a repository to maintain the information provided by an object implementation when it does a bind. Information kept in the repository

include the server name, the services being provided, server location (i.e., port number and the address of the network node on which the object implementation resides).

- **Rebind** - A server wishing to relocate to another location or wanting to add some other new services issues this type of request to the broker. For a rebind request, the server name remains the same plus the services that it was providing before the rebind operation (at least as far as the interfaces are concerned).
- **Unbind** - This service is used by an object implementation that wants to shut down gracefully. Clients wishing to interact with the server after unbinding are informed of the non-availability of this service. The server entry in the repository is removed.
- **Resolve** - A client wishing to communicate directly with an object implementation can do so by issuing a resolve request to the broker. In the request, it specifies the object name it wishes to communicate with. A reply is returned by the broker containing all the relevant information necessary to communicate with the object implementation.
- **Forward** - A client uses this service when it wants the broker to handle all server location and connection establishments.

3.3.1 Broker Implementation

All the broker functionality such as bind, rebind, unbind, forward and resolve were implemented in a class known as `Orb_Handler`. This class inherits from the `Svc_Handler` and the `handle_input()`, and `handle_output()` and `handle_close()` virtual functions were over-

ridden to implement broker specific functionality such as bind, unbind, rebind, resolve and forward. The broker prototype implemented via the ACE toolkit was the forwarding broker model.

Additional functions and data members were added to the derived class to achieve this. The broker implementation has a database containing the names, location (internet address and port number) of servers that have had their services bounded. A class known as `Orb_Request_Reply` was created. The purpose of this class was to provide both client, server, and broker processes a uniform way of packing data to be exchanged, be it local or remote. This class takes care of byte order issues by converting data into a standard network order through the use of the unix `hton*` and `ntoh*` functions.

Multi-threading issues were then tackled. The various threading strategies used are discussed below. Intra-nodal interprocess communication was done using the pipe mechanism since it is the most efficient form of interprocess communication, while inter-nodal communication was done via UNIX socket streams.

3.4 Server and Client Task Design and Implementation

The design and implementation of the client process, using components provided by the ACE toolkit, was straightforward. The `ACE SOCK Connector` was used for active connection establish with the broker process. After connection has been established with the broker, the client process enters an infinite loop and randomly chooses which of the server implementations it wishes its request forwarded to, and encodes this in its request to the broker. It then waits the reply from the broker (the probability of choosing any one the server implementation is equal). This is done for a number of times equal to the number of

server implementations, after which it enters a think time phase by remaining dormant using the UNIX `sigset()` call together with the `setitimer()` call. All the above steps (from the point of sending its requests to the broker to the think phase) is repeated in an infinite loop. The client process are identical across the various broker implementation.

The server process implemented was single threaded and uses the Reactor component to listen for connection establishment, and forwarded requests from the broker process. For connection events, the server uses an extended version of the `ACE_Acceptor` (`Server_Acceptor`) to create a `Server_Handler` class and registers this with the Reactor object, so that when events arrives, the appropriate `Server_Handler` is dispatched automatically by the Reactor object. The `Server_Handler` object `handle_input()` method (called by the Reactor object when requests come in) receives the request, does some CPU spinning and returns a reply to the process (i.e., broker) that sent the request. The server process are identical across the various broker implementation.

3.5 Multi-threading Models in the Broker Design

One way to improve performance of a system is through the use of multi-threading.

Threads provides multiple flows execution and share the same address space as the parent process. A broker process may contain many lightweight threads. To this end, the thesis investigates the performance of various threaded broker implementations. The threading paradigm looked at in this thesis are the Thread-Pool, Thread-per Client and the Single-thread-Reactive models. Multi-threading the broker component helps simplify the broker design since every request can be handled by a separate thread running in the broker process. This also helps in the abstraction of the design.

3.5.1 Thread-per-Client Broker

This thread paradigm handles each client request in a separate thread of control over the duration of the conversation. It is useful in cases where a client communicates with broker component over a long period of time such as forwarding requests to an object implementation. It amortizes the thread creation costs over several multiple request per client.

- Thread-per-Client Broker Implementation

The implementation of this design was done using the components provided by the ACE toolkit. Figure 3-9 depicts the ACE components used in the implementation of the broker component. The thread-per-client design used by Schmidt [18][19] was used as a guide in the design.

- **Reactor** - In this model, an instance of this class is responsible solely for the initial connection establishment between the clients and servers to the broker, after which each individual thread listens for incoming requests.
- **Orb_Handler** - This class inherits from the ACE_Svc_Handler base class in ACE. An ACE_Svc_Handler defines a generic class for a communication service that exchanges data between peers over communication networks. The Orb_Handler extends this base class interface by defining additional methods and data members. One important additional data member is the forward_stream object member that is used in the forwarding broker model. This is used by the broker to connect the appropriate server implementation for forwarding purposes. An instance of the Orb_Handler class is created and used when requests come in from clients. It receives requests from both object implementation and clients through the reply_stream and forward_stream

Figure 3-9 Thread-per-Client Broker model using ACE Components

objects respectively. Requests types that it handles are bind, rebind, unbind, resolve and forwarding requests. When requests do come in, its type is determined and the appropriate method is called.

- **Orb_Acceptor** - This class inherits from the ACE_Acceptor. This class is a factory that implements the strategy for passively initializing an Orb_Handler. An instance of this class registers itself with the Reactor object that listens for

incoming connection requests using the select system call. The broker now goes into an infinite loop accepting clients connection requests and creating Orb_Handler's. When connection requests come in, this instance of Orb_Acceptor class is responsible for creating a new Orb_Handler object, accepting the new connection into it and activating the Orb_Handler by calling its open method. The open method of the newly created Orb_Handler spawns a thread that handles both clients request and server replies to the client.

- Orb_Request_Reply - This class is responsible for packing all parameters to be sent between brokers, clients and servers. It has a fixed interface but it is overloaded to allow for only a subset of parameters to be packed. It mimics the marshalling that takes place in a full blown implementation of a broker system.

3.5.2 Thread Pool Broker

The thread- pool model spawns a fixed number of threads at start up to handle incoming requests. Client and server requests and replies can execute concurrently until the number of simultaneous requests exceeds the number of threads in the pool. When this point is reached, additional requests are queued until a thread becomes available. This model avoids excessive resource consumption associated with the thread-per-client model when the number of requests become very large.

- Thread Pool Broker Implementation

The thread-pool design used by Schmidt [17] was used as a guide in the design. This broker implementation uses five main class. These are described below:

Figure 3-10 The Thread-per-Pool model using components provided by ACE.

- **Reactor** - Initially, the Reactor listens on a single file descriptor (which is that of the Orb_Acceptor) and awaits connection requests from clients and servers. Only the main thread of the broker uses the Reactor in this thread-pool design. When connections requests arrive on this file handle (known to clients and

servers as a well advertised port number), the Reactor dispatches the `handle_input()` method of the `Orb_Acceptor`. This creates and associates an `Orb_Handler` object and file handle with the party requesting connection and subsequent communication with this party is done via the `Orb_Handler` object and new file descriptor. In addition, the new file handle and `Orb_Handler` is registered with the Reactor so that when messages arrive on this handle, the appropriate `Orb_Handler` object is activated (by calling its `handle_input()` method) to process the request. The `handle_input()` method of the `Orb_Handler` receives this message into a buffer in its data structure and enqueues this request into a request queue on a FCFS basis (which is actually a pointer to the `Orb_Handler` object that also contains the request). This is later removed for processing when one of the threads in the pool becomes available. When there are no more pending events, the Reactor waits for new events to occur by listening on all the registered file descriptors using the UNIX `select()` system call.

- `Orb_Acceptor` - This class inherits from the `ACE_Acceptor`. This class is a factory that implements the strategy for passively initializing a `Orb_Handler`. The `Orb_Acceptor` registers itself with the Reactor object that listens for incoming connection requests using the `select` system call. The broker now goes into an infinite loop accepting clients connection requests and creating `Orb_Handler`. When connection requests come in, this class is responsible for creating a new `Orb_Handler` object, accepting the new connection into it and activating the `Orb_Handler` by calling its `open` method (that registers the newly created `Orb_Handler` with the Reactor component).

- **Orb_Handler** - This class inherits from the **ACE_Svc_Handler** base class in ACE. An **ACE_Svc_Handler** defines a generic class for a communication service that exchanges data between peers over communication networks. The **Orb_Handler** extends this base class interface by defining additional methods and data members. One important data member is the **forward_stream** object member that is used in the forwarding broker model. This is used by the broker to connect the appropriate server implementation for forwarding purposes. The thread assigned is responsible for processing the request to completion and blocks if necessary on any file descriptor associated with the **Orb_Handler**. An instance of this class is created and used when requests come in from clients. It receives requests from both object implementations and clients through the **reply_stream** and **forward_stream** respectively. Requests types that it handles are bind, rebind, unbind, resolve and forwarding requests. When requests do come in, the **handle_input** method of the associated **Orb_Handler** is called by the main thread that enqueues the request by storing a pointer to the **Orb_Handler** object in a fixed finite buffer. When a thread pool becomes available, it dequeues the stored **Orb_Handler** object pointer and invokes the appropriate method to handle the request.
- **Orb_Request_Queue** - This class queues requests from the main thread to the pool threads on a FCFS basis. It is modelled after a fixed finite circular buffer.
- **Orb_Request_Reply** - This class is responsible for packing all parameters to be sent between brokers, clients and servers. It has a fixed interface but it is overloaded to allow a subset of parameters to be packed. It mimics the marshalling

that takes place in a full blown implementation of a broker system.

3.5.3 Single-thread-Reactive Broker

To avoid the problem of blocking in this Reactive broker, the Reactor component is used. The Reactor is responsible for managing several file descriptors corresponding to a client or server connection. The Reactor detects an incoming event, demultiplexes the event to the appropriate Orb_Handler and dispatches an application specific callback method (i.e handle_input or handle_output).

- Reactive Broker Implementation

The main class used in this single-threaded reactive broker are described below.

- **Reactor** - In this design, all processing takes place in a single thread of control. The Reactor dispatches the handle_input() method of the Orb_Acceptor when connection events arrive from the clients or servers. This creates and associates an Orb_Handler object and file handle with the party requesting connection and subsequent communication with this party is done via the Orb_Handler object and new file descriptor. In addition, the new file handle and Orb_Handler is registered with the Reactor so that when messages arrive on this handle, the handle_input() method of the appropriate Orb_Handler object is activated by the Reactor to process the request. The Router pattern is used in the Reactive broker design to prevent the broker from blocking on any single file descriptor [29][35]. This design involves the use of non-blocking send and receive operating system primitives. In this pattern, when a message is to be

sent (say a request is to be forwarded to the server), the forward operation checks to see that there is no flow control on network link to the server. If there is no flow control, the entire message is sent to the server in one send operation. On the other hand, if at some point in the forwarding of the request a flow control condition arises, the non-blocking send primitive, used in transmitting the message, does not block the entire process and returns the number of bytes of the message sent. The forward operation of the Orb_Handler makes use of the number of bytes returned by the send operation to determine what part of the message was not transmitted and makes a note of this. It then instructs the Reactor to call the `handle_output()` method of the Orb_Handler when the flow control conditions abate on the file handle associated with the network link to the server, and returns to the main event loop. When it is possible to try the send operation again, the Reactor calls the `handle_output` method() of the appropriate Orb_Handler, which then continues from the point the last send operation ended. This sequence of steps may be repeated until the entire message is transmitted successfully. After the entire message is transmitted, the forward operation now registers the file descriptor associated with the `forward_stream` object contained in the Orb_Handler with the Reactor, so that when replies come in from the server, the Reactor can detect this and call the `handle_input()` method of the appropriate Orb_Handler. These messages are received in fragments using non-blocking operating system receive primitive, and if an entire message is not immediately available, the `handle_input()` method buffers the fragment and returns control to the event loop. When the

other parts of the message arrives, the Reactor detects this and calls the `handle_input()` method of the appropriate `Orb_Handler` to receive them. These steps are repeated if necessary until the entire message is received. Using these strategies, it is guaranteed that the process would not block on any single file descriptor.

- `Orb_Acceptor` - This class inherits from the `ACE_Acceptor`. This class is a factory that implements the strategy for passively initializing a `Orb_Handler`. The `Orb_Acceptor` registers itself with the Reactor object that listens for incoming connection requests using the `select` system call. The broker now goes into an infinite loop accepting clients connection requests and creating `Orb_Handler`'s. When connection requests come in, this class is responsible for creating a new `Orb_Handler` object, accepting the new connection into it and activating the `Orb_Handler` by calling its `open` method which registers the newly created `Orb_Handler` with the Reactor component.
- `Orb_Handler` - This class inherits from the `ACE_Svc_Handler` base class in ACE. An `ACE_Svc_Handler` defines a generic class for a communication service that exchanges data between peers over communication networks. The `Orb_Handler` extends this base class interface by defining additional methods and data members. One important data member is the `forward_stream` object member that is used in the forwarding broker model. This is used by the broker to connect the appropriate server implementation for forwarding purposes. An instance of this class is created and used when requests come in from clients. It receives requests from both object implementation and clients through the

reply_stream and forward_stream respectively. Requests types that it handles are bind, rebind, unbind, resolve and forwarding requests. When requests do come in, its type is determined and the appropriate method is called to handle the request.

- Orb_Request_Reply - This class is responsible for packing all parameters to be sent between brokers, clients and servers. It has a fixed interface but it is overloaded to allow for only a subset of parameters to be packed. It mimics the marshalling that takes place in a full blown implementation of a broker system.

Figure 3-11 The Single-Threaded Reactive Broker model using ACE components.

Chapter 4

Measurements

This chapter presents measurement and performance results of the CBS prototypes described in the previous chapter. The performance results are used in the validation process of the CBS simulators. The first section presents the results of the measurements obtained for components that make up the CBS system. Measurements results are given for the Reactor component, bus sub-system, thread synchronization (for the case of thread-pool designs), and the send and receive methods used in the CBS system. Section 2 presents the performance results of the forwarding broker obtained by measuring the prototype system.

4.1 Characterization of the CBS Components

The components used in the forwarding broker of the CBS system were measured for use as input parameters into the simulation models of the CBS system that were to be built. The hardware configuration used in the execution of the CBS prototypes were four *Sparc2* workstations and one *SparcIPC* workstation running on a 10Mbits/sec ethernet network. Measurement results presented in this section are for the Reactor component, thread synchronization, send and receive operations (for both ACE_SOCKET_Stream and pipes), and the bus sub-system. The 95% and 99% confidence intervals of these measurements are within 2.92% and 4.67% of the measured values unless otherwise stated.

4.1.1 Reactor Component

The principal methods in the Reactor class (as far as the CBS system built is concerned) are the `handle_events()`, the UNIX `select()`, `dispatch()`, `register_handler()` and `remove_handler()` functions. In the broker process, the Reactor's `handle_events()` is used in the event demultiplexing. The methods that are called when this is done are

- *handle_events()* - This is called in the broker component's main function. It calls in turn two important functions, namely the `wait_for_multiple_events()` and the `dispatch()` function. From studying the Reactor code, it can be seen that the UNIX `select()` call is the only method in `wait_for_multiple_events()` method whose execution times would vary with the number of file descriptors. Hence, execution times can be decomposed into two types (for convenience): one which is dependent on the number of file descriptors and the other which is independent of the number of file descriptors. Therefore, the cpu times consumed in the `wait_for_multiple_events()`, other than the `select()` call, is taken to be consumed directly in the `handle_events()`.
- *select()* - The ACE Reactor class uses the UNIX `select()` call to demultiplex incoming events. It allows a process to wait on multiple file descriptors. This function was instrumented to examine how the cpu times would vary with the number of file descriptors.
- *dispatch()* - this method is called in the Reactor's `handle_events()` method when event(s) occur on the file descriptors waited on in the `select` call. It is responsible for dispatching the appropriate timer and I/O event handler. A analysis of the dispatch code reveals that the cpu times that would be con-

sumed by this method would vary with the number of file descriptors.

- *register_handler()* - this method of the Reactor is used to instruct the Reactor to execute certain methods in an event handler (which is one of the arguments to this function) when certain events occur.
- *remove_handler()* - instructs the Reactor to stop responding to events associated with a particular handler (that must have be previously been registered with the Reactor). When this method is called, the Reactor removes this handler from its internal table.

4.1.1.1 Reactor Characterization

Figure 4-1 to Figure 4-5 presents the results obtained from instrumenting the Reactor class on all the workstations used in the experiments. The results confirm the supposition made that only the select call, invoked in the *wait_for_multiple_events()* method, would vary with the number of file descriptors. It can be seen from Figure 4-1 to Figure 4-5 that the cpu times in *handle_events()* remain almost constant as the number of file descriptors varied. Hence the simulation model assumes that the cpu time in the *wait_for_multiple_events()* method is independent of the number of file descriptors. Figure 4-1 to Figure 4-5 also shows the cpu time consumed for the *select()* and *dispatch()* methods, and an almost linear relation can be observed to exist between the cpu times consumed in these methods and the number of file descriptors.

Table 1 and Table 2 shows the measured results for both the *remove_handler* and *register_handler()* method in the Reactor class, for each machine in the network on which the experiments was conducted.

Figure 4-1 CPU Times for relevant Reactor methods on workstation helicon

Figure 4-2 CPU Times for relevant Reactor methods on workstation mira

Figure 4-3 CPU Times for relevant Reactor methods on workstation alcor

Figure 4-4 CPU Times for relevant Reactor methods on workstation sputnik

Figure 4-5 CPU Times for relevant Reactor methods on workstation bellatrix

Table 1 Reactor remove_handler() cpu times

Machine	bellatrix	sputnik	helicon	mira
CPU Time (msec)	0.356	0.322	0.258	0.263

Table 2 Reactor register_handler() cpu times

Machine	bellatrix	sputnik	helicon	mira
CPU Time (msec)	0.301	0.298	0.237	0.235

4.1.2 Bus Sub -System

The bus is one of the components in CBS systems that could have a great impact on the performance of the system. Therefore, measuring this component is of great importance in accurately predicting the performance of CBS systems.

4.1.2.1 Bus Speed Measurements

The bus speed was measured by using a UNIX system command known as netstat. This call displays how many packets have been received in a user specified interval. The command line for this call to report the network statistic for every 30 seconds is “netstat -i 30” where the option “-i” shows the statistic for TCP/IP traffic.

Since the bus used in running the experiment is not a perfectly quiet one, the assumption that the effective maximum speed as seen by the applications in the experiment is 10Mbps/sec (the speed of the ethernet bus used in the experiment) would be inaccurate. To measure the bus speed, two classes of processes were written. The first one, which will be referred to as the *sender process*, sends messages of 1460 bytes to the second class referred to as the *receiver processes*. The receiver process operates in loop and receives messages sent by the sender process. The sender process does not wait for a reply from the receiver process before sending the next message, and neither does the receiver process send a reply back to the sender. For the bus speed measurement, two sender and receiver processes were instantiated, to ensure that the maximum bus capacity is used (since flow control can take place on a per sender basis). The sender processes were run on a workstation and the receiver processes on another.

4.1.2.2 Bus Speed Measurement Results

Netstat was run before the start of this experiment to know what the network traffic was like before the run of the actual experiment (which helps to determine the background noise). Netstat was then run during the experiment to collect figures that would be necessary to estimate the effective speed of the bus as seen by future experiments. Table 3 and

Table 4 show the network traffic before and during the experiment respectively.

Table 3 Netstat report taken on receiver workstation before running experiment

Replication	Number of Input Packets	Number of Output packets
1	22	3
2	24	2
3	32	2
4	37	2
5	44	2
6	22	4
7	19	6
8	32	2
9	24	2
10	23	2
Mean	27.9	2.7

Table 4 Netstat report taken on receiver workstation while experiment was taking place

Replication	Number of Input Packets	Number of Output Packets
1	23322	11716
2	23458	11719
3	23426	11699
4	23425	11699
5	23375	11720
6	23422	11731

Table 4 Netstat report taken on receiver workstation while experiment was taking place

Replication	Number of Input Packets	Number of Output Packets
7	23462	11716
8	23386	11713
9	23435	11707
10	23383	11679
Mean	23409	11710

The measurements were taken 10 times as indicated in Table 3 and Table 4, to obtain the average and also to derive a tight confidence interval for the measurements taken. The third column in Table 4, corresponding to the number of output packets sent, must be the ACK sent in the TCP/IP layer for ensuring reliable transmission, since no reply is sent in the receiver process application space.

Comparing the number of input packets before running the experiment to that obtained during the experiment, one can see that the background traffic constitutes only 0.1% of the total number of packets when the experiment was run. Hence, one can assume that the background network traffic is negligible and was therefore ignored in the calculation below.

Even though netstat does not report the sizes of the packets, information gathered on TCP/IP and the ethernet packet structures and protocols [21][22] reveal that sending messages containing 1460 bytes of user data translates to a single packet of size 1530bytes on the network wire. Also, assuming that the packets size of the ACK's is 40bytes, the size of the header information used in the TCP/IP protocol, the effective bus speed can be calculated as

$$\begin{aligned}
 \text{EffectiveBusSpeed} &= (23456 \times 1530 + 40 \times 11713) / 30 && (\text{bytesPerSecond}) \\
 &= 1209472 && (\text{bytesPerSecond}) \\
 &= 9.6758 \times 10^6 && (\text{bitsPerSecond})
 \end{aligned}$$

This value was used as the bus speed in the PARASOL models of the CBS system built. The 95% and 99% confidence intervals, calculated using column 3 in Table 4, are 0.13% and 0.20% respectively.

4.1.2.3 CPU measurement results for Send and Receive operations

Figure 4-6 shows the cpu times returned by the displacement technique for various message sizes. Since pipes were used in the live implementation of the CBS system, instrumenting the pipe send and receive operation would also be necessary for input into the simulator. Figure 4-7 shows the CPU measurement results obtained from instrumenting the send and receive operations of pipes.

4.1.2.4 Thread Pool Insert and Remove CPU times

The cpu time necessary on enqueue(insert) and dequeue (remove) requests in the thread-pool broker design was instrumented and the results are shown in Table 5.

Table 5 Remove and Insert methods CPU times on workstation bellatrix

Average Insert CPU Time(msec)	Average Remove CPU Time(msec)
0.5325	0.6842

Figure 4-6 CPU times for send and receive methods of ACE SOCK Stream on workstation bellatrix

Figure 4-7 CPU times for send and receive methods of pipes on workstation bellatrix

4.1.2.5 Model Input Parameters

With all the important component of the CBS system characterized, the measurement results were then used as input parameters for the PARASOL models. The comparison between the actual system performance and the simulator prediction will be discussed in section 2 of chapter 5.

4.2 Forwarding Broker Performance Results

To study the effect of multi-threading and distributing the broker component in the CBS system, the broker forwarding model was used. Later when the issues of multi-threading and broker distribution have been addressed, comparison between the various broker models and architectures are looked into. This section presents only a preliminary result. The performance results presented here were primarily used for the validation of the CBS simulator which are described in chapter 5. It does not fully address the issue of multi-threading and broker distribution. Further investigation will based on the simulator whose performance results are discussed in chapter 6.

The system architecture used consists of several clients, two servers, and one or two brokers, depending on whether it is the distributed broker or the non-distributed version. The interaction pattern of the client consists of sending one request to each of the servers followed by a sleep phase. The choice of which particular server to send a request to is done randomly with equal probability. The duration of the sleep activity is $TT=47.0\text{msec}$ and it is deterministic. The broker on receiving the request, examines the request to determine which server to route the request to. It then checks its tables to determine the location of the appropriate server. The server on receiving the forwarded request

does some CPU intensive activity followed by a reply message to the broker. The service time of the two servers in the system are $S_1=11.6\text{msec}$ and $S_2=12.7\text{ msec}$ and these values are also deterministic. The service time does not include the receive and send times involved in receiving a request and sending a reply.

Using DECALS, the client, server and broker process were distributed across the workstations nodes and performance figures obtained. The performance parameter obtained was the response time of the client process, and this is the time from when the client sends its first request to the time when it receives the last reply. This was accomplished by using DECALS's logging services.

The following nomenclature will be used in referring to the various CBS system designs: TPC will refer to the CBS system with the non-distributed broker and employing the thread-per-client model, and DB_TPC will refer to the distributed version. Similarly, TPOOL will refer to the CBS system with the non-distributed broker and employing the thread-pool model, and DB_TPOOL the corresponding distributed-version. Finally, TREAC will refer to the CBS system with the non-distributed broker employing the reactive model and DB_TREAC the corresponding distributed version. The reply message size (RPMS) is varied while the request message size (RQMS) is fixed at 63bytes.

TPOOL has 4 threads operating in the broker while DB_TPOOL also has a total of 4 threads in the brokers but since there are two brokers in this system, this means that each broker has 2 threads each.

Figure 4-9 and Figure 4-8 show the performance results of these systems for RPMS=100 bytes. The 95% and 99% confidence intervals for all the response times shown in the figures are within 1.96% and 3.24% of the indicated values. As can be seen

Figure 4-8 Response time versus number of clients for CBS systems employing TPC, TPOOL, TREAC, DB_TPC, DB_TPOOL and DB_TREAC broker. $S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 100 bytes and RQMS = 63 bytes.

Figure 4-9 System Throughput versus number of clients for CBS systems employing TPC, TPOOL, TREAC, DB_TPC, DB_TPOOL and DB_TREAC broker. $S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 100 bytes and RQMS = 63 bytes

from these results, which are bunched up for DB_TREAC, DB_TPC, TPC and DB_TPC, distributing the broker across the system has little impact on the performance of such systems for these set of parameters. TPOOL and DB_TPOOL do not perform as well as the other designs. This obviously is as a result of the threading level used in these systems which is 4. Therefore, there must exist some software bottleneck in these two systems. Chapter 5 discusses how the parameter space was explored to find out what threading level for DB_TPOOL and TPOOL would result in performance that is comparable to those of other broker designs. In addition, for a single client and considering only the non-distributed designs, one can see that TPOOL and TREAC have a higher response time when compared to TPC. This can be attributed to the fact that TREAC and TPOOL make extensive use of the reactor component. The same argument applies for the corresponding distributed designs. Chapter 5 investigates further, the performance impact of this.

Figure 4-10 and Figure 4-11 show that for large reply messages (RPMS = 10000 bytes). The 95% and 99% confidence intervals are within 1.71% and 2.83% of the indicated values for the response times. Here, it can be seen that distributing the broker has a significant impact on performance of these systems. Comparing the performance results for TPC and DB_TPC, one can see that as these systems approach maximum throughput capacity, DB_TPC has about 50% more throughput than TPC. Similar performance results apply when DB_TREAC and TREAC are compared. Figure 4-9 and Figure 4-8 also show that DB_TPOOL has about 37% more throughput than TPOOL.

These performance results were later used to validate the CBS simulators which are described in the next chapter.

Figure 4-10 System throughput versus number of clients for CBS systems employing TPC, TPOOL, TREAC, DB_TPC, DB_TPOOL and DB_TREAC broker. $S_1 = 13.39$ msec, $S_2 = 16.05$ msec, RPMS = 10000 bytes and RQMS = 63 bytes.

Figure 4-11 Client response time versus number of clients for CBS systems employing TPC, TPOOL, TREAC, DB_TPC, DB_TPOOL and DB_TREAC broker. $S_1 = 13.39$ msec, $S_2 = 16.05$ msec, RPMS = 10000 bytes and RQMS = 63 bytes.

Chapter 5

CBS Modelling and Model Validation

This chapter presents the evolution of the PARASOL models of the CBS prototypes, and discusses how the model prediction comes close to the performance results obtained from the actual live experiments described in chapter 4. This chapter also describes other CBS architectures that were built using the PARASOL simulator.

5.1 Forwarding Broker PARASOL Model Evolution

Figure 5-1 shows the process involved in building a PARASOL model of the CBS prototypes. It starts with building of some CBS prototypes to obtain the desired model input parameters, and also the taking of performance results for validating the models.

The software entities of the CBS prototypes (i.e., the client, server, and broker processes) were modelled using one or more PARASOL tasks (for the multi-threaded broker, a PARASOL task was used to represent each thread). The hardware environment used in the running of the CBS prototypes (i.e., the workstation nodes and the communication network) were accounted for in the model by using the PARASOL hardware constructor calls. The send and receive calls used for communication between these software entities in the CBS prototypes were also modelled using a combination of PARASOL Inter Task Communication primitives and some derived message transmission and reception calls (this is explained in sub-section 5.1.1). The same client interaction used in the CBS prototypes was also used in the model.

Figure 5-1 CBS Model Construction Process

In the following sub-sections, the details involved in the modelling of all the components in the forwarding broker system including their interactions are presented.

5.1.1 Bus Modelling

While PARASOL provides the CSMA/CD style of bus arbitration, it does not cater for the fact that message sizes greater than the maximum message size allowed on the network wire have to be broken down into smaller messages (or packets) for transmission along the bus. This was the case in the CBS system built using the TCP/IP mechanism, as observations revealed that when a message with a size greater than the maximum allowed on the bus hardware is being sent, it is broken down into smaller packets. This maximum is 1460

bytes of user data [21]. Furthermore, when messages with user data size greater than 1460 bytes is sent between two nodes, there is an overlap in the send operation on the sending node, the transmission of the message on the bus connecting the two nodes, and the reception of the message on the receiving node.

In addition, the TCP layer combined with the IP layer adds a minimum of 20 bytes each of header information, making a total of 40 bytes, to the original message being sent. The data link layer also adds some other control information to the message being sent for a total of 30 bytes. In addition, the data link layer includes some additional bytes in the form of padding for cases where the message being sent has less than 46 bytes of user data. All this was accounted for in the model's bus sub-system.

The send and receive CPU times used in the bus sub-model was that reported by the displacement technique developed by Courtois and Woodside [34].

To model the concurrency issue mentioned above into the model, two PARASOL calls were written, namely `ps_send_over_bus()` and `ps_recv_mess()`. `ps_send_over_bus()` takes all the aforementioned TCP/IP bus characteristics and behavior into consideration. For example, a message containing 1460 bytes of user data or less can be modelled by a CPU consumption followed by the PARASOL send primitive, `ps_bus_send()`, since this fits one packet in the real world of TCP/IP protocol. However, modelling the transmission of a message with user data size greater than 1460bytes was done by carrying out a number of the above steps i.e., for a message size greater than 1460 but less than 2 x 1460bytes (in actual TCP/IP world, this will consist of splitting the message into two packets as the maximum user data allowed in a packet is 1460 bytes), the model send operation, `ps_send_over_bus()`, is made up of a PARASOL CPU consumption followed by a

ps_bus_send(), followed by another CPU consumption and finally a ps_bus_send(). The first CPU consumption corresponds to the amount of real CPU that will be necessary to transmit the first full packet, and the ps_bus_send() operation that follows corresponds to the transmission of this packet. The second PARASOL CPU consumption corresponds to the amount of CPU required to pack the remaining user data into the second packet and the second ps_bus_send corresponds to the actual wire transmission of this second packet. The first and second CPU times were assumed to be proportional to the normalized size of the user data in each packet being transmitted i.e., for a total CPU time of 2.46 msec and a total user data size of 1800bytes, the first CPU time will be

$$CPU_1 = 2.46 \times 1460/1800 \quad m \text{ sec}$$

and the second CPU time will be

$$CPU_2 = 2.46 \times (1800 - 1460)/1800 \quad m \text{ sec}$$

This approach therefore captures the concurrency between the CPU consumption on the sending node and the actual transmission of the message along the network wire for user data size greater than 1460 bytes. Hence for messages greater than 1460 and less than $n \times 1460$ (where n is greater than 1), this will consist of n PARASOL CPU consumption interleaved by n PARASOL ps_bus_send() operations.

Also modelled was the overall concurrency that takes place between CPU con-

sumption corresponding to a send operation, the transmission of the message along the bus and the CPU consumption that takes place when the packets of the message are received at the destination (for messages with sizes greater than 1460 bytes of user data). This was achieved by having `ps_rcv_mess()` start to receive immediately, as soon as the first packet of the message arrives at the destination. The complete message is delivered by `ps_rcv_mess()` to the “calling task” only when all the packets that make up the original message have been received. The individual CPU times for each individual packet was again assumed to be proportional to the normalized size of user data in each packet.

As discussed in the previous chapter, ACK's are sent between machines when the TCP/IP protocol is used. Accounting directly for the ACK's in the PARASOL model would result in a more complex implementation of the bus sub-model for the simulator and will invariably lead to a slower simulation run. An alternative (which is an approximation to real thing) was to assume that no ACKs are sent. This can be justified by the fact that the time on the bus for an ACK packet is small (i.e., approximately 0.032msec) when compared to the cpu time to send or receive data, even for small message sizes.

5.1.2 Client Modelling

The modelling of this task in PARASOL was a straight forward process. A PARASOL task was used to represent the client prototype. There was a one to one mapping between the calls in the client prototype to calls in the PARASOL model. For example, a send call in the prototype corresponds to a `ps_send_over_bus()` or `ps_send_over_pipe()`, depending on if the send operation was to a local broker (in the case of the distributed broker) or a remote broker (in the non-distributed broker). Each PARASOL client task had one port on

which it receives replies from the broker task. The choice of which server to request service was done on a random basis as was the case in the prototype. The send and receive times and other cpu times used in all the PARASOL tasks was assumed uniformly distributed. The minimum and maximum values used in the distribution was based on the observed cpu times in the prototype measurements. The think time, TT, was also made uniformly distributed.

5.1.3 Server Modelling

The modelling here was not as straight forward as the client task, since the Reactor component had to be modelled. Here, a single PARASOL task was used to represent the server prototype. The Reactor component was modelled by creating an additional call, `ps_select()`, that mimics the UNIX `select` call. This call was needed since the cpu time consumed in the Reactor's `dispatch()` and `select()` methods varies directly with the number of registered file descriptors on which events have occurred. Hence, knowing the number of active file descriptors was necessary to model correctly the amount of cpu resource consumed in the Reactor components when events occurred. Rather than using a PARASOL port for each client in the server task model, a single port was used and requests arriving from different clients are differentiated by a unique acknowledgment port number that is supplied by the broker task that forwarded the request. The `ps_select()` call scans the messages in this port, placing them in a some data structure for later processing, and returns the number of complete requests that have arrived from the broker. This number is then used to determine the cpu consumption in the Reactor model.

The modelling of the send and receive operations in the prototype had a one to one

mapping. The service time was modelled using a using a set of uniformly distributed values. The maximum and minimum values for this distribution was obtained from observing the cpu times in the server prototype measurements.

5.1.4 Broker Modelling

The Reactor modelling discussed above in the server task model was also used to model the Reactor component used in all the broker prototypes.

For the thread-per-client broker modelling, a PARASOL task was used to represent each thread spawned to handle a request from a client in the prototype. Each task had a separate port on which it accepted requests from client tasks or replies from server tasks. The message type is used to differentiate between these two types of events.

In the thread-pool broker modelling, a PARASOL task were used to represent each thread in the thread-pool broker prototype i.e., the main thread and the threads in the pool. The PARASOL shared port was used to represent the request queue in the prototype. Requests from client tasks are stored in the shared port by the main thread. The threads in the pool then remove these requests for processing.

The reactive broker model uses just one PARASOL task to represent the single-threaded reactive broker and the same mechanism described in the reactive broker prototype was used to avoid the blocking of this single PARASOL task.

5.2 PARASOL Model Prediction Results

As mentioned previously, the measurement results of all the important components that make up the CBS system were used as input parameters into the PARASOL models. This

section shows how well the PARASOL simulators predict the performance of the CBS prototypes.

5.2.1 TPC PARASOL Model Validation

Figure 5-2 and Figure 5-3 show the actual performance results and those predicted by the simulator for TPC. The 95% and 99% confidence intervals for the simulator results shown here are all within 1.0% and 1.6% respectively. Examination of the results reveal that, for a reply message size of 100 bytes, there exist a 6% error (for large number of clients) in the simulator response time prediction. Also, for a single client in the system, there is also some difference (about 1.5%) between the simulator and the prototype response time. This initial error can be attributed to the fact that in the prototype, when no message is available, a receive operation would consume some cpu time before the message arrives and the remaining cpu time will be consumed when the message arrives. The UNIX system call, `getrusage()`, however returns only the total cpu time and this cpu time is assumed in the simulator to be consumed when the message arrives.

With larger message sizes, the maximum percentage error is less (about 3.5% for the response time for large number of clients in the system). In general, it can be said that the simulator for TPC does provide a fairly accurate prediction of the performance for the CBS prototype with the TPC broker.

Figure 5-2 Response time versus number of clients for TPC prototype and simulator. $S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 100 bytes, RQMS = 63 bytes.

Figure 5-3 Response time versus number of clients for TPC prototype and simulator. $S_1 = 13.39$ msec, $S_2 = 16.05$ msec, RPMS = 10000 bytes, RQMS = 63 bytes

5.2.2 DB_TPC PARASOL Validation

Figure 5-4 and Figure 5-5 compares the performance of the actual DB_TPC prototype and that predicted by the DB_TPC PARASOL simulator. The 95% and 99% confidence intervals for the simulator results shown in these figures are all within 0.9% and 1.4% respectively. The results show that for a small reply message size (RPMS = 100

Figure 5-4 Response time versus number of clients for the DB_TPC prototype and simulator. $S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 100 bytes, RQMS = 63 bytes

bytes), the maximum error observed is about 6.5% and this occurs for large number of clients. Here also, there exist an error of about 2.2% in response time for just one client. This as explained previously is due to assumption made in the simulator that all the receive cpu time is consumed when the message arrives.

For a larger reply message size (RPMS = 10000 bytes), the maximum error for any number of clients is about 2%.

Figure 5-5 Response time versus number of clients for the DB_TPC prototype and simulator. $S_1 = 13.39$ msec, $S_2 = 16.05$ msec, RPMS = 10000 bytes, RQMS = 63 bytes

5.2.3 TPOOL PARASOL Validation

Figure 5-6 and Figure 5-7 show the comparison between the simulator prediction and the actual prototype results for TPOOL. The 95% and 99% confidence intervals for the simulator results shown are all within 0.74% and 1.2% respectively. For small reply size (RPMS = 100 bytes), the maximum error in the simulator for large number of clients is about 5.5%. For RPMS = 10000 bytes, the maximum error observed is about 4.5% and this is seen for large client numbers.

Figure 5-6 Response time versus number of clients for the TPOOL prototype and simulator. $S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 100 bytes, RQMS = 63 bytes

Figure 5-7 Response time versus number of clients for the TPOOL prototype and simulator. $S_1 = 13.39$ msec, $S_2 = 16.05$ msec, RPMS = 10000 bytes, RQMS = 63 bytes

5.2.4 DB_TPOOL PARASOL Validation

Figure 5-8 and Figure 5-9 show the comparison between the simulator and prototype for DB_TPOOL. The 95% and 99% confidence intervals for the simulator results shown are all within 0.9% and 1.4% respectively. For RPMS = 100 bytes, the maximum error in the simulator results is about 10.7% for large number of clients. However, for RPMS = 10000 bytes, the maximum error observed is about 6.9% and this occurs for large client numbers.

Figure 5-8 Response time versus number of clients for the DB_TPOOL prototype and simulator. $S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 100 bytes, RQMS = 63 bytes

Figure 5-9 Response time versus number of clients for the DB_TPOOL proto-
type and simulator. $S_1 = 13.39$ msec, $S_2 = 16.05$ msec, RPMS = 10000 bytes,
RQMS = 63 bytes

5.2.5 TREAC PARASOL Validation

Figure 5-10 and Figure 5-11 show the comparison between the simulator and prototype for TREAC. The 95% and 99% confidence intervals for the simulator results shown are all within 1.1% and 1.78% respectively. For RPMS = 100 bytes, the maximum error in the simulator results is about 6.3% for large number of clients. However, for RPMS = 10000 bytes, the maximum error observed is about 4.3% and this is seen for a large number of clients.

Figure 5-10 Response time versus number of clients for the TREAC prototype and simulator. $S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 100 bytes, RQMS = 63 bytes

Figure 5-11 Response time versus number of clients for the TREAC prototype and simulator. $S_1 = 13.39$ msec, $S_2 = 16.05$ msec, RPMS = 10000 bytes, RQMS = 63 bytes

5.2.6 DB_TREAC PARASOL Validation

Figure 5-12 and Figure 5-13 show the comparison between the simulator and prototype for DB_TREAC. The 95% and 99% confidence intervals for the simulator results shown are all within 0.75% and 1.25% respectively. For RPMS = 100 bytes, the maximum error in the simulator results is about 6% for large number of clients. However, for RPMS= 10000bytes, the maximum error observed is about 1.3% for large number of clients.

Figure 5-12 Response time versus number of clients for the DB_TREAC prototype and simulator. $S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 100 bytes, RQMS = 63 bytes

Figure 5-13 Response time versus number of clients for the DB_TREAC prototype and simulator. $S_1 = 13.39$ msec, $S_2 = 16.05$ msec, RPMS = 10000 bytes, RQMS = 63 bytes

5.3 Simulator Model of other Broker Architectures

The sub sections below describes the other broker architectures that were built by extending the earlier ones.

5.3.1 Half Forwarding Broker (distributed and non-distributed) Model

The client task was modelled exactly as the forwarding broker modelling. The broker task model was modified slightly by not having it wait for a reply from the server. The server task was also changed by having it send the reply directly to the client, rather than back to the broker.

5.3.2 Handle Broker Model

The client task was modified to send the request directly to the server task. The server task model was also changed to have it return the reply directly to the client. The broker task is ignored in this model since it is assumed that server binding is rare and the clients caches the object handle after obtaining it from the broker. Hence, the handle-broker effectively becomes a client-server model.

5.3.3 Broker Model with object replication

The broker paradigm used in the PARASOL implementation of the CBS architecture with object replication was the forwarding broker. The client task used previously in the forwarding broker PARASOL model was modified slightly in this implementation by having it make two types of requests: a read request and a write request. The broker task was modified to include the role of unicasting a request from a client to all the object replicas in the system (in the case of a write type request) or sending the request to the just the local replica (in the case of a read type request). The server task was also modified to implement the algorithm described in chapter 3 for keeping the states of each replicated object in the system consistent. This was done using the PARASOL primitives together with the ones derived.

Chapter 6

Performance Results

This chapter investigates further the comparison between the various architectures used in the broker design through the use of the PARASOL simulator. Section 1 discusses the workload pattern in the CBS system. In section 2, the confidence interval of the measurements are discussed. Section 3 presents the performance impact of broker distribution on the forwarding broker. Section 4 discusses the performance impact of the choice of threading model for the broker component. Section 5 compares the performance of the half-forwarding, forwarding and the handle driven broker models and the impact of broker distribution. Finally, Section 6 investigates the impact on performance of replicating stateful objects in CBS systems.

6.1 Workload pattern

Two servers were considered adequate for investigating the performance of the various broker architectures since what we are really interested in is how the distribution of the broker component impacts performance, and also the effect of the threading model choice at the broker on performance. While the server carries out this request, the client blocks awaiting a reply from the server in question. A server response to a client request consists of some cpu intensive computation (corresponding to a service time which is exponentially distributed) followed by a reply (directly to the client in this case of the handle-driven and half-forwarding broker architecture or via the broker in the case of the forwarding broker architecture). After the receipt of these two replies, the client goes into an inac-

tivity phase by sleeping (corresponding to a think time which is also exponentially distributed). The elapsed time from the start of the client's first request to the time it receives the last reply is referred to as the response time. The word throughput would mean the system wide throughput.

The service time indicated does not include the receive or send time. The service time was also made to vary with reply or request message size to reflect the fact that more cpu time would be needed to unpack or pack larger request and reply messages. Unless otherwise stated, a complete client interaction consists of a request to each of the two servers in the system (via the broker in the case of the forwarding and half-forwarding broker architecture or directly in the case of the handle-driven broker architecture).

In section 3, where the performance impact of broker distribution on the forwarding broker for both small and large replies are discussed, the request message size is kept small at 100 bytes and the reply message size is made to vary. This would correspond to situation where the requests are read types (the request message would contain a description of the data needed and the reply would contain the actual data whose size could be small or large).

In section 4, where the choice of the appropriate threading level is discussed for the thread-pool broker system, only read type requests are used in the interactions.

In section 5, where a performance comparison is made between various broker architectures, the parameter space involving the reply and request message sizes was explored. For example, in one of the investigations, the reply message size is fixed at a small value (100 bytes) and the request message size is made to vary. This scenario would correspond to situation where the requests are of a write nature i.e., the request message

contains information that needs to be written and the reply corresponds to an acknowledgment that the write request was successful (hence a small size of 100 bytes).

In section 6, where the performance impact of stateful object replication is discussed, the proportion of read and write requests are implicitly considered by the use of the read probability. The read message size is made equal to the write message size.

6.2 Confidence Intervals

In performance studies, the confidence interval of the results need to be known to be able to judge their accuracy. The confidence intervals of the result presented in this chapter were obtained using the PARASOL built in confidence interval feature. Confidence intervals estimation in PARASOL can be done in two ways. The first involves replicating experiments by using different random number generator seeds to obtain default measurements reported by PARASOL, and using the Student-t distribution to obtain the confidence interval. The second way involves the use of a long duration simulation run which is blocked into the quasi-independent sections. This second approach was the one chosen in estimating the confidence intervals of the measurements.

A simulation duration run between 2500000 and 3000000 time units was used in the PARASOL experiments to obtain a 95% and 99% confidence level that is within 1% and 1.6% for the results reported in this chapter, unless otherwise stated.

6.3 Performance Impact of broker distribution on the forwarding broker

In this section, we present the impact on performance of the choice of threading model for the broker. Figure 6-1 to Figure 6-4 shows the performance results of TPC, TPOOL,

Figure 6-1 Throughput versus number of clients for CBS systems employing TPC, TPOOL, TREAC, DB_TPC, DB_TPOOL and DB_TREAC broker. $S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 100 bytes, TT = 1sec

Figure 6-2 Response time versus number of clients for CBS systems employing TPC, TPOOL, TREAC, DB_TPC, DB_TPOOL and DB_TREAC broker. $S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 100 bytes, TT = 1sec

TREAC, DB_TPC, DB_TPOOL and DB_TREAC. Figure 6-1 and Figure 6-2 shows the performance result for small reply sizes and service times of $S_1 = 11.7$ msec and $S_2 = 12.7$ msec. The service times were chosen to make the server the bottleneck component in the system. The response time and throughputs of these systems can be seen to be very close. For this point in the parameter space, the results show that broker distribution does not have an impact on the performance of the systems.

However, with a smaller service time ($S_1 = 1.17$ msec and $S_2 = 1.27$ msec) and $RPMS = 100$ bytes, broker distribution does have a significant impact on the performance of these systems as can be seen in Figure 6-3, which shows the response times for TPOOL and DB_TPOOL. Here, the benefits of broker distribution can readily be seen. The results show that when the number of clients in the system exceeds 30, DB_TPOOL outperforms TPOOL. This can be attributed to the fact that for smaller service times, the bottleneck shifts from the object implementations to the broker component. This is because of the fact that there are more brokers in DB_TPOOL than in TPOOL, and they make use of the spare CPU power of other workstation nodes in the system, hence the broker does not become a bottleneck in DB_TPOOL for this point in the parameter space.

This trend can be observed from Figure 6-4 and Figure 6-5 which shows the cpu utilizations for TPOOL and DB_TPOOL for both the small and large service times. It can be seen from these two figures that for TPOOL, changing the service times from $\{S_1 = 11.5$ msec, $S_2 = 12.7$ msec $\}$ to $\{S_1 = 1.15$ msec, $S_2 = 1.27$ msec $\}$, the maximum utilizations for the second server and the broker goes from a 0.99 and 0.58 to 0.53 and 0.93 respectively. Hence the change service time results in the broker becoming the most utilized component in the system. However, for DB_TPOOL, the change in the service time

Figure 6-3 Response time for DB_TPOOL and TPOOL. $S_1 = 1.15$ msec, $S_2 = 1.27$ msec, RPMS = 100 bytes, TT = 1sec.

Figure 6-4 Utilization figures for the participants in TPOOL. $S_1 = 1.15$ msec, $S_2 = 1.27$ msec, RPMS = 100 bytes, TT = 1sec

Figure 6-5 Utilization figures for the participants in DB_TPOOL. $S_1 = 1.15$

msec, $S_2 = 1.27$ msec, RPMS = 100 bytes, TT = 1sec

from $\{S_1 = 11.5 \text{ msec}, S_2 = 12.7 \text{ msec}\}$ to $\{S_1 = 1.15 \text{ msec}, S_2 = 1.27 \text{ msec}\}$ results only in a slight increase in CPU utilization for each broker in DB_TPOOL (see Figure 6-5).

For service times $\{S_1 = 11.5 \text{ msec}, S_2 = 12.7 \text{ msec}\}$ and a larger reply message size, (i.e., RPMS = 10000 bytes), the gain in performance resulting from broker distribution can be seen in Figure 6-6. This gain in improvement is because of the fact that with broker distribution, only two network messages are needed to complete make a request and obtain a reply. For systems with no broker distribution, majority of client interactions require a total of four network messages, hence a higher utilization of the bus. Figure 6-7 shows the utilizations for the systems. It can be seen that the bus utilizations for DB_TPOOL, DB_TPC and DB_REAC have a lower value when compared to their corresponding non-distributed counterparts.

Figure 6-6 Response time versus number of clients for CBS systems employing TPC, TPOOL, TREAC, DB_TPC, DB_TPOOL and DB_TREAC broker. $S_1 = 13.39$ msec, $S_2 = 16.05$ msec, RPMS = 10000 bytes, TT = 1sec

Figure 6-7 Bus utilization versus number of clients for CBS systems employing TPC, TPOOL, TREAC, DB_TPC, DB_TPOOL and DB_TREAC broker. $S_1 = 13.39$ msec, $S_2 = 16.05$ msec, RPMS = 10000 bytes, TT = 1sec

Hence, it can be said that for large reply message sizes, the improvement in performance due to broker distribution becomes very evident even for a single client and becomes even better for a large number of clients in the system, as can be seen in Figure 6-6. However, for small reply message sizes and large service times, broker distribution does not have any significant impact on the performance of CBS systems.

It can thus be said that there is nothing to lose by having brokers distributed in CBS systems and possibly more to gain. To conclude, distributing brokers helps to reduce the potential for a bottleneck at the broker and also reduces the amount of network traffic generated as a result of client-broker-server interactions.

6.4 Performance comparison of threading models

In this section, the relative performance of the threading models for various message sizes and server service times is shown. The thread-level for TPOOL is initially fixed at 10.

Figure 6-8 shows the results show that for small service times $\{S_1 = 1.15 \text{ msec}, S_2 = 1.27 \text{ msec}\}$. These service times were chosen to make the broker the bottleneck component in the system and hence most of the time is spent at the broker. The TPOOL is seen to have the worst performance because of the extra overheads incurred to enqueue and dequeue requests (the TPOOL threading models uses a master-slave structure which has the master thread enqueue the requests in a queue and the slave threads dequeue these requests for processing). Furthermore, the TPOOL uses the Reactor component in the master thread to detect and demultiplex an incoming requests from multiple clients.

The TREAC also incurs some overhead from the using the Reactor as it relies solely on the Reactor to detect request from clients and replies from servers. TPC only

Figure 6-8 Response time for TPOOL, TREAC and TPC. $S_1 = 1.15$ msec, $S_2 = 1.27$ msec, RPMS = 100 bytes, TT = 1sec

uses the Reactor in the initial connection stage but does not use it to detect and demultiplex future requests from clients or replies from servers. This is because there is a broker thread for each client and this thread blocks waiting for either a request from the corresponding client or the reply from the server for this client. For a larger reply message size (i.e., RPMS = 10000 bytes), the bottleneck shifts from the broker to the bus and hence the response times of TPC, TPOOL, and TREAC become comparable. This can be seen in Figure 6-9.

However, for larger server service times $\{S_1 = 11.5$ msec, $S_2 = 12.7$ msec $\}$, chosen such that the bottleneck in the system shifts from the broker to the servers, and keeping RPMS at 100 bytes, the overheads involved in TPOOL and TREAC are not apparent as can be observed from the response times of these systems, shown in Figure 6-10. Here we

Figure 6-9 Response time for TPOOL, TREAC and TPC. $S_1 = 1.39$ msec, $S_2 = 1.61$ msec, RPMS = 10000 bytes, TT = 1sec

Figure 6-10 Response time for TPOOL, TREAC and TPC. $S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 100 bytes, TT = 1sec

see that TPOOL, TREAC and TPC do have comparable response times for this point in the parameter space as the bottleneck shifts from the broker to other components in the system.

6.4.1 Choosing an appropriate threading level for TPOOL

Previously, the thread level of the TPOOL and DB_TPOOL had been fixed at 10. In this section, we shall present the results obtained from searching the parameter space for the thread level (TL), for which increasing the number of threads beyond this value does not improve significantly the response time of the clients. This thread level is taken as the optimum thread level.

Figure 6-11 shows the response time for TPOOL for various threading levels and a reply message size of 100 bytes. The results show that increasing the threading level above 10 does not appreciably improve performance. Hence, one can choose this threading level as the optimum threading level.

The results for a reply message size of 10000 bytes are shown in Figure 6-12, where we have the optimal threading level being 6, as the bottleneck in this scenario becomes the bus network.

Hence, it can be concluded that obtaining the optimal thread level in advance for TPOOL is not possible as the threading level depends on a lot of things, such as the message size, service times etc. One way to solve this would be to have threads created and destroyed as their utilizations increase or decrease.

Figure 6-11 Response time for TPOOL for $S_1 = 11.5$ msec, $S_2 = 12.7$ msec,

RPMS = 100 bytes and TT = 1sec

Figure 6-12 Response time for TPOOL for $S_1 = 13.39$ msec, $S_2 = 16.05$ msec,

RPMS = 10000 bytes and TT = 1sec

6.5 Performance comparison of the handle-driven, half-forwarding and forwarding broker model and the impact of broker distribution

This section presents the performance comparison between various broker architectures, namely, the handle-driven broker (HD_BROKER), the forwarding broker (F_BROKER), and the half-forwarding broker (HF_BROKER) architectures. The impact on performance of broker distribution on these architectures is also presented.

A total of 12 nodes are used in these experiments, 2 of which are server nodes (each node containing a server task). The remaining 10 nodes are client nodes. Both write and read type of requests are involved in client-server-broker interactions. Sub-sections present the performance comparison of these architectures where interactions involve only read type requests (i.e., the request message size is small and fixed at 100 bytes with the replies being small or large). Sub-section 2 shows the performance comparison of these architectures for write type of requests (i.e., the request could be large or small with the replies being small and fixed at 100 bytes). Sub-section 3 discusses the performance results of these architectures for interactions involving both read and write type requests.

6.5.1 Interactions involving only Read type requests

The performance comparison between F_BROKER, HF_BROKER and HD_BROKER is presented in this sub-section for read type requests. Figure 6-13 shows the response time for these architectures for service times $\{S_1 = 11.5 \text{ msec}, S_2 = 12.7 \text{ msec}\}$. These service times were again chosen to make the object implementations initially the bottleneck component in the systems as can be observed in Figure 6-14, which shows the utilizations for the various components in the systems. It can be seen from Figure 6-13 that for 78 clients

Figure 6-13 Response time for F_BROKER, HF_BROKER and HD_BROKER. $S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 100 bytes, RQMS = 100 bytes and TT = 1 sec

Figure 6-14 Utilizations for F_BROKER, HF_BROKER and HD_BROKER. $S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 100 bytes, RQMS = 100 bytes and TT = 1 sec

or less in the system, HD_BROKER has the best response time with HF_BROKER having the worst response time of the three architectures but the percentage differences in response times are small. This difference begins to fade away as the number of clients in the system exceeds 80.

Figure 6-15 shows that with a larger reply (i.e., larger read requests), the percentage difference in response time between HF_BROKER and HD_BROKER is much lower than the previous case. Furthermore, only for HF_BROKER and HD_BROKER do you find that the response time become comparable as the number of clients in the systems exceeds a critical value (which is also less than the previous case i.e about 50 clients). The response time of F_BROKER is the largest of the three architectures and increases faster than the response time of the other two architectures as the number of clients in the system increases. This is because the F_BROKER makes more use of the bus than the other two architectures. However, with broker distribution (see Figure 6-16) a completely different performance characteristic results for the F_BROKER with broker distribution (i.e., DB_F_BROKER). Here, the response time of DB_F_BROKER approaches that obtained in the other two architectures as the number of clients in the system increases. Little or no improvement in performance is seen with HD_BROKER and HF_BROKER with broker distribution.

Up to this point, the service time was chosen to make the server the bottleneck component in the system for small message sizes. The performance characteristic of these architectures is examined when the service time at the object implementations is such that more processing takes place at the broker. In other words, the broker becomes a bottleneck in the system. The service time chosen for the two servers are $\{S_1 = 1.15 \text{ msec}, S_2 = 1.27$

Figure 6-15 Response time for F_BROKER, HF_BROKER and HD_BROKER.

$S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 10000 bytes, RQMS = 100 bytes and

TT = 1 sec

Figure 6-16 Response time for the distributed version of the architectures (i.e.,

DB_F_BROKER, DB_HF_BROKER and HD_BROKER). $S_1 = 11.5$ msec, S_2

= 12.7 msec, RPMS = 10000 bytes, RQMS = 100 bytes and TT = 1 sec

msec}. Figure 6-17 shows the response time for a RPMS value of 100 bytes.

HD_BROKER is seen here to out perform the other two architectures. This is due to the fact that for smaller service times, little processing takes place at the server, and the broker becomes heavily used for F_BROKER and HF_BROKER, since all client-server interac-

Figure 6-17 Response time for F_BROKER, HF_BROKER and HD_BROKER. $S_1 = 1.15$ msec, $S_2 = 1.27$ msec, RPMS = 100 bytes, RQMS = 100 bytes and TT = 1 sec

tions must take place via the broker. HF_BROKER does not perform as badly as F_BROKER, since the broker is involved only once in a complete client/server interaction, compared to two needed for F_BROKER. Figure 6-18 shows the utilizations for these architectures. It can be seen that for F_BROKER, the broker becomes the bottleneck in the system as the number of clients in the system exceeds 125.

Figure 6-19 shows the impact of broker distribution on the performance of these architectures. It can be seen from this figure that broker distribution does help to reduce the potential for a bottleneck at the broker.

Figure 6-18 Utilizations for F_BROKER, HF_BROKER and HD_BROKER. $S_1 =$

1.15 msec, $S_2 = 1.27$ msec, RPMS = 100 bytes, RQMS = 100 bytes and TT = 1 sec

Figure 6-19 Response time for DB_F_BROKER, DB_HF_BROKER and

HD_BROKER. $S_1 = 1.15$ msec, $S_2 = 1.27$ msec, RPMS = 100 bytes, RQMS =

100 bytes and TT = 1 sec

6.5.2 Interactions involving only Write Type Requests

For small request message sizes (i.e., small write requests), the behavior of the system for write type requests can be expected to be the same as that for read type request discussed earlier. This is because messages of the same size are involved (i.e., small message sizes irrespective of type). However, for large write requests, there is a change in the performance characteristic as can be seen in Figure 6-20. Here, unlike for large read requests where HF_BROKER performs as well as HD_BROKER, HF_BROKER in this case performs as badly as the F_BROKER for large write type requests. This can be attributed to the fact that there is little gain in performance by having the server return the reply directly to the client, since replies are small in this case. Figure 6-21 shows the impact of broker distribution. Here again, broker distribution makes the performance almost independent of the type of broker architecture chosen.

Figure 6-20 Response time for F_BROKER, HF_BROKER and HD_BROKER. $S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 100 bytes, RQMS = 10000 bytes and TT = 1 sec

Figure 6-21 Response time for DB_F_BROKER, DB_HF_BROKER and HD_BROKER. $S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 100 bytes, RQMS = 10000 bytes and TT = 1 sec

6.5.3 Interactions involving Equal proportion of Read and Write Type requests

Here also, for small message sizes (i.e., small write and read requests), the behavior of the system for equal proportion of read and write type requests can be expected to be the same as that for read type or write type requests which was discussed earlier. This is because the same messages sizes are involved (i.e., small message size irrespective of type). However, for equal proportion of large write and read type requests, the performance characteristic is different (see Figure 6-22) from both the two previous cases where there was only large read requests or large write requests. Here, HF_BROKER does not perform as well as HD_BROKER but it does not perform as badly as F_BROKER. HF_FORWARD is seen to have a performance characteristic that is midway between the performance of

F_BROKER and HD_BROKER. Figure 6-23 shows the performance impact of broker distribution. Again, broker distribution makes the performance of the CBS system almost independent of the type of broker architecture chosen.

Figure 6-22 Response time for F_BROKER, HF_BROKER and HD_BROKER.

$S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = 10000 bytes, RQMS = 10000 bytes and

TT = 1 sec

Figure 6-23 Response time for DB_F_BROKER, DB_HF_BROKER and HD_BROKER.

$S_1 = 11.5$ msec, $S_2 = 12.7$ msec, RPMS = RQMS = 10000 bytes and TT = 1 sec

6.6 Performance impact of replicating stateful objects in CBS systems

In this section, the impact on performance of replicating stateful objects is presented. The broker architecture used in this case study is the distributed forwarding broker architecture using the thread-per-client threading model. The performance of this broker architecture with and without server replication was explored and the results are presented. The study involves 10 nodes each of which supports a local broker for local clients, which are distributed evenly over the nodes. The non-replicated server case is realized with a single server on a distinguished node. The replicated server case uses a server on each node whose states are kept write consistent through group communication (using the unicasting mechanism). The performance obtained for the architecture with object replication would represent a worst case scenario. Message size (MS) will refer to both the size of read and write requests.

Figure 6-24 and Figure 6-25 show the comparison between the read and write response time for the CBS system with and without server replication, with a read probability of 0.75 and a message size of 2000 bytes. The CBS with server (object) replication will be referred to as OR_BROKER, while that without server replication will be referred to as NOR_BROKER. The results shows that server replication does greatly improve performance in this case. For large number of clients in the system (i.e, 100), the order of magnitude for both read and write response time between these two architectures is about 100. However, as can be seen from Figure 6-26, there is a price to paid for this performance gain as the bus utilization is considerably more for OR_BROKER. This is as a result of the network messages necessary to maintain the states of these server replicas consistent. Figure 6-26 also shows the server utilization for these two architectures. The

Figure 6-24 Read response time for $S_W = 50$ msec, $S_R = 50$ msec, $rp = 0.75$, MS
= 2000 bytes and $TT = 10$ sec

Figure 6-25 Write response time for $S_W = 50$ msec, $S_R = 50$ msec, $rp = 0.75$, MS
= 2000 bytes and $TT = 10$ sec

Figure 6-26 Utilizations for $S_W = 50$ msec, $S_R = 50$ msec, $rp = 0.75$, $MS = 2000$

bytes and $TT = 10$ sec

server utilization for NOR_BROKER, as expected, is higher than that for OR_BROKER. This is expected since there exists only a single copy of the server and majority of the requests are reads.

However, as the read probability is reduced, the dramatic improvement in performance can no longer be seen. Figure 6-27 to Figure 6-29 shows the performance figures of NOR_BROKER and OR_BROKER for a reduced read probability of 0.25. The ratio (for 250 clients) of response times between the NOR_BROKER and OR_BROKER architectures for both reads and writes is only about 3. The reason for this can be found in Figure 6-29, where it can be seen that the server utilization for NOR_BROKER is just lower than that for OR_BROKER. This is because majority of the requests are now writes and hence all the server replicas have to handle this large number of write requests in OR_BROKER as it is the case for NOR_BROKER which handles all types requests in one server copy.

Figure 6-27 Read response time for $S_W = 50$ msec, $S_R = 50$ msec, $rp = 0.25$, MS = 2000 bytes and TT= 10 sec

Figure 6-28 Write response time for $S_W = 50$ msec, $S_R = 50$ msec, $rp = 0.25$, MS = 2000 bytes and TT= 10 sec

Figure 6-29 Utilization for $S_W = 50$ msec, $S_R = 50$ msec, $rp = 0.25$, $MS = 2000$

bytes and $TT = 10$ sec

6.7 Critical Message Size for NOR_BROKER and OR_BROKER

Exploring the parameter space of these two architectures revealed that not in all cases does server replication lead to an improvement in performance in CBS systems. A typical point in the parameter space where this is the case is shown in Figure 6-31, where it can be seen that the write response time for OR_BROKER is larger than that of NOR_BROKER for large client numbers. This can be attributed to the fact that for small read probabilities, a lot more write messages are sent across the network to keep the states of all the server replicas consistent, hence the demand on bus network becomes enormous and this leads to the bus becoming a bottleneck component in the OR_BROKER architecture. Hence the sys-

Figure 6-30 Read response time for $S_W = 50$ msec, $S_R = 50$ msec, $rp = 0.25$, MS = 10000 bytes and TT= 10 sec

Figure 6-31 Write response time for $S_W = 50$ msec, $S_R = 50$ msec, $rp = 0.25$, MS = 10000 bytes and TT= 10 sec

Figure 6-32 Bus utilization for $S_W = 50$ msec, $S_R = 50$ msec, $rp = 0.25$, $MS = 10000$ bytes and $TT = 10$ sec

tem performance suffers because of this. This behavior can readily be seen from the bus and server utilizations of these two architectures which are shown in Figure 6-32. This shows that for OR_BROKER architecture, the bus reaches 100% utilization as the number of clients in the system exceeds 210, while the server utilization is less than this (82%). Hence write requests take much longer time to get answered because they require the use of an highly utilized bus network. However, the read response time in OR_BROKER architecture is less than that for NOR_BROKER because reads can be done locally in OR_BROKER architecture.

To capture all these behavior in a concise and efficient manner, an exhaustive search of the parameter space of these two architectures was carried out to determine at what point does one architecture start to perform better than the other architecture. The performance metric chosen for determining the performance crossover of these two archi-

tures was the response time. The methodology for going about this search was to initially fix the number of clients and for different message sizes, observe for which message size was there a crossover. Obtaining the exact crossover point would require using very small message size increments and due to the lengthy time it takes to run just one experiment, this was considered too expensive. The alternative, which was the approach used, was to use large message steps of size 500 bytes together with linear interpolation. This involved observing and recording the message size just before and after the crossover occurred. This approach assumes that the region over which the crossover occurred is linear (i.e., response time varies directly as the message size). This step was repeated for different number of clients and the corresponding message size before and after the crossover point was recorded.

Using linear interpolation, and applying the general expression for the equation of a straight line,

$$y = nx + c \quad (1)$$

the following equations was used to determine the slopes n_1, n_2 and intercepts c_1 and c_2 of the two lines describing the response times around the crossover region of these two architectures.

$$R_{1b} = n_1 M_b + c_1 \quad (2)$$

$$R_{1a} = n_1 M_a + c_1 \quad (3)$$

$$R_{2b} = n_2 M_b + c_2 \quad (4)$$

$$R_{2a} = n_2 M_a + c_2 \quad (5)$$

when these two lines intercept, we have

$$n_1 M^* + c_1 = n_2 M^* + c_2 \quad (6)$$

Solving the last equation, and using the values of n_1 , n_2 , c_1 and c_2 obtained from solving

equations 2- 5, we have

$$M^* = \frac{M_b(R_{2a} - R_{1a}) - M_a(R_{2b} - R_{1b})}{(R_{1b} - R_{2b}) + (R_{2a} - R_{1a})} \quad (7)$$

One issue that came up was that of the confidence interval of this crossover message size (M^*) just derived. One approach to obtain the confidence interval would be to repeat the process of finding the crossover message size several times and then calculating a confidence interval for the critical message size. However, this approach would be very expensive time wise. An alternative approach involves recognizing the fact that M^* is dependent on R_{2a} , R_{1a} , R_{2b} , R_{1b} , M_a and M_b . However, R_{2a} , R_{1a} , R_{2b} , R_{1b} are not exact (since they involved taking sample statistic of the response times which means that all these four parameters have a certain confidence interval). On the other hand, the values of M_a and M_b are known precisely. Hence, one could do some sensitivity analysis of M^* with respect to changes in R_{2a} , R_{1a} , R_{2b} , and R_{1b} by obtaining the differential of the above equation. The analysis for the derivation of this confidence interval estimation is presented below.

$$M^* = f(R_{1a}, R_{1b}, R_{2a}, R_{2b}, M_a, M_b) \quad (8)$$

and the differential of M^* is given by,

$$dM^* = \frac{\partial f}{\partial R_{1a}} dR_{1a} + \frac{\partial f}{\partial R_{1b}} dR_{1b} + \frac{\partial f}{\partial R_{2a}} dR_{2a} + \frac{\partial f}{\partial R_{2b}} dR_{2b} \quad (9)$$

since M_a and M_b have exact values in the crossover region. For small confidence intervals, this can be approximated as

$$\Delta M^* \approx \frac{\partial f}{\partial R_{1a}} \Delta R_{1a} + \frac{\partial f}{\partial R_{1b}} \Delta R_{1b} + \frac{\partial f}{\partial R_{2a}} \Delta R_{2a} + \frac{\partial f}{\partial R_{2b}} \Delta R_{2b} \quad (10)$$

Where the deltas for R are the confidence intervals around the crossover region.

The final expression for delta M^* turns out (after several steps) to be

$$\frac{M_a - M_b}{[R_{1b} - R_{2b} + R_{2a} - R_{1a}]^2} [(\Delta R_{1a} - \Delta R_{2a})(R_{2b} - R_{1b}) + (\Delta R_{2b} - \Delta R_{1b})(R_{2a} - R_{1a})] \quad (11)$$

The results of this parameter space search, based on the above approach, is shown in Figure 6-33 to Figure 6-38 for a read service time $S_R = 50$ msec, a write service time $S_W = 50$ msec, think time $TT = 10$ sec, and various read probability r_p . In each figure, the regions in which one architecture has better performance (in terms of response time) can be observed. Each figure shows this in terms of a upper bound on the critical message size and a smoothed line drawn in between these bounds. The critical message size is the message size (which depends on read probability and number of clients in the system) for which CBS interactions involving message sizes greater than this value results in NOR_BROKER performing better than OR_BROKER. The smoothed lines are drawn for an asymptotically behaved critical message size. The upper bounds are based on the 99% confidence intervals of the deltas in equation (11).

These result show that the region favoring server replication grows as the read probability increases. These results summarizes the performance characteristics of CBS

Figure 6-33 Critical message size for read requests. $S_R = 50$ msec, $S_W = 50$ msec, $rp = 0.3$, $TT = 10$ sec

Figure 6-34 Critical message size for read requests. $S_R = 50$ msec, $S_W = 50$ msec, $rp = 0.5$, $TT = 10$ sec

Figure 6-35 Critical message size for read requests. $S_R = 50$ msec, $S_W = 50$ msec,

$rp = 0.7$, $TT = 10$ sec

Figure 6-36 Critical message size for write requests. $S_R = 50$ msec, $S_W = 50$ msec,

$rp = 0.3$, $TT = 10$ sec

Figure 6-37 Critical message size for write requests. $S_R = 50$ msec, $S_W = 50$ msec, $rp = 0.5$, $TT = 10$ sec

Figure 6-38 Critical message size for write requests. $S_R = 50$ msec, $S_W = 50$ msec, $rp = 0.7$, $TT = 10$ sec

systems employing object replication with the states maintained through the use of group communication (using the unicast mechanism). It shows that no gain in performance is achieved through object replication for either read or write requests if the message size is greater than the read critical message size. The results also indicate that in some regions there is some gain in performance for read requests and none for write requests. There is also a region in which a gain in both read and write request can be achieved through object replication.

From these results, it can be seen that for a small number of clients in the system, the response time in the CBS system employing object replication is slightly higher for write requests as there is some overhead in updating the under-utilized replicas. However, the response time seen for read type requests is better in OR_BROKER. This is because reads can be done locally with no network traffic or delay. As the number of clients in the system exceeds a certain critical value (depending on the read probability rp), the performance advantage of object replication becomes apparent for both read and write type requests involving message sizes less than the critical message size. This is because the gain in having additional copies now outweighs the overhead in maintaining the copies as more and more clients make use of the copies for both reads and writes. Furthermore, the single server copy in NOR_BROKER becomes heavily utilized as the rate of both reads and writes increases.

Chapter 7

Conclusions and Future Work

In this thesis, a number of CBS prototypes were built using the state-of-the-art ACE technology, a tool that allows the rapid development of distributed applications. To facilitate the investigation of the parameter space of these systems, simulation models were built for each of these prototypes. Simulation offers several advantages over prototype measurements in investigating the performance of these systems. One of the advantages observed is the efficiency of simulators with respect to time of investigation, as simulators take much less time than measurements of real systems. Another advantage of simulators over using prototypes is the ability to control with ease the experiments that is used in collecting performance results (i.e., the setup and running of experiments). Also, the collection of performance figures is much easier in simulators (especially utilizations of hardware components involved in the experiments like the cpu and bus network). Simulators also place less restriction on the choice of design parameters (i.e., think time, service time, number of tasks etc.) plus an almost unlimited choice in the number of hardware and software entities when compared to their corresponding prototype. This allows for large scale experiments to be conducted. Furthermore, simulators allow the elimination of noise (i.e., interference from other entities external to the experiment being run) which are prevalent in the prototypes and are impossible to eliminate. Using simulators, an extension of a system is relatively easier to achieve than the corresponding prototype, since less debugging and code is required in building simulators.

The input parameters into the simulators were obtained from the prototypes by

running an instrumented version of the prototype in a carefully controlled environment.

Later, performance figures were collected from the non-instrumented version of the prototypes to validate the simulator. The validation process revealed that for most of the simulators built, the percentage error difference between the simulator prediction and that collected from the prototype were within 6.5% for small message sizes. This however improves to within 4.5% when large message sizes are involved.

Given the vast size of the design parameter space, it is impossible to state explicit, general conclusions. However, this thesis does point the way to analyze the performance of CBS systems. The performance investigations carried out revealed that for small message sizes and large service time at the object implementation, the basic handle driven broker, half-forwarding broker and forwarding broker architectures had comparable performance. However, for large message size and/or small service time at the object implementation, the basic handle driven broker architecture had the best performance, followed by that of the half-forwarding broker architecture. The forwarding broker architecture had the worst performance of the three architectures. This difference in performance can be readily observed when there are large number of clients in the system. Also, it is found that the performance characteristics of the CBS architectures investigated depends on whether client requests are of a read or write nature. For example, the half-forwarding broker performed as well as the handle-driven broker for large read requests but performs as poorly as the forwarding broker for large write requests.

In conclusion, the decision regarding the architecture of a CBS system should be based on the concrete service time of the object implementation and the size of the messages involved in a client interaction. For a system in which the object service time is rela-

tively small compared to the broker service time, the handle-driven broker architecture should be used. However, the burden of detecting and recovering from failures is passed onto the client. This will lead to a larger executable client file due to the extra code for detecting and recovering from failures. For an environment in which clients reside on cheap processors (that usually have a limited amount of memory) this can create performance problems because of the high paging activity (thrashing) that could occur. For CBS systems in which the object service time is large compared to the broker service time, and the average message size involved in the interactions between the various components of the system is small, either the forwarding or half-forwarding broker architecture should be used. In systems in which client requests are mainly writes (i.e., a logging system) and the object service time is large, either the forwarding or half-forwarding broker architecture should be used. However, for systems where most client requests are of a read nature (i.e., a read only database system) and the service time is large, the half-forwarding broker architecture should be used.

In this thesis, a number of architectural characteristics are found to have an impact on the performance of CBS systems. These are broker distribution, threading model and server replication.

Broker distribution has been shown to dramatically improve the performance of CBS system for cases where the service time at object implementation is small and/or large message sizes are involved. In addition, distributing brokers tend to make the performance of these systems independent of the basic broker architecture used. For example, when broker distribution is added to the basic half-forwarding broker and forwarding broker architectures, the large differences in performance which was previously noticed for

large message sizes and/or small service times at the object implementation becomes less pronounced. These results therefore suggest that when using the CBS model for building software, multiple brokers that are distributed across the system should be used as opposed to employing just a single broker component in the system.

The choice of the threading model for the broker is also found to impact the performance of CBS systems for small service time at the object implementation. The thread-pool model is found to have the worst performance of three threading models investigated for relatively small service times at the object implementation. This is because the thread-pool model incurs the greatest overhead of the three threading models investigated due to the extra CPU consumed by main thread to enqueue requests and that consumed by the threads in the pool to dequeue these requests for processing. However, for large service times at the object implementation, the difference in performance between these three threading models is no longer noticed as the bottleneck shifts from the broker to the object implementations. For systems in which the maximum number of clients is known beforehand and this number does not exceed the maximum number of threads allowed to exist simultaneously by the OS in a single process, the thread-per-client model should be used. However, if this is not the case, then either the thread-pool or the reactive model should be used.

While object replication, which has been suggested for incorporation into the CORBA standard by Maffeis [10], could lead to an improvement in performance of these systems, some careful consideration is needed when the object implementation involved is stateful. This is because stateful object replication is not guaranteed to improve performance all the time it is applied and in fact it is found to lead to a degradation in perfor-

mance in some cases. For example, when the number of client in the system is small, the response time for clients in CBS systems employing object replication is slightly higher for write requests as there is some overhead in updating the under-utilized replicas, and this offsets any gain to be derived from having extra replicas of the object implementation. Read type requests do however benefit from object replication for small client numbers, as they can be done locally without any network traffic or delay. As the number of clients in the system exceeds a certain critical value, which depends on the read probability, the performance advantage of object replication become apparent for both write and read type requests involving message sizes less than the critical message size. This is because the gain in having additional copies now outweighs the overhead in maintaining the replicas, as more and more clients make use of these copies for both read and write transactions. While the critical message size curves determined in this thesis are not absolute, as there can exist many other possible configurations and parameters different from the ones used in this investigation, the results do indicate that for object replication to be useful, the average message size of the interaction should be below a certain value. Hence designers of CBS systems should be aware that a critical message size exists for any point in the parameter space, and determine through some means (i.e., simulation) what these are for the area of the parameter space the system is likely to operate in. Future work could carry out a more exhaustive search of the parameter space (taking into account other factors that would influence the critical message size curve, such as the number of distinct objects in the system, network speed, service time at the object implementations, etc.).

While it has being demonstrated that broker distribution does help improve performance of CBS systems, this was shown for cases where either the clients are evenly dis-

tributed across and the workstation nodes have equal processing power. In situations where this is not the case, some modification of the distributed broker architecture would be needed. One way would be to have all the local brokers in the system maintain information on the power of the workstation nodes in the system together with their utilization levels, and have each local broker in the broker group use this information to maintain the location of the local broker with the lightest load (taking also into consideration the network utilization) for use by clients residing on that machine. Future work could also look at the best case scenario in replicated object state consistency in CBS systems by using multicasting or broadcasting mechanism instead of the unicasting mechanism used in this thesis. Another direction for future work would be the development of simple analytical models given the “classic” look to performance curves. Analytical models would provide much greater speed in the investigation of these systems.

References

- [1] A. Gokhale, D.C. Schmidt, "Measuring the Performance of Communication Middle-ware on High Speed Networks", in Proceedings of the Symposium on Communications Architecture and Protocols (SIGCOMM). Stanford University, August 28-30, 1996.ACM.
- [2] D.C. Schmidt, "Introduction to Distributed Object Computing", SIGS C++ Report, Vol. 7, No. 1, January 1995.
- [3] S. Vinoski, "Distributed Object Computing with CORBA", SIGS C++ Report, July 1993.
- [4] D.C. Schmidt, T. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-Speed Network Programming", in Proceedings of the Conference on Object-Oriented Technologies, (Monterey, CA), USENIX, June 1995.
- [5] D.C. Schmidt, "An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching", in Pattern Languages of Program Design.
- [6] D. C. Schmidt, "Reactor: An Object-Oriented Interface for Event-Driven UNIX I/O Multiplexing", C++ Report, SIGS, Vol. 5, No. 2, February, 1993, pp. 1--12.
- [7] D. C. Schmidt, "The Object-Oriented Design and Implementation of the Reactor: A C++ Wrapper for UNIX I/O Multiplexing", C++ Report, SIGS, Vol. 5, No. 7, September, 1993, pp. 1--14.
- [8] M. Stal, "The Broker Architectural Framework", Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95), Maitland, Florida, U.S.A.
- [9] R. Adler, "Distributed Coordination Models for Client/Server Computing", IEEE Computer, pp 14-22, April 1995.

-
- [10] Maffeis, "Adding Group Communication and Fault-Tolerance to CORBA", in Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies. Monterey, CA. The USENIX Association, June 1995.
 - [11] S. Maffeis, "Piranha -- A CORBA Tool for High Availability", IEEE Computer (April 1997).
 - [12] Object Management Group, "The Common Object Request Broker: Architecture and Specification", Revision 2.0, July 1995.
 - [13] Post Modern Computing, ORBELINE User's Guide.
 - [14] D. C. Schmidt, "The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications", 11th Sun Users Group Conference, June 1994.
 - [15] D. C. Schmidt, "IPC SAP: An Object-Oriented Interface to Operating System Interprocess Communication Services", C++ Report, SIGS, Vol. 4, No. 8, November/December, 1992, pp. 1--10.
 - [16] D. C. Schmidt, "Acceptor and Connector, Design Patterns for Initializing Communication Services", 1st European Conference on Pattern Languages of Programs, Kloster Irsee, July, 1996.
 - [17] D. C. Schmidt, S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded Servers -- the Thread-Pool Concurrency Model", SIGS, Vol 8. No 4. April 1996.
 - [18] D. C. Schmidt, S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded Servers -- the Thread-per-Request Concurrency Model", SIGS, Vol 8. No 2. February 1996.

-
- [19] D. C. Schmidt, S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded Servers -- the Thread-per-Session Concurrency Model", SIGS, Vol 8. No 7. July 1996.
 - [20] D. C. Schmidt, "An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit", Washington University technical report WUCS-95-31.
 - [21] J. Warland, Communication Networks: A First Course. Richard D. Irwin Inc., and Aksen Associates Inc., 1991.
 - [22] F. Halsall, Data Communications, Computer Networks and Open Systems. Fourth Edition Addison-Wesley Publishing Company Inc., 1996.
 - [23] A. S. Tanenbaum, Distributed Operating Systems. Englewood Cliffs, NJ: Prentice Hall, 1995.
 - [24] B. Meyer, Object Oriented Software Construction. Prentice Hall, 1988.
 - [25] PARASOL Users' Manual, Version 3.1, School of Computer Science, Carleton University.
 - [26] A PARASOL Tutorial, School of Computer Science, Carleton University.
 - [27] RADS: Department of Systems and Computer Engineering, Carleton University, DECALS, User's Guide and Reference manual, August 1994.
 - [28] H.W. LOCKHART, OSF DCE Guide to Developing Distributed Applications, J. Ranade Workstation Series McGraw-Hill, Inc. 1994.
 - [29] D.C. Schmidt. "A Family of Design Patterns For Flexibly Configuring Network Services in Distributed Systems", in Proceedings of the International Conference on Configurable Distributed Systems, Annapolis, Maryland, May 6--8, 1996.
 - [30] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of

- Reusable Object-Oriented Software”, Reading, MA: Addison-Wesley, 1994.
- [31] A.O. Allen, Probability, Statistics and Queueing Theory with Computer Science Applications, Second Edition. Academic Press, Inc., 1990.
- [32] C.M. Woodside, J.E. Neilson, D.C Petriu and S. Majumdar, “The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server like distributed Software”, IEEE Transactions on Computer, vol. 44, no. 1, pp. 20-34, January, 1995.
- [33] M.K. Molloy, “Performance Analysis Using Stochastic Petri Nets”, IEEE Transactions On Computers, Volume C-31, Number 9, September 1982, 913-917.
- [34] C.M Woodside, M. Courtois, “Resource Functions: Measurement and Modelling”, Real-Time and Distributed System (RADS) group, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada. June, 1997.
- [35] A System of Reusable Design Patterns for Application-level Gateways, The Journal of Theory and Practice of Object Systems Special Issue on Patterns and Pattern Languages, 1996.
- [36] Jon Siegel, CORBA-Fundamentals and Programming, John Wiley & Sons, Inc., 1996.
- [37] J.R. Nicol, C.T. Wilkes, and F.A. Manola, “Object Orientation in Heterogeneous Distributed Computing Systems”, IEEE Computer 26, June 1993.
- [38] R.G. Lavender, D.C. Schmidt, “Active Object: an Object Behavioral Pattern for Concurrent Programming”, in Proceedings of the 2nd Pattern Languages of Programs Conference, September 1995.
- [39] A. Istabrak, “Performance of COBRA-based client-server architectures”, Carleton

University, Ottawa, Master of Engineering (M.Eng) Thesis, 1997.

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc.
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved