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

**OO Multi-Threaded Servers: Design with reusable
components, Performance Measurements and Analysis**

by

**Gurudas Somadder
B.E.**

**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**

February 12, 1997

© 1997, Gurudas Somadder

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.

0-612-22128-8

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

*OO Multi-Threaded Servers: Design with reusable components,
Performance Measurements and Analysis*

Submitted by Gurudas Somadder
in partial fulfillment of the requirements for
the degree of Master of Engineering

Dr. D.C. Petriu, Thesis Supervisor

Dr. S.A. Mahmoud, Chair, Department of Systems and Computer Engineering

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
February 12, 1997

Abstract

The first object of this research was to develop an efficient approach to build multi-threaded servers by using reusable components provided in the object-oriented framework Adaptive Communication Environment (ACE) described in [Sch94Apr1]. The thesis presents the pattern-based design, implementation and performance measurements of different alternatives of multi-threaded servers for distributed applications. The multi-threaded approaches are compared based on their performance under different loads and system architectures, their execution overheads and consumption of shared system resources (such as memory and I/O ports). It is shown that for certain system architectures, server multi-threading can improve the overall performance of the system by allowing for a more efficient use of the hardware resources. The impact of assigning priorities to the service requests (as opposed to the clients) and a rationale for establishing such priorities depending on the workload characteristics are also discussed.

Another objective of the research was to find a way to exploit the design patterns used in object-oriented systems for performance analysis. Design patterns describe recurring solutions to common problems from various application domains, and provide a more abstract view of the system behavior as they concentrate on the main interactions between the system components. It is shown that patterns can guide developers in choosing what to measure, in instrumenting the code for measurements and in interpreting the results. It is emphasized that an abstract behavioral model, such as the one provided by design patterns, is especially useful in systems built with reusable components, both for understanding what the reused software does and its contribution to the overall performance.

Acknowledgments

I would like to express my deep gratitude to my supervisor, Dr. D. C. Petriu, for her valuable guidance, advice and encouragement throughout this research.

I would also like to thank Dr. C.M. Woodside and Dr. J. Rolia of the Real-Time and Distributed Systems (RADS) Group of the Department of Systems and Computer Engineering for their support and advice during the research.

Many thanks to my colleagues, Istabrak, Fahim, Hesham, Marc, Alex, Cheryl and Leslie for their support, advice, help and encouragement during the course of this thesis.

The financial assistance of the Telecommunications Research Institute of Ontario (TRIO), through its operating grant program, is gratefully acknowledged.

Finally I would like to thank my parents, sisters and last but not the least, my brother-in-laws, for their invaluable support, patience, understanding and encouragement which made this thesis possible.

Contents

1. Introduction	1
1.1 Research Background	1
1.1.1 OO Technology for Distributed Systems	3
1.2 Research Motivation	5
1.3 Research Goals	6
1.4 Contributions and Publications	6
1.5 Thesis Overview	8
2. Threads and Threading Models	10
2.1 Introduction	10
2.2 Solaris 2.5 Multi-threading Architecture	12
2.2.1 Processes and Threads	13
2.2.2 Solaris Multi-threaded Architecture	14
2.3 Threading Models	17
2.4 Thread Priorities and Thread-based Measurements	18
2.4.1 Thread Priorities	18
2.4.2 Thread-based Measurements	19
2.5 Benefits/Drawbacks of Thread-based Concurrent Programming	20
2.5.1 Benefits of using Thread-based Concurrent Programming	20
2.5.2 Drawbacks of using Thread-based Concurrent Programming ...	21
3. The Adaptive Communication Environment	24

3.1	Introduction	24
3.2	The ACE Toolkit	25
3.2.1	C++ Wrappers	26
3.2.2	Class Categories and Frameworks	26
3.3	Design Patterns: Concept and Use	32
3.3.1	The Reactor Pattern	33
3.3.2	The Acceptor Pattern	35
3.3.3	The Connector Pattern	38
3.3.4	The Active Object Pattern	41
4.	Client/Server Design and Implementation	45
4.1	Introduction	45
4.2	Client/Server Architecture from an ACE Viewpoint	47
4.2.1	The Client	47
4.2.2	The Server	51
4.2.2.1	Single Threaded Server	52
4.2.2.2	The Thread-per-Request Server	55
4.2.2.3	The Thread-per-Client Server	57
4.2.2.4	The Thread-Pool Server	59
4.3	System Architecture	60
4.3.1	Single-Threaded C/S Architecture	61
4.3.2	Thread-per-Request C/S Architecture	62
4.3.3	Thread-per-Client C/S Architecture	62

4.3.4	Thread-Pool C/S Architecture	64
4.4	A Use Case Map Representation of the System Execution Cycle	64
4.5	Systems with Layered Servers	66
5.	Experimental Setup and Measurements	68
5.1	Introduction	68
5.2	Experimental Setup	69
5.2.1	Experiment Controller and Data Logger	70
5.2.2	DECALS Architecture	71
5.3	Measurement Results and Performance Analysis	72
5.4	Pattern-guided Measurements and Analysis	79
5.4.1	Frequently executed patterns	80
5.4.1.1	Reactor Pattern	80
5.4.1.2	Active Object Pattern	83
5.4.2	Detailed Measurement Results and Performance Analysis	86
5.5	Comparison between the Thread-per-Client and the Thread-Pool Server Models	92
6.0	Priority Software Servers	96
6.1	Priority Server Design	97
6.2	Priority Queue Design	97
6.3	Service Priority	99
6.4	Experimental Setup	100
6.5	Measurement Results and Analysis for Service Priority	101

6.5.1 Measurement Results for the Pure Server Architecture.....	102
6.5.1.1 Effect of Changing Service Time Ratios	106
6.5.1.2 Impact of Changing the Number of Threads in the Thread-Pool	108
6.5.2 Measurement Results for the Layered Server Architecture	109
6.6 Measurement Results and Analysis for Class Priority	112
7. Conclusions	115
7.1 Summary	115
7.2 Future Work	117
Appendix A Notations for Use Case Maps and UML	118
A.1 Use Case Map Notations	118
A.2 UML Notation	118
Appendix B Batch Script for Conducting Experiments	120
B.1 Overview	120
B.2 Batch Scripts	120
B.3 Logging Data	125
B.4 Overview of Data gathering	125
References	126

List of Figures

Fig. 1	Solaris Multi-Threading Architecture	14
Fig. 2	Structure of the Adaptive Communication Environment Toolkit	26
Fig. 3	ACE Class Categories	28
Fig. 4	Participants in the Reactor Pattern	34
Fig. 5	Participants in the Acceptor Pattern	36
Fig. 6	Participants in the Connector Pattern	39
Fig. 7	Participants in the Active Object Pattern	42
Fig. 8	Active/Passive Connection Roles	45
Fig. 9	A High Level Hardware View of the System	46
Fig. 10	Class Components for the Client	48
Fig. 11	Interaction Diagram for the Client	50
Fig. 12	Class Components for the Single-Threaded Server	52
Fig. 13	Interaction Diagram for the Single-Threaded Server	54
Fig. 14	Interaction Diagram for the Thread-per-Request Server	56
Fig. 15	Interaction Diagram for the Thread-per-Client Server	58
Fig. 16	Interaction Diagram for the Thread-Pool Server	60
Fig. 17	Single-Threaded C/S Architecture	61
Fig. 18	Thread-per-Client C/S Architecture	62
Fig. 19	Thread-Pool C/S Architecture	63

Fig. 20	Use Case Map Representation of the System for a Single-Threaded Server	65
Fig. 21	Pure/Layered Server Architecture	66
Fig. 22	Experimental Setup	70
Fig. 23	Measurement Results for the Pure Server Architecture	73
Fig. 24	Server Utilizations for the Pure Server Architecture	74
Fig. 25	Measurement Results for the Layered Server Architecture	76
Fig. 26	Mid-level Server Utilization for the Single-Threaded Server	77
Fig. 27	Mid-level Server Utilization for the Thread-Pool Server	78
Fig. 28	Execution Sequence Chart for the Reactor Pattern	80
Fig. 29	Execution Sequence Chart for the Active Object Pattern	84
Fig. 30	Measurement Results for the <code>handle_events()</code> loop	85
Fig. 31	Measurement Results for the <code>register_handler()/remove_handler()</code> methods	88
Fig. 32	Measurement Results for the <code>insert()/remove()</code> methods	89
Fig. 33	Memory Consumption due to Threads	91
Fig. 34	Effect of Different Service Times and Message Sizes	92
Fig. 35	Heap Implementation of the Priority Queue	97
Fig. 36	Measurement Results for the Pure Server Arch. (Low Priority - Low Service Times)	102

Fig. 37	Measurement Results for the Pure Server Arch. (High Priority - Low Service Times)	103
Fig. 38	Thread-Pool Server Utilizations for the Pure Server Arch. (With Priority)	104
Fig. 39	CPU Overheads of the Different Components of the Thread-Pool Server	106
Fig. 40	Effect of Changing Service Time Ratios.	107
Fig. 41	Impact of Changing the Number of Threads in the Thread-Pool	108
Fig. 42	Measurement Results for the Layered Server Arch. (Low-Priority - Low Service Times)	110
Fig. 43	Measurement Results for the Layered Server Arch. (High Priority - Low Service Times)	111
Fig. 44	Comparison of CPU Utilizations for the Layered Server Arch. (With Priority)	112
Fig. 45	Measurement Results for Class Priority for the Layered Server Architecture	113

List of Symbols and Terminology

ACE	Adaptive Communication Environment
API	Application Programming Interface
ASX	Adaptive Service Executive
COOTS	Conference on Object-Oriented Technologies and Systems
C/S	Client/Server
DECALS	Distributed Experiment Control and Logging System
EuroPloP	European Conference on Pattern Languages of Programming
HWP	Heavyweight Process
ICDP	International Conference on Distributed Platforms
ICODP	International Conference on Open Distributed Processing
IP	Internet Protocol
IPC	Inter-process Communication
LWP	Lightweight Process
MT	Multi-Threaded
OO	Object-Oriented
OS	Operating System
RPC	Remote Procedure Call
SAP	Service Access Point
ST	Single-threaded
TPC	Thread-per-client

TPOOL	Thread pool
TPR	Thread-per-request
UML	Unified Modelling Language

Chapter 1

Introduction

1.1 Research Background

A *distributed system* is a combination of hardware, software, and network components in which the software components execute on two or more computers (also known as *nodes*) and communicate via a network. A distributed system possesses certain unique advantages over computing with multi-user systems (typically mainframes and minicomputers) and single-user systems (typically personal computers and workstations). It provides the advantages of a multi-user system that are lacking in a single-user system like resource sharing, widespread access to data, centralized services, multi-user applications etc. It preserves the advantages of single-user systems that are lost in multi-user systems like price to performance ratio, standard software packages, ability to use current computing platforms etc. It also has some inherent advantages like flexibility (in terms of location of hardware, mobility, reconfigurability etc.), scalability (in terms of additional components that can be added), efficiency (due to the fact that they consist of heterogeneous components which can be so selected that they are optimal for the task at hand) and availability (as critical components can be duplicated and thus the system can be made immune to single points of failure).

In order to take advantage of distributed computing, a new paradigm was developed called the Client/Server (C/S) paradigm which involved *distributing* applications onto

different *nodes* in a network, in which certain applications (the *clients*) would generate requests for service from other applications (the *servers*) who would service these requests.

Although experience has shown [Sch95Jan, Sch95Feb, Dilley95] that distributed computing, based on a C/S paradigm, can indeed offer these benefits when applied properly, developing distributed applications whose components collaborate efficiently, reliably, transparently and scalably is a complex task. Much of this complexity arises from the limitations with conventional tools and techniques used to develop distributed application software. For example, most standard network programming mechanisms (such as BSD sockets and Windows NT named pipes) lack typesafe, portable, reentrant and extensible interfaces. A typical example of this is the weakly-typed I/O handles used by sockets and named pipes to identify endpoints of communication. These handles increase the potential for subtle run-time errors since compilers cannot detect type mismatches at compile-time.

Another factor that makes distributed systems more difficult to implement is their involvement of *multiple technologies* that are the result of the partitioning of computer technology into specialized areas. Each area deals with solving the problems of its own domain using concepts and terminology that have been developed to serve its specialized needs. Although this would have been fine in a typical monolithic system that can be built using tools from just one of these specialties, distributed systems invariably involve several of them. It is often difficult even to understand how a concept from one field works with a concept from another. It is also difficult to design a system that mandates an interface

between two technologies when you cannot determine if they are even intended to be compatible. These, and other such problems arising due to distributed computing, are most apparent in the area of computer communication which involves communication between heterogeneous nodes in a network, each with its own individual data formats and communication protocols. In order to overcome these difficulties, *middleware* standards and products have appeared to facilitate distribution like the Distributed Communication Environment (DCE), proposed by the Open Software Foundation (OSF) group [Lockhart94] and Common Object Request Broker Architecture (CORBA), proposed by the Object Management Group (OMG) [Siegel96].

However, although the middleware products based on these standards comprise a wide variety of tools and services, they are quite often *heavyweight and slow* in nature [Sch95Jun]. Due to this fact, there was felt a need for *lightweight* libraries of reusable components which can be used as building blocks for distributed applications. An example of such a product is the Adaptive Communication Environment (ACE) which is used in this thesis [Sch94Apr1].

1.1.1 OO Technology for Distributed Systems:

OO technology provides distributed computing with many of the same benefits (such as encapsulation, interface inheritance and reuse, parameterized types, object-based exception handling, portability and extensibility) as it does for non-distributed computing. *Encapsulation* promotes the separation of interface from implementation, which is crucial for developing highly extensible architectures that decouple reusable application-independent mechanisms from application-specific policies. Interface inheritance and

parameterized types promote *reuse* and emphasize commonality in a design. Object-based exception handling often simplifies program logic by decoupling error-handling code from normal application processing. In addition, the inherently decentralized nature of OO computing provides a natural way to design distributed applications, which are themselves based on a decentralized architecture. For example, most distributed applications do not have direct access to global resources, as a result of which they interoperate by passing messages. This message passing mechanism is very similar to method invocation on an object in OO programming. Thus, OO technology is ideally suited to the development of extensible, robust, portable and efficient distributed applications.

Even with OO technology, developing communication software that is reusable across OS platforms is hard. Constraints imposed by the underlying OS platforms may make it impractical to reuse existing algorithms, detailed designs, interfaces implementations directly. However, most solutions to fundamental problems in communication software follow certain basic patterns. Such patterns, known as *design patterns* [Gam94, Sch96, Coplien94, Gilbert95], concentrate on the key collaborations between participants in a software architecture, instead of overwhelming the developer with details. Design patterns capture the static and dynamic structures of solutions that occur repeatedly when developing applications in a particular context, and help in experience reuse at the architectural level, and thus ease development efforts and costs. Due to their benefits, developers are turning more and more to a OO design pattern based approach, for the design of communication software.

1.2 Research Motivation

Most distributed applications, based on a C/S paradigm, have an inherently concurrent nature, as more than one service request from different clients may arrive at the servers at the same time. For example, some distributed applications use software servers like name servers, networks file servers etc. They can benefit from using a concurrent model of execution to perform their functions, as queuing delays caused by contention between requests for services are reduced when several requests can be processed at the same time. Although concurrency can be provided by using multiprocessor platforms, the complexity and cost of such applications usually makes it infeasible. Instead, many uni-processor applications rely on *multi-threading* to provide concurrent services. A *thread* is an independent series of instructions executed concurrently with other threads *within* a single process address space. Multi-threaded server applications handle concurrent requests by allowing each thread to deal with one request at a time, in parallel with other threads. This not only simplifies the code and improves its understanding, but also, for some system architectures, improves the overall performance of the system. Various communication midwares (like DCE and CORBA implementations) make use of multi-threading to provide concurrency. However, different implementation techniques introduce different overheads and consume different system resources. This thesis compares several approaches for implementing multi-threaded servers in a distributed environment using the thread-encapsulation library of the Adaptive Communication Environment (ACE) reusable component object-oriented toolkit [Sch95Apr1], and identifies cases where multi-threading yields performance dividends.

1.3 Research Goals

The main goal of the thesis is to analyze and compare different approaches for building multi-threaded servers in a distributed environment by using the reusable components contained in ACE. The conclusions of such an analysis may be used by future developers of distributed applications.

The analysis and comparison takes into account several factors, such as the overall system response times and throughputs obtained under different workloads; the execution overheads of various threading models and their causes; the use of other system resources (such as memory, I/O ports etc.). This thesis also takes the relationship between design patterns and performance a step further, promoting the idea that patterns can be used not only to document and explain reusable components, but also to guide the performance measurements and analysis of such components and of the systems incorporating them. The thesis also assesses the role played by OO design patterns in building distributed systems with reusable components and in analysing the performance of such systems. This research also studies the impact of assigning priorities to service requests, and comes up with a rationale for assigning priorities to client requests [Li95], depending on the workload characteristics, for different server architectures.

1.4 Contributions and Publications

Research contributions of the thesis:

- Experiment in software reuse: Designed and implemented several multi-threaded servers based on the ACE reusable component toolkit. Built distributed test beds for layered C/S

systems.

- Developed a technique to automate system measurements (like CPU usage, memory consumption, etc.) on a *per-thread* and *per-process* basis, based on UNIX batch scripts. These scripts set up, launch, control and cleanup after the different sets of distributed experiments. They change the different workload parameters (like number of threads, service times, think times, number of clients etc.) on the fly for each set of experiments, collect statistics and process them by computing averages and confidence intervals.
- Compared several approaches for implementing multi-threaded servers in the ACE environment and identified cases where multi-threading yields performance dividends. The results are presented in the technical report SCE-97-02 (also accepted at the International Conference on Open Distributed Processing, ICODP/ICDP, 1997).
- Used design patterns to identify key interactions between components in the software architecture, and thus guide performance measurements. This thesis emphasizes that an abstract model, such as provided by design patterns, is especially useful when a system is built with reusable components, both for understanding what the reused software does, and what is its contribution to system performance. This idea is presented in the technical report SCE-97-03 (also submitted to the Conference on Object Oriented Technologies and Systems, COOTS, 1997).
- Studied the impact of assigning priorities to service requests, and put forward a rationale for assigning priorities to service requests, depending on workload characteristics.

Publications issued from the thesis are:

- G. Somadder and D.C. Petriu, "Performance Measurements of Multi-Threaded Servers

in a Distributed Environment,” *Report: SCE-97-02, Dept. of Systems and Computer Eng., Carleton University* - accepted at the ICODP/ICDP’97, Toronto, Canada, May 1997.

- G. Somadder and D.C. Petriu, “Pattern-guided Performance Measurements and Analysis,” *Report: SCE-97-03, Dept. of Systems and Computer Eng., Carleton University* - submitted to the USENIX COOTS’97, Portland, U.S.A., June 1997.
- G. Somadder and D.C. Petriu, “Performance Analysis Patterns for Multi-Threaded Servers,” - in progress for submission to the European Conference on Pattern Languages of Programming, EuroPLoP, June 1997.

1.5 Thesis Overview

Chapter 2 provides an overview of concurrent programming and multi-threading, defines key terminology, highlights the salient points of the various alternative mechanisms available for concurrent programming and discusses the measurement techniques used in collecting data for the case studies.

Chapter 3 has two parts. The first part describes the structure and functionality of the ACE toolkit, while the second part presents the concept of design patterns, and discusses some of the ACE design patterns relevant to the thesis.

Chapter 4 provides an in-depth discussion of the design and implementation of the C/S system implemented and used for measurement purposes.

Chapter 5 has two parts: the first describes the experimental setup used for conducting performance measurements, and the second presents a detailed analysis of the results obtained, first from a system level viewpoint, and then from a pattern-guided viewpoint.

Chapter 6 studies the design and implementation of the priority software servers, and presents a detailed analysis of the results obtained. It comes up with a rationale for assigning priorities to client requests depending on the workload characteristics for different server architectures.

Finally, Chapter 7 presents conclusions and directions for future research.

Chapter 2

Threads And Threading Models

This chapter provides an overview of concurrent programming and multi-threading, defines key terminology, highlights the salient points of the various alternative mechanisms available for concurrent programming and discusses the measurement techniques used in collecting data for the case studies. It concentrates only on the Solaris 2.5 operating system because the ACE toolkit (discussed in Chapter 3) used in this thesis was compiled on Solaris 2.5.

2.1 Introduction

Distributed applications are designed to take advantage of connectivity, inter-networking, parallel processing, replication, extensibility and cost effectiveness offered by distributed computing. Some distributed applications use software servers like name servers, network file servers etc. These applications are difficult to develop using single threaded processes. For example, a single-threaded network file server cannot block for extended time periods handling one client request since the responsiveness for other clients would suffer. Such applications can benefit from using a concurrent model of execution to perform their functions, as queuing delays caused by contention between requests for services are reduced when several requests can be processed at the same time. In order to provide concurrent services, various workarounds have been developed, some of which are described.

- **Event demultiplexer and dispatcher** - This technique is widely used to manage multiple input devices in *single-threaded* frameworks. The main event demultiplexer/dispatcher detects an incoming event, demultiplexes the event to the appropriate event handler, and then dispatches an application-specific callback method associated with the event handler. The primary drawback with this approach is that long duration services drastically degrade the responsiveness of the system.
- **User-level co-routines** - This technique enables tasks to *voluntarily* suspend their *own* execution until another co-routine resumes them at a later point. It involves developing non-preemptive, user-level co-routine packages that explicitly save and restore context information. An example of this system is the multi-tasking mechanisms available on Windows 3.1 systems. However, co-routines are difficult to develop on account of their complexity (due to task preemption). It is also difficult to program co-routines correctly, since developers must modify their programming style to avoid certain operating system (OS) features (such as asynchronous signals). Also, they are mostly suitable for short duration tasks as otherwise they would lead to significant queuing delays.
- **Multi-processing** - This technique tries to alleviate the complexity of designing single-threaded applications meant to handle concurrent requests, by making use of the coarse grained multi-processing capabilities provided by the UNIX *fork* and *exec* system calls. *Fork* spawns a separate child process that executes a task concurrently with its parent. Inter-process communication (IPC) is achieved by using IPC mechanisms such as shared memory and memory-mapped files. However, the overhead and inflexibility of creating and using processes using *fork* and *exec* makes dynamic process invocation prohibitively

expensive and overly complicated for most applications. Moreover, it is difficult to exert fine-grain control over scheduling and process priority using *fork* and *exec*.

- **Multi-Threading** - Multi-threading mechanisms provide a more elegant, and sometimes more efficient, way to overcome the limitations of the traditional concurrent processing techniques described above.

Every thread deals with one request at a time, in parallel with the other threads. This simplifies the code and improves its understanding. An added bonus is that, for some system architectures, multi-threading services improves the overall performance of the system. Intuitively, multi-threading is a means to improve the performance of software servers by allowing concurrent processing of requests. Various communication midwares (like DCE and CORBA implementations) make use of multi-threading to provide concurrency (It should be mentioned at this point that CORBA itself does not have threads, but its implementations can have, by incorporating them from standard packages). However, different implementation techniques introduce different overheads and consume different system resources. The material presented in the rest of the chapter discusses the strategies and tactics of concurrent programming techniques using threads, and describes different threading models which can be used to effectively implement multi-threading.

2.2 Solaris 2.5 Multi-threading architecture

This section summarizes relevant background material on the multi-threading (MT) mechanisms provided by Solaris 2.5.

2.2.1 Processes and Threads

A traditional UNIX *process* is a collection of resources like virtual memory, I/O descriptors, a run-time stack, signal handlers, user and group ids, and access control tokens executing in a *single-thread* of control. Such a process is also sometimes known as a *heavy-weight process* (HWP). A thread, on the other hand, is an independent series of instructions executed *within* a single process address space. In addition to its own instruction pointer, a thread contains other resources such as a run-time stack of function activation records, a set of general purpose registers, and thread-specific data. A multi-threaded process consists of one or more independently executing threads. These threads, often known as *light-weight processes* (LWP), maintain a minimal state information, require less overhead to spawn and synchronize, and inter-communicate via shared memory rather than message passing. Conventional workstation operating systems (such as variants of UNIX and Windows NT) support the concurrent execution of multiple processes, each of which may contain one or more threads. A process serves as the unit of protection and resource allocation within a separate hardware protected address space. A thread serves as the unit of execution that runs within a process address space that is shared by zero or more threads.

2.2.2 Solaris Multi-threaded architecture

A typical Solaris 2.5 multi-threading architecture [Pow91, POSIX] operates at two levels (kernel space and user space) and consists of the following components, as shown in Fig. 1:

- **CPUs** - They execute user-level and kernel-level instructions. The semantics of the Solaris MT model are intended to work for both uni-processors and symmetrical multi-processors on shared memory hardware.
- **Kernel Threads** - These are the *fundamental entities* that are scheduled and executed by the CPU in kernel space. Each kernel thread has a small data structure and stack, maintained by the OS kernel, and thus makes context switching between kernel threads relatively fast, as virtual memory information remains unchanged.
- **Lightweight Processes (LWPs)** - These are associated with kernel threads, and can be thought of as a “*virtual CPU*”, upon which application threads are scheduled and multiplexed by a user-level thread library. In Solaris 2.5, each UNIX process, instead of having a single thread of control, contains one or more LWPs, each of which is associated with a kernel thread. Application level threads are scheduled by

the Solaris 2.5 kernel level scheduler using LWPs. Context switching between LWPs is relatively slow, compared to context switching between kernel level threads, as each LWP maintains a large amount of state (virtual memory address ranges, timers, register data etc.). LWPs are scheduled by the kernel onto the available CPU resources according to their scheduling class and priority. However, switching between LWPs in a process is still much faster than switching between processes.

- **Application Threads** - Application threads are spawned by the application. Each application thread has a unique stack and register set, although it shares the process address space with other threads. Application threads are scheduled and multiplexed onto available LWPs by a user-level thread library. Within a process, each of these application threads execute independently, although not necessarily in parallel (depends on the hardware). Solaris 2.5 provides a multi-level concurrency model that permits application threads to be spawned and scheduled using one of the following two modes:

1. **Bound** - Each bound thread is mapped onto its corresponding LWP. If the underlying hardware consists of multiple CPUs, then programmers can take advantage of multi-processing by forcing independent tasks to execute in parallel on multiple CPUs. Thus, if two application threads are running on separate LWPs (and thus separate kernel threads), they may execute in parallel, provided that they are running on a multiprocessor or using asynchronous I/O. A kernel-level context switch is required to re-schedule bound threads. Also, OS kernel intervention is required for synchronization operations on bound threads. The number of bound threads is limited by the underlying kernel resources, as each bound thread requires the allocation of kernel resources.

2. *Unbound* - These are scheduled onto a pool of LWPs by a user-defined thread library which implements a non-preemptive cooperative multi-tasking concurrency model. Preemption would require saving the state of each thread preempted, in order for the thread to resume execution from the same state, which would lead to highly complex library designs with no obvious benefits due to the overheads involved. It invokes LWPs as needed and assigns them to execute runnable threads (there may be a many-to-one relationship between unbound threads and runnable LWPs). After assuming the state of the thread, the LWP executes its instructions until completion, or till it gets blocked on a synchronization mechanism, at which time the thread library schedules another LWP to run. Unbound threads consume relatively less system resources and incur relatively lower overheads to spawn, context switch, and synchronize in comparison with bound threads. Also, since each unbound thread does not allocate kernel resources, it is possible to allocate a very large number of unbound threads without significantly degrading performance.

2.3 Threading Models

Multi-threaded servers are designed to handle multiple client requests simultaneously. They help not only in simplifying program design (as they allow multiple server tasks to proceed independently using conventional programming abstractions) but also in improving system performance by using the parallel processing capabilities of multi-processor hardware platforms and overlapping computation with communication. This section discusses some of the common models [Sch96Feb] adopted in the design of multi-threaded servers.

- **Thread-per-request** - In this model, the server spawns off a thread for each request from a client, which means that the creational overhead for a thread is incurred for every request. In order to minimize consumption of OS resources (when many clients make requests simultaneously) and the overheads associated with spawning a thread, this model is useful only for long-duration requests from clients.
- **Thread-per-client** - The server spawns off a thread for each client, which is exclusively associated with it, for the entire period during which the client is connected to the server. This amortizes the cost of spawning a thread across multiple requests. It is useful in scenarios where the servers carry on long-duration conversations with multiple clients. Given sufficient OS resources, and multiple CPUs, this model can achieve the highest throughput. This model reduces to the thread-per-request model for scenarios in which the client makes only one request.
- **Thread pool** - This model tries to alleviate the cost of spawning a thread by pre-spawning a pool of threads, whose number may be fixed, or changed dynamically as per requirements. It is useful for scenarios in which OS resources are limited, so that resource consumption is minimized. However, it has the overhead of queuing delays to gain access to threads, when the number of simultaneous client requests exceeds the number of threads in the pool, at which time all the threads are busy. This model requires the most effort by the programmer to function as intended.
- **Thread-per-object** - In this model, each *service* in the server, associated with an object, is executed in a separate thread of control. This model is useful when the client requests

are distributed evenly among the objects, otherwise objects receiving a higher percentage of requests will become a performance bottleneck.

2.4 Thread Priorities and Thread-based Measurements

This section provides a brief overview of the different system calls and techniques used to set thread priorities and conduct measurements on a per-thread basis.

2.4.1 Thread Priorities

Solaris threads are implemented as a library, using underlying LWPs which are supported by the kernel. The threads library schedules threads on a pool of LWPs in the process, in much the same way as the kernel schedules LWPs on a pool of processors. LWPs in the system are scheduled by the kernel onto the available CPU resources according to their scheduling class and priority. Scheduling of threads is controlled through the Solaris threads library functions *thr_setconcurrency()*, *thr_setprio()*, and the *THR_NEW_LWP* option of *thr_create()* [Solaris]. Unbound thread scheduling uses simple priority levels with no adjustments and no kernel involvement. Thread scheduling regulates only how threads are assigned to LWPs. It has no effect on scheduling of LWPs by the kernel. The LWP's system priority is usually uniform and is inherited from the creating process.

Solaris 2.5 kernel has three classes of scheduling which are maintained for each LWP. Each scheduling class maps the priority of the LWP it is scheduling to an overall dispatching priority according to the configurable priority of the scheduling class. The three scheduling classes, in terms of descending order of priority, are: *real time* (RT), *system*, and *timeshare* (TS - default scheduling class). When a process is created, its one

initial LWP inherits the scheduling class and priority of the parent process. Unbound threads in a process have the same scheduling and class priority. On the other hand, bound threads have the scheduling class and priority of their underlying LWPs. Each bound thread in a process can have a unique scheduling class and priority that is visible to the kernel. Bound threads are scheduled with respect to all other LWPs in the system. Scheduling class (i.e. RT or TS) for both bound and unbound threads can be set by the *prctl(2)* system call. This system call is also used to change the priority of a bound thread, by changing the priority of the LWP to which it is bound. In order to dynamically change the priority of an unbound thread, the *thr_setprio()* system call can be used by passing the calling threads identifier and the required priority. Currently, Solaris supports a priority level ranging from 0 to 127. The target thread will *preempt* lower priority threads, and will yield to higher priority threads in their contention for LWPs, *not for CPUs*.

Thread priorities set in this manner regulate access to LWPs, not CPUs, and hence are different from real-time priorities, which regulate and enforce access to CPU resources. A thread's priority set via these system calls is more like a hint in terms of guaranteed access to execution resources. The case studies performed on priority servers, in Chapter 7, use *thr_setprio()* to change the priorities of *unbound* threads only.

2.4.2 Thread-based Measurements

One of the major problems in thread-based experiments is the collection of data like CPU usage etc. on a per-thread basis. This arises due to the fact that most system calls used for measuring various parameters, like *getrusage()* which is used for measuring the CPU

usage, are on a per process basis, and do not provide data on a per-thread basis. However, Solaris does provide system calls which can measure resource usage on a per-LWP basis (For e.g. this thesis uses the *ioctl()* system call, which is a catchall of all system calls under Solaris). Thus, in order to measure resource usage on a per-thread basis, there has to be a one to one relationship between the thread and the LWP, which in turn means that the only threads which can be used to conduct per-thread measurements *have to be bound threads*. In Chapter 5, which explains the behavior of multi-threaded servers from a design pattern perspective, *ioctl()* system call has been used to obtain the process identifier for the servers, which in turn is fed back into the *ioctl()* function to obtain LWP identifiers. These identifiers are then used to calculate resource usage for the *bound* threads associated with them.

2.5 Benefits/Drawbacks of Thread-based Concurrent Programming

When used correctly, multi-threading provides a more elegant and potentially more efficient means to overcome the limitations with the other concurrent processing techniques described in Section 2.1. Some of the common advantages and disadvantages of using multi-threaded applications are outlined below:

2.5.1 Benefits of using Thread-based Concurrent Programming

- ***Creational Overhead*** - Spawning a thread consumes relatively fewer system resources as compared to forking off a new process, as it does not require duplicating the parent's address space memory or setting up new kernel data structures. In addition, it avoids using up a process slot in order to perform a subtask within a larger application.

- **Context switching Overhead** - As threads maintain minimal state information, context switching overhead is correspondingly reduced as less state information must be stored and retrieved, in comparison with context switching between UNIX heavy-weight processes. Moreover, threads that run strictly in user-level do not incur any kernel-level context switching overhead at all, as they are managed by the user-defined thread library.
- **Synchronization Overhead** - Thread synchronization is less expensive than process synchronization as it may not be necessary to switch between kernel-mode and user-mode when scheduling and executing an application thread. Also, entities being synchronized are most often local ones, so they do not involve any kernel intervention.
- **Communication Overhead** - Communication between separate threads usually takes place using shared memory, which is much faster than IPC message passing for inter-process communication, as it avoids the overhead of *explicit data copying*. A common example of this is cooperating database services that frequently reference common memory-resident data structures implemented using threads. In general, *using the shared address space of a process to communicate between threads is easier and more efficient than using shared memory mechanisms to communicate between processes*.

2.5.2 Drawbacks of using Thread-based Concurrent Programming

- **Concurrency Control Overhead** - Regardless of the hardware platform i.e. uni-processor or multi-processor, programmers must ensure that access to shared resources (such as files, databases records, network devices, terminals, shared memory etc.) is serialized to prevent *race conditions* - an erroneous condition arising due to the execution of two or more concurrent LWPs. Race conditions may be eliminated by using the Solaris

2.5 synchronization mechanisms, which introduce additional overheads (like queuing delays for gaining access to shared resources), and increase the complexity of the applications.

- **Stable Applications** - Multi-threaded application robustness may be reduced due to the fact that separate threads execute within the same process address space and are not protected from each other (in order to reduce context switching and synchronization overhead), which might lead to one (or more) thread(s) inadvertently corrupting the address space of other threads, thereby leading to unpredictable system behavior. *It requires a great deal of programming effort to develop stable complex multi-threaded applications.*

- **Security** - Since all threads in a process *share the same userid and access privileges* to files and other protected resources, it may not be possible to prevent accidental or intentional access to unauthorized resources. As a result of this, network services like Internet *ftp* and *telnet*, which base their security mechanisms on process ownership, are typically implemented in separate processes.

- **System Performance** - *A common misconception associated with threads is that multi-threading an application would automatically improve its performance.* In many scenarios, as will be shown in the following chapters, multi-threading does not improve performance, *specially for compute-bound applications running on a uni-processor workstation*, as it the CPU itself which saturates, and becomes the system bottleneck.

To conclude, programming single threaded distributed applications meant to handle concurrency is difficult, especially for server applications. A single threaded application

must be so designed that service requests are not *starved*, either by quickly handling all requests or by using heavyweight mechanisms like *fork* and *exec*. However, in practice, most non-trivial requests can't be served quickly enough to avoid starvation, neither can new processes be created such that system resource consumption is minimized, while at the same time keeping program design simple.

On the other hand, with multiple threads, each request can be serviced in its own thread, independent of other requests, thereby ensuring that clients are not starved. At the same time, system resource consumption is kept to a minimum, thereby freeing up resources for some other useful purposes. Thus, although multi-threaded systems are difficult to design and debug, their benefits often outweigh their drawbacks and help in creating simple designs and implementations than single-threaded programming.

Subsequent chapters will introduce the concept of ACE design patterns and will explain and evaluate the design and implementation of the above multi-threaded server models based on these design patterns.

Chapter 3

The Addaptive Communication Environment

The Adaptive Communication Toolkit (ACE) was developed at the department of Computer Science, Washington University, St. Louis, U.S.A., by Douglas C. Schmidt and his team [Sch94Apr]. It implements a set of reusable C++ wrappers, classes and frameworks, *based on a set of fundamental design patterns*, that perform common communication tasks such as event demultiplexing, event handler dispatching, connection establishment, routing, dynamic configuration of application services and concurrency control.

The primary goal of ACE is to simplify the development of concurrent event-driven communication software by simplifying the use of OS mechanisms that provide interprocess communication (IPC), communication port demultiplexing, explicit dynamic linking, and concurrency.

This chapter provides an overview of the structure and functionality of the ACE toolkit, introduces the concept of design patterns, and discusses some of the ACE design patterns relevant to the thesis [Sch94Apr1, Sch94Apr2, Sch95, Sch95Jan, Sch95Sept].

3.1 Introduction

Developing distributed applications whose components collaborate efficiently, reliably, transparently, and scalably is a highly complex task., mainly due to the limitations of conventional tools and techniques used to develop these applications. For instance, many

standard network programming mechanisms (such as BSD sockets and Windows NT named pipes) and reusable component libraries (such as sun RPC) lack typesafe, portable, reentrant, and extensible interfaces. A typical example would be the weakly typed I/O handles used to identify communication endpoints by both sockets and named pipes. These handles increase the potential for subtle run-time errors since compilers can't detect type mismatches at compile time. In addition, since most distributed applications are developed using functional decomposition techniques, extending such applications becomes a highly complex task. *One of the solutions to this problem is to shift to object-oriented technology* which provides distributed applications with benefits like encapsulation (which promotes the separation of interface from implementation, thereby decoupling reusable application-dependent mechanisms from application-specific policies), reuse (which emphasizes commonality in a design), portability and extensibility (by providing wrappers around actual system calls, thereby enabling developers to program distributed applications using familiar techniques such as method calls on objects) etc.

The next section provides an overview of the structure of the object-oriented ACE toolkit and describes its functionality.

3.2 The ACE Toolkit

The ACE toolkit is designed using a layered architecture, providing C++ wrappers that encapsulate OS mechanisms as well as higher level class categories and frameworks which implement strategic and tactical communication software design patterns (by extending the C++ wrappers).

3.2.1 C++ wrappers

The ACE toolkit improves application robustness by encapsulating OS communication, threading, and virtual memory mechanisms within type-secure *C++ wrappers* thereby reducing the need to directly access the underlying weakly-typed C library interfaces directly, which in turn allows the C++ compilers to detect type system violations at compile-time, instead of run-time.

These C++ wrappers help to simplify the design and implementation of robust, compact, portable, and extensible communication software.

3.2.2 Class Categories and Frameworks

A framework is an integrated collection of components that collaborate to produce a

reusable architecture for a family of related applications. Frameworks enable larger-scale reuse of software components than is otherwise obtained by reusing individual classes and stand-alone functions.

The lower level C++ wrappers, shown in Fig. 2, are grouped into *class categories*. These class categories consist of a collection of components that collaborate to provide a set of related interfaces and services [Bch93]. Some of the primary class categories supported by ACE, shown in Fig. 3 using the UML notation [Appendix A], include:

- **Reactor** - This class category performs event demultiplexing and event handler dispatching by encapsulating OS event demultiplexing mechanisms (such as *select* and *poll*). The Reactor shields application developers from error-prone, low-level details associated with programming existing I/O demultiplexing system calls (such as setting and clearing bitmasks, handling time-outs and interrupts, and dispatching callback methods). To facilitate application portability, the Reactor provides the same interface regardless of the event demultiplexing mechanism used (such as *select* and *poll*), thus allowing developers to concentrate on higher-level application-related issues, rather than repeatedly wrestling with lower-level event demultiplexing details. Moreover, in order to work correctly in a multi-threaded event processing environment, the Reactor also contains mutual exclusion mechanisms designed to perform callback-style dispatching correctly and efficiently. In addition, the Reactor class category supports transparent extensibility by using inheritance, dynamic binding and parameterized types, which decouples the lower-level I/O demultiplexing and service dispatching mechanisms (such as detecting events on multiple I/O descriptors, handling timer expiration etc.) from the higher-level application

processing policies (such as connection establishment, data transmission and reception, processing service requests from other participating hosts etc.).

Fig. 3 ACE Class Categories

- **IPC_SAP** - The Inter Process Communication Service Access Point (IPC_SAP) class category encapsulates the standard OS I/O handle-based local and remote IPC mechanisms that offer connection-oriented and connectionless protocols, such as the SOCK_SAP (which encapsulates the **socket** API), TLI_SAP (which encapsulates the TLI API), SPIPE_SAP (which encapsulates the UNIX SunOS 5.x **STREAM** pipe API) and the FIFO_SAP (which encapsulates the UNIX **NAMED** pipe API). Each of these subclasses provide a well-defined interface to a subset of local or remote communication mechanisms which, when put together, comprise the overall functionality of a particular communication abstraction (such as the Internet-domain or UNIX-domain protocol families). Compared to stand-alone functions, *these subclasses help to simplify network programming* by shielding applications from error-prone details (by providing a type-

secure C++ interface), combining several operations into one (For example, the SOCK_Acceptor class category *constructor* performs the various socket system calls like *socket*, *bind* and *listen*), parameterizing IPC mechanisms into applications (C++ features such as default parameter values and templates help to develop applications, parameterized at compile time, to operate correctly over either a *socket-based* or *TLI-based* transport interface), and finally, by enhancing code sharing (*inheritance-based* hierarchical decomposition increases the amount of common code that is shared amongst the various IPC mechanisms).

- **Connection** - It encapsulates *active/passive* connection establishment mechanisms. Its two subclasses are the *Connector* (which *actively* initiates a connection to a communication endpoint) and the *Acceptor* (which *passively* establishes an endpoint of communication). These subclasses decouple the passive and active connection roles, once the connection has been established, from the services performed. They help to enable flexible strategies for executing network services concurrently (once a connection is established, peer applications use the connection to exchange data to perform some type of service), reusing existing initialization code for new services (as initialization strategies tend to remain unchanged, even though service characteristics may change), efficiently establishing connections with large number of peers (by using *asynchrony* to initiate and complete connections in non-blocking mode) and making connection establishment code portable across platforms containing different network programming interfaces (by *parameterizing* platform dependent mechanisms for accepting and initiating connections).
- **Concurrency** - This class category encapsulates the corresponding Solaris and POSIX

Pthreads multi-threading and synchronization mechanisms. It automates the initialization of synchronization objects that appear as fields in C++ classes and also simplifies typical usage patterns for the threading and synchronization mechanisms. Each mutual exclusion wrapper subclass (encapsulating the *mutex*, *condition*, *semaphore* and *RW_Lock* mechanisms) shares a common interface (i.e. *acquire/release*), but possesses different serialization and performance properties. ACE also provides a *Thread_Manager* class that contains a set of mechanisms to manage groups of threads that collaborate to implement collective actions (like creating, suspending, resuming threads etc.).

- **Service Configurator** - It is responsible for automatic configuration and reconfiguration by encapsulating explicit *dynamic linking* mechanisms (such as *dlopen* and *dlsym*). Dynamic linking, in contrast to static linking (which only allows object files binding at compile-time and/or static link-time), enables the addition and/or deletion of object files into the address space of a process at initial program invocation or at any point later during runtime. Although SunOS 4.x and 5.x support both *implicit* dynamic linking (used to implement shared object files or libraries) and *explicit* dynamic linking (provides interfaces that allow applications to obtain, utilize, and/or remove the run-time address bindings of symbols defined in shared object files), the ACE Service Configurator class only supports the explicit dynamic linking mechanisms of SunOS. Using the other ACE components, the Service Configurator class extends the functionality of conventional daemon configuration and control frameworks (such as *listen*, *inetd* etc.) by providing automated support for static and dynamic configuration of concurrent, multi-service communication software, monitoring sets of communication ports for I/O activity and

dispatching incoming messages received on monitored ports to the appropriate application-specified services.

- ***Stream*** - This class category is the primary focus of the ACE toolkit. This class category contains the Adaptive Service eXecutive (ASX) framework [Sch94Apr2], which integrates the lower-level C++ wrapper components (like IPC_SAP) and higher-level class categories (like the Service Configurator). The ASX framework embodies, encapsulates, and implements key design patterns (design patterns help to enhance software quality by addressing fundamental challenges in large-scale development, as explained in the subsequent sections) that are commonly used to develop communication software. Moreover, the ASX framework separates communication software development into two distinct categories: (i) *application independent* - common to all (or most) communication software, such as event demultiplexing, queuing, port monitoring etc.) and (ii) *application specific* - which depend on an individual application. This helps in reusing the C++ wrappers and frameworks provided by ACE, thereby allowing developers to concentrate on key higher-level functional requirements and design concerns that constitute a particular application(s), instead of spending their time on reinventing solutions to commonly recurring tasks. It permits applications to consolidate one or more services into a single administrative unit, thereby helps in simplifying development by performing common service initialization activities automatically, reduces OS resource consumption by spawning service handlers *on demand*, and allows application services to be updated without modifying existing source code or terminating an executing dispatcher process (such as the *inetd* superserver). It allows run-time services to be performed by several

different types of process and thread execution agents, and thus increases the range of application concurrency configuration alternatives available to developers (as service functionality is decoupled from the execution agent).

3.3 Design Patterns: Concept and Use

Christopher Alexander, an architect, developed the idea of a *pattern language* to enable people to design their own homes and communities [Alex77]. A pattern language is a set of patterns, each of which describes how to solve a particular kind of problem. Alexander's pattern language, which starts by explaining from a world viewpoint, then to a nation view, then to smaller regions in a nation, until it finally it reaches to a room in a house, is supposed to be a document that non-architects can use to design their own communities and homes, without requiring any specialized training. It focuses on common design problems that non-architects encounter (like building bedrooms), rather than uncommon ones (like building cathedrals). Although Alexander talks in terms of *pattern languages*, it is not a formal language (like a context-free language), but is more like a structured essay. Thus, his term has been replaced by the more generic term *patterns*.

Any recurring solution(s) to a design problem(s) can be characterized by a specific pattern(s) which captures the static and dynamic structures of its solution(s). Such a pattern is known as a *design pattern* [Sch95Aug, Gam94, Sch96]. Design patterns facilitate architectural level reuse by providing *blueprints* that guide the definition, composition, and evaluation of key components in a software system. In general, a large amount experience reuse is possible at the architectural level. Reuse of such patterns helps in developing better software architectures due to their emphasis on the strategic

collaborations between key participants in a software architecture without overwhelming developers with excessive details. If used properly, design patterns can significantly reduce and simplify development efforts.

These ideas have been transferred to the field of OO design. Objects are the design elements that form patterns, the patterns are discernible in classes of cooperating objects, linked by certain relationships, that are repeated in the solutions to a particular class of problems. Originally, the OO paradigm promoted component reuse at the system-component level by establishing libraries of reusable classes organized in hierarchies of inheritance (frameworks of classes). However, now the focus is more on the relationship between patterns in any system design i.e. on *architecture reuse* rather than component reuse.

The subsequent sections describe, in detail, the structure, collaborations and working of the different participants in the ACE design patterns relevant to the thesis.

3.3.1 The Reactor Pattern

The Reactor is an object behavioral pattern that decouples event demultiplexing and event handler dispatching from the services performed in response to events. It provides a portable interface to an integrated collection of extensible, reusable and type-secure C++ classes that encapsulate and enhance the OS *select* and *poll* I/O demultiplexing mechanisms [Sch95]. By decoupling the policies (which are specific to an application) and mechanisms (which may be independent of an application, and thus reusable), a number of software quality factors, such as reusability and extensibility of system components, is increased.

The Reactor pattern addresses a key issue in single-threaded communication software - multiplexing different types of events from multiple events sources within a single thread of control. This is achieved by providing a coarse-grained concurrency control that serializes application event handling within a process at the event demultiplexing level, thereby minimizing the need for more complicated threading,

synchronization, or locking within an application.

Fig. 4 shows the collaborations between the different participants in the Reactor pattern, using the UML notation [Appendix A]. The key participants are:

- **Reactor** - The *Reactor*, which defines an interface for registering, removing and dispatching *Event_Handler* objects, provides a set of application-independent mechanisms which perform event demultiplexing and dispatching of application-specific event handlers in response to events.
- **Event Handler** - It defines an interface used by the *Reactor* to dispatch callback methods defined by objects that are pre-registered to handle certain events.
- **Concrete Event Handler** - It is responsible for implementing the callback methods that process events in an application-specific manner.

WORKING - A *Reactor* maintains a table of objects that are derived from the *Event_Handler* base class. Its interface provides public methods to register

(*register_handler()*) and remove (*remove_handler()*) these objects from this table at runtime. The *Reactor*'s dispatching mechanism is usually implemented as the main event loop of an event-driven application. Its *dispatch()* method may be implemented using an OS event demultiplexing system call (like *select* or *poll*). This method blocks on the OS event demultiplexing system call until one or more events occur. On occurrence of any such event(s), it returns from this demultiplexing call and dispatches the *handle_event()* method on any *Event_Handler* object(s) that are registered to handle these events. This callback method executes user-defined code and returns control to the *Reactor* when it completes.

Due to this pattern, the application-independent mechanisms become reusable components whose main function is to demultiplex events and dispatch the appropriate callback methods defined by *Event_Handlers*, while the application-specific mechanisms provide a specific type of service. Moreover, it provides applications with a coarse-grained form of concurrency control by serializing the invocation of *Event_Handlers* at the level of event demultiplexing and dispatching within a process or thread, thereby minimizing the need for more complicated synchronization or locking mechanisms within an application.

3.3.2 The Acceptor Pattern

The Acceptor is an object creational pattern that decouples *passive* connection establishment from the service performed, once the connection has been established [Sch95Dec]. This decoupling of services enables the application-specific portion of a service to vary independently of the mechanism used to establish the connection i.e. it enables the tasks performed by network services to evolve independently of the strategies

thread of control.

- *Acceptor* - It implements the strategy for establishing connections with peers. It is parameterized by concrete types that conform to the interfaces of the formal template arguments *SVC_HANDLER* (which performs a service in conjunction with a connected peer) and *PEER_ACCEPTOR* (which is the underlying IPC mechanism used to passively establish the connection). The *Acceptor's* *handle_event()* method implements the strategy for initializing a *Svc_Handler* by passively connecting it with a peer. The *Reactor* performs a callback to this method automatically when a connection arrives for the *Acceptor*.

- *Svc_Handler* - The *Svc_Handler* is a concrete type that defines a generic interface for a service. It inherits from the *Event_Handler* (shown in Fig. 4), which allows it to be dispatched by the *Reactor* when connection events occur. The *Svc_Handler* is parameterized by a *PEER_STREAM* endpoint, which is used by the *Acceptor* to associate it with its peer when a connection is established successfully. Implementors supply concrete types for the parameters to produce an *Instantiated_Acceptor*.

WORKING - Among the different key participants in this pattern, the *Event_Handler* and *Reactor*, which are reused from the *Reactor* pattern, encapsulate the OS event demultiplexing system calls (like *select* or *poll*), and provide the same functionality as described in Section 3.3.1. The *Acceptor* is responsible for creating a service handler (*Svc_Handler*) object, passively connecting this handler to its peer, and activating the handler once its connected. The *open()* method of a *Svc_Handler* object is called by the *Acceptor* after a connection is established (This is a pure virtual function which *has* to be

defined by a subclass, which performs service-specific initializations). Since the *Acceptor* inherits from the *Event_Handler* class, the *Reactor* can automatically callback to the *Acceptor's handle_event()* method when a connection arrives from a peer. The *open()* method of the *Acceptor* is passed the local network address used to listen for connections. It forwards this address to the passive connection acceptance mechanism defined by the *PEER_ACCEPTOR*. This mechanism initializes the listener endpoint, which then advertises its Internet Protocol (IP) address and port number (i.e. Service Access Point) to clients interested in connecting with the *Acceptor*. After the listener endpoint has been initialized, the *open()* method registers itself with the *Reactor*, which then uses the *get_handle()* method to obtain the underlying I/O file descriptor (or *handle*). This *handle* is then used by the *Reactor* to detect and demultiplex incoming connection from clients in order to dispatch the *Acceptor's handle_event()* method, which implements the strategies for creating a new *Svc_Handler*, accepting a connection into it, and activating the service.

3.3.3 The Connector Pattern

The Connector pattern, which is a complement of the Acceptor pattern, enables the tasks performed by network services to evolve independently of the mechanisms that *actively* initialize the services i.e. it decouples active service initialization from the tasks performed once a service is initialized [Sch96Jan]. This pattern permits key characteristics of services (such as the concurrency strategy or the data format) to evolve independently and transparently from the mechanisms used to establish the connections, which in turn helps in increasing code reuse as connection establishment mechanisms change far less often

when compared to service characteristics. Like the Acceptor pattern, it makes use of asynchrony to *actively* establish connections with a large number of peers efficiently.

The structure of the key participants in the Connector pattern is shown in Fig. 6:

Fig. 6 Participants in the Connector Pattern

- **Connector** - It connects and activates a *Svc_Handler*. The *connect()* method of the *Connector* is used to actively initiate a connection between a *Svc_Handler* and its remote peer, while the *handle_output()* method is used to activate the *Svc_Handlers* whose connections were initiated and completed asynchronously.
- **Svc_Handler** - As in the Acceptor pattern, it defines a generic interface for a service. The *Svc_Handler* contains a communication endpoint (*PEER_Stream*) that encapsulates an I/O file descriptor (i.e. *handle*). This endpoint is used to exchange data between the *Svc_Handler* and its connected peer. The *Connector* uses the *open()* method of the *Svc_Handler* to activate its endpoint, when a connection completes successfully.

- *Reactor* - As stated earlier in Section 3.2.1, the *Reactor* handles the completion of connections that were initialized asynchronously. The *Reactor* allows multiple *Svc_Handlers* to have their connections initiated and completed asynchronously by a *Connector* configured in a single thread of control.

WORKING - As in the Acceptor pattern, the *Event_Handler* and *Reactor*, reused from the Reactor pattern, encapsulate the OS event demultiplexing calls (like *select* or *poll*) and provide application-independent (*Reactor*) mechanisms that perform event demultiplexing and dispatching of application-specific (*Svc_Handler*) event handlers in response to events. The *Connector* uses the *open()* method of a *Svc_Handler* to perform service-specific initializations, once a connection is completed. On completion of a connection, the *Reactor* can automatically callback to the *Connector's handle_event()* method (as the *Connector* inherits from the *Event_Handler*). One of the parameters of the *Connector*, the PEER_CONNECTOR provides the transport mechanism used by the *Connector* to actively establish the connection synchronously or asynchronously, while the other, SVC_HANDLER, provides the service that processes data exchanged with its connected peer. The *Svc_Handler* can be activated either in a synchronous or asynchronous mode (by setting the underlying socket mechanisms into a *blocking* or *non-blocking* mode respectively). The *Connector* maintains a map of *Svc_Handlers* whose asynchronous connections are pending completion. Once an asynchronous connection completes successfully, the *Reactor* makes a callback to the *Connector's handle_event()* method, which in turn finds and removes the connected *Svc_Handler* from its internal map, transfers the I/O handle to the *Svc_Handler* and initializes it by calling the

activate_svc_handler() method. Once the initialization is completed, the *Reactor* makes appropriate callbacks to the pre-registered concrete *Svc_Handlers* to implement the application-specific service functionalities.

3.3.4 The Active Object Pattern

This pattern is used for simplifying synchronized access to a shared resource (by methods invoked in *different threads of control*). It decouples method execution from method invocation, so that independently executing threads can access data (modeled as a single object) in an interleaved fashion [Lav95Sept]. This pattern is well suited to a broad class of producer/consumer and reader/writer problems and is commonly used in distributed systems requiring multi-threaded servers. Even though it is possible to build robust single-threaded servers handling concurrent requests for services, it requires complex concurrent programming (to ensure that undue serialization is not caused) and it may not be possible to alleviate performance bottlenecks (like queuing delays caused due to serialization of requests) arising due to adopting a single-threaded approach. The Active Object pattern provides a solution to the above problem by allowing a method to execute in a different thread than the one that invoked the method originally (In contrast, *passive objects* execute in the same thread as the object that called a method on the passive object). Using the Active Object pattern helps in simplifying flow control (since each active object has its own thread of control, and hence can block, waiting for flow control to abate, without affecting other executing objects), concurrent programming (since it allows multiple methods to be executed in parallel) and takes advantage of parallelism (on multi-processor platforms). Moreover, it shields applications from low-level synchronization mechanisms,

rather than having them acquire and release locks explicitly and allows methods, which are invoked asynchronously, to be executed according to synchronization policies, and not on the order of invocation.

As shown in Fig. 7, the key participants in the Active Object pattern are:

- **Client Interface** - This is a *proxy* object that presents a method interface to client applications. The invocation of a method defined by it triggers the construction and queuing of a *Method Object*.
- **Method Objects** - These are objects that are created for any method call that requires synchronized access to a shared resource managed by the *Scheduler*. Each of these objects maintains context information necessary for executing an operation following a method invocation and for returning results of that invocation through the *Client Interface*.

- **Activation Queue** - It is managed by the *Scheduler*. It maintains a priority queue of pending method invocations, which are represented as *Method Objects* created by the *Client Interface*.
- **Scheduler** - It is responsible for managing the *Activation Queue* containing *Method Objects* requiring execution (based on mutual exclusion and condition synchronization constraints).
- **Resource Representation** - It represents the shared resource that is being modeled as an Active Object and typically defines methods that are defined in the *Client Interface*, in addition to methods that are used by the *Scheduler* to compute runtime synchronization conditions for determining the scheduling order.
- **Result Handle** - When a method is invoked on the *Client Interface*, a *Result Handle* is returned to the caller. It allows the method result value to be obtained after the *Scheduler* finishes executing the method.

WORKING - The client application invokes a method defined by the *Client Interface* which triggers the creation of a *Method Object*. Each *Method Object* maintains the argument bindings to the method, in addition to any other bindings required to execute the method and return a result. On creation of *Method Objects*, the *Scheduler* acquires a mutual exclusion lock, consults the *Activation Queue* to determine which *Method Object(s)* meet the synchronization constraints, binds the *Method Object(s)* to the current *Resource Representation* and allows the method to access/update this *Resource Representation* and create a *Result Handle*. This *Result Handle* is used to bind the Result value, if any, to a *future* object which passes return values back to the caller when the method finishes

executing. A future object [Lis88Jun] is a synchronization object that enforces *write-once,read-many* synchronization. Subsequently, any *readers* that rendezvous with the future will evaluate the future object and obtain the *Result* value. The future and the *Method Object(s)* are garbage collected when they are no longer needed.

To sum up, design patterns help in developing communication software components and frameworks that are reusable across OS platforms by facilitating the reuse of abstract architectures that are decoupled from concrete realizations of these architectures. In addition, they provide a structured means of documenting software architectures by capturing the structure and collaboration of participants in a software architecture at a higher level than source code, which helps in capturing the essential architectural interactions while suppressing unnecessary details.

Chapter 4 discusses the design and implementation of the multi-threaded server models, discussed in Chapter 2, based on ACE design patterns.

Chapter 4

Client/Server Design and Implementation

This chapter, using chapters 2 and 3 as background, provides an in-depth discussion of the design and implementation of the Client/Server (C/S) system used for measurement purposes.

4.1 Introduction

This thesis uses a C/S architecture to study the impact of using different multi-threaded

server models to implement concurrent services. In every C/S system, communication channels have to be established between the clients

and servers. As shown in Fig. 8, establishing connections between endpoints involves the following two roles:

- **Passive Role Element** - which initializes an endpoint of communication at a particular address and waits passively for the other endpoint(s) to connect with it.

- **Active Role Element** - which actively initiates a connection to one or more endpoints that are playing the passive role.

Chapter 3 introduced the ACE toolkit and highlighted the usefulness of pattern-based design. Using the background provided by Chapter 3, one can form a picture as to how the design patterns, discussed there, can be used to implement a C/S system based on the ACE toolkit, as shown in Fig. 9:

The server uses the *Acceptor* pattern to establish an endpoint of communication (*passive role*), and advertises its local port and address. The generic clients make use of the *Connector* pattern to actively initiate a connection (*active role*) to the server, on its predefined local port. The server accepts these connections and creates *Svc_Handlers*, responsible for providing the requested service, for each client. Both the client and server make use of the *Reactor* pattern for demultiplexing of I/O events and dispatching the appropriate method(s) in the pre-registered *Svc_Handlers*. As explained in Section 3.3.4,

the thread-pool server makes use of the *Active Object* pattern to implement a pool of threads, each of which services a client request in parallel with the other threads in the pool.

4.2 Client/Server Architecture from an ACE Viewpoint

This section discusses the design and implementation of the client and servers from an ACE viewpoint. The client implementation is generic, with the same implementation being used for all the different servers based on the different threading models. The entire discussion is from a design-pattern perspective because, as explained in Section 3.3, design patterns help to focus on the interactions between key participants in a software architecture, rather than going into all the nitty-gritty details and overwhelming the reader.

4.2.1 The Client

The client, due to its active role in the experimental setup, is based on the Connector/Reactor patterns [Sch96Jan, Sch95]. The *Reactor* encapsulates the operating system *select* mechanism, while the *Connector* encapsulates the various UNIX domain *socket* mechanisms. Due to its generic role in the experiments i.e. it is responsible only for sending service requests to the server, sleeping and sending again, its implementation remains the same for all the different multi-threaded server models. As shown in Section 3.3.1 and 3.3.3, the Reactor and Connector patterns can be broken down into different modules, and since the client is based on these patterns, its design can also be broken down into smaller modules. Fig. 10 shows the object oriented design of the client, modularized into its class components, using the UML notation, developed by Booch and

Rambough [Appendix A].

As shown in Fig. 10, the client components are divided into three layers:

Fig. 10 Class Components for the Client

- **The Reactive Layer** - This layer, along with the *Connection* layer, performs generic, application-independent strategies for handling events and establishing connections. The two participants at this layer, the *Reactor* and *Event Handler*, are reused from the *Reactor* pattern [Sch95]. The *Reactor* defines an interface for registering, removing and dispatching *Event_Handler* objects (such as the *Connector* and *Svc_Handler*). Its interface provides a set of application-independent mechanisms that perform event demultiplexing and dispatching of application-specific handlers in response to events.
- **The Connection Layer** - This layer is responsible for actively connecting a *Svc_Handler* to its peer and activating the handler once its connected. This layer delegates to the *Reactor* pattern in order to establish connections asynchronously without requiring multi-threading. The *Svc_Handler* abstract class provides a generic interface for

processing services, which are customized by applications. The *open()* method of a *Svc_Handler* is called by the *Connector* after a connection is established for performing service-specific initializations. A subclass of the *Svc_Handler* is responsible for determining the service's concurrency strategy (for e.g. a *Svc_Handler* might employ the Reactor pattern to demultiplex events in a single-thread of control, or it might employ the Active Object pattern for providing multi-threaded services). The *Connector* abstract class implements the generic strategy for initializing network services. Since the *Connector* inherits from the *Event_Handler*, the *Reactor* can automatically callback to the *Connector's handle_event()* method when a connection completes. The *Connector* is parameterized by a particular type of PEER_CONNECTOR (which provides the transport mechanism used by the *Connector* to actively establish the connection synchronously or asynchronously) and SVC_HANDLER (which provides the service that processes data exchanged with its connected peer).

- **The Application Layer** - The application layer is responsible for supplying a concrete IPC mechanism and a concrete service handler. The SOCK_Connector class is used to encapsulate the OS *socket* mechanisms for actively initiating a connection to a peer. The *Concrete Svc_Handler* implements the application-specific service activated by a *Concrete Connector*. Each *Concrete Svc_Handler* is instantiated with a specific type of C++ IPC wrapper that exchanges data with its connected peer. The *Concrete Connector* class instantiates the generic *Connector* factory with concrete parameterized type arguments for SVC_HANDLER (which is the *Request_Issuer* here) and PEER_CONNECTOR (which is the SOCK_Connector here).

EXECUTION CYCLE - Fig. 11 depicts the execution cycle of the client, using an interaction diagram. Each client has the following two major phases:

Fig. 11 Interaction Diagram for the Client

- **Connection and Service Initialization Phase** - In this phase, one or more *Svc_Handler(s)* are actively connected with their peers, either synchronously or asynchronously. The main client driver program uses the *Connector::connect()* method to actively initiate a connection to the server, creating and registering a service handler (*Request_Issuer*) with the *Reactor*, thus ensuring that the *Reactor* performs application-

specific callbacks as events concerning it occur. The *open()* method of the *Svc_Handler* performs service-specific initialization.

- **Service Processing Phase** - Once the connection has been established actively and the service has been initialized, the client enters into a service processing phase. It blocks on the *Reactor::handle_event()* loop an event occurs (like an I/O on the port its listening on). As events occur, the *Reactor* automatically calls the appropriate method (i.e. the *svc()* loop) in the previously registered *Request_Issuer* to perform the necessary application-specific services.

4.2.2 The Server

This section provides a brief synopsis of the design and implementation of the different server models. In contrast to the client, the server fulfills the passive role in the experimental setup and uses the *Acceptor* pattern [Sch95Dec] to accomplish it. Event demultiplexing is handled by the *Reactor* pattern [Sch95]. The design of the different threaded-model servers remains essentially the same, although the implementation details vary, as will be shown in the subsequent sections.

Two main design issues in server design relate to *demultiplexing* [SchMar94, Stevens90] of service requests from clients, and *concurrency*. Accordingly, server design can fall into two fundamental categories:

- **Single Threaded Servers** - These handle client service requests sequentially by *iterating* through an infinite loop. While processing the current request, an iterative server typically queues new client requests. An iterative design is most suitable for short-duration services

that exhibit relatively little variation in their execution time (as otherwise queuing delays would reduce the level of concurrent services provided).

- **Concurrent Servers** - These servers, either by using multi-processing or multi-threading techniques, handle multiple requests from clients simultaneously. They help to improve the system throughput when the arrival rate of requests is greater than the rate at which they are processed. They are most useful for *I/O bound* or *variable execution time services*, as each client incurs a relatively smaller queuing delay for getting access to the server.

4.2.2.1 Single Threaded Server

The single threaded server implements the Acceptor/Reactor patterns. As shown in Fig. 12, the class components of the iterative server are similar to the ones in the client, with the *Acceptor* pattern replacing the *Connector* pattern:

Fig. 12 Class Components for the Single-Threaded Server

- **The Reactive Layer** - This layer performs the same services as those described in Section 4.2.1. The *Reactor*, which demultiplexes connections requests received on one or

more communication endpoints to the appropriate *Acceptor*, allows multiple *Acceptors* to listen for connections from peers.

- ***The Connection Layer*** - This layer is responsible for creating a *Svc_Handler*, passively connecting it to its peer, and activating it once it is connected. Due the generic behavior of this layer, the classes in this layer delegate to the concrete IPC mechanism and concrete SVC_HANDLER instantiated by the application layer. The *Svc_Handler* abstract class provides a generic interface for processing services, which are customized by applications. The *open()* method of a *Svc_Handler* is called by the *Acceptor* after a connection is established for performing service-specific initializations. As in the case of the client, a subclass of the *Svc_Handler* is responsible for determining the service's concurrency strategy (for e.g. a *Svc_Handler* might employ the Reactor pattern to demultiplex events in a single-thread of control, or it might employ the Active Object pattern for providing multi-threaded services). The *Acceptor* abstract class implements the generic strategy for passively initializing network services. Since the *Acceptor* inherits from the *Event_Handler*, the *Reactor* can automatically callback to the *Connector's handle_event()* method when a connection completes. The *Acceptor* is parameterized by a particular type of PEER_ACCEPTOR (which provides the transport mechanism used by the *Acceptor* to passively establish the connection) and SVC_HANDLER (which provides the service that processes data exchanged with its connected peer).

• **The Application Layer** - The application layer is responsible for supplying a concrete IPC mechanism and a concrete service handler. The `SOCK_Acceptor` class is used to encapsulate the OS *socket* mechanisms for passively accepting a connection request from a peer. The *Concrete Svc_Handler* implements the application-specific service activated by a *Concrete Acceptor*. Each *Concrete Svc_Handler* is instantiated with a specific type of C++ IPC wrapper that exchanges data with its connected peer. The *Concrete Acceptor* class instantiates the generic *Acceptor* factory with concrete parameterized type arguments for `SVC_HANDLER` (which is the *Request_Issuer* here) and `PEER_ACCEPTOR` (which is the `SOCK_Acceptor` here). The execution cycle of the single threaded server goes through the following two phases, as shown in Fig. 13:

Fig. 13 Interaction Diagram for the Single Threaded Server

- **Endpoint and Service Initialization Phase** - In this phase, a *passive-mode* endpoint is created and bound to a network address such as an IP address and port number. The passive-mode endpoint listens for connection requests from peers. The main server driver program uses the *Acceptor::open()* method to passively establish an endpoint of connection. On receiving connection requests, it creates, activates (by calling the *Svc_Handler's open()* method) and registers a *Svc_Handler* with the *Reactor*, to facilitate callbacks to the *Acceptor's handle_event()* method. The *open()* method of the *Svc_Handler* then performs service-specific initialization.
- **Service Processing Phase** - This phase performs application-specific tasks that process the data exchanged between the *Svc_Handler(s)* and its connected peer(s). The server enters the *Reactor::handle_input()* method that blocks till an I/O event occurs. On occurrence of these pre-registered events, the *Reactor* automatically calls back to the previously registered *Svc_Handler* (*Request_Handler* in this case) to perform the necessary application-specific services.

4.2.2.2 The Thread-per-Request Server

The thread-per-request server, like the single-threaded server, also implements the *Acceptor/Reactor* patterns. However unlike the single-threaded server, which has a single thread of control only, the thread-per-request server handles each and every service request in a separate thread of control. Each callback by the *Reactor* to the appropriate method in the *Svc_Handler* (*Request_Handler* in this case) triggers the spawning of a thread, which processes and returns the reply to the client *in a separate thread of control*. Threads are spawned and managed by the *ACE_Thread_Manager* [Sch95Sept] class of

the ACE thread library, which encapsulates a set of mechanisms to manage a group of threads that collaborate to implement collective actions. The `ACE_Thread_Manager` class also shields applications from many incompatibilities between different types of multi-threading mechanisms (such as Solaris, POSIX, and Win32 threads). As shown in Fig. 14, the key phases in the thread-per-request server are:

- **Endpoint and Service Initialization Phase** - This phase is exactly identical to the corresponding phase in the single-threaded server. The main server driver program uses the `Acceptor::open()` method to passively establish an endpoint of connection. On

Fig. 14 Interaction Diagram for the Thread-per-Request Server

receiving connection requests, it creates, activates (by calling the `Svc_Handler`'s `open()` method) and registers a `Svc_Handler` with the `Reactor`, to facilitate callbacks to the

Acceptor's handle_event() method. The *open()* method of the *Svc_Handler* then performs service-specific initialization.

- **Service Processing Phase** - Once the connection has been established passively and the service has been initialized, the server enters into a *service processing* phase. The *Reactor*, which blocks on the *handle_event()* method, makes callbacks to the appropriate *Svc_Handler(s)* on occurrence of events concerning those *Svc_Handlers*. The *Svc_Handlers*, in turn, invoke the *Thread_Manger::spawn()* method for each and every request. This method, which spawns off a thread to handle that particular request, processes the request and returns the results, before destroying the thread.

4.2.2.3 The Thread-per-Client Server

As explained in Section 2.3, the thread-per-client server is similar to the thread-per-request server, except that a thread is spawned for each client, which processes requests from the client for the whole interval during which the client is connected with the server [Vin96Jun]. One of the major differences between the server described in Section 4.2.2.2 and this is that since each thread exists for the duration of a client's connection to the server, the corresponding *Svc_Handler* is not registered with the *Reactor* as it can function in its own thread of control and does not require callbacks by the *Reactor* to invoke the appropriate method in the *Svc_Handler*. Fig. 15 shows the major phases in the execution cycle of the thread-per-client server:

Fig. 15 Interaction Diagram for the Thread-per-Client Server

- **Endpoint and Service Initialization Phase** - The implementation of this phase is exactly similar to the other models, with one major difference. This phase is responsible for passive-mode endpoint initialization and *Svc_Handler* creation and activation. However, unlike the other models, the *Svc_Handlers* are not registered with the *Reactor*. The reason behind this is that since each *Svc_Handler* has its own thread of control, it can block on its own OS demultiplexing call, without affecting the execution of other *Svc_Handlers*. The main server driver program, after initializing and activating *Svc_Handlers*, invokes the *Thread_Manager::spawn()* method to spawn off threads which are associated with a *Svc_Handler*, and hence its corresponding peer, for the whole interval during which the peer remains connected to the *Svc_Handler*.

- **Service Processing Phase** - In this phase, each and every *Svc_Handler* performs application-specific services in their independent threads of control . When the *Acceptor* terminates, either due to errors or due to a proper shutdown, the *Reactor* calls the *Acceptor::handle_close()* method to release dynamically acquired resources.

4.2.2.4 The Thread-Pool Server

The thread pool server, which has the most complex design of all the models considered, is based on the *Acceptor/Reactor* patterns. In addition, it implements the *Active Object* pattern to provide a functional pool of reusable threads [Sch96Apr]. It pre-spawns a fixed number of threads (a design decision) at start-up which processes all incoming requests. Its design, which is a compromise between the single-threaded server and the thread-per-client server, consists of initializing and activating *Svc_Handlers*, similar to the other models, and processing service requests as they occur. However, a major difference is that although each *Svc_Handler* executes in a separate thread of control , the number of threads are limited, and hence the response seen by the client also consists of the queuing delay incurred by the *Svc_Handlers* to get a thread of control. The major phases in the execution cycle of the thread-pool server are shown in Fig. 16:

- **Endpoint and Service Initialization Phase** - This phase is exactly similar to the initialization phase described in the other server models. The *Acceptor::open()* method, invoked by the main driver program, creates a passive-mode endpoint that is bound to a network address. This endpoint listens for connection requests from peers. When a connection arrives, the *Reactor* makes a callback to the *Acceptor::handle_event()* method, which performs *Svc_Handler* specific initialization, by calling the *open()* method of the

Svc_Handler.

- **Service Processing Phase** - In this phase, as events occur, the *Reactor* makes a callback to the *handle_input()* method of the appropriate pre-registered *Svc_Handler*. This *Svc_Handler*, in turn, enqueues this request onto to a global queue, which is then dequeued by any of the inactive threads in the thread pool for service processing and returning results back to its peer.

Fig. 16 Message Sequence Chart for the Thread-Pool Server

4.3 System Architecture

The previous section provided an overview of the design and implementation issues involved in the different servers based on different threading models. This section further

expands on the previous section, and provides an overall view of the system architecture.

4.3.1 Single-Threaded

C/S Architecture

Fig. 17 shows the system architecture for the single-threaded server. The server starts its execution cycle by creating an *Acceptor* object, which advertises its local address. Each client begins its execution cycle by issuing

Fig. 17 Single-Threaded C/S Architecture

a *connect* request to the server. On receiving such a connection request, the server creates a *Request_Handler* object (which is the application-specific concrete instance of the generic *Svc_Handler*). It accepts the connection into the *Request_Handler* and activates it. It then informs the client about the acceptance of its connection request, at which point the client starts its service cycle. In case the client doesn't receive this acknowledgment within a certain time period (a design decision), it performs an exponential backoff and retries. In case, after a given number of retries, it is not able to connect, it informs the experiment controller about its status and shuts down, thereby aborting the experiments.

4.3.2 Thread-per-Request C/S Architecture

This architecture is very similar to the thread-per-client architecture shown in Fig. 18, the only difference being in the way requests are handled. In this case, each service request causes the server to spawn off a thread, which services the request and gets destroyed at the end of the service for that request.

4.3.3 Thread-per-Client C/S Architecture

In contrast to the thread-per-request architecture, the thread-per-client server spawns off threads which service a client for the entire duration during which its is connected to the server. When a connection request arrives from a client, the server creates, accepts and activates a *Request_Handler*. In addition to service-specific

initialization, the *Request_Handler::open()* method also spawns off a thread, which gets associated with it as long as it has a connected peer. Each *Request_Handler* does its own event-demultiplexing and hence does not register itself with the *Reactor* as it does not need any callbacks to be performed.

4.3.4 Thread-Pool C/S Architecture

As can be seen from Fig. 19, the thread-pool server implements the *Active Object* pattern, in addition to the *Acceptor/Reactor* patterns. The startup of the experiments follows the same pattern as the other cases. However, an additional part in the server initialization is the

spawning of a pre-determined number of threads to form the thread pool. This thread pool is managed by the *Active Object*, which is an instance of the *ACE_Task* class, and contains a global message queue, in addition to the thread pool. When the server receives a connection request from a client, it creates, accepts and activates a *Request_Handler* object. On receiving an acknowledgment from the server, the client issues a service request, which is directed by the *Reactor* (similar to the single-threaded server case), in the *main thread*, to the appropriate *Request_Handler*. The *Request_Handler*, instead of receiving the actual message in the main thread, enqueues a pointer to it in the global message queue. This is then taken off the global queue by any of the inactive threads in the

thread pool, which then goes on to receive the actual request, process it and return the result back to the client. In this way, the *Reactor*, which blocks till a *Request_Handler* finishes executing, is freed up as soon as possible to handle service requests from other clients, and the actual service processing is done in a separate thread of control.

In order to get a feel of the execution cycle of the system, a dynamic diagramming technique is presented in the next section.

4.4 A Use Case Map Representation of the System Execution Cycle

This section provides another way to describe the system execution cycle. It presents a study of the execution cycle of the C/S system based on the single-threaded server model, using a *diagramming technique* known as *use case Maps* [Buhr96, Hubb96]. A use case map consists of superimposing paths of related scenarios of system operation on diagrams of system architecture. *Paths* in use case maps trace cause-effect sequences that traverse a system from points of inputs to points of outputs. The structural components of the system are represented by differently shaped boxes and the behavioral parts are shown using paths superimposed on such boxes. Use case maps are easy to comprehend due to their visual design and are particularly helpful in providing an insight into the structural and behavioral patterns of complex software systems.

As shown by the use case paths in Fig. 20, the server initialization starts with a UNIX command to start up the server, which causes the server to create a Request Acceptor object, which is an instance of the *ACE_Acceptor* class. This stimulus, interpreted by the *Request_Acceptor*., causes it to establish a *passive endpoint* of communication and instantiate a *Request_Handler* object, an instance of the ACE *Svc_Handler* class, which

processes requests received from clients. As a result of this initialization process, the server is ready to service requests from clients. The client initialization proceeds in the same manner, except that this time *Request_Connector* object is instantiated, which *actively* initiates a connection to the server, and creates an instance of a *Request_Issuer* object, which sends (and receives) requests (and replies) to (from) the server.

In addition to the above, some error handling has been shown by *use case paths* (represented by gs4), in which a client makes a maximum of 20 connection retries, shown by a timer like object. On detection of a communication error, it performs retries, based on an exponential backoff, failing which it shuts down. Use case maps for the other models follow the same pattern essentially and so have not been shown. Previous sections, including this, an overview of the design and implementation details of the client and servers of the C/S system used for conducting experiments. The next section provides an overview of the distributed layered server architecture, which was used as a case study.

4.5 Systems with Layered Servers

This section provides a brief overview of the distributed layered server system considered as a case-study.

Many distributed applications decompose naturally into a series of hierarchically co-operating tasks that fit the Client/Server paradigm. For instance, a database system can be designed and implemented using a layered architecture in which applications, *acting as clients* running on one layer, make use of the services of other applications, *acting as servers*, running below them. These servers, in turn, may require the services of other servers further below, i.e.

the same module acting as a server in one layer, may become a client for a server running on another layer. Fig. 22 represents an example of a layered server architecture. Software entities are represented as parallelograms, and hardware devices as circles. Accordingly, we have two types of software servers:

- **Pure Server** - A pure software server executes on its own the requests of its clients, needing only a CPU to run on. It does not perform any I/O nor does it issue any request for services to other servers. *If the server requires more servers than its own CPU, it becomes a mid-level server.*
- **Mid-level Software Server** - In this case, the server requires the services of other server(s) below (besides its own CPU), while servicing the requests of its own clients. The example shown in Fig. 21 contains two software servers, a front-end server (*mid-level*), and a back-end server (*low-level*). The mid-level server receives a request from a client above it, processes part of the request, sends down the remaining portion to the low-level server, waits for the result to come back, and then sends the reply back to the client.

This chapter provided an overview of the design and implementation issues involved in designing the C/S system. The next chapter discusses the experimental setup and analyses the results obtained.

Chapter 5

Experimental Setup and Measurements

Software performance characteristics (response times, throughputs etc.) are crucial in most application domains, specially distributed domains. Furthermore, it is important to understand the dependence of these characteristics on various factors such as design alternatives, system load, resource allocation, distribution etc. In order to analyze the performance of a system, one should be able to fully understand its runtime behavior, and to identify frequent, critical execution paths. *Object-oriented* (OO) patterns offer an invaluable support in those steps, as they focus on the interactions between key participants in a software architecture, rather than overwhelming the developer with details. This chapter describes the experimental setup used for conducting performance measurements and presents a detailed analysis of the results obtained, first from a system level viewpoint, and then from a pattern-guided viewpoint, using chapters 3 and 4 as background.

5.1 Introduction

Design patterns describe recurring solutions to common problems that occur in various application domains. By capturing the static and dynamic nature of these solutions, patterns help to design and implement new systems either *from scratch* or *based on reusable components*. In the latter case, patterns are useful not only to the designers and implementors of the components meant for reuse, but also to those who build software systems using such components. For example, in [Johnson92] it is shown how patterns can

help to explain and document a framework (i.e. a hierarchy of reusable classes), and to teach application programmers how to use the framework for building new systems.

It is a well known fact that the performance characteristics of the system are very important in some application domains (like embedded controllers, telecommunication, distributed databases etc.). It is only natural that some of the patterns applied to the design and implementation of such systems do reflect a strong concern for efficiency, as shown in [Coplien96, Sch95Aug, Gam94, McKenny95].

As will be shown in the subsequent sections, this thesis takes the relationship between patterns and performance a step further, promoting the idea that patterns can be used not only to document and explain reusable components, but also to guide the performance measurements and analysis of such components and of the systems incorporating them.

The overall performance measures depend on all the parts composing a system (application software, operating system, hardware devices, etc.) and on their runtime interaction and behavior. For example, in order to gain insight into the factors contributing to the response time for different types of requests coming to a system, one has to identify the execution paths through the system for each type of request, the software entities executed along the path, and the main causes for waiting and delays. Patterns can help in this process because they provide a more abstract view of the system behavior, concentrating on the main interactions between the system components, and getting away from implementation details. Moreover, patterns can guide developers in choosing what to measure, in instrumenting the code for performance measurements, and in interpreting their results.

5.2 Experimental Setup

This section describes the experimental setup of the system which was used for evaluating its performance. The system consisted of two SPARCstation-2, one SPARCstation-5,

three SPARCstation-10/20 and one SPARCUltra machine, respectively. The network was based on an Ethernet topology, having a maximum bit rate of 10Mbps. The clients were distributed among the different machines, however the server(s) were run alone.

5.2.1 Experiment Controller and Data Logger

In order to setup the server and clients on different machines, and synchronize their data gathering, **DECALS** (Distributed Experiment Control and Logging System) - a set of tools and libraries used to run and control distributed applications over a network and collect measurement and trace related data, was used [Elgillani94]. DECALS reads a set of configuration files (for the system configuration), a list of experiments file- LOE (which specifies the number of different experiments to be conducted) and one or more experiment description files - EXP (which indicate what is to be done for the experiments to be performed).

An *experiment*, in DECALS terminology, is the allocation of application processes to network nodes, the execution until completion of those application processes and the collection of measurement and trace information from those processes. Once the

Fig. 22 Experimental Setup

configuration information has been read, DECALS performs each experiment in turn (Fig.

22). For each experiment, DECALS *executes and synchronizes* all the application processes (server(s) and clients), and simultaneously runs the *collector*, which is used to collect the application's measurements and trace information, using a procedural interface provided by DECALS libraries. In order to terminate the experiments, DECALS provides a time-out period, in addition to the *natural death* (completion of the experiment) of the experiments.

5.2.2 DECALS Architecture

DECALS consists of many processes, as shown in Fig. 22:

- **Experiment Controller** - It is the main process in the DECALS system. It is in charge of running and initializing all other processes. There is exactly one experiment controller in operation for a given set of experiments and it may be executed on any network node.
- **RPTD Processes** - These processes are used by the *experiment controller* to invoke the remaining processes. Initially, there is one *RPTD process* per workstation node, which acts as a daemon server. It then forks a copy of itself to deal with each application process to be run on its node according to the instructions received from the experiment controller. These forked *RPTD processes* remain until the application processes die.
- **Application Processes** - These are the set of processes for which the measurements and analysis are being performed. They communicate with the *Experiment Controller* and the *Collector* via the constituent services provided by the *Collector* library.
- **Collector** - This process is responsible for collecting measurements from the application processes and storing in a file. Exactly one *Collector process* is running on each node in the network being used by the current application.

However, it was found that a number of problems cropped up as the number of users of DECALS increased. So, although the first half of this chapter uses DECALS to set up and control experiments, the remainder of the thesis uses UNIX batch scripts to implement the same approach [explained in Appendix B].

5.3 Measurement Results and System Performance Analysis

A series of tests were conducted for all the different threading models, corresponding to the two architectures. The message size was kept constant at 4 bytes, while the socket size was fixed at the default size of 8K. Although, it is known from previous studies [Courtois96] that message size variation has a strong impact on the total response time, we kept the message size fixed and small for most of our experiments because our goal was to study multi-threading overheads. In [Courtois96, Pozzetti95], it is shown that the CPU cost-per-byte is non-monotonic and causes large fluctuations in the service time, and we wanted to avoid such fluctuations which would hide the effect of the overheads we were interested in. Each client execution cycle consisted of an exponential think time with 100 msec average, and an active time, during which it sent requests and received replies from the server. The server service time was fixed at 10 msec *in order to highlight the impact of various multi-threading overheads, which would be of the order of a few milliseconds*. As will be shown in Fig. 34, larger service times and larger message sizes give the same overall performance behavior, but *mask* the difference between various multi-threading techniques. Although the think time could be used to vary the load on the system (as the arrival rate of service requests would change), the load on the system was varied by varying the number of clients in the system. Each test consisted of 10

replications of 300 sends/recvs each, in order to obtain a reasonable accuracy and to account for performance variation due to transient load on the networks and hosts. The test results obtained were within a confidence interval $\pm 2\%$ of the mean at 95% confidence level [Appendix B].

Fig. 23 summarizes the performance results for the pure server architecture, for a single-threaded case and all the three threading models. The single threaded server shows the lowest cycle times of all the four models compared, which corresponds to the highest throughput achieved in the system. Since the pure-server architecture comprises of a single CPU, the single threaded model avoids the overheads of context switching between threads incurred by the other models (although context-switching between processes is present for all the four models).

In case of the threaded models, the cycle times of all the three models are close to each other. Higher cycle times can be explained by the extra overheads incurred due to thread creation, scheduling and context switching, since all threads are time-multiplexed onto a single CPU. Detailed performance measurements in Section 5.4 will show that, on an average, a typical call to the `ACE_Thread::spawn()` method, (which encapsulates the O/S thread spawn system call), consumes around 3msec of CPU time, which compared to the overall service time, constitutes a highly significant overhead, which explains the fact that

the thread-per-request model exhibits the poorest performance, as can be seen from its throughput. Fig. 24 shows the utilization of the server instances and their CPU for the single-threaded and thread-pool servers respectively. It can be seen that although the maximum CPU utilization for both cases is limited to around 50%, the cumulative effect of the different overheads (shown, and explained, by the elapsed time graphs in Section 5.4) cause the software server instance of the thread-pool server to saturate at around 9 clients. For the single-threaded case, the server takes from its queue only one request at a time, which produces significant queuing delays, thereby reducing its throughput. To sum up, multi-threading servers in a pure-server architecture does not provide dividends in terms of system performance, and for cases where service requests are CPU bound, it is the CPU itself which becomes the system bottleneck i.e. we have a *hardware bottleneck*.

However, the performance results for a mid-level server are entirely different. Fig. 25 highlights the superior performance of the threaded server compared to the single threaded server, for the layered server architecture. The service requested by the clients is assumed to consist of a CPU intensive operation performed by the mid-level server, and an I/O intensive operation performed by the low-level server, although the total service time seen by the client remains the same as in the case of the pure server architecture. The low-level server is based on the thread-per-client model, and remains the same for all the models being compared. The highest throughput is achieved by the thread-per-client model, followed by the thread-pool model. In case of the thread-per-client model, each thread can perform an I/O operation, from the mid-level server to the low-level server, in parallel, and since there are as many threads as clients, the interval of time for which a

client gets blocked reduces, thereby increasing the overall throughput of the system. It also minimizes the overheads of thread creation, unlike the thread-per-request model, since each thread is created for the interval during which a client remains connected to the server.

The single threaded model, as expected, shows the lowest throughput. In this case, the

mid-level server takes from its queue only one request at a time. The server is busy even when it is waiting for a reply from the low-level server, thereby producing a queuing delay which leads to its low throughput. Therefore, the mid-level server becomes the system bottleneck, even if its CPU and the low-level server are not used at their full capacity. Such a bottleneck is known as a *software bottleneck* [Neilson95]. This effect is illustrated in Fig. 26, which shows the utilization of the mid-level server and the low-level server for the single-threaded model.

The utilization of a resource represents the percentage of time the resource is busy. For example, the mid-level server is *busy* with a request from the moment it accepts it to the moment it completes its service, including the time it waits for the low-level server. As it can be seen in the single-threaded mid-level server case, the system bottleneck is the mid-level server (i.e. we have a *software* bottleneck). Both the mid-level server and the lower-level server have equal service times, due to which their curves are overlapped in Fig. 26.

The mid-level server causes the system to saturate at around 11 clients (Fig. 26). Its CPU and the low-level server remain under-utilized, with the maximum utilization at around 16%.

In such situations, the bottleneck can be removed or mitigated, by increasing the number of threads in the mid-level server. Stated in another manner, we *push down* the request queue to the low-level servers (i.e. the low-level software server instance *and* its CPU), thereby increasing their utilization, as demonstrated by the mid-level server utilization for the thread-pool server in Fig. 27. In case of the thread-pool model, the mid-level server still remains the bottleneck. However, the system now saturates at around 19 clients and the low-level server utilization increases to around 33%.

The system-level performance analysis presented in this section, based on end-to-end measurements, does not explain the differences in performance between the different threading models. In order to gain insight into these issues, we will have to conduct more

fine-grained measurements. These measurements are based on design patterns, as explained in the subsequent sections.

5.4 Pattern-guided Measurements and Analysis

Chapter 4 discusses the patterns used to implement the generic client and different servers (based on the different threading models). Out of the patterns discussed, the *Acceptor/Connector* patterns are executed only during the initialization of the system under consideration. The *Acceptor* pattern initializes a passive-mode endpoint of communication that is bound to a network address (such as an IP address and port number) which is used to listen for communication requests from peers. Once the connection has been established passively and the service has been initialized, the *Acceptor's* role ends and the application enters into a *service processing phase*. Similarly, the *Connector* pattern is used to actively initiate a connection to its remote peer, either synchronously or asynchronously. Like the *Acceptor*, its role ends as soon as the application enters into the service processing phase. Thus, *the Acceptor/Connector patterns do not have a significant impact on the system performance as they form part of the startup services*, and so have not been instrumented for measurement purposes.

The *Reactor* pattern is invoked at each and every service request issued by a client, as it is responsible for event demultiplexing and performing call-backs to the appropriate registered service handlers for providing application specific functionality. In addition, the *Active Object* pattern is also involved in every service request as it manages a pool of threads which provide the actual service requested by the clients. Thus, both the *Reactor*

and *Active Object* patterns have a significant impact on system performance due to their repeated invocations.

5.4.1 Frequently executed patterns

This section presents a discussion on the design and implementation of the *Reactor* and *Active Object* patterns and provides an insight to issues critical to system performance.

5.4.1.1 Reactor Pattern

The *Reactor* pattern, as discussed in Section 3.3.1 is an object behavioral pattern used to perform event demultiplexing and event handler dispatching. This pattern is responsible for demultiplexing of events (such as I/O events, timers, UNIX signals) and dispatching the appropriate methods of pre-registered event handler(s) to process these events. An execution sequence chart (which further elaborates on the interaction diagram charts by showing the different components of the execution cycle) is shown in Fig. 28, which

shows the interactions between key participants in the *Reactor* pattern.

As explained in Section 3.3.1, the *Reactor* defines an interface for registering, removing and dispatching *Event Handler* objects and provides a set of application-independent event demultiplexing mechanisms. It triggers *Event Handler* methods in response to events (An *Event Handler* object specifies an interface used by the *Reactor* to dispatch callback methods defined by objects that are pre-registered to handle certain events). When the *Reactor* registers an *Event Handler* subclass object using the *register_handler()* method, it obtains the object's underlying file descriptor (or *handle*) by invoking the *get_handle()* method in the corresponding *Event Handler*. This handle is then used by the *Reactor*, along with other registered *Event Handler* handles, to wait for I/O events to occur. The *Reactor's dispatch()* method blocks on the OS event demultiplexing system call (*select* or *poll*), until one or more events occur. On occurrence of such events, the *Reactor* invokes the *handle_events()* method which uses these handles to perform call backs to these pre-registered *Event Handlers* to perform application-specific functionality, after which control returns back to the *Reactor*.

Thus, the critical execution path (or the most frequent path of execution) in the *Reactor* pattern is the *dispatch()* - *select()* - *handle_events()* path. In case of multi-threaded applications, as mentioned below, this path also includes *remove_handler()*-*register_handler()* methods, as they are used for unblocking the *Reactor* after each call-back, and subsequent re-registering after application-specific execution to ensure call-backs.

Overheads Associated With The Reactor Pattern

From the critical execution path description, it can be seen that the Reactor pattern has the following major overheads associated with it:

- ***Event Handler registration/de-registration*** - *Event Handlers* registered with the *Reactor* are *not pre-empted* while they are executing, which means that they should not perform blocking I/O or long duration operations such as bulk data transfer etc. Also, since the *Reactor* is blocked till a particular *Event Handler* finishes execution, it is imperative that it should be freed as early as possible, in order to service other registered *Event Handlers*. Otherwise, this effect of the *Reactor's serialization* will degrade the performance of multi-threaded servers, as the *Reactor* will become the bottleneck. The main advantage of doing this becomes apparent when message sizes are large. If the *Reactor* were to receive the message itself and pass on the service request to the worker threads, larger message sizes would make the *Reactor* do the work of the actual message reception (and *thus reduce its serializing effect*), the *Reactor* just informs a *Event_Handler* about a service request and returns. The *Event_Handler*, on being informed about this pending message, enqueues a service request which is taken off by one of the worker threads in the thread-pool. This worker thread, *in parallel with other threads*, then does the actual message reception. To free up the *Reactor* as quickly as possible, it is necessary for the *Event Handler* to de-register itself, by invoking the *remove_handler()* method, as soon as it is invoked by the *handle_events()* loop of the *Reactor*, thereby removing itself from the *Reactor's* internal tables. However, at the end of its service, it is necessary for it to re-register itself using the *register_handler()* method, in

order to facilitate call-backs once more. As shown in the previous section, when this overhead of registering/de-registering is most apparent for small service times, and short messages. However, this technique pays off for large message sizes.

- **Locking Overheads** - In order to *ensure atomicity of registration/de-registration* and other modifications of the *Reactor's* internal tables invoked in different threads of control in multi-threaded applications, mutexes have to be obtained and released for each and every call. This introduces locking/unlocking overheads which may prove to be significant if done repeatedly.

- **Event demultiplexing Overheads** - The I/O semantics of the underlying OS significantly affect the performance of applications implementing the *Reactor* pattern. For this case study, the *select* mechanism of the operating system was used for I/O demultiplexing. The *select* mechanism indicates which file descriptors (or handles) out of the registered set of handlers have become ready for I/O. Thus, as the number of registered handles increases, the overhead associated with the *select* call increases correspondingly.

5.4.1.2 Active Object Pattern

This pattern enables a method to execute in a different thread than the one that invoked the method originally (as opposed to passive objects which execute in the same thread of control as the object that called a method on the passive object) i.e. the order of method execution can differ from the order of method invocation [Lav95Sept].

An execution sequence chart for the *Active Object* pattern is shown in Fig. 29, which represents the key interactions between participants in that pattern:

The client application invokes a method defined by the *client interface* (a proxy that provides a method interface to client applications) whose invocation creates a queuing *method object*, which maintains all context information required to execute the method and return a result. On request for a particular service from this *active object*, the *scheduler* acquires a mutex and binds the *method object(s)* to the current *representation* after determining the *method object(s)* which meet the synchronization constraints, by consulting the *activation queue*. This method then accesses/updates this *representation* and creates a result, which is then bound to a *future* object (a synchronization object that enforces *write-once/read-many* synchronization) which passes back return values to the caller when the method finishes executing.

As it can be seen from the execution sequence chart in Fig. 28, the critical execution path in the *active object* pattern involves queuing/dequeuing of service requests i.e. the *insert()-remove()* operations which are executed for every service request to the *active object*.

Overheads Associated With The Active Object Pattern

- **Scheduling and Execution Overheads** - This pattern can potentially increase context switching, data movement and synchronization overheads, *depending on how the scheduler is implemented*, due to scheduling and execution overheads (user-space or kernel-space) of multiple active objects. With the increase in the number of independently executing threads, this overhead might become more significant than the gains obtained by increasing the level of concurrency.
- **Locking/Unlocking Overheads** - Since this pattern consists of multiple threads of

control executing independently, mutual exclusion mechanisms are required to ensure atomicity of operations on shared resources like global data, variables etc. This implies that this pattern has a locking/unlocking overhead associated with each service request, whose impact depends on the manner in which these mutual exclusion mechanisms have been implemented.

5.4.2 Detailed Measurement Results and Performance Analysis

This section describes the experimental setup of the system which was used for conducting measurements on the Reactor and Active Object patterns, and the analysis of obtained measurements. The system parameters were kept the same as discussed in Section 5.3, with the server service time fixed at 10 msecs, and the exponential mean of the client think time fixed at 100 msecs. At this point it should be mentioned that the accuracy of the measurements presented in this section is not very good (around 90% confidence interval) due to the low order of magnitude of the measured overheads and to the fact that the operating system collects this type of measurements by sampling. Even though the curves are *noisy*, the general trend can be easily observed.

- **Measurements on the Reactor pattern** - As discussed in Section 5.4.1.1, the execution path of interest in the *Reactor* pattern is the *dispatch()-select()-handle_events()* operation, including the *register_handler()* and *remove_handler()* operations. Accordingly, these methods were instrumented for both elapsed time (which would capture queuing delays, including those caused by mutexes) as well as the actual CPU execution time for these methods.

Fig. 30 shows the measurement results for the *handle_events()* loop for the *Reactor* pattern. This loop is the main event driver for the *Reactor* pattern and it consists of the *wait_for_multiple_events()* method (which encapsulates the *select()* mechanism), as well as the *dispatch()* method, which performs the actual call-back to the appropriate *Event Handler*. Two separate sets of measurements were conducted, one including the *select()* system call, and one without it. From the graph of CPU usage, it can be seen that the *Reactor* pattern overhead is constant, but the overhead of the *select()* system call is increasing when the number of clients goes up, due to the increase in the number of registered file descriptors (or *handles*) that the *select()* has to check. From the graph of elapsed time, it can be seen that the elapsed time with *select()* is consistently higher than the one without *select()*, which can be attributed to the fact that the application blocks on the *select()* system call, waiting for events to occur. This measurement result is more *dependent on the system wide behavior*, rather than on the pattern itself (as it depends on the frequency of arrivals of service requests).

As shown in Fig. 31, the CPU demand for registering a particular *Event Handler* with the *Reactor*, as well as the demand for de-registering, are constant. For this particular implementation, as mentioned before in Section 5.4.1.1, the main thread (which enqueues a service request) is *alone* responsible for de-registering a particular *Event Handler*, using the *remove_handler()* method, and hence does not incur any queuing overheads for acquiring mutexes (as shown in the graph of elapsed time v/s system load). On the other hand, a thread which is carrying out a particular service for an *Event Handler* is responsible for re-registering it, using the *register_handler()* method, and so has to

complete with other threads for acquiring mutexes (in order to ensure atomicity of updates of the *Reactor's* internal tables). Thus, re-registering an *Event Handler* by a thread involves blocking for mutexes acquired by other threads, the effect of which is felt on the elapsed time (Fig. 31). However, the *actual CPU overhead is insignificant*, compared to the total overall overhead (as summarized in Fig. 39).

From these results, we can further explain the differences between the different threading

models shown in Fig. 23 and Fig. 24. The *Reactor* is responsible for event demultiplexing and performing call-backs to the appropriate *Event Handler* object. The thread-pool server requires such an event demultiplexer to act as a work producer for the threads in the thread pool. When an event occurs, the *Reactor* calls-back to a pre-registered *Event Handler* which enqueues the request received, in order for it to be served by one of the threads in the thread pool. This introduces an additional queue in the server, in addition to the system level queue. This repeated event demultiplexing and callback, enqueueing and dequeuing and the associated acquiring and releasing of mutexes introduces additional overheads in the thread-pool server, compared to the thread-per-client server, due to which its throughput is correspondingly lower.

On the other hand, the thread-per-client server has a thread permanently associated with a client, each of which is assigned an independent socket to listen to. Hence, it does not require an event demultiplexer like the *Reactor* and thus avoids the above mentioned overheads which are incurred by the thread-pool server.

- **Measurements on the Active Object Pattern** - Section 5.4.1.2 shows that the most frequent path of execution, and hence the path causing the greatest impact on system performance, involves enqueueing and dequeuing of service requests, performed by the *insert()/remove()* methods. As in the *Reactor* pattern, these methods were instrumented and the results are shown below. As can be seen from Fig. 31, the CPU demands for both the *insert()* and *remove()* methods are constant and similar to each other. The enqueueing of a service request is done by the main thread, while dequeuing of a service request, and subsequent service is carried out by the threads in the thread pool. Due to this, all threads in the thread pool try to acquire mutexes to execute the *remove()* method, with the result that one thread in the pool has acquired the mutex and is blocking for a service request to be enqueue, while all other threads are blocked waiting to acquire the mutex. Thus, this would lead to large variations in the measured elapsed time for dequeuing requests and so elapsed time measurements were not done.

Fig. 32 provides further insight into the system under study. The thread-pool server requires a global queue in which the *Event Handlers* can enqueue service requests, and the worker threads in the pool can dequeue them for providing service. Each queue/dequeue operation is synchronized by mutexes for ensuring atomic operations, which however, incurs the cost of additional overhead of acquiring/releasing mutexes. These overheads, combined with the ones mentioned above, lead to the better performance of the thread-per-client server compared to the thread-pool server.

In all the cases considered, the thread-per-request model's performance is consistently

poorer than the others. This can be explained by the fact that this model incurs the overhead of creating a thread for each and every request, as well as the overhead of cleaning up after each request has been serviced (around 3 msec of CPU), as well as the overheads associated with threaded servers, like thread-scheduling, context-switching between threads (in addition to context switching between processes), acquiring/releasing mutexes etc., all of which contribute to its poorer performance. Also, if each spawned thread allocated additional system resources like virtual memory etc., the performance of the thread-per-request server would degrade further.

Thus, as can be seen from the previous sections, design patterns can be used to guide the performance measurements and analysis of performance-sensitive object-oriented systems, like the present case study. The next section presents a comparison between the thread-per-client model and the thread-pool model, from the point of view of system resource consumption.

5.5 Comparison between the Thread-per-Client and the Thread-Pool Server models

In general, whenever the issue of multi-threaded servers arise, it is usually a case of making a choice between a thread-pool server and a thread-per-client (also known as a thread-per-session) server. Spawning a thread consumes vital system resources. Fig. 33 shows the increase in the size of a process (in memory and resident, for Solaris 2.5) , as the number of threads spawned increases. This consumption of system resources can be an extremely strong incentive for choosing the thread-pool model implementation over the thread-per-client (even if the latter has a better performance than the former, as shown in the previous section).

If the system load is so high that it causes an unacceptable amount of context switching

between threads in the thread-per-client model, it is highly probable that the thread-pool model's performance will be on par, or even better than the than the former. In the previous implementation for the thread-pool model, the message size itself was insignificant, thereby causing the Reactor overhead to become significant. As shown in Fig. 34, two more experiments were performed, for the layered server architecture: (i) Message size was increased to 1024 bytes (as compared to the previous cases when it was 4 bytes) and (ii) Server service times were increased by twenty times to 200 msecs, keeping all the rest of the parameters constant. In case of increase in service times, it can be seen that the thread-pool model's performance equals that of the thread-per-client model, which can be explained by the fact that the Reactor overhead now becomes insignificant.

In case of increase of message size, the thread-pool model performs as well as the thread-per-client until the system load increases beyond 15 clients. This can be attributed to the fact that the CPU demand for the operating system's read/write calls (carried out by the threads) is significant, compared to the overhead incurred by the Reactor's demultiplexing of I/O events, and since this overhead is incurred by both the models, they are both equally affected. Two factors contribute to the thread-pool model's poor performance, compared to the thread-per-client, beyond 15 clients: (i) the Reactor overhead of registering and de-registering (3 msecs) is significant compared to the overall service time of 10 msecs. and, (ii) The number of threads is fixed at 15, which means that all of them become busy as soon as the system load increases beyond 15 clients, thereby causing internal queuing in the mid-level server, in addition to queuing at the operating system level.

Another deciding factor in choosing between the two models could be the number of system file-descriptors available. These experiments were conducted on a system running Solaris 2.5, which had a maximum of 64 file descriptors. In case of the thread-per-client model, the number of file-descriptors required equals the number of clients that request services. This limitation arises due to the fact that there is no event demultiplexer, like the Reactor in the thread-pool model, so that each thread has to be assigned a unique socket for communication with the Client (and hence a unique file-descriptor). The thread-pool model, on the other hand, can avoid this limitation, by assigning a socket for one or more Clients, thereby freeing file-descriptors for other operations, like communicating with other servers.

To sum up, multi-threading a pure server does not produce performance gains, and in fact may cause it to deteriorate due to the *hardware* system bottleneck. In order to gain benefits from multi-threading, a layered server architecture has to be adopted so that the bottleneck can be pushed down to lower levels by increasing the concurrency (i.e. number of threads) level of the mid-level server. Also, whenever the issue of multi-threading arises, as shown in the previous sections, it is often better to opt for the thread-pool model in comparison to the thread-per-client model, specially from the point of view of system resource consumption. Another important point is that an abstract behavioral model, such as provided by design patterns, is especially useful when a system is built with reusable components, both for understanding what the reused software does, and what is its contribution to system performance. The ACE patterns used to design and implement the various multi-threading alternatives helped (i) to understand how the reusable ACE

components work (ii) to design and implement different multi-threaded servers (iii) to identify execution paths critical for performance and finally (iv) to decide what to measure and how to instrument the code. In this case, the *Reactor* and *Active Object* patterns hold the key to understanding the system behavior, as they control the concurrency level in the server.

The next chapter concentrates solely on the thread-pool server and examines the impact of assigning priorities to clients with different workloads.

Chapter 6

Priority Software Servers

In a client/server distributed system, as shown in the previous chapters, the server is often the *bottleneck* (could be a hardware bottleneck, or a software one). It was shown that it is possible to reduce the effect of the bottlenecking by using multiple copies of the servers, either by having multiple independent server processes, or by using multi-threading. This chapter studies the impact of priority scheduling on multi-threaded servers. Two cases are considered: (i) associate priorities to client classes (as required in certain applications) and (ii) associate priorities to different services provided by the server. The idea behind the second approach is to use the priorities in such a manner that the overall client waiting time for a response is reduced.

This chapter studies the design of the head-of-line (HOL) priority server, applied to the thread-pool server, for both the pure and layered server architectures. HOL software priority server was chosen due to the fact that it avoids the complex programming required in the case of pre-emptive software server (which requires saving the state of each service request being pre-empted and reproducing it on its resumption, thereby leading to unnecessary complexity). It provides measurement results for both these architectures for different priority combinations and presents detailed analysis of the results obtained.

6.1 Priority Server Design

This section provides an overview of the design and implementation of the priority software server, for both the pure and layered server architectures. The priority server is based on the thread-pool server model, (explained in Section 4.2.2.3), and implements the *Acceptor/Reactor* patterns, in addition to the *Active Object* pattern (used to implement the pool of threads). The design and implementation of the priority server is exactly similar in all respects to the thread-pool server, the only exception being in the design of the global message queue used to exchange service requests between the main thread and the worker threads in the thread pool. The next subsection discusses the design of the priority queue.

6.2 Priority Queue Design

A standard FIFO queue is simply a big buffer. The first object pushed into the queue is also the first object removed. This is the kind of data structure a simple operating system might use to handle, say, requests to access a hard drive. However, most operating systems want to include the concept of *priority* attached to each task. The scheduler

would then give the next slot of CPU time to the task (or service request) with highest priority. A simple implementation of such a queue would be to store all entries, according to priority, in a list and then simply remove the head of the list. Although this is a simple implementation, it is relatively inefficient. The code will run in $O(N)$ time, where N is the length of the queue. A faster solution is to implement the priority queue as a *heap*, shown in Fig. 35.

Any list containing N elements is considered a heap if each element i is greater than, or equal to, each of its two children, elements $2i+1$ and $2i+2$. These two nodes can be considered to be children in a binary tree. For example, as shown in Fig. 36, `Element[1]` has a value of 8, which is greater than or equal to each of its two children, `Element[3]` and `Element[4]` respectively. Although it looks like a sorted binary tree, it is not, which is illustrated by `Element[4]`, which is larger than `Element[2]` and thus violates the rules for a sorted binary tree. Some of the advantages of using a heap as a priority queue are outlined below:

- For any given heap, `Element[0]` *must* be the largest element in the array. Thus, as long as the data structure is a heap, the highest priority element in the queue can be accessed in constant time.
- There are no additional storage requirements. The shape of the heap is a simple binary tree, with each node's children always residing in the same spot. Children or parents can be found by using simple index math.
- Adding a node to the heap is an $O(\log N)$ operation.
- Removing a node from the tree is an $O(\log N)$ operation.

- Its performance is superior to other sorted tree implementations because it does not sort the entire tree. Instead, it just ensures that the element at the top of the tree is the largest in the heap. Thus, adding or deleting leaf nodes do not have to shuffle the entire tree.

The priority queue used for the experiments is designed as a parameterized class, accepting the type of mutual exclusion mechanism required, the user-defined *Svc_Handler*, and the comparison operator (type of the priority i.e. integer, double etc.) as its parameters. It is based on the priority queue implementation provided by the HP Standard Template Library (STL), which contains a set of class templates and their implementations for performing various tasks [Nelson95].

6.3 Service Priority

In some applications, it is necessary to associate different priorities to different classes of clients (depending on how important a client is, how much it pays, etc.). The priority of a client issuing a service request gets inherited by the server thread executing the request. The *service priority* experiments were conducted with four classes of clients, *distinguished by their service requirements*, with all classes having the *same* number of clients. The priority of a client remained constant throughout the duration of the experiment. Also, for simplicity reasons, we assumed that each class of clients called a single service with different service requirements for each class. In order to study the impact on performance of service priority combined with different service requests, we ordered the client classes by the length of service required from the server, then conducted two sets of experiments: (i) the class having the *highest* service requirements was given the maximum priority and (ii) the class having the *lowest* service requirements was given

the maximum priority. The experiments were done for both the pure and layered server architectures.

6.4 Experimental Setup

Although the first half of the thesis used DECALS to set up and control the experiments, UNIX batch scripts were used to conduct these experiments [as explained in Appendix B]. Initially, the priority of all the threads in the thread-pool were kept equal, independent of the priority of the client they were serving. However, it was found that the differences between the different cases were *more* accentuated when the worker threads themselves inherited the priority of the client, so all cases presented here are of this kind.

The ratio of the four different service times was fixed first at 1:2:5:10 with the lowest service time of 10 msec. However, different service ratios were also studied: a ratio of 1:2:3:4 and one of 1:5:50:100 are also presented in Section 6.5.1.1. The message size was kept constant at 24 bytes (this increase from the 4 bytes in Chapter 5 was due to the additional information which had to be passed back to the client about its priority, for logging purposes) and the socket size was fixed at the default size of 8K. Each client execution cycle consisted of an exponential think time with 500 msec average, and an active time, during which it sent requests and received replies from the server. The number of worker threads in the thread-pool server were kept constant at 10, due to the following reasons:

- The performance gains achieved by increasing the number of threads in the thread pool is diminishing as the number of threads increases above a certain level. The increase in throughput achieved by changing the number of threads from 10 to 20 was insignificant.

Cases with 2 and 5 threads were also studied, showing a significant reduction in the achieved throughput, compared to the case with 10 threads. Thus keeping the number of threads at 10 represents a good tradeoff in this system (as shown in Section 6.5.1.2).

- For the layered server case, each thread, serving a *Svc_Handler* in the mid-level server, had an open connection to the corresponding *Svc_Handler* in the low-level server. Although each *Svc_Handler* in the mid-level server could have an open connection to the corresponding *Svc_Handler* in the low-level server (instead of having connections on a per-thread basis), the number of connections active at a time would be defined by the number of worker threads in the pool, as they were responsible for communicating with the low-level server. Consequently, the fewer the number of threads, the fewer connections were required, which meant that there were that many more unused file descriptors in the system. Thus, the number of threads was fixed at 10.

Due to the increase in the number of users and corresponding increase in the network traffic, the number of sends/recvs were increased from 300 to 100 for every client. All tests were repeated for 10 times to account for performance variation due to transient load on the networks and hosts. The test results obtained were within a confidence interval $\pm 2\%$ of the mean at 95% confidence level [Appendix B].

6.5 Measurement Results and Analysis for Service Priority

This section presents the measurement results for the different cases, for the pure and layered server architectures.

6.5.1 Measurement Results for the Pure Server Architecture

Fig. 36 summarizes the performance results for the pure server architecture, with the highest priority given to the class of clients having the largest service requirements. Initially, the classes with lower service times (and hence lower priority) outperform the classes with higher priorities (and higher service times), which is a combination of two facts - (i) their service times are the lowest, and hence require the least time to complete and more importantly (ii) as the number of clients in the system is fewer than the number

of threads in the pool, there are some threads free to service the lower priority class of clients. However, at around 10 clients, when all the number of threads in the pool (fixed at 10) become busy serving requests, the effect of priority kicks in, and the throughput of the higher priority class of clients (with higher service times) starts climbing, with the lower priority clients spending more time *waiting* in the priority queue and getting lesser amount of CPU resources, (i.e. they undergo service starvation) due to the fact that all threads in the pool start giving priority to service requests coming from higher priority class of clients.

On the other hand, as shown in Fig. 37, when the class of clients with lowest service times are given maximum priority, the results are quite different. In this case, clients with lower service times have higher priority, and so more such clients get serviced, with the result that the throughput achieved is much higher. This can be attributed to the fact that the average waiting time in the service queue, per class of clients, is reduced considerably. Fig. 38 compares the efficiency of CPU usage in the two cases, showing that at lower loads, the CPU is used more efficiently when the class with the highest service request has higher priority (Note that the clients with longer services contribute the most to CPU utilization per request). The situation changes at higher loads, when higher priority for the shortest clients leads to a higher CPU utilization and a higher throughput overall. The total CPU utilization, in Fig. 38, is a combination of the CPU utilizations by the different classes of clients, as well as the *class independent* CPU resources consumed due to overheads.

In order to get a better idea of the consumption of CPU resources by different components in the server, a breakup of the *class independent* server components, and their corresponding CPU requirements, is outlined. When a client issues a service request to the server, the *Reactor::handle_events()* loop, blocking on a given set of I/O file descriptors, demultiplexes it and calls back to the *handle_input()* method of the appropriate *Svc_Handler*. On receiving this callback, this *Svc_Handler* then de-registers itself from the *Reactor* (as explained in Section 5.4.2), locks the global priority queue, inserts a reference to itself (and hence to the client) at the appropriate place in the heap-based queue, according to its priority and re-constructs the heap. It then returns, thereby freeing up the *Reactor* to service other requests. At some later time, one of the idle worker threads in the pool removes the reference to the *Svc_Handler* from the queue, and then uses this reference to perform the actual message reception, to service the request and to return the results back to the appropriate client. Thus, in addition to the actual service processing done, CPU overheads are incurred in the *Reactor::handle_events()* (responsible for event demultiplexing), the *Svc_Handler::handle_input()* itself (which receives the callback from the *Reactor*), *Reactor::remove_handler()* (which de-registers the *Svc_Handler* from the *Reactor* in order to free it), *queue::insert()* (which inserts the item in the queue), *queue::rebuild_heap()* (which rebuilds the heap structure after item insertion) and *queue::remove()* (which removes the head of the queue), *Svc_Handler::recv()* and *Svc_Handler::send()* (actual message reception and sending), and finally *Reactor::register_handler()* (which re-registers the *Svc_Handler* with the *Reactor* to facilitate callbacks) operations. These methods were instrumented to find the

out the actual CPU overheads in the multi-threaded server, (measurements were carried out using the *ioctl()* system call, as explained in Section 2.4.2), which are summarized in Fig. 39 below:

Component of the Multi-threaded Server	CPU Overhead (Approx.) (msecs)
Reactor::handle_events() with select()	0.6
handle_input() for the user defined <i>Svc_Handler</i>	0.7
Reactor::remove_handler() for de-registering	0.16
Queue::insert() for inserting item into priority queue	0.27
Queue::rebuild_heap() for rebuilding the heap	0.33
Queue::remove() for removing item from priority queue	0.2
Svc_Handler::recv() for receiving message	0.09
Svc_Handler::send() for sending message	0.13
Reactor::register_handler() for re-registering	0.18
Total approximate CPU Overheads	2.66

Fig. 39 CPU Overheads of the Different Components of the Thread-Pool Server

6.5.1.1 Effect of Changing Service Time Ratios

Fig. 40 shows the measurement results obtained by changing the ratio of the service times between classes from 1:2:5:10 to 1:2:3:4 and 1:5:50:100, for the server in which lower service time clients have lower priority, with the lowest service time still fixed at 10 msecs. It can be seen that, although the general pattern of the results is the same as in Fig. 36, the throughputs of the different classes are much closer to each other for the case where the

service time ratios are 1:2:3:4, due to reduction in the impact of service starvation of lower priority clients. However, in the case where service ratios were increased to 1:5:50:100, the impact is dramatic. Initially, the throughput of the lower priority clients (with lower service times) is much higher than the higher priority clients (with higher service times), as there are still some threads in the thread pool free to serve them.

However at around 12 clients, when all the threads in the pool become busy serving the higher priority clients, the lower priority clients suffer from *severe* service starvation and

at around 24 clients, when all the threads in the pool are serving only higher priority requests, their throughput drops to near zero.

Thus, although the general pattern of the results remains the same as those obtained in Fig. 36, service time ratios effect the degree of service starvation felt by the lower priority clients.

6.5.1.2 Impact of Changing the Number of Threads in the Thread-pool

In order to achieve optimum performance with the least possible number of threads, the above experiments were repeated with 2, 5, 10 and 20 threads in the pool. Fig. 41 shows the throughputs of the class with the lowest service time(10 msec) and lowest priority, for 2, 5, 10 and 20 threads. It can be seen that there is a marked improvement in the class throughput, when the number of threads in the pool is 10 or 20, compared to the cases with 2 and 5 threads.

In the case where the thread pool has 2 and 5 threads, the *bottleneck* is the number of

threads itself, as there are too few threads in the thread-pool to handle the number of simultaneous requests coming in, thereby increasing the total queuing delay seen by the lower priority clients. However, with 20 threads, the improvement in the class throughput is not very significant, compared to the case with 10 threads. This can be attributed to the fact that with 20 threads, the overheads due to scheduling, managing and context switching between these threads is quite significant, compared to the case with 10 threads. Thus, the thread-pool with 10 threads is able to achieve a better balance between the overheads incurred and the service requested, and so the number of threads in the thread-pool was chosen to be 10.

6.5.2 Measurement Results for the Layered Server Architecture

Fig. 42 summarizes the results for the layered server architecture, for the case in which clients having lower service times have lower service priorities.

As in Section 5.3, the service requested by the clients is assumed to consist of a CPU intensive operation performed by the mid-level server, and an I/O intensive operation, performed by the low-level server, although the total service time seen by clients of a particular class remains the same as in Section 6.5.1. Also, as in Section 5.3, the low-level server is based on the thread-per-client model and has a one-to-one communication channel open with each of the threads in the thread pool, so there is no queuing at the low-level server. As expected, although the pattern of the results obtained remains the same as for the pure server case, the obtained throughput is much higher. This is because, for the layered server architecture, the actual CPU consumption of the mid-level server is halved (as the total service time requested by the client is divided equally between the

CPU intensive operation and the I/O intensive operation, although the total is the same as in the pure server architecture), consequently there is that much more free CPU capacity.

Fig. 43 presents the measurement results for the layered server architecture, for the case where clients with lower service times have higher priority. In this case also, the service requested by the clients is assumed to consist of a CPU intensive operation performed by the mid-level server, and an I/O intensive operation, performed by the low-level server, although the total service time seen by clients of a particular class remains the same as in

Section 6.5.1. As explained in Section 6.5.1, this design reduces the overall waiting time in the priority queue seen by the clients with lower priority (i.e. higher service times) due to the increase in the number of lower service time clients (i.e. higher priority) being served. In other words, this design utilizes the system in a better way compared to the case where clients with lower service times have lower priority, which can be seen from Fig. 44,

which compares the CPU utilizations of the mid-level server for the two cases.

Initially, the CPU utilization of the mid-level server for the case where clients with lower service times have lower priority is approximately the same as the case where clients with

lower service times have higher priority due to more higher service time (and hence higher priority) clients being served first. However, as the number of clients increases, the total utilization of the former begins to fall, compared to the latter case, because of the increase in the overall waiting time seen by clients of lower priority.

All the cases studied above had clients whose priority was decided according to their service requirements. The next section examines the impact of assigning priorities according to the class of the client, instead of its service requirement.

6.6 Measurement Results and Analysis for Class Priority

This section presents measurement results for the layered server architecture for the case where, unlike the previous cases, each client had a choice of selecting one of two different service times with equal probability, for every service request. The system was assumed to consist of two classes of clients, *with their class deciding their priority*.

Fig. 45 presents a comparison of the mean response times between the two classes of

clients, for the case where priority was decided by the class of the client. The service times requested by the clients were taken to be the extreme of the cases presented in the previous sections, with the ratio between them fixed at 1:20, with the lowest service time fixed at 5 msecs.

It can be seen that the class of clients belonging to the lower priority class are penalized because of their class, even though their service requirements are the same as that for the class of clients having higher priority. Such an approach is usually not practical, unless it is

the application itself which imposes class priority restrictions. To sum up, if workload dependent priorities are to be imposed by the application, assigning higher priorities to shorter jobs makes best use of the available system resources. In real systems, it might happen that exact service requirements might not be known beforehand. However, even rough estimates can be used to decide which service requests would be of shorter duration, and hence should be served first. For example, in a database application, *retrieving* data most often consumes lesser CPU resources compared to *modifying* data, which would lead us to assigning higher priority to a *retrieve* operation as compared to a *modify* operation.

Furthermore, *if system performance is to be optimized, it is better to opt for a layered server architecture, compared to a pure server architecture, as it can take advantage of the processing power of multiple CPUs* (although this is true where network delays are not significant, most modern networks have much higher bandwidth and are as fast, if not faster, than existing CPUs, thereby making the use of such architectures extremely feasible).

Finally, the next chapter present some concluding remarks and directions for future research.

Chapter 7

Conclusions

7.1 Summary

Over the past decade, there has been an increasing trend in building distributed applications, based on a C/S paradigm. Most of these distributed applications have an inherently concurrent nature, as more than one service request from different clients may arrive at the servers at the same time. Such distributed applications can benefit from using a concurrent model of execution to perform their functions, as queuing delays caused by contention between requests for services are reduced when several requests can be processed at the same time. Although concurrency can be provided by using multiprocessor platforms, the complexity of such applications usually makes such implementations infeasible. Instead, most uni-processor applications rely on multi-threading to provide concurrent services. Multi-threaded server applications can handle simultaneous requests for service by allowing each thread to deal with one request at a time, in parallel with other threads. This not only simplifies the code and improves its understanding, but also, for some system architectures, improves the overall performance of the system. Various communication midwares (like DCE, CORBA and ACE implementations) make use of multi-threading to provide concurrency.

This thesis compares several approaches for implementing multi-threaded servers in a distributed environment using the thread-encapsulation library of the ACE object-oriented

reusable component communication toolkit, and identifies cases where multi-threading yields performance dividends.

This thesis also introduces the idea of design-pattern guided measurements to analyze the performance of the object-oriented systems under study and promotes the idea of using patterns to fully understand system runtime behavior, and to identify frequent, critical execution paths due to their focus on the interactions between key participants in a software architecture, rather than overwhelming the developer with details. It shows that patterns are very useful not only for documenting reusable components and teaching application programmers how to use them, but also for characterizing their performance and providing a basis for performance measurements and analysis.

In this thesis, several different multi-threaded server models based on two different architectures were examined from a performance viewpoint. In case of the pure-server architecture, multi-threading a software server which does not require the services of any other server except of its own CPU does not produce any performance gains, as *the system bottleneck is a hardware one* i.e. the CPU itself. Multi-threading in such a case can be detrimental to system performance due to execution overheads, as illustrated by the measurement results. However, in case of the layered server architecture, the mid-level server uses the services of the low-level server, in addition to its own CPU. In such a case, multi-threading the mid-level server, which represents a software bottleneck, will bring substantial performance gains. By increasing the number of concurrent threads in such a server, the bottleneck is *pushed down* to lower levels, allowing for a more efficient use of the servers below and thereby increasing the overall system performance. Also, whenever

the issue of multi-threading arises, as shown by the measurements, it is often better to opt for the thread-pool model, in comparison to the thread-per-client model, specially from the point of view of system resource consumption.

This thesis also examined the impact of assigning priorities to service requests, for the pure and layered server architectures. Two different cases were studied for both the architectures - (i) assigning lower priorities to clients whose jobs had lower service times and (ii) assigning higher priorities to clients whose jobs had lower service times. It was found that if workload dependent priorities are to be imposed by the application, assigning higher priorities to shorter jobs makes a better use of the available system resources. Also, due to the increasing speed and bandwidth of modern networks, network induced delays are almost insignificant, which makes it better to opt for a layered server architecture, compared to a pure server architecture, as it can take advantage of the processing power of the different workstations present in the network.

7.2 Future Work

The work presented in this thesis can be extended in several ways:

- Extend the range of parameters for the different cases studied, and cover more thoroughly the parameter space.
- Use results from the thesis to build performance models for multi-threaded servers and to validate those models.
- Apply the measuring techniques to real servers (such as database servers, name-servers etc.)

Appendix A

Notations for Use Case Maps and UML

A.1 Use Case Map Notations

This section shows a subset of the notations used for representing use case maps [Buhr96].

A.2 UML Notation

This section presents a subset of the notations used in the *unified modelling language* (UML) developed by G. Booch and Rumbaugh, which combines their individual notations (Booch and UMT respectively).

Fig. A.2 UML Notation

Appendix B

Batch Script for Conducting Experiments

B.1 Overview

The first half of this thesis used DECALS [Elgillani94] to set up experiments and log data. However, it was found that as the number of users of DECALS increased, it became increasingly difficult to setup and manage experiments due to some technical difficulties. As a result, instead of DECALS, it was decided to use UNIX batch scripts to accomplish the same purpose. These batch scripts, as explained in the subsequent sections in this appendix, were started on the command line in the appropriate directory and accepted input which was very similar to DECALS input, as shown below:

```
$ <script> <clients> <experiment file> <multiplication factor> <stats filename>
```

where batch script name: name of the main batch script

clients: number of clients for the experiment

experiment file: file containing the names of the machines and processes (similar to .LOE file)

multiplication factor: incremental steps in the number of clients in the system

stats filename: filename to use, when looking for data files in the logging directory

B.2 Batch Scripts

The main batch script is responsible for actually setting the different parameters which are

used by the experiment file as run-time arguments to the different servers and clients. A portion of the initializing section of the script is shown:

```
## Use the "C" shell script with the fast option
#!/bin/csh -f
unset noclobber
## On receiving an interrupt, like a Ctrl-C, go to CLEANUP
onintr CLEANUP
.....
## Set the server service times and client think times
set service = (10 .....)
set think = (100 500 .....)
## Get the current working directory
set CWD = $cwd
.....
## Remove former logging directories and start afresh
(rm -r ./expt/graphs; rm -r ./expt/stats;) >& /dev/null
(mkdir ./expt/graphs; mkdir ./expt/stats;) >& /dev/null
```

Once the initialization process is over, the script enters into a *for* loop which is executed for the different combinations of service and think times. During each iteration through the loop, different parameters like logging directories, think times, service times, number of threads etc. are modified using a UNIX utility called *gawk*, similar to *awk*, as shown below:

```
## for each service time and think time
foreach serviceTime ($service)
    foreach thinkTime ($think)
        set timeForServer = -s$serviceTime
        set timeForClient = -t$thinkTime
.....
## Change service times for the low level server
gawk '{if(match($3, "remote")) {print $1 " "$2 ".....varTime"\" &";"} else print $0}' \
    varTime = $timeForServer ./expt/$argv[2] >> tempFile
## Copy the changed file to the new experiment file
rm ./expt/$argv[2]; mv tempFile ./expt/$argv[2];
.....
## make logging directory
mkdir ./expt/$serviceTime.$thinkTime
```

Although the *gawk* command here seems confusing, it becomes quite easy to understand once we take a look at the experiment file. *Gawk* goes through each line of the file given to it i.e. the experiment file, and checks for a variable called *remote* (which is the name of the low-level server). Once it encounters such a line, it takes the variable called *varTime* (which contains the value of the service time for that particular iteration, for this case) and inserts it into the appropriate place in the command line arguments for the low-level server. Here \$1... represent the different fields in each line of the experiment file. Once this stage is completed, it enters a *while* loop in the inner *for* loop, which executes as many clients as needed for a particular combination of server service time and client think times.

```
## Set counter for clients
@ index = 1
## Start loop
while ($index <= $argv[1])
    .....
    ## Create logging directory for a particular set of clients
    mkdir ./expt/$serviceTime.$thinkTime/client.$index;
    ## Set server logging option, passed to the experiment file
    set serverLogFile = -d$CWD/expt/$serviceTime.$thinkTime/client.$index/
    .....
    ## Execute as many clients as needed, according to the multiplication factor
    @ multiplier = 1
    if ($#argv == 3) then
        while ($multiplier <= $argv[3])
            gawk -f multiplier.awk ./expt/$argv[2];
            rm ./expt/$argv[2]; mv tempMult ./expt/$argv[2];
            @ multiplier ++
        end
    else
        echo " $0 : ...."
    endif
endwhile
```

The *gawk* file *multiplier.awk* is used to go through the experiment files for as many times as needed, in order to make the required number of clients active, according to the multiplication factor. At this stage, the experiment file is ready to be run, and data processing starts.

```
## Start experiment
chmod 777 ./expt/$argv[2];
./expt/$argv[2];
.....
## Get the names of the data files for post processing of data
cd expt/$serviceTime.$thinkTime/client.$index
set filename = 'ls | grep stats'; cd ../.././
## Perform post processing for each data file
foreach name ($filename)
    ## If it is a stats file
    if($name =~ stats*) then
        echo " Post processing file: /expt/$serviceTime.$thinkTime/client.$index/$name..."
        gawk ....
    endif
end
```

Here, *\$argv[2]* represents the second argument fed into the main batch script i.e. the experiment file. First of all, it is made into an executable file, and then launched. At this juncture, it is necessary to stop the main batch script from proceeding further, until the experiment file has finished its experiment. To achieve this, all commands in the experiment file were written to be executed in the background, and since UNIX stops the execution of such a batch file till all its background processes have terminated, the experiment file, and hence the main batch script which invoked it, stops executing at this point till all executing processes terminated. The syntax of the experiment file is shown below:

```
## Experiment file for launching experiment
#!/bin/csh -f
.....
## Get executable file directory
cd ../; set EXE = $cwd;
rsh sunrise "$EXE/remoteserver -aPORT -sSERVICE -Tthreads -dDIR" &;
.....
## Allow the servers sufficient time to setup
```

```

sleep 20;
## Execute as many clients as necessary
# CLNT# rsh ariane "$EXE/client -nSERVERHOST -aPORT -dDIR -tTHINK" &;
# CLNT# rsh ranger "$EXE/client -nSERVERHOST -aPORT -dDIR -tTHINK" &;
.....

```

Here the letters in capitals represents the different command line parameters for the clients and servers which are modified at run-time by the main batch script. In order to execute a certain number of clients, the main batch script removes the comment `# CLNT #` from as many lines as required, which makes those commands active and thus get executed by the experiment file at the time of the experiment. Once all launched processes have terminated, the control returns back to the main batch script, which then launches *matlab*, a mathematical software, for calculating the different statistics required

```

## Launch matlab
matlab << monitor
    clear;
    %% Open the different results file
    outfile1 = fopen .....
    .....
    %% Calculate the different statistics
    load ./expt/$serviceTime.$thinkTime/client.$index/stats.file
    data = stats(:, ...)
    avg = mean(data);
    ....
    %% Record results
    fprintf( outfile1, .....);
    .....
    fclose(outfile1)
monitor
## Increment the index and repeat for a different combination of input parameters
.....

```

The label *monitor* is used by *matlab* to define the portion of the batch script which is to be used as an input file to *matlab* i.e. all commands between these labels are treated as *matlab* commands. The final portion of the batch script relates to the interrupt controller,

whose function is to restore the state of the experiment files, if it catches an interrupt signal i.e. Ctrl-C, from the user.

Interrupt controller

CLEANUP:

Catch all interrupts till cleanup is finished

onintr -

Perform cleanup

mv

.....

Finish script

The label *CLEANUP* is used by the “C” shell script to jump to the portion of the batch script which acts as an interrupt signal handler. Once in that portion, all further interrupts are disabled till the experiment files and directories have been put back into their original state, at which time the interrupts are enabled and the batch script continues.

B.3 Logging Data

In order for the batch scripts to work with all the different server and client models, a logging template was set up, which basically defined the manner in which each server and client, for all the different experiments, would log data. A typical log file from a client would look as shown below:

PID: xxxxxxxx CYCLETIME: xxx.xxxxxxxx REJECTS: xxx/xxxxx ACCURACY:

Each log file would have a common name, followed by the process identifier of the client, or server logging data (For e.g. stats.PID). These log files, as explained in the previous section, were then processed by the main batch script to obtain relevant data, which was then fed into matlab to obtain the final results.

B.4 Overview of data gathering

For both types of experiments, i.e. the ones run using DECALS and the ones run using

UNIX batch scripts, the clients were responsible for actual data gathering. Each client performed 10 batches of 300 sends/recvs (later increased to a 1000 per batch) and logged the corresponding data to their global data structures used for storing these individual data. After completion of the experiment, each client calculated the batch average for each of these batches of measured data, which were then averaged to find out the final average statistic for a particular parameter like cycle time etc. (this was done to ensure that the final average statistic had a normal distribution). The mean and standard deviation of the final statistic, for the required confidence interval, were then calculated.

References

- [Alex77] Christopher Alexander, Sara Ishikawa and Murray Silverstein, with Max Jacobson, Ingrid Fiksdahl-King and Shlomo Angel, "A Pattern Language," *Oxford University Press*, New York, 1977.
- [Bch93] G. Booch, "Object Oriented Analysis and Design with Applications(2nd Edition)," *Benjamin/Cummings*, Redwood City, California, 1993.
- [Buhr96] R.J.A. Bur, R.S. Casselman, "USE CASE Maps for Object Oriented Systems," *Prentice Hall*, 1996.
- [Coplien94] J. Coplien, "A Development Process Generative Pattern Language," in *Proceedings of PLoP*, Monticello, August 1994.
- [Coplien96] J. C. Coplien, "After All We Can't Ignore Efficiency," in *C++ Report*, *SIGS C++ Report Magazine*, April 1996.
- [Courtois96] M. Courtois, "Resource Function Aggregation," *Real-Time and Distributed Systems (RADS) group*, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, September 1996.
- [Dilley95] J. Dilley, "OODCE: A C++ Framework for the OSF Distributed Computing Environment," in *Proceedings of the Winter USENIX Conference*, USENIX Association, January 1995.
- [Elgillani94] T. Elgillani, A. Hubbard, C. Schramm, F. Sheikh, "DECALS (Distributed Experiment Control and Logging System) User's guide and

- Reference Manual, *Real-Time and Distributed Systems (RADS) group*, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada., 1994.
- [Gam94] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Reading, MA: Addison-Wesley, 1994.
- [Gilbert95] J.W. Gilbert, "Private Thread: A Software Pattern for the Implementation of Autonomic Object Behavior," in *Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems Workshop Submission, OOPSLA*, October 1995.
- [Hubb96] A. Hubbard, R.J.A. Buhr, "Use Case Maps for Engineering Real-Time and Distributed Computer Systems: A Case Study of an ACE framework Application," in *Proceedings of the 30th Annual Hawaii International Conference on System Science*, Vol. 5: Advanced Technology Track, pages 218-229, Maui, Hawaii, January 1997.
- [Johnson92] R.L. Johnson, "Documenting Frameworks using Patterns," in *Proceedings of OOPSLA*, 1992.
- [Lav95Sept] R.G. Lavender, D.C. Schmidt, "Active Object: An Object Behavioral Pattern for Concurrent Programming," at *Pattern Languages of Programming Conference*, Monticello, Illinois, U.S.A., September 6-8, 1995.
- [Lazowska84] E.D. Lazowska, J. Zahorjan, G.S. Graham, K.C. Sevcik, "Quantitative

- System Performance: Computer System Analysis Using Queueing Network Models," *Prentice-Hall, Inc.*, Englewood Cliffs, New Jersey, 1984.
- [Li95] Y. Li, R. Campbell, "A Dynamic Priority-based Scheduling Method in Distributed Systems," *Dept. of Computer Science*, University of Illinois at Urbana-Champaign, IL, U.S.A, 1995.
- [Lis88Jun] B. Liskov and L. Shriram, "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems," in *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 260-267, June 1988.
- [Lockhart94] H.W. Lockhart Jr., "OSF DCE: Guide to Developing Distributed Applications," *McGraw Hill*, 1994.
- [McKenny95] P.E. McKenny, "Selecting Locking Designs for Parallel Programs," in *PLoP*, 1995.
- [Neilson95] J.E. Neilson, C.M. Woodside, D.C. Petriu, S. Majumdar, "Software Bottlenecking in Client/Software Systems and Rendezvous Networks," in *IEEE Transactions on Software Engineering*, Vol. 21, No. 9, September 1995.
- [Nelson95] M. Nelson, "C++ Programmer's Guide to the Standard Template Library," *IDG Books Worldwide, Inc.*, 1995.
- [Pow91] M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, "SunOS 5.0 Multithreaded Architecture," 1991, Sun Microsystems Inc.,

A White Paper.

- [POSX] POSIX P1003.4a, "Threads Extension for Portable Operating Systems," IEEE.
- [Pozzetti95] E. Pozzetti, V. Vøland, J. Rolia and G. Serrazi, "Characterizing the Resource Demands of TCP/IP," B. Hertzberger, G. Serrazi Eds., *Lecture Notes in Computer Science*, Springer Verlag, No. 919, pages 79-85, 1995.
- [Sch94Mar] D.C. Schmidt, "A Domain Analysis of Network Daemon Design Dimensions," in the *C++ Report*, Vol. 6, March/April 1994.
- [Sch94Apr1] D.C. Schmidt, "ACE: An Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.
- [Sch94Apr2] D.C. Schmidt, "ASX: An Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 11-15, 1994.
- [Sch95] D.C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J.O. Coplien and D.C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.
- [Sch95Jan] D.C. Schmidt, S. Vinoski, "Object Interconnections: Introduction to

- Distributed Object Computing (Col. 1),” in *SIGS C++ Report Magazine*, January 1995.
- [Sch95Feb] D.C. Schmidt, S. Vinoski, “ Object Interconnections: Modeling Distributed Object Applications (Col. 2),” in *SIGS C++ Report Magazine*, February 1995.
- [Sch95Jun] D.C. Schmidt, T. Harrison, E. Al-Shaer, “ Object-Oriented Components for High-speed Network Programming,” in *Proceedings of the USENIX Conference on Object-Oriented Technologies*, Monterey, CA., June 1995.
- [Sch95Aug] D.C. Schmidt, Paul Stephenson, “Experience Using Design Paterns to Evolve Communication Software Across Diverse OS Platforms,” in *Proceedings of the 9th European Conference on Object-Oriented Programming*, August 7-11, 1995.
- [Sch95Sept] D.C. Schmidt, “An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit,” in the *Technical Rep.*, WUCS-95-31, Washington University, St. Louis, September 1995
- [Sch95Dec] D.C. Schmidt, “Acceptor: A Design Patern for Passively Initializing Network Services,” *C++ Report, SIGS Vol. 7, No. 8*, Nov/Dec 1995.
- [Sch96] D.C. Schmidt, “A Family of Design Patterns for Application-level Gateways,” in *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)* (S.P. Berczuk, ed.), Wiley and Sons, 1996.

- [Sch96Jan] D.C. Schmidt, "Connector: A Design pattern for Actively Initializing Network Services," *C++ Report, SIGS Vol. 8, No. 1*, January 1996.
- [Sch96Feb] D.C. Schmidt, Steve Vinoski, "Object Interconnections: Comparing Alternative Programming Techniques for Multi-threaded Servers (Col. 5)," in *C++ Report, SIGS C++ Report Magazine*, February 1996.
- [Siegel96] J. Siegel, "CORBA: Fundamentals and Programming," *John Wiley and Sons, Inc.* , 1996.
- [Solaris] SunSoft , "SunOS 5.2 Guide to Multi-Thread Programming Manual," *Part No. 801-3176-10, Revision A*, Sun Microsystems Inc., May 1993.
- [Stevens90] W.R. Stevens, "UNIX Network Programming," *Englewood Cliffs, NJ: Prentice Hall*, 1990.
- [Vin96Apr] S. Vinoski, D.C. Schmidt, "Object Interconnections: Comparing Alternative Programming techniques for Multi-threaded CORBA Servers (Col. 6)," in *C++ Report, SIGS C++ Report Magazine*, April 1996.
- [Vin96Jun] S. Vinoski, D.C. Schmidt, "Object Interconnections: Comparing Alternative Programming techniques for Multi-threaded CORBA Servers (Col. 7)," in *C++ Report, SIGS C++ Report Magazine*, June 1996.