1

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

Scheduling in Client-Server Systems

by

William C. Graham, B.Eng.

**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**

April 18, 1996

©1996, William C. Graham

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file *Votre référence*

Our file *Notre référence*

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-13841-0

Canada

Name William C. Graham

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

"Scheduling in Client-Server Systems"

SUBJECT TERM

0544

U·M·I

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Jazz 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0608
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0480
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Science 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language
General 0679
Ancient 0289
Linguistics 0290
Modern 0291
Literature
General 0401
Classical 0294
Comparative 0295
Medieval 0297
Modern 0298
African 0316
American 0591
Asian 0305
Canadian (English) 0352
Canadian (French) 0355
English 0593
Germanic 0311
Latin American 0312
Middle Eastern 0315
Romance 0313
Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion
General 0318
Biblical Studies 0321
Clergy 0319
History of 0320
Philosophy of 0322
Theology 0469

SOCIAL SCIENCES

American Studies 0323
Archaeology
Archaeology 0324
Cultural 0326
Physical 0327
Business Administration
General 0310
Accounting 0272
Banking 0770
Management 0454
Marketing 0338
Canadian Studies 0385
Economics
General 0501
Agricultural 0503
Commerce-Business 0505
Finance 0508
History 0509
Labor 0510
Theory 0511
Folklore 0358
Geography 0366
Gerontology 0351
History
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science
General 0615
International Law and
Relations 0616
Public Administration 0617
Recreation 0814
Social Work 0452
Sociology
General 0626
Criminology and Penology 0627
Demography 0938
Ethnic and Racial Studies 0631
Individual and Family
Studies 0628
Industrial and Labor
Relations 0629
Public and Social Welfare 0630
Social Structure and
Development 0700
Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
General 0473
Agronomy 0285
Animal Culture and
Nutrition 0475
Animal Pathology 0476
Food Science and
Technology 0359
Forestry and Wildlife 0478
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0817
Range Management 0777
Wood Technology 0746
Biology
General 0306
Anatomy 0287
Biostatistics 0308
Botany 0309
Cell 0379
Ecology 0329
Entomology 0353
Genetics 0369
Limnology 0793
Microbiology 0410
Molecular 0307
Neuroscience 0317
Oceanography 0415
Physiology 0415
Radiation 0821
Veterinary Science 0778
Zoology 0472
Biophysics
General 0786
Medical 0760
EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0796

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleocology 0426
Paleontology 0418
Paleozoology 0985
Polynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
General 0566
Audiology 0300
Chemotherapy 0992
Dentistry 0567
Education 0330
Hospital Management 0769
Human Development 0758
Immunology 0982
Medicine and Surgery 0564
Mental Health 0347
Nursing 0569
Nutrition 0570
Obstetrics and Gynecology 0380
Occupational Health and
Safety 0354
Ophthalmology 0381
Pathology 0571
Pharmacology 0419
Pharmacy 0572
Physical Therapy 0382
Public Health 0573
Radiology 0574
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences
Chemistry
General 0485
Agricultural 0749
Analytical 0486
Biochemistry 0487
Inorganic 0488
Nuclear 0738
Organic 0490
Pharmaceutical 0491
Physical 0494
Polymer 0495
Radiation 0754
Mathematics 0405
Physics
General 0605
Acoustics 0986
Astronomy and
Astrophysics 0606
Atmospheric Science 0608
Atomic 0748
Electronics and Electricity 0607
Elementary Particles and
High Energy 0798
Fluid and Plasma 0759
Molecular 0609
Nuclear 0610
Optics 0752
Radiation 0754
Solid State 0611
Statistics 0463
Applied Sciences
Applied Mechanics 0346
Computer Science 0984

Engineering
General 0537
Aerospace 0538
Agricultural 0539
Automotive 0540
Biomedical 0541
Chemical 0542
Civil 0543
Electronics and Electrical 0544
Heat and Thermodynamics 0348
Hydraulic 0545
Industrial 0546
Marine 0547
Materials Science 0794
Mechanical 0548
Metallurgy 0743
Mining 0551
Nuclear 0552
Packaging 0549
Petroleum 0765
Sanitary and Municipal 0554
System Science 0770
Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0381
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0969
Psychobiology 0349
Psychometrics 0622
Social 0481

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

"Scheduling in Client-Server Systems"

**Submitted by William C. Graham, B.Eng.
in partial fulfillment of the requirements for
the degree of Master of Engineering**

A handwritten signature in dark ink, appearing to read "Shakhareh Majumdar", is written over a horizontal line.

Thesis Supervisor

A handwritten signature in dark ink is written over a horizontal line.

Chair, Department of Systems and Computer Engineering

Carleton University

May 13, 1996

Abstract

Client-server systems are now a common way to achieve distributed computing in many organizations. The performance of these systems has become an important factor in design and implementation. This thesis presents scheduling and multithreading as a way to increase performance in client-server systems. Multithreading is effective in reducing software bottlenecks in client-server systems. Scheduling is useful in reducing response times at servers. A simulation model is used to investigate several client-server scheduling policies on multithreaded and non-multithreaded systems. In particular, policies based on client request characteristics are studied. Two important characteristics of client requests have been identified. Path length is the number of blocked servers waiting on the response to a request. Service time is the amount of CPU time required by the request. Policies based on path length and service time were found to improve performance in many client-server systems. The effect of system architecture and workload characteristics on the performance of the policies are investigated. Multiple servers may share the same CPU in some client-server systems. A comparison of the performance of two-level and single-level scheduling for these types of systems is presented. The effect of architecture and workload characteristics on these systems is investigated.

Acknowledgments

I would like to thank my supervisor Dr. Shikharesh Majumdar for his valuable guidance, encouragement and advice while performing this research.

I would also like to thank Dr. C.M. Woodside, Dr. D.C. Petriu and Dr. J. Rolia of the Real-time and Distributed Systems Group of the Department of Systems and Computer Engineering for their support and advice during the research.

Thanks also to Dr. J. Neilson of the School of Computer Science for creating Parasol and his help and advice on its use.

The financial assistance of the Telecommunications Research Institute of Ontario (TRIO) is gratefully acknowledged.

I would like to thank my wife, Marianne, for her continued support and patience during my studies and research.

Table of Contents

1	Introduction	1
1.1	Scheduling and Multithreading in Client-Server Systems	2
1.2	Motivation for the Thesis	3
1.3	Goals of the Thesis	4
1.4	Outline of Thesis	5
2	Client-Server Systems	6
2.1	Overview	6
2.2	Factors Affecting Performance in Client-Server Systems	7
2.2.1	Communication Mechanism	7
2.2.2	Software Bottlenecking	8
2.2.3	Workload Characteristics	9
2.3	Performance Models of Client-Server Systems	10
2.4	Scheduling and Multithreading to Improve Performance	11
2.5	Evaluation Criteria for Scheduling Policies	14
3	Simulation Model	15
3.1	Overview	15
3.2	Open/Closed System Model	16
3.3	Client Model	16
3.4	Server Model	19

3.5	Scheduling Policies	21
3.5.1	Server Scheduling Policies	23
3.5.2	Node Scheduling Algorithms	26
3.6	Inter-arrival and Service Time Distributions	27
3.7	Simulator Design	29
3.7.1	Parasol	29
3.7.2	System Startup and Configuration	29
3.7.3	Client Design	31
3.7.4	Server Design	33
3.7.5	Custom Scheduler	33
4	Performance of Single-Level Scheduling	34
4.1	Overview	34
4.2	Non-Multithreaded Servers with Non-preemptive Scheduling	35
4.3	Multithreaded Servers with Non-preemptive Scheduling	36
4.4	Multithreaded Servers with Preemptive Scheduling	38
4.5	Effects of Relative Loading of Servers	43
4.6	Effects of Service Time Variability	45
4.7	Effects of Inter-arrival Time Variability	50
4.8	Effects of System Architecture	53
4.9	Discussion of Results	60
5	Performance of Systems with Servers Sharing a CPU	63
5.1	Overview	63
5.2	Performance of Scheduling Policies in Systems with Servers Sharing a CPU	64
5.3	Effect of System Architecture	68
5.4	Effect of Allocation of Active and Pure Servers	72
5.5	Effect of Workload Characteristics	80

5.6	Discussion of Results	81
6	Conclusions	85
6.1	Summary	85
6.2	Unique Characteristics of Client-Server Systems	87
6.3	Contributions	88
6.4	Further Research	89
	References	90
A	Other Node Scheduling Policies	94
B	Additional Results of Single-Level Scheduling	96
C	Additional Results of Node Scheduling Performance Experiments	101
D	Simulation Input File Format	104

List of Figures

2.1	Example of a nested client-server system.	13
3.1	Example of closed and open client-server system.	17
3.2	Client processing cycle.	18
3.3	A closed client-server system with two clients.	20
3.4	An open client-server system showing interface agents.	20
3.5	A server processing cycle.	22
3.6	A nested client-server system with two servers.	24
3.7	The bi-phase hyperexponential number generator.	28
3.8	The internals of an open systems client-server system.	32
3.9	The internals of a closed systems client.	32
4.1	An open client-server system with two servers.	35
4.2	Performance of non-multithreaded two server system.	37
4.3	Performance of multithreaded two server system with non-preemptive scheduling.	39
4.4	Impact of level of multithreading on performance.	40
4.5	Performance of multithreaded two server system with preemptive scheduling	42
4.6	Comparison of preemptive and non-preemptive versions of the SRF and LPF/SRF policy.	43
4.7	Performance of two server system with heavily loaded pure server. . .	44

4.8	Performance of two server system with heavily loaded active server. . .	45
4.9	Impact of variability in request service time for the active server. . . .	47
4.10	Impact of variability in request service time for the pure server. . . .	48
4.11	Impact of variability in request service time for the pure server with preemptive scheduling.	49
4.12	Impact of variability in inter-arrival time.	51
4.13	Impact of variability in inter-arrival time on a multithreaded system.	52
4.14	Impact of variability in inter-arrival time on a multithreaded system with preemptive scheduling.	53
4.15	An open client-server system with three servers.	54
4.16	Performance of three server system with no multithreading.	55
4.17	Performance of three server system with multithreading.	56
4.18	An open client-server system with one active and two pure servers. . .	57
4.19	Performance of two pure server system with no multithreading.	58
4.20	Impact on performance on level of multithreading.	59
5.1	Three server client-server architecture.	65
5.2	Comparison of the performance of node scheduling policies for FCFS server scheduling.	65
5.3	Comparison of the performance of node scheduling policies for LPF server scheduling.	66
5.4	Comparison of the performance of node scheduling policies for SRF server scheduling.	67
5.5	Three server client-server system with different node mapping.	68
5.6	Comparison of the performance of node scheduling policies.	69
5.7	Comparison of the performance of node scheduling policies.	70
5.8	Comparison of the performance of node scheduling policies.	71
5.9	Three server client-server architecture with different node mapping. .	72
5.10	Comparison of the performance of node scheduling policies.	73

5.11 Comparison of performance of two different node mappings.	74
5.12 Comparison of the performance of node scheduling policies.	75
5.13 Four server client-server system.	77
5.14 Comparison of the performance of node scheduling policies.	77
5.15 Comparison of the performance of node scheduling policies.	78
5.16 Comparison of the performance of node scheduling policies.	79
5.17 The effect on performance of the variation of the service time distribution	80
5.18 The effect on performance of the variation of the service time distribution	81
5.19 The effect on performance of the variation of the inter-arrival time distribution	82
 B.1 Performance of two server system with heavily loaded active server. .	97
B.2 Performance of two server system with heavily loaded pure server. . .	97
B.3 Performance of two server system with heavily loaded active server. .	98
B.4 Performance of two server system with heavily loaded pure server. . .	98
B.5 Impact of variability in request service time for the active server. . . .	99
B.6 Impact of variability in request service time for the active server. . . .	99
B.7 Performance of three server system of Figure 4.15	100
B.8 Performance of three server system of Figure 4.18	100
 C.1 Comparison of the performance of node scheduling policies.	102
C.2 Comparison of the performance of node scheduling policies.	102
C.3 Comparison of the performance of node scheduling policies.	103
C.4 Comparison of the performance of node scheduling policies.	103

Chapter 1

Introduction

Client-server systems have become a popular paradigm for the development of distributed systems. The adoption of this paradigm has been prompted by the vast increase in the performance, storage capacity and graphics capabilities of desktop workstations and the availability of high-performance, reliable, and inexpensive networking capabilities. The price/performance advantage of client-server systems has caused many corporations to replace traditional mainframe-based systems. *Clients* are processes running on user workstations that handle the details of the interaction with the user, typically through a Graphical User Interface (GUI). Clients generate requests for service from servers. A *server* is a process running on a remote workstation that is connected to the client by a communication network. The server accepts the request and places it in a queue if it is busy. After processing the request the server replies with the result to the client. The server is only responsible for receiving and replying to requests for service. This has a clear performance advantage over the mainframe and terminal-based systems since the mainframe system was encumbered with all aspects of the application, including the provision of a user interface at every terminal connected.

Client-server systems distribute the processing of an application across a number of platforms to improve performance and reliability and to reduce costs. There is

a reliability advantage with client-server systems because of the redundancy that is present on all clients since they run similar client application software and much of the application processing is performed locally on the user's workstation. Reliability is also enhanced with redundant servers that provide backup support when one server fails. There is a cost advantage with client-server systems because many companies have an existing network and workstations and the hardware cost of a server is significantly less than an entire mainframe system.

An example of a Management Information System (MIS) consisting of a database server and several networked Personal Computers (PC's) can be found in [26]. The fact that PC hardware has reached the stage where it is powerful enough to provide enterprise-wide computing is indicative of the growth of workstation price/performance ratio. A typical PC system is two orders of magnitude cheaper than a mainframe system and doesn't require extensive maintenance and support hardware. Other examples of client-servers systems include on-line transaction processing systems, printer and file servers on a company network, database servers for business applications, compute servers for intense numerical computation and name servers for domain name resolution. Client-server systems are also used to provide a middle layer between the user and an older mainframe-based application, known as a three-tiered system [14]. This architecture provides a modern GUI interface to the users and relieves the mainframe of much of the processing required to handle user interaction. This is used in systems where porting the application to a pure client-server system would be prohibitive.

1.1 Scheduling and Multithreading in Client-Server Systems

Effective resource management is required in any system in order to achieve high system performance. Examples of resources to be managed are memory, CPU time,

processes, and I/O devices. Resource management in multiprogrammed operating systems running on conventional mainframe computers is well understood [25]. These techniques may or may not, however, be applicable to client-server systems. Scheduling, that is of particular interest in this thesis, is concerned with the management of CPU processing resources. The specific meaning of scheduling as used throughout this thesis is the intelligent scheduling of client requests and server threads according to some pre-determined scheduling policy. Currently, client-server systems rely on the scheduler built into host operating systems on which they run. A common example would be UNIX. Relying on such general purpose strategies may negatively impact system performance and alternative policies need investigation. Client-server systems have unique performance concerns that are not present in traditional uniprocessor systems.

Threads or Light Weight Processes (LWP) are becoming a common feature in many operating systems, including versions of UNIX [28]. Threads have a single flow of control and share the same address space as the parent process. A server process may contain many threads. Threads are called light weight because the context for each thread is small and the system overhead for context switching of threads within the same process is much less than among separate processes. Threads are added to server or application software to prevent blocking on asynchronous events. The addition of threads to an application is known as multithreading. In the OSF DCE threads are an integral part of the RPC interface specification [22]. Multithreading is an important consideration in scheduling in client-server systems because the scheduler can allocate client requests to threads and then schedule threads accordingly.

1.2 Motivation for the Thesis

The growth of client-server systems has created the situation where performance concerns are becoming important in system design. One way to improve performance

is the use of multithreading and scheduling. Multithreading is becoming a common part of many client-server systems although the nature of their performance may not be understood. This thesis investigates the effect of multithreading on system performance and identifies places where multithreading is most useful. No previous investigation of client-server scheduling has been found in the literature. Much of the research in scheduling applies to conventional mainframe systems or to multiprocessor systems running scientific applications. The results of that work are not directly applicable to client-server systems although many of the existing heuristics used are investigated in the context of client-server systems.

1.3 Goals of the Thesis

The primary goal of this thesis is to provide a basis for improving client-server system performance through the use of multithreading and intelligent scheduling. Several scheduling policies are proposed. Using a simulation model of different architectures and workloads, the relative performance improvement possible with each policy is discussed. Multiple server processes may share a CPU in certain systems. A two-level scheduling mechanism is introduced for the case when software servers share a CPU resource. The single-level scheduler uses client request characteristics to make scheduling decisions. The two-level scheduler schedules server processes sharing the same node. This thesis is motivated by the following questions regarding multithreading and client-server scheduling:

- What is the effect of multithreading on client-server system performance?
- What is the effect of scheduling on client-server system performance?
- How do scheduling policies perform on multithreaded servers.
- Is the use of preemptive policies more beneficial than non-preemptive ones?

- What are the effects of system architecture and workload characteristics on the performance of scheduling policies?
- What characteristics of client requests are most useful in scheduling for high system performance?
- If multiple servers share CPU resources, is it worthwhile to use a two-level scheduler?

The secondary goal of this thesis is to provide a rationale for selecting a scheduling policy in light of system architecture and workload characteristics of the system. It is hoped that given the workload and architecture of a system, a designer can select a scheduling policy that improves performance based on the understanding gained in this thesis. If these characteristics are unknown then the selection of a scheduling policy may be on the basis that it is insensitive to workload or architecture. Insight into the behaviour of client-server systems gained through this thesis will be beneficial in systems design.

1.4 Outline of Thesis

This thesis is divided into six chapters. An introduction to performance issues, performance models and scheduling in client-server is given in Chapter 2. Chapter 3 discusses the simulation model and simulator design. The results of the investigation into the performance of single-level scheduling are given in Chapter 4. The results of investigation into the performance of two-level scheduling are given in Chapter 5. Chapter 6 provides the conclusions, the contributions made by this thesis, and recommendations for future research.

Chapter 2

Client-Server Systems

2.1 Overview

The client-server paradigm has been discussed for some time as a practical way to implement distributed processing. Early work by White in [30] discusses network-based resource sharing. In [21] the implementation of the Remote Procedure Call (RPC) is presented. The RPC has become the defacto standard for client-server communication although client-server systems are not specifically mentioned in [21]. The term client-server computing seems to appear in the literature in the late 1980's even though the concept appeared much earlier in [21] and [30]. The practice of resource sharing (i.e. printers and disk systems) was commonplace before the client-server paradigm became popular. In [26] some of the history of client-server systems is discussed. The growth of the PC has fueled the push for client-server systems although many corporations still rely on mainframe systems for so-called mission critical applications. Corporations are now spending the time and effort to *down-size* to client-server systems. The term down-sizing refers to the porting of older mainframe applications running on expensive and slow mainframe computers to a new client-server based system. There is much discussion as to how to approach this down-sizing effort and what technology to adopt in order to achieve it, for example,

see [13].

2.2 Factors Affecting Performance in Client-Server Systems

2.2.1 Communication Mechanism

Clients generate requests and communicate the request to the server, the client then waits for the response from the server. This sequence is called a send-wait-reply sequence or rendezvous. A typical communication mechanism used by operating systems to implement a network-wide rendezvous is the Remote Procedure Call (RPC). The RPC is an extension of the well-known procedure call mechanism that allows a client to call a procedure on a remote server over a network. A description of the implementation of the RPC can found in [21]. The use of RPC for communication in client-server distributed systems has become standardized as part of the Open Software Foundations (OSF) Distributed Computing Environment (DCE) [22]. The simulator used in this thesis is based on the DCE RPC model because of its importance as a standard.

The performance of the RPC was investigated when first proposed in [21]. This first analysis measured the response time for RPCs with varying parameter data sizes. The performance of the RPC in the DCE environment has also been studied. Detailed measurements of DCE RPC performance can be found in [10]. This work used detailed measurements of the RPC runtime system on the order of microseconds. Low traffic measurements of RPC turn around time were done for varying parameters sizes. Initial RPC calls were found to take longer than subsequent ones. Models of DCE RPC performance are presented in [24] and [11] based on the measurements of real DCE systems. In [24] a SRVN model was built to model RPC performance. This model was then used to evaluate the relative effects of the level of multithreading and

various RPC overheads on performance. These studies are based on as-built DCE systems and do not specifically investigate scheduling.

2.2.2 Software Bottlenecking

Software bottlenecking is a unique problem in client-server systems. One of the first people to report this phenomena was Fontenot [4]. He observed that software servers were fully occupied while the underlying hardware was poorly utilized. Traditionally performance problems were thought to be easily solved by running the software on a faster platform. This new phenomena indicated that this approach is not sufficient in improving performance for some systems. This phenomena is the result of the rendezvous communication mechanism used in client-server systems. During a rendezvous an active servers is blocked until the reply from another server is received. While blocked the server is unable to perform any other processing until the reply is received. As load increases an active server can become a software bottleneck because it is spending all of its time processing requests and blocking on replies from other servers. The server process is considered to have a *software utilization* of 100% when there is no time between executing requests and waiting for replies from other servers. An active server may reach this fully utilized state long before the CPU on which the server runs is heavily utilized. Traditional analytical models failed to predict software bottlenecks which means new models are required such as the Hyperbolic Model proposed in [4].

Multithreading of software servers was investigated in [20] as a possible solution for software bottlenecks. A simulation model of a client-server system was used to measure improvements in response time and throughput when non-multithreaded servers were replaced with multithreaded servers. Improvements in response time and throughput were observed when active servers were multithreaded. Threads allow the server process to accept and process requests while blocking on a reply from another server. Consequently the throughput increases because more requests can be handle

at once and the utilization of the CPU is improved.

2.2.3 Workload Characteristics

Most models of computer systems, particularly queuing models assume that inter-arrival and service time distributions are exponential. This is partly because exponential distributions are easier to analyze when solving queuing models. Exponential inter-arrival times are common in computer systems although recent evidence suggests that inter-arrival times may be hyperexponential for certain classes of client-server systems [3] [27]. A hyperexponential distribution models bursts of client activity that is typical in some client-server systems. Large variability in client activity is evident in network traffic of local and wide area networks on which distributed applications run (for example, see [7], [9], and [31].) Another network measurement study reported that inter-arrival times for a distributed DECWindows application had a Gamma distribution [18]. However the parameters for the gamma distribution were not reported. Hyperexponential service time can be exhibited by servers that provide a set of services with varied service times for each service. Large differences in service time can be expected when different services are requested from the server.

These measurement studies indicate that inter-arrival time is rarely exponential and that hyperexponential models are more realistic. These measurements apply to rather simple client-server systems such as file servers. Very little work been done on retrieving inter-arrival and service times for other types of client-server systems such as database servers. Since little is known about the workload characteristics of more complicated client-server systems, scheduling policies are evaluated for their performance under different workloads.

2.3 Performance Models of Client-Server Systems

To date, much of the work on performance analysis of client-server systems has been concerned with performance models of “in-place” client-server systems. The use of queuing theory for performance modeling of client-server systems with an emphasis on capacity planning can be found in [17]. In [8] Stochastic Reward Nets are used to model client-server systems, in particular, the interaction of clients and servers on a Ethernet network. This and other Petri net based solutions are difficult to apply to complex systems because the solution state space becomes unbounded as system complexity increases. One solution to this problem is the Stochastic Rendezvous Network (SRVN) model which was proposed in [32] as a useful model for client-server distributed systems, including complex systems with many clients and servers. These models although very useful are difficult to use in an investigation on scheduling for several reasons :

- Analytical models assume FCFS scheduling and require modification for each scheduling policy used by the system.
- Some scheduling policies are probably impossible to model analytically. For example, policies that rely on request characteristics.
- Real client-server systems are more complex than those used in analytical modeling research. Applying analytical methods to complex systems with multithreading would be difficult.
- Analytical models generally make assumptions about system behaviour. For example, exponential inter-arrival and service time distributions are often assumed. In real systems, workload can be highly variable.

To overcome these difficulties, a simulation model was built to investigate the performance of multithreading and client-server scheduling. Simulation models have several advantages over analytical models :

- Simulation models can be developed for any reasonable scheduling policy that can be implemented as a software algorithm.
- Simulation models can be developed for client-server systems of any reasonable complexity.
- The modeling of multithreading is easy in simulation models.
- Various random number distributions can be generated in a simulation model. In particular, uniform, exponential, hyperexponential and hypoexponential distributions are useful in investigations of scheduling.

Simulation models do have drawbacks; the most important being speed of execution. Analytical models, when implemented in software, are extremely fast and can calculate solutions several orders of magnitude faster than simulation models. Experiments with simulators require thousands if not millions of iterations in order to achieve acceptable confidence intervals. Each point on a graph produced by a simulator may take several minutes of CPU time to calculate. This disadvantage was considered acceptable because of the availability of powerful workstations and a comprehensive simulation environment provided by Parasol, which is described further in Chapter 3.

2.4 Scheduling and Multithreading to Improve Performance

In conventional multiprogrammed uniprocessor systems scheduling is a resource management technique for deciding which job to run and when. Scheduling is an important part of mainframe operating systems design because many users are sharing one very valuable resource, i.e. the CPU. The proliferation of desktop computers has removed some of the urgency of scheduler design from operating system design. For

example, MS-DOS does not contain or need any scheduler. Other more sophisticated multitasking operating systems that run on desktop workstations tend to rely on a Round Robin (RR) scheduler which gives equal quantum of CPU time to all processes. This is a good general purpose policy for a multiprogrammed workstation since it provides good interactive response times to short commands (i.e. user shell processes have prompt response times) and background jobs share the leftover CPU time equitably. Other systems may schedule processes in a First Come First Served (FCFS) fashion. This policy is simple but can lead to unacceptable response times for short jobs when a large job is running. Scheduling in conventional computer systems is well studied and optimal policies such as Shortest Job First (SJF) and Shortest Remaining Processing Time (SRPT) are well known. Whether or not these principles are applicable to client-server systems needs investigation. Priority inversion is a well known phenomenon in conventional computer systems and maybe present in client-server systems using priority scheduling. This thesis is concerned with scheduling on individual servers and therefore priority inversion is not considered in this investigation.

In current client-server systems the scheduling is done by the operating system, such as UNIX, on which the client-server system runs. In DCE, for example, the threads in a server process are scheduled in a FCFS fashion by the DCE runtime [14]. The server process is then scheduled by the underlying operating system, most probably in the same fashion as UNIX. These general purpose strategies may or may not always be applicable to client-server systems. Client-server systems differ in several ways from conventional multiprogrammed computer systems. For this reason, the following characteristics were identified as unique to client-server systems :

Nested Architecture : Active servers may make calls to other servers which creates a nested chain of requests from client to final server. This nested chain has a path length which is defined as the number of active servers blocked on the response to the request. This path length which is a characteristic of the client

Figure 2.1: Example of a nested client-server system.

request can be used by the scheduler to improve performance. An example of server nesting is illustrated in Figure 2.1. In such a system the client, C_1 makes a request to the active server, S_1 which then makes a request to a pure server S_2 . The client also makes requests to directly to the pure server S_2 . The number of servers blocked on the response to a service requests is called the *path length*.

Communication Mechanism : Clients and active servers communicate with servers using the rendezvous mechanism. This is typically implemented by RPC. The rendezvous implies that the caller is blocked until the response is received, unable to perform any useful work until the response is received. This method of communicating can create software bottlenecks. The use of multithreading in these servers can remove software bottlenecks and increase throughput.

Fixed Set of Services : Traditionally, in general purpose systems service time is not know *a priori* and is hard to predict. Servers typically provide a set of procedures that clients can call by RPC and the number of procedures is finite and typically small. It may be possible to characterize the execution time

of each procedure call and use this estimate in scheduling decisions. For this reason, scheduling policies based on request service time are considered feasible.

These characteristics of client-server systems provide the basis for developing scheduling heuristics that exploit these characteristics to the benefit of better performance. The characteristics of client requests to implement these scheduling heuristics are described below :

Service Time : The service time is the amount of CPU time required by the request. In the thesis it is assumed that service time is known for each request *a priori* in light of the possibility of having execution time characteristics of services.

Path Length : The number of active servers blocked on a response to a request is defined as the path length of the request. Servers can use the path length to give priority to requests that have servers blocked on their response. In the example given in Figure 2.1 the path length for requests arriving at server S_2 from server S_1 would be one. The path length for requests arriving at server S_2 directly from the client would be zero.

2.5 Evaluation Criteria for Scheduling Policies

The mean response time at the client is used as the main performance metric in rating scheduling policies. The policy that delivers the lowest mean response time for a given workload is considered the best policy. The arrival rate at which the system saturates is also an important factor. The saturation point can be extended for some client-server systems by multithreading and scheduling.

Chapter 3

Simulation Model

3.1 Overview

A simulation tool provides a test bed for investigating the effects of scheduling and multithreading on client-server systems. The simulator is highly configurable and allows for any combination of clients, active servers, and pure servers. The simulation tool supports open and closed system modeling. Client request inter-arrival and service time distributions are configurable as deterministic, exponential and hyperexponential. The mean service time at each server is configurable. The mean number of calls made to other servers for each client or active server can be specified.

The simulation model simulates the communication between client and servers in a similar fashion to the DCE RPC as described in [22] [23]. The emphasis of the simulation model is the performance of scheduling policies therefore a detailed RPC model was not deemed necessary. Some features such as security and non-idempotent RPCs were not considered. The DCE RPC was chosen because it is an industry standard and is becoming used widely by client-server systems developers. The following sections describe the simulator model and design.

3.2 Open/Closed System Model

The simulation model supports open and closed system models. In a closed system the number of clients is fixed. An example of a closed system can be seen in Figure 3.1 (a). This system consists of four clients, $C_1 \dots C_4$, each making requests to a single server, S_1 . Closed systems are not investigated in this thesis. The simulation model contains facilities to support closed system modeling which maybe useful in future research. This thesis concentrates on the performance of open client-server systems. In an open system, client requests arrive in a Poisson stream with a given rate, λ . To support this in software, the simulator uses *interface agents* which are created dynamically at a rate equivalent to the arrival rate of client requests. Interface agents are described in detail in the next section. An example of an open system is shown in Figure 3.1(b). This diagrams shows a set of interface agents (shaded circles), representing pending client requests, communicating with a server, S_1 . Any number of pending requests (within the limits of simulator) can accumulate depending on the load on the server. As each request is completed the interface agent waiting for the response is terminated and removed. Open systems are more applicable to modeling large systems with a variable number of clients, as such only open system models are used in the experiments used in this thesis.

3.3 Client Model

Clients perform activities in cycles. An example of a client cycle is shown in Figure 3.2. The first part of the client cycle is called the *think time* where the client process is performing other processing possibly not associated with the client-server application. In the second part of the cycle, the client performs local processing associated with the request generation. In the third part of the cycle, the client generates requests and sends them to one or more servers. After generating a request the client waits for a response. The time between generating the request and receiving the reply is

Figure 3.1: (a) A closed client-server system with four clients $C_1 \dots C_4$. (b) An open client-server system with interface agents dynamically created at same rate as the client request arrival rate.

called the *response time*. Once the client receives the reply more local processing is performed and the cycle begins again. A client may make several consecutive calls to the server as specified by the configuration information. The number of server calls is generated by a geometric distribution. The local processing occurs before and after each call to a server. In the example shown in Figure 3.2 the client is making two consecutive calls to the server. The local processing time used during the request is split into three intervals, t_1 , t_2 and t_3 . If three consecutive calls are made then there would be four intervals and so on. The total length of local processing time, T , is modeled using an exponential distribution and the time intervals t_1 , t_2 and t_3 are equal portions of T (i.e. $t_1 = T/3$, $t_2 = T/3$ and $t_3 = T/3$.) This local processing model is used because real clients may generate consecutive requests with short time intervals between each call. This model creates bursts of client activity separated by think times of no activity. This simulates the behaviour of distributed systems observed in [3], [27].

Figure 3.2: A client-server interaction showing the client processing cycle. Two consecutive calls are made by the client to the server.

In a closed system model, the client is modeled as a single process running on a single CPU executing the client cycle described above. There is a fixed number of these client processes in the system. Each client process waits until each request is completed before generating another request. In an open system a different client model is used. To simulate an open system, interface agents are used to generate client requests and wait for the response. The arrival rate of requests to the system is equivalent to the creation rate of interface agents. If the inter-arrival time of requests is exponential then this creation of interface agents is a Poisson process. An interface agent is used to model a runtime element of the RPC library on the client workstation which handles the server interaction on the behalf of the client, see [1] for a detailed description of this terminology. In essence, the interface agent represents a client who has arrived at the system, requests some service and disappears from the system. Each client request is handled by a separate interface agent which is responsible for calling the server, waiting for the result and recording the response time to complete the request. The client cycle for each interface agent is identical to the cycle followed by the closed client described above. Interface agents are created when a client

request arrives and terminates when the request is complete, for example, see Figure 3.4. In an open system in which the servers are highly utilized it is possible for too many interface agents to accumulate thus causing the simulator to exceed its memory bounds. This is due to the dynamic creation of interface agents and is an acceptable limitation of the simulator. In general, however, servers in the experiments in this thesis are not more than 90% utilized.

3.4 Server Model

The server model is fashioned after the design of a DCE server [22] [23]. A server process contains a listener thread, request queue and executor threads. The diagram of Figure 3.3 shows a closed system with two clients and one server. The internals of the server are visible. All requests are received by the listener thread which places the requests on to the request queue. The requests are placed on the queue in the order they arrive (First In First Out – FIFO order). The diagram in Figure 3.4 shows an open system simulation model with open system clients and one server. The server design remains the same for open and closed system simulations.

Requests are removed from the queue by the executor threads and are then processed. The order in which requests are removed from the queue depends on the server scheduling policy for the server. If, for example, a First Come First Served (FCFS) policy is used then requests are removed in the order they arrived. Other policies use other criteria and are explained later in the chapter. Each request has a service time which is the amount of CPU time the executor thread must consume in order to satisfy the request. Executors may, while processing a request, make calls to other servers in the system. The number of calls to make is a geometric random variable with a specified mean. The execution cycle is similar to that of a client, described in the previous section. An example of this cycle is shown in Figure 3.5, this example shows a client making a request to an active server which then makes

Figure 3.3: A closed client-server system with 2 clients C_1 and C_2 and a single server.

Figure 3.4: An open client-server system. Interface agents are created dynamically by the open client manager. Interface agents exist until server has completed requests then terminate themselves. Server interacts with interface agents directly.

two consecutive calls to a pure server. The service time for the request at the active server is divided equally among the time intervals t_1 , t_2 and t_3 shown in the diagram. That is, if the total service time for the request is T then;

$$t_1 = t_2 = t_3 = T/3 \quad (3.1)$$

If a request does not require calling another server (i.e. requests arriving at a pure server) then the executor consumes all the service time as one contiguous execution slice. Once the request is complete the executor responds to the client and reads the queue for another request. If the queue is empty the executor thread blocks. The executor is unblocked when a request arrives at the queue.

Each server node contains a scheduler process which performs the scheduling of threads for all processes running on the node. This is necessary when multiple servers are running on the same node. This scheduler is described later in the chapter.

3.5 Scheduling Policies

The simulator supports several scheduling policies at the server and node level. A server scheduling policy schedules client requests and executor threads within a server process. A node level scheduling policy schedules server processes on a node. Node scheduling is only applicable when multiple server processes are sharing a node. If a single server resides on a node then only server level scheduling is used. The combination of server and node scheduling is called *two-level scheduling*. Clients may have explicit priorities associated with their requests to a server. This possibility is not considered in this research which is concerned with the effect of workload characteristics and system architecture on the performance of scheduling policies in client-server systems. The policies investigated in this thesis are described in the following sections.

Figure 3.5: A server-server interaction showing the server processing cycle. Two consecutive calls are made by the server. Execution of request at active server is broken up into three equal time slices, t_1 , t_2 , and t_3 .

3.5.1 Server Scheduling Policies

A number of both non-preemptive and preemptive policies that work with non-multithreaded as well as multithreaded systems are considered in this thesis.

First Come First Served (FCFS)

The First Come First Served (FCFS) algorithm is used as a basis for comparison since it is simple and commonly used in many systems. The FCFS algorithm does not use any knowledge of client request characteristics. Requests are processed in the order they arrive.

Shortest Request First (SRF)

The Shortest Request First (SRF) algorithm uses *a priori* knowledge of the amount of CPU time (service time) required by a request at a given server. The request with the least amount of required service time at the server is scheduled first. In conventional general purpose computer systems, request service time is not known and is hard to predict. However, in client-server systems a fixed set of procedures are invoked by RPC's, approximate execution time of these procedures may be made available to the scheduler.

Longest Path First (LPF)

An important characteristic of client requests is the *path length*. The path length of a request at a given server is the number of servers that are already blocked waiting for a response from this server. For example, the path length of a request arriving at the active server S_1 in Figure 3.6 is zero while that of requests at the pure server S_2 may be one or zero. Using LPF scheduling, the pure server S_2 will assign higher priority to requests coming from server S_1 which have a path length of one. Requests coming directly from the client will have a path length of zero and thus are given

Figure 3.6: A nested client-server system with an active server S_1 and a pure server S_2 .

lower priority in LPF scheduling. Requests with equal path lengths are executed in a FCFS fashion.

Adaptive Longest Path First (ALPF)

The LPF policy was shown in [5] to perform worse than FCFS in a multithreaded system where active servers in a nested request were not software bottlenecks. Conversely, the LPF policy performed well in situations where software bottlenecks existed despite multithreading. The Adaptive Longest Path First (ALPF) policy is a combination of the LPF and FCFS policies. A server using the ALPF policy will schedule requests in the same fashion as LPF if any of the previous servers blocked on the request had a request queue size of at least one when the request was generated. In a multithreaded server, a request queue size greater than zero indicates that all threads are busy processing requests. This may be a potential software bottleneck so a flag is set in the request before it is passed on to the next server. If this flag remains unset

then the next server will schedule requests in a FCFS fashion.

Longest Path First / Shortest Request First (LPF/SRF)

Combining path length and service time characteristics produces another algorithm called the LPF/SRF algorithm. Since path length is likely to be an integer number in the range $0 \dots N$ (where N is the longest path length) it is possible that requests with the same path length appear in the request queue for a server. To break ties in path length the LPF/SRF algorithm processes the request with the shortest service time first.

Preemptive Algorithms

The simulator supports preemptive versions of the ALPF, LPF, SRF and LPF/SRF algorithms. A preemptive Round Robin (RR) policy is also available, it is described later in this section. In the non-preemptive versions of the scheduling policies a new scheduling decision is made only when the currently executing thread blocks or completes and another thread is ready to run in the server. In preemptive scheduling a higher priority request is allowed to preempt a currently executing request of lower priority. Preemptive scheduling policies only work in conjunction with multithreading since more than one thread must be ready to run for preemption to occur.

Round Robin (RR)

The preemptive RR policy gives fixed slices (quantums) of CPU time to every thread that is ready to run in the system. A currently running thread (regardless of priority) is preempted when its quantum is complete. The next thread to run is picked from the ready to run queue in a FCFS fashion. Rescheduling will also occur if a thread blocks or completes before its quantum is complete. A typical quantum used in the experiments in this thesis is 100 ms.

Other Policies

Experiments have been performed with a number of other algorithms, such as Shortest Path First (SPF), Longest Alive (LA) and Shortest Alive (SA). The SPF policy gives higher priority to requests with short path lengths. The LA policy gives higher priority to older requests (age of request is determined by time stamp given to each request when it is created.) The SA policy gives high priority to newer requests. These policies were observed to perform poorly in most situations [16]. The simulator supports these policies but these policies were not investigated in this thesis.

3.5.2 Node Scheduling Algorithms

Given a number of server processes the node scheduler determines which server to run. The following node scheduling policies are investigated :

Single Level (SL)

The SL node scheduling policy allows the server scheduling policy to pick the next thread to run regardless of which server the thread belongs to. It is called single level scheduling because it is similar to kernel-level thread scheduling as supported by some operating systems. Essentially the kernel is aware of all threads in the system and can schedule them individually. This is in contrast to user-level threads which are scheduled by the process in which they are running. The kernel has no knowledge of user-level threads. For example, if the server scheduling policy is SRF and the node scheduling policy is SL, the next thread to run is determined by searching the entire ready to run queue for the node. The thread with the shortest request time will be picked from all the threads available.

Fair Share (FS)

The FS policy gives equal quanta of CPU time to all servers on a node. It is similar in operation to the RR server scheduling policy except the FS policy switches between server processes rather than individual threads. The FS policy has a time quantum amount that can be specified by the simulator input file. If a quantum of 0.1 is used then each server is given 0.1 sec of CPU time and is then preempted. The next server is picked in a FCFS order. If a server has no threads ready to run then it forfeits its quantum and the next server is picked to run. Once a server is selected to run it may employ any of the server scheduling policies described in the previous section. Several threads within the server may be scheduled to run before the server's quantum is complete. Once the quantum is complete execution within the server is suspended until the next quantum arrives.

Other Policies

Other node scheduling policies were implemented. Investigating the performance of these policies is beyond the scope of this thesis. These policies may be useful in further research and a description of these policies is provided in Appendix A.

3.6 Inter-arrival and Service Time Distributions

The simulator supports exponential and hyperexponential inter-arrival time distributions. As discussed in the previous chapter there is evidence that both distributions are important in client-server systems. The simulator also supports deterministic, exponential, and hyperexponential distributions for client service times. Exponential random numbers are generated by Parasol using an underlying uniform random number generator (drand48). The hyperexponential random numbers are generated using the bi-phase hyperexponential method described in [12]. The diagram in Figure 3.7 is a graphical representation of this generation technique. Two exponential

Figure 3.7: A bi-phase hyperexponential number generator. Small numbers have probability p , large numbers have probability $1 - p$.

distributions with means s and l are used. Numbers with a smaller mean, s , occur with a probability p whereas numbers with a larger mean, l , occur with a probability $1 - p$. The hyperexponential random number generator requires a mean, m , and a Coefficient of Variation, CV . In experiments using this hyperexponential number generator, the value of p is 0.95 which means smaller numbers occur 95% of the time and larger numbers occur 5% of the time. The values for s and l are calculated as follows.

$$s = m - \frac{\sqrt{(CV^2 - 1)}}{2p(1 - p)}(1 - p)m \quad (3.2)$$

$$l = m + \frac{\sqrt{(CV^2 - 1)}}{2p(1 - p)}pm \quad (3.3)$$

In most experiments, the client request inter-arrival times are modeled by exponential random numbers although in some experiments a hyperexponential inter-arrival time distribution is used to demonstrate the effect of higher variance on the performance of the scheduling policies. The service times for most experiments are modeled using an exponential distribution. Hyperexponential models are also used to demonstrate the effect of variability in service time on the performance of the scheduling policies. The execution time for the interface agents is modeled by an exponential distribution whose mean is fixed at 1.0 for all the scheduling experiments.

3.7 Simulator Design

The simulator is a C program that runs in a UNIX environment. The program reads an input file which contains all the configuration information necessary to perform the simulation. The simulation output is a text file containing a summary of the input file and the statistics collected during the simulation. The Parasol runtime library is used to collect these statistics while the simulation is running. Parasol generates 95% and 99% confidence intervals for each statistic. The output is processed using *awk* to generate input for the graphs shown in later chapters. The graphs are drawn using the *gnuplot* utility. The key part of the simulator is the Parasol runtime library which provides the necessary functions to simulate distributed systems.

3.7.1 Parasol

Parasol is a runtime library of functions for developing simulations of multithreaded distributed and multiprocessor systems. Parasol is an event-driven simulation tool that can be used to simulate hardware and software entities. It was developed by Professor John Neilson of the School of Computer Science at Carleton University [19]. The Parasol library provides a rich set of functions to create nodes, CPU's, tasks (threads), and communication busses. Parasol provides facilities to generate random numbers of various distributions and to collect statistics as simulation is running. Parasol allows for a user-defined scheduler for any node in the system. This capability was exploited in the simulator to implement the scheduling policies under investigation.

3.7.2 System Startup and Configuration

The simulator reads all configuration information from an input file. The format of this file is described in Appendix D. The configuration information consists of the following :

- **Simulation duration**
- **Node scheduling policy**
- **Server scheduling policy**
- **Number of nodes**
- **Number of servers**
- **Number of clients**
- **For each client :**
 - **Mean, CV of inter-arrival time**
 - **Servers to call**
 - **Mean number of calls to each server**
- **For each server :**
 - **Node server resides on**
 - **Mean, CV of service time**
 - **Number of executor threads**
 - **Servers to call**
 - **Mean number of calls to each server**

At startup the *genesis* thread is the first thing to run. This thread is responsible for setting up the simulation. The configuration of the simulation is held in a database that can be accessed by all servers and clients to retrieve any needed information. Once the information is read by the *genesis* thread it terminates itself and is removed from the now running simulation. The simulator runs each simulation in blocks of 10 to produce confidence intervals for each statistic collected. The first 10% of simulation

time is disregarded by the simulator to remove any startup transients. The following statistics are collected during a simulation :

- Mean response time recorded at each client
- Mean number of outstanding client requests
- Mean queue size for each server
- Mean residence time for each server
- Utilization for each node.

3.7.3 Client Design

There are two types of clients in the simulator. For open systems simulation the *open client* is used. The diagram in Figure 3.8 shows the internal of the open client. The open client consists of an *client request manager* which dynamically creates interface agent threads at a rate equivalent to the client request arrival rate. The open client manager sleeps for the required amount of time (i.e. think time) then creates an interface agent and then sleeps again. The interface agents perform the client cycle (excluding think time), record response time for each server call and then terminate themselves. In an open systems simulation only one open client is created although many interface agents are created dynamically at runtime within the open client process. For closed system simulations a fixed number of client processes are created, each running on its own CPU. Each client executes the client cycle and records the response time for each server call. Closed clients do not terminate but run continuously and the population of clients is fixed throughout the simulation. The internals of the closed system client are shown in Figure 3.9.

Figure 3.8: The internals of an open systems client-server system.

Figure 3.9: The internals of a closed systems client.

3.7.4 Server Design

The server design is the same for open or closed system simulations. The server contains a listener thread which accepts requests from clients and places them in the request queue. The request queue is protected by two semaphores one semaphore is used to indicate the number of requests in the queue. If the queue is empty, executor threads are blocked on the size semaphore. The other semaphore is used to provide mutual exclusion to the queue to prevent concurrent access from threads. A non-multithreaded server contains one executor thread and a multithreaded server contains more than one. Executor threads are created at start up by the listener thread. The executor threads may make requests to other servers and are responsible for replying to the client with the result. The diagram in Figure 3.8 shows the internals of the server. System services such as semaphores are provided by Parasol.

3.7.5 Custom Scheduler

The custom scheduler is a process which runs in all server nodes in the simulator. For nodes containing the custom scheduler, Parasol notifies the scheduler process that scheduling is required. This notification occurs every time a thread in the system becomes ready to run. Parasol also supplies the ready to run queue. The custom scheduler searches the ready to run queue and picks the next thread to run based on the node scheduling policy (if applicable) and the server scheduling policy. The node scheduling policy is used to decide which server to run next. The scheduler knows which server each threads belongs to and only schedules the next thread to run based on which server has been picked by the node scheduling policy. Which thread to run from the server is based on the server scheduling policy. Once a thread has been scheduled to run, Parasol is notified and the scheduler returns control to Parasol. In the RR server scheduling policy or the FS node scheduling policy the scheduler is awakened at regular intervals to reschedule. The currently running thread is preempted and another is picked from the ready to run queue.

Chapter 4

Performance of Single-Level Scheduling

4.1 Overview

This section presents the results of the simulation experiments for single-level scheduling on open systems. The results presented in this chapter are representative of the results gained through the experiments. Further results are available in Appendix B. In all experiments each node has a single server and a single CPU executing the server. In most cases the results are in terms of mean response time in seconds versus the arrival rate of client requests to the system in terms of requests per second. In some cases the results are in terms of mean response time in seconds versus CV. In these experiments the arrival rate is fixed and the CV of the service time or inter-arrival rate is varied. Simulation results are within a $\pm 2\%$ interval at a 95% confidence level.

Figure 4.1: An open client-server system with two servers.

4.2 Non-Multithreaded Servers with Non-preemptive Scheduling

The investigation of non-multithreaded client-server systems is important since many existing systems are still non-multithreaded. Many PC-based servers are non-multithreaded [26]. The performance of non-preemptive client-server scheduling policies were investigated for a two server client-server system as shown in Figure 4.1. The two servers S_1 and S_2 are non-multithreaded. The numbers labeling the arcs from clients to servers and from servers to servers indicate the mean number of calls made to the server. This system, for example, can represent a database server (S_1) that makes use of a remote file server (S_2). The client makes calls to the database and file server during execution of some database-oriented application. This architecture creates two paths for requests arriving at the file server (S_2) one path directly from the client and the second from the database server (S_1). The existence of these two paths allows the LPF and LPF/SRF algorithm to exploit path length to achieve higher performance.

The graph in Figure 4.2 shows the relative performance of the scheduling policies. The results show that ALPF, LPF, SRF and LPF/SRF policies perform better than FCFS in this case. In a uniprocessor system FCFS scheduling tends to allow large

requests to monopolize the CPU which causes increased response times for small requests. The LPF policy performs better in this case because higher priority is given to requests from server S_1 since they have a larger number of blocked requests waiting on their response. This helps alleviate the software bottleneck at server S_1 which in turn improves client response time. The ALPF policy demonstrates the same performance as the LPF policy. The SRF policy performs better than ALPF, LPF, and FCFS. The SRF policy gives higher priority to small requests which reduces queuing delay at all servers and reduces client response time. The combined LPF/SRF policy takes advantage of path length and service time request characteristics. The LPF/SRF policy produces the lowest client-response times in this system. Based on these observations, this policy seems to be the best choice for this client-server system with non-multithreaded servers and with non-preemptive scheduling. Service time and path information is useful in improving performance in a non-multithreaded client-server system. Combining path and service time information in the scheduler increases performance further.

4.3 Multithreaded Servers with Non-preemptive Scheduling

Threads were added to the active server S_1 of the client-server system of Figure 4.1. The performance of the scheduling policies are shown in Figure 4.3. One of the important observations is that with multithreading the client response time is relatively low at high arrival rates ($\lambda = 0.45$ requests per second). In the non-multithreaded case (see Figure 4.2) the system saturated around $\lambda = 0.375$ requests per second. In the non-multithreaded case server S_1 is a software bottleneck because the server process is fully occupied executing requests and waiting on responses from server S_2 even though the hardware utilization is low (35% when $\lambda = 0.35$ requests per second). Use of path information by the scheduler is actually detrimental to

Figure 4.2: Performance of non-multithreaded two server system with non-preemptive scheduling. Mean service time at both servers is 1.0 and CV is 1.0.

performance in this system. The addition of threads is very effective in clearing the software bottleneck at server S_1 and increasing the maximum throughput sustainable by the system. The efficiency of server S_1 is increased because additional requests can be processed while blocking on responses from the server S_2 . The processor at server S_2 becomes the new performance bottleneck (90% utilization when $\lambda = 0.45$ requests per second). Another significant result is that the ALPF, LPF and LPF/SRF policies perform comparably with FCFS in this client-server system when multithreaded servers were used. In the previous experiment they performed better than FCFS in the same client-server system with non-multithreaded servers. The SRF policy performs slightly better than FCFS in this case although the performance of all the policies is close together. Similar results are obtained for a different system, shown in Figures B.1 and B.2 in Appendix B.

To investigate the level of multithreading required in order to improve performance, the number of threads in server S_1 was varied from 1 to 10. The results of which are shown in Figure 4.4. A significant drop in response time is observed even when the active server S_1 contains two threads. Also, adding more than two threads does not seem to improve performance for this system at the arrival rate of $\lambda = 0.375$ requests per second).

4.4 Multithreaded Servers with Preemptive Scheduling

The scheduler can be made preemptive which means the scheduler will preempt the execution of a lower priority thread if a higher priority thread becomes ready to run. For example, consider a situation in which LPF scheduling is used at a server and the currently executing request has a path length of zero. A new request arrives with a path length of one and is assigned to a thread for execution. Once this thread is ready to run the currently executing thread (executing request with path length zero)

Figure 4.3: Performance of multithreaded two server system with non-preemptive scheduling. Server S_1 has five threads. Mean service time at both servers is 1.0 and CV is 1.0.

Figure 4.4: Impact of level of multithreading on the performance of the two server system of Figure 4.1 with non-preemptive scheduling. Arrival rate was fixed at $\lambda = 0.375$ requests per second. Both servers have a mean service time of 1.0 and CV = 1.0.

is immediately preempted to allow the new request to execute. In order to support preemption the server must be multithreaded. There must be another thread ready to run for preemption to occur and thus is impossible in a non-multithreaded server. The listener thread in each server is given the highest priority and may preempt any executor thread currently running. This prevents the listener thread from being starved of CPU access and prevents a backlog of client requests from forming.

The results of using preemptive scheduling in the two server system of Figure 4.1 are shown in Figure 4.5. The preemptive ALPF and LPF policies perform worse than the other policies including FCFS (which is non-preemptive but included for comparison). In fact, the preemptive LPF policy performs worse than the non-preemptive version. This behaviour is related to that observed in the previous experiment using multithreaded servers with non-preemptive scheduling. The LPF and ALPF policy were observed to perform poorly in this case. The use of preemptive scheduling further diminishes the performance of these policies since requests with shorter path lengths (i.e. directly from the client) are further starved from CPU access. This further increases the response time of requests directly from the client and increases the overall client response time. The preemptive SRF policy demonstrates the best performance. The preemptive LPF/SRF policy performs slightly worse than the preemptive SRF policy but better than ALPF, LPF and FCFS policies. The RR policy performs equally with FCFS in this system. In this client-server system the preemptive policies using service time as a scheduling criterion were observed to perform better than policies using path information as a scheduling criterion. Similar results are obtained for a different system, shown in Figure B.4 in Appendix B. The LPF policy demonstrates better performance when the active server is heavily loaded in a multithreaded system using preemptive scheduling (see Figure B.3 in Appendix B.)

A comparison of preemptive and non-preemptive versions of the SRF and LPF/SRF policies is shown in Figure 4.6. Preemption improves the performance of both policies. A preemptive scheduler ensures that a new request with smaller service time

Figure 4.5: Performance of multithreaded two server system with preemptive scheduling. Server S_1 and S_2 have five threads. Mean service time at both servers is 1.0 and CV is 1.0.

Figure 4.6: Comparison of response times for preemptive and non-preemptive versions of the SRF and LPF/SRF policy. Server S_1 and S_2 have five threads. Mean service time at both servers is 1.0 and CV is 1.0.

is executed when it arrives thus preempting a currently running request. With non-preemptive scheduling a thread must voluntarily relinquish the CPU before the scheduler allocate another thread to run. This can cause smaller requests to wait for execution while large jobs are still executing. Preemptive scheduling was observed to be beneficial in this client-server system particularly for the SRF policy.

4.5 Effects of Relative Loading of Servers

In the previous results each server had equal mean service times of 1.0. Increasing the ratio between mean service time changes the performance of the scheduling policies. This effect is more pronounced when the service time of the pure server is increased. The performance of the scheduling policies in the system of Figure 4.1 with non-

Figure 4.7: Performance of non-multithreaded two server system with non-preemptive scheduling. Mean service time at server S_1 is 1.0 and 5.0 at server S_2 . CV is 1.0.

multithreaded servers and the mean service time for the pure server S_2 increased to 5.0 is shown in Figure 4.7. The ALPF, LPF, and LPF/SRF policies perform the best in this case. The software bottleneck at active server S_1 increases in strength because the pure server S_2 is more heavily utilized and significant queuing delay is experienced by all calls to this server. As discussed earlier, the ALPF, LPF, and LPF/SRF policies are effective in alleviating the software bottleneck at server S_1 . The SRF policy performs slightly better than FCFS but worse than ALPF, LPF, and LPF/SRF policies. Therefore the strength of the software bottleneck at a nested server, such as S_1 , affects the performance of path length based policies. Service time information becomes less useful as the software bottleneck increases at server S_1 .

The performance of the scheduling policies when the service time of the active server S_1 is increased to 5.0 is shown in Figure 4.8. The mean service time of the pure

Figure 4.8: Performance of non-multithreaded two server system with non-preemptive scheduling. Mean service time at server S_1 is 5.0 and 1.0 at server S_2 . CV is 1.0.

server S_2 is 1.0. The SRF policy performs the best in this situation. The ALPF and LPF policies perform worse because all requests arriving at server S_1 have the same path length so these policies perform the same as FCFS for this server. The ALPF and LPF policies are only effective on nested servers (like S_2) where previous servers in the request chain are software bottlenecks. In the case where the active server is under high utilization relative to the pure server the ALPF and LPF policies provide no benefit over FCFS scheduling.

4.6 Effects of Service Time Variability

The effect of variability in service time is an important consideration in evaluating scheduling policies because certain real-life systems tend to have high variability in

workload (see [2], [3], and [27] for example). To investigate the impact of high variability in service time on the performance of the scheduling policies experiments were run using the client-server system of Figure 4.1. The CV of the service time was varied from one to five with a fixed workload ($\lambda = 0.3$ requests per second). The results of varying CV of the service time for the active server S_1 is shown in Figure 4.9. The results of varying CV of the service time for the pure server S_2 is shown in Figure 4.10. As the variability of service time increases the effect of large requests monopolizing the CPU increases. As CV increases the difference in size between small and large requests increases. Scheduling policies such FCFS, ALPF and LPF do not use service time characteristics and consequently their performance suffers. The SRF and LPF/SRF policy perform better because higher priority is given to smaller requests which prevents large requests from monopolizing the CPU. Thus these policies are less affected by variability than FCFS, ALPF and LPF policies but response times still increase significantly as CV increases. Comparing the graphs of Figures 4.9 and 4.10 the effect of variability in service time for the pure server seems to have a stronger effect on performance than the active server S_1 . The pure server S_2 is more heavily utilized than server S_1 thus the variability in service has a more dramatic effect. As in the previous case the policies using service time information perform better in comparison to policies not using service time information. It was observed that path information does not benefit performance in this client-server system with high variability in service times. The results of varying CV of the service time for the pure server S_2 with multithreaded servers is shown in Figure B.6 in Appendix B. The results of varying CV of the service time for the active server S_1 with multithreaded servers is shown in Figure B.5 in Appendix B. Both of these figures demonstrate the strong effect of variability in service time on the relative performance of these policies.

The experiment corresponding to Figure 4.10 was repeated but with multithreaded servers and preemptive scheduling. The results of the experiment are shown in Figure 4.11. A dramatic improvement in performance was observed when using the SRF

Figure 4.9: Impact of variability in request service time for the active server S_1 . $\lambda = 0.3$ requests per second. Server S_1 and S_2 are non-multithreaded and non-preemptive scheduling is used. Mean service time is 1.0, CV for pure server S_2 is 1.0. CV of service time for active server S_1 is varied.

Figure 4.10: Impact of variability in request service time for the pure server S_2 . $\lambda = 0.3$ requests per second. Server S_1 and S_2 are non-multithreaded and non-preemptive scheduling is used. Mean service time is 1.0, CV of active server S_1 is 1.0. CV of service time for pure server S_2 is varied.

Figure 4.11: Impact of variability in request service time for the pure server S_2 . $\lambda = 0.3$ requests per second. Server S_1 and S_2 are multithreaded and preemptive scheduling is used. Mean service time is 1.0.

and RR policy. The Round Robin performs very well in this situation because it gives equal quantum of CPU time to all requests which prevents large requests from monopolizing the CPU. The RR policy still allows large requests to execute because all requests are given equal opportunity to share the CPU. Therefore the RR policy produces a nearly flat response time curve as CV is increased. The preemptive SRF policy performs the best in this system, in fact, the response time decreases slightly as CV increases. The ALPF policy performs better than LPF and almost equally as well as the LPF/SRF policy. The LPF policy performs worse than the other policies. In situations where preemptive scheduling is possible the SRF policy is a good choice in this client-server system with large variations in service time. Unfortunately preemptive scheduling may not be possible at a server. A file server, for example, cannot preempt disk access to allow another request to be processed. A

file server must wait for the current request to complete before scheduling the next request. Servers providing access to I/O devices such as disks may only support non-preemptive scheduling.

4.7 Effects of Inter-arrival Time Variability

To investigate the effect of inter-arrival time variability, the CV of the inter-arrival time was varied from one to five for the client-server system of Figure 4.1. The results for non-multithreaded servers using non-preemptive scheduling is shown in Figure 4.12. As CV increases the response time increases dramatically. The policies using service time characteristics, the SRF and LPF/SRF policies, perform better than the others. The policies using path length, the ALPF and LPF policies, perform slightly better than FCFS scheduling.

Performance of all policies improve with the addition of multithreading to the active server, S_1 . These results are found in Figure 4.13. The SRF policy better than the other policies. The ALPF, LPF, and LPF/SRF policy perform slightly worse than FCFS. This observation has been made in previous experiments, the use of path length characteristics can be detrimental to performance when software bottlenecks have been removed through the addition of multithreading.

The results of adding preemptive scheduling to this system is shown in Figure 4.14. The results of FCFS scheduling, although non-preemptive, is included in the results as a point of reference. The policies using path length characteristics, the ALPF and LPF policies, perform much worse than FCFS. The SRF policy performs better than all other policies and the rate of increase in response time versus an increase in the CV of inter-arrival time is much small than the other policies. The LPF/SRF policy performs slightly better than FCFS due its use of service time characteristics which benefits the performance of the SRF policy. The FCFS and RR policy have near identical performance in this experiment. The RR policy does not provide any

Figure 4.12: Impact of variability in inter-arrival time. $\lambda = 0.3$ per second. Server S_1 and S_2 are non-multithreaded and non-preemptive scheduling is used. Mean service time is 1.0 and CV of service time is 1.0.

Figure 4.13: Impact of variability in inter-arrival time. $\lambda = 0.3$ per second. Server S_1 and S_2 are multithreaded and non-preemptive scheduling is used. Mean service time is 1.0 and CV of service time is 1.0.

Figure 4.14: Impact of variability in inter-arrival time. $\lambda = 0.3$ per second. Server S_1 and S_2 are multithreaded and preemptive scheduling is used. Mean service time is 1.0 and CV of service time is 1.0.

benefit when there is large variation in inter-arrival time and low variance in service time.

4.8 Effects of System Architecture

The architecture of the client-server systems plays a role in the performance of the system. Given the two server system of Figure 4.1 it is possible to increase the depth of the system by adding more active servers in the request chain. Such a system is illustrated in Figure 4.15. This system consists of two active servers S_1 and S_2 and one pure server S_3 . The clients in this system makes calls to all three servers as part of its regular processing.

The performance of the scheduling policies in the client-server system of Figure

Figure 4.15: An open client-server system with three servers.

4.15 with non-multithreaded servers and non-preemptive scheduling is shown in Figure 4.16. The LPF/SRF policy performs the best in this case. The LPF and ALPF policies perform almost as well as the LPF/SRF policy. These policies use path length characteristics and this provides more benefit than service time characteristics in this system with a larger number of nested servers. The FCFS policy perform poorly in this experiment.

The performance of the scheduling policies when multithreading is added to the client-server system of Figure 4.15 is shown in Figure 4.17. As observed in previous experiments, the maximum sustainable throughput of the system is improved dramatically. In the non-multithreaded case the system saturated at $\lambda = 0.19$ whereas in the multithreaded case the system saturated above $\lambda = 0.3$. There is approximately a 58% increase in throughput with the addition of 10 threads in servers S_1 and S_2 . Similar to results for the two-server system of Figure 4.1, the policies using path length characters, the ALPF, LPF, and LPF/SRF policies, perform worse than FCFS with the addition of multithreading. The results for the same system with multithreaded servers and preemptive scheduling is shown in Figure B.7 in Ap-

Figure 4.16: Performance of three server system of Figure 4.15. Servers are non-multithreaded and non-preemptive scheduling is used. Mean service time is 1.0 and $CV = 1.0$.

Figure 4.17: Performance of three server system of Figure 4.15. The active servers S_1 and S_2 are multithreaded with 10 threads in each server. Non-preemptive scheduling is used. Mean service time is 1.0 and $CV = 1.0$.

pendix B. Using preemptive scheduling increases the differences in performance of the scheduling policies.

It is also possible to increase the breadth of the system by adding more pure servers to the request chain as in Figure 4.18. This system consists of one active server S_1 and two pure servers S_2 and S_3 . This system could represent a partitioned database that consists of one database server, S_1 , any two files servers, S_2 and S_3 , which each contain segments of the database. The client in this example makes requests directly to both file servers, S_2 and S_3 , and to the database server S_1 .

The performance of the scheduling policies when the active server S_1 is multithreaded is shown in Figure 4.19. The SRF and LPF/SRF policies perform the best in this situation. The LPF and ALPF policies perform better than FCFS. This is an interesting result since in previous experiments these policies tend to perform worse

Figure 4.18: An open client-server system with one active server, S_1 , and two pure servers, S_2 and S_3 .

than FCFS in multithreaded systems. These results tend to reinforce the relationship between software bottlenecks and the performance of policies using path length characteristics. The active server S_1 although multithreaded is still a software bottleneck and policies such as LPF and ALPF can still improve performance. Figure 4.20 shows the performance of the scheduling policies when the number of threads in active server S_1 is varied. The response time curve bottoms out at approximately five threads although it is still decreasing somewhat at ten threads. The number of threads required to remove a software bottleneck is dependent on the architecture of the client-server system. Server S_1 must make a call to pure servers S_2 and S_3 to complete one request. This increases the amount of blocking time experienced by the server and increases the opportunity to improve performance through multithreading. The results for the same system with multithreaded servers and preemptive scheduling is shown in Figure B.8 in Appendix B. Using preemptive scheduling increases the differences in performance of the scheduling policies.

Figure 4.19: Performance of the server system of Figure 4.18. The active server S_1 is multithreaded with 5 threads. Non-preemptive scheduling is used. Mean service time is 1.0 and $CV = 1.0$.

Figure 4.20: Impact of level of multithreading on performance of three server system of Figure 4.18. Number of threads applies to active server S_1 only. Non-preemptive scheduling is used. Mean service time is 1.0 and $CV = 1.0$.

4.9 Discussion of Results

The results of the experiments presented in this chapter reveal a number of relationships between scheduling, multithreading, system characteristics and performance. These results are summarized as follows :

Scheduling and Multithreading : Both of these techniques have a strong impact on system performance. Multithreading improves efficiency at active servers and increases the maximum attainable throughput of the system. Appropriate scheduling policies improve system response times for many situations.

Multithreading and Preemption : On non-multithreaded systems, non-preemptive policies based on request characteristics such as path length and service time can improve system performance. In multithreaded servers this improvement is somewhat diminished although still observable, particularly policies based on service time characteristics such as the SRF and LPF/SRF policy. The FCFS policy performs comparably with other policies in multithreaded systems using non-preemptive scheduling. Preemptive scheduling produces further improvement in system performance. In particular, policies using service time characteristics such as the SRF and LPF/SRF policy perform better than non-preemptive versions. The RR policy has similar performance to FCFS in systems where the service time distribution is exponential.

Workload Characteristics : Workload characteristics have significant impact on the performance of the scheduling policies. An increase in the variability of service time at active and pure servers is detrimental to performance. The effect is more pronounced in the active server since they are more heavily utilized in the experiments. In non-preemptive scheduling the policies using service time characteristics, the SRF and LPF/SRF policies, tend to perform better than the other policies when the CV of workload increases. In preemptive scheduling the SRF and RR policies perform better than other policies as the CV of service

time increases. Both policies demonstrate nearly flat response time curves as CV of service time increases. In RR scheduling, giving equal quantum of CPU time to all requests, the starvation of large requests is prevented. In preemptive SRF scheduling small requests are given the highest priority and currently running requests are preempted if an even smaller request arrives. This improves the response time to small requests significantly. Even though large requests tend to be starved with SRF scheduling they constitute a small portion of requests. The overall effect of preemptive SRF policy seems to be better than other policies when CV of service time is high.

An increase in the variability of inter-arrival times is also detrimental to performance. Policies using path length characteristics performed poorly, particularly with multithreaded servers using preemptive scheduling. The preemptive version of the SRF policy performs the best because it is effective in reducing queuing delays at all servers. The RR and FCFS policies performed equally well when the CV of service time is low.

System Architecture : Policies using path length characteristics show more benefit as the depth of the system (number of vertically nested servers) increases and as the breadth (the number of nested pure servers) of the system increases. In non-multithreaded systems, as the depth of the system increases the benefit of path length based policies increases. This benefit is reduced and may even disappear when multithreaded servers are used. However, as the breadth of the system increases, the benefit of using path length increases even when multithreaded servers are used. This is due to the fact that the software bottleneck at the active server increases in strength as the breadth of the system increases. Path length characteristics in scheduling can still be beneficial if software bottlenecks exist.

Scheduling Based on Knowledge of Request Characteristics : Using request characteristics in scheduling can improve performance. The use of service time characteristics tends to improve performance in all situations. In particular, preemptive policies using service time characteristics tend to improve performance more than other policies. In general, associating a high priority to small requests reduces queuing delays at all servers thus reducing response time of the system.

Policies based on path length characteristics are useful in non-multithreaded systems. These policies associate a higher priority to requests with a larger number of blocked servers waiting on their reply. This tends to alleviate software bottlenecks at active servers to some degree. In the absence of these software bottlenecks, these policies provide no benefit and can, in some cases, perform worse than FCFS scheduling. These policies seem to be more beneficial on systems where software bottlenecks are known to exist. The ALPF policy, in most cases, performed the same as the LPF policy with one notable exception. The preemptive ALPF policy performs better than LPF when CV of service time is high. The combined LPF/SRF policy which uses path length and service time characteristics tends to perform better than the other path length based policies, LPF and ALPF. In systems where LPF and ALPF perform well, the LPF/SRF policy tends to perform even better. In general, path length is an important characteristics for scheduling although, based on the observations in the preceding experiments, its application is more useful in systems where software bottlenecks exist.

Chapter 5

Performance of Systems with Servers Sharing a CPU

5.1 Overview

The previous chapter dealt with scheduling at servers where each server resided on its own node with its private CPU. It is possible for a node to host several software servers. An example of such a node would be a workstation that hosts a file and name server for a company network. When software servers share the same node the scheduling of the server processes becomes important to system performance. In current systems, for example DCE, this responsibility is left to the host operating system. The scheduling of threads within each software process is left to the process to manage. This creates a *two-level scheduling* mechanism, the first level being the node scheduling policy (traditionally the host operating system) which manages the software server processes. The second level is the server scheduler which manages the execution of threads within the server process. Another possibility for node scheduling is having the host operating system handle all executable elements (threads and processes) together. This is termed *single-level (SL) scheduling* because there is one level of scheduling for all executable elements in the operating system. This

is equivalent to kernel-level scheduling where the operating system kernel manages all scheduling decisions. Kernel-level scheduler evaluates all ready to run threads regardless of which server process they belong to.

A comparison of single-level and two-level node scheduling presented in this chapter. In particular the effects of system architecture and workload on the performance of node scheduling policies are examined. Many operating systems employ a fair share scheduling policy for process management. A FS policy gives equal quantum of CPU time to all processes. The Fair Share (FS) two-level scheduling policy was compared against SL scheduling policy to investigate the benefits of single-level against two-level scheduling.

5.2 Performance of Scheduling Policies in Systems with Servers Sharing a CPU

A client-server system consisting of one active server and two pure servers is shown in Figure 5.1. The active server S_1 and the pure server S_2 share the same node. The pure server S_3 resides on its own node. For all the systems considered in this paper, each node has one CPU. A comparison of the performance of the FCFS server scheduling policy for different node scheduling policies is shown in Figure 5.2. The same comparison for the LPF and SRF policy is shown in Figure 5.3 and 5.4 respectively. In this case, FS node scheduling improves performance for all three server scheduling policies. Therefore using a two-level scheduling policy that gives equal quantum of CPU to each server process is beneficial in this client-server system. In fact, the combination of FS node scheduling and FCFS server scheduling is similar to that found in current systems such as DCE. However, further experiments show that system architecture plays an important role in the performance of node scheduling and is discussed in the next section.

Figure 5.1: Three server client-server architecture. The active server S_1 and the pure server S_2 share the same node. The pure server S_3 resides on its own node. Each node has one CPU. Service time is 1.0 at each server, $CV = 1.0$.

Figure 5.2: Comparison of the performance of SL and FS node scheduling. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

Figure 5.3: Comparison of the performance of the LPF algorithm with SL and FS node scheduling. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

Figure 5.4: Comparison of the performance of the SRF algorithm with SL and FS node scheduling. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

Figure 5.5: **Three server client-server architecture.** The active servers S_1 and S_2 share the same CPU. The pure server S_3 resides on its own CPU. Service time at all servers is 1.0, $CV = 1.0$

5.3 Effect of System Architecture

The client-server system in Figure 5.1 is only one example of many possible configurations. There seems to be a correlation between scheduling policy performance and architecture. A three server architecture with two active servers S_1 and S_2 and one pure server, S_3 is shown in Figure 5.5. The servers S_1 and S_2 share the same node. This architecture has two nested servers thus requests arriving at pure server S_3 will have a path length of 0, 1 or 2.

A comparison of the performance of the FCFS server scheduling policy for SL and FS node scheduling is shown in Figure 5.6. Results for the LPF and SRF server scheduling policies are shown in Figures 5.7 and 5.8. These results show that FS node scheduling does not improve performance and at high workloads tends to perform worse than SL node scheduling. This indicates that FS node scheduling may not be applicable to all architectures.

Figure 5.6: Comparison of the performance of the FCFS policy with SL and FS node scheduling in client-server system of Figure 5.5. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

Figure 5.7: Comparison of the performance of the LPF policy with SL and FS node scheduling in client-server system of Figure 5.5. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

Figure 5.8: Comparison of the performance of the SRF policy with SL and FS node scheduling in client-server system of Figure 5.5. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

Figure 5.9: Three server client-server architecture. The pure servers S_2 and S_3 share the same node. The active server S_1 resides on its own node. Each node has one CPU. Service time at all servers is 1.0, $CV = 1.0$

5.4 Effect of Allocation of Active and Pure Servers

A node can contain any combination of active and pure servers. This allocation of active and pure servers can impact the performance of the system. The client-server system of Figure 5.1 has been modified in Figure 5.9 so that both pure servers S_2 and S_3 share the same node and active server S_1 resides on its own node.

A comparison of the performance of the FCFS server scheduling policy with the FS and SL node scheduling policy is shown in Figure 5.10. In this case the SL node scheduling policy demonstrates superior performance to the FS node scheduling policy. The SL node scheduler picks threads from both servers regardless of which process owns the thread. For pure servers in this client-server system, this leads to a more efficient allocation of the CPU because the scheduler can pick the best thread to run from all ready to run threads in the node. The threads in a pure server only consume CPU time and do not block on calls to other servers. The FS node scheduling policy would tend to delay smaller requests because they are preempted

Figure 5.10: Comparison of the performance of the SL and FS node scheduling in client-server system of Figure 5.9. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

while executing to allow other threads to run. The SL node scheduling policy allows requests to run to completion. Similar relative performance between the FS and SL node scheduling policies were found with the LPF and SRF server scheduling policies (see Figures C.1 and C.2 in Appendix C.)

A comparison of the performance of the two systems of Figure 5.1 and Figure 5.9 are shown in Figure 5.11. In this system there is a performance advantage in allocating server S_1 and S_2 on the same node compared to the two pure servers S_2 and S_3 on the same node. The active server S_1 is not as heavily utilized as either S_2 or S_3 . When S_1 and S_2 are combined on a node the overall utilization of the node is lower than when S_2 and S_3 share the same node. Another difference is the relative performance of the FS and SL node scheduling policies. In the first mapping (CS 1), an active and pure server share the same node. The FS node scheduling gives better

Figure 5.11: Comparison of the performance of FCFS server scheduling policy for the client-server system of Figures 5.1 (CS 1) and 5.9 (CS 2). Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

performance. In the second mapping (CS 2), two pure servers share the same node and the SL policy gives the best performance.

To investigate this effect further the client-server system of Figure 5.5 was modified so that servers S_2 and S_3 share the same node and server S_1 resides on its own node. The results of this experiment are shown in Figure 5.12. In this case the FS and SL node scheduling policy perform equally until higher workloads in which the SL policy gives better performance. Similar relative performance between the FS and SL node scheduling policies were found with the LPF and SRF server scheduling policies (see Figures C.3 and C.4 in Appendix C.) This result indicates that the performance improvement attained by FS node scheduling may only occur in certain systems such as in Figure 5.1. In this system, the active server, S_1 , makes calls to two pure servers, one of which is external to its node. The FS policy improves performance in this

Figure 5.12: Comparison of the performance of the SL and FS node scheduling in client-server system of Figure 5.9. Servers S_2 and S_3 share the same node and server S_1 resides on its own node. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

case because it allows the active server to run more often than would be the case with SL node scheduling. Further experimentation has shown that by increasing the ratio of service time between the active and pure server in this client-server system increases the performance of the FS policy, the results of this experiment are in the Appendix C. The pure server which is more heavily utilized than the active server in this system tends to have more threads ready to to run and dominates the ready to run queue. The active server with most of its threads blocked has less threads in the ready to run queue and has to wait for CPU time longer when a response is received and the thread becomes unblocked. The FS policy gives equal share of CPU time to each process which allows the active server to complete pending requests. The active server in client-server system of Figure 5.1 also makes calls to an external pure server, S_3 , by freeing up the threads blocked on this server the overall client response time decreases. This performance gain is lost in the client-server system of Figure 5.5 because the active server is blocked on requests to a server which resides on the same node. The FS node scheduling policy does not seem to improve performance on nodes that have active servers that do not make calls to external servers. To further understand this phenomenon, the client-server system of Figure 5.1 was modified so that a fourth external server was added, S_4 , as illustrated in Figure 5.13.

A comparison of the performance of the FCFS server scheduling policy for the FS and SL node scheduling policies for the client-server system of Figure 5.13 is shown in Figure 5.14. The results for the LPF and SRF server scheduling policy are shown in Figures 5.15 and 5.16. In this case the FS policy performs better than SL policy, in fact the performance improvement ratio (FS/SL) for FCFS server scheduling is higher than in Figure 5.2. This indicates that the effectiveness of the FS policy depends on the number of calls made by the active server, S_1 , to servers external to its own node. The FS policy seems to prevent starvation of the active server which improves overall response time.

Figure 5.13: Four server client-server architecture. The active server, S_1 , and pure server, S_2 , share the same node. The pure servers S_3 and S_4 reside on their own node.

Figure 5.14: Comparison of the performance of the SL and FS node scheduling in client-server system of Figure 5.13. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

Figure 5.15: Comparison of the performance of the LPF server scheduling policy in conjunction with the SL and FS node scheduling policy in client-server system of Figure 5.13. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

Figure 5.16: Comparison of the performance of the SRF server scheduling policy in conjunction with the SL and FS node scheduling policy in client-server system of Figure 5.13. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

Figure 5.17: The effect on performance of varying the CV of the service time distribution for client-server system of Figure 5.1. Each server has 5 threads and mean service time is 1.0

5.5 Effect of Workload Characteristics

The effect of varying CV of the service time distribution on the performance of the client-server system of Figure 5.1 is shown in Figure 5.17. The benefit of the FS policy in this system improves as variability increases. The response time when using the FS policy remains relatively flat as CV of service time increases. As the variability of service time increases the probability of large requests increases. The FS policy prevents monopolization of the CPU by large requests because each server process is given equal quantum of CPU time. It behaves similarly to the RR server scheduling policy discussed in Chapter 4.

The effect of varying CV of the service time distribution on the performance of the client-server system of Figure 5.9 is shown in Figure 5.18. In this system the FS policy performs worse than the SL policy as CV increases. The allocation of

Figure 5.18: The effect on performance of varying the CV of the service time distribution for client-server system of Figure 5.9. Each server has 5 threads and mean service time is 1.0

servers on nodes seems to play an important role in performance even under different workloads. Further results are provided in Appendix C.

The effect of varying the CV of inter-arrival time on the performance of the client-server system of Figure 5.1 is shown in Figure 5.19.

5.6 Discussion of Results

The results of the experiments in this chapter reveal a relationship between the performance of node scheduling policies and the allocation of servers to nodes. The results are summarized as follows :

Effects of Server Allocation : The SL node scheduling policy was observed to give better performance in a system with two or more pure servers sharing the

Figure 5.19: The effect on performance of varying the CV of the inter-arrival time distribution for client-server system of Figure 5.1. Node scheduling is FCFS. Each server has 5 threads and mean service time is 1.0

same node. This policy also gives better performance in a system with two active servers sharing the same node. The SL node scheduling policy was observed to give better performance when used in conjunction with server scheduling policies using service time characteristics such as the SRF policy. This is particularly true when two or more pure servers are sharing the same node. The SL policy picks the next thread to run based on the decision of the server scheduling policy. For example, if the SRF policy is used, the thread with the smallest request time is picked to run regardless of which server it belongs to. This improves performance because the scheduler is picking the thread with the smallest request out of all possible requests in the system.

The FS node scheduling policy was observed to give better performance on a system with an active and pure server sharing the same node. In particular, the performance improvement is dependent on the number of calls made to servers external to the node the active server resides on. As the number of servers called by the active server increases so does the performance improvement. The FS node scheduling policy, by allowing the active server a fair share of the CPU resources, decreases the completion time for threads making calls to external servers. The FS policy is effectively aiding a software bottleneck at the active server by allocating more CPU time than is available in SL node scheduling.

Effects of Architecture : The depth of the client server systems (the number of vertically nested active servers) and the breadth of the system (the number of pure servers) affects the performance of node scheduling policies. It was observed that the SL node scheduling policy provided better performance as the depth, i.e. nested active servers was increased. The FS node scheduling policy was observed to give better performance on systems with more breadth, i.e. more pure servers. In both cases, the allocation of active and pure servers seemed to be a more dominant factor in the performance of the policy than system architecture.

Effects of Workload : The variability of service time has a dramatic effect on the performance of the scheduling policies. With higher CV of service time the FS policy provides good performance in systems where FS already gives better performance at a low CV. In systems where the SL node scheduling policy gives better performance at a low CV it tends to perform better as CV of service time increases.

Single-Level Versus Two-Level Scheduling : A case can be made for using single-level scheduling since it tends to perform better in many of the systems investigated. It allows the scheduler to pick the best thread to run from all ready to run threads. When used in conjunction with server scheduling policies that use request characteristics such as SRF and LPF it gives even better performance. The SL node scheduling policy was observed to be the best choice for nodes that contained two or more servers of the same type (i.e. two pure servers or two active servers.)

The FS policy is a two-level node scheduling policy that picks the server to run and then allows the server to pick the next thread to run. This policy is similar to the scheduling policies used on host operating systems in client-server systems. The FS policy was observed to give equal or worse performance than SL node scheduling and only provided good performance in certain situations. The advantage of the FS policy is that it prevents starvation of a server in a node with multiple servers. This was observed to be beneficial when active and pure servers shared the same node, in particular, when the active server made calls to servers outside its own node.

Chapter 6

Conclusions

6.1 Summary

The growth of client-servers systems as a paradigm for achieving distributed computing has made performance an important issue in design and implementation. This thesis has investigated two methods to achieve higher performance in client-server systems: multithreading of servers and scheduling. A simulation model and simulator program were developed to carry out this investigation for an open system. The simulation program is configurable and supports any combination of clients, active servers, and pure servers. Although closed system models are not investigated in this thesis, facilities to support modeling of closed systems are included in the simulation to support future research. Any combination of active and pure servers may be mapped to nodes in order to support node scheduling experiments. It also supports a number of distributions for service time and inter-arrival time.

In the experiments performed, multithreading was observed to reduce software bottlenecking in servers and increase the maximum throughput of the system. Client request service time and path length were found to be useful characteristics for scheduling. Scheduling policies using characteristics of client requests were observed to perform better than FCFS scheduling in many cases. Scheduling was observed

2 of/de 2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

to increase performance in non-multithreaded servers which is an important results since many existing systems do not support multithreading. Non-preemptive scheduling is also effective in multithreaded systems although to a lesser degree. Preemptive scheduling using policies based on service time characteristics were found to achieve better performance than the corresponding non-preemptive policy.

System architecture has an important effect on the performance of scheduling policies in client-server systems. The use of path length and service time in scheduling was effective in increasing the performance of systems. The use of path length for scheduling in non-multithreaded systems becomes more beneficial as the depth (the number of nested active servers) and the breadth (number of pure servers) is increased. In multithreaded systems policies based on path length tend to perform poorly as the depth of the system increases. Conversely, the same policies are more beneficial in systems as the breadth of the system increases. The performance of policies based on path length are dependent on the presence of software bottlenecks in the active servers in the system. In multithreaded systems these bottlenecks are lessened and may even disappear, which decreases the effectiveness of path length based policies. Policies based on service time were observed to be less affected by architecture, particularly in multithreaded systems.

The system workload characteristics were found to play an important role in performance of the scheduling policies. Scheduling policies based on path length were found to be adversely affected by increasing variability in service time. Policies based on service time were found to provide better performance as the variability of service time increases. In particular, the variability of service time was observed to have little effect on the response time of the preemptive SRF and RR policies. The variability of inter-arrival time was found to be detrimental to performance of all policies.

An investigation into two-level scheduling for client-server systems with multiple servers sharing one CPU was presented. A two-level policy, Fair Share (FS), was compared to a Single-Level (SL) policy. The FS policy gives equal quantum of CPU

time to each server process in a node. The FS policy is similar to the scheduler found in many operating systems. The SL policy is similar to kernel-level thread scheduling available in some operating systems. The predominant factor affecting performance of both policies was the allocation of active and pure servers to nodes. The FS policy was found to be effective in systems where active and pure servers shared the same node and the active server made calls to other servers external to its own node. In most other cases studied, the SL policy gave better performance. The scheduler on nodes using SL scheduling can pick the best thread to run from all possible threads in the system. This has the effect of reducing mean response times for all servers sharing the node.

6.2 Unique Characteristics of Client-Server Systems

Client-server systems differ from conventional mainframe systems in several ways. Client-server systems can be arranged in nested architectures where clients make calls to servers that in turn make calls to other servers. This nested architecture allows for the use of path length information in scheduling which has been shown to improve performance in many systems. Many client-server systems employ a blocking communication mechanism which means that performance may degrade because clients and active servers are blocked until the reply to a request is received. The use of multithreading to alleviate this problem in active servers has been shown to increase maximum throughput sustainable by a client-server system.

Scheduling policies that have been shown to improve performance in conventional systems also perform well in client-server systems. The SRF policy is similar to the Shortest Job First policy that is a known optimal non-preemptive policy in conventional systems. The SRF policy has been shown to perform well in most client-server systems studied. In particular, the SRF policy performs well in multithreaded client-

server systems with preemptive scheduling. Other policies that are well known in conventional systems such as FCFS and RR tend to demonstrate similar relative performance in client-server systems.

6.3 Contributions

The main contributions of this thesis are :

- An investigation of the performance of scheduling and multithreading in client-server systems, of which little previous research existed.
- The identification of path length as an important characteristic of client-server systems that can be used in scheduling to improve performance.
- The identification of important factors that affect the performance of scheduling and multithreading in client-server systems.
- A preliminary investigation into the relative performance of two-level and kernel-level scheduling in client-server systems.
- The development of a client-server simulation program for the investigation of scheduling and multithreading. This simulator program is configurable and supports many service time and inter-arrival time distributions. The program can be easily modified to support further research, for example, the simulation of multiprocessor servers.

A portion of the work presented in this thesis has been summarized in a paper entitled "The Performance of Multithreading and Scheduling in Client-Server Systems" [6] which has been submitted for publication to the Eighth IEEE Symposium on Parallel and Distributed Systems (SPDP'96).

6.4 Further Research

An important area for future research into scheduling in client-server systems would be the investigation of scheduling in multiprocessor client-server systems. In such systems the server nodes would be multiprocessor computers which adds a new dimension to the scheduling problem. The scheduler would then require processor allocation policies as well as server and thread scheduling policies. New policies may be required to take advantage of the characteristics of multiprocessor client-server systems. Along this same line, client-server systems are now evolving into peer-to-peer systems in which the distinction between client and server is less clear. In these systems clients may become servers for some services and servers may become clients for other services (as is the active server in the client-server models used in this thesis). Essentially, the roles as client and server are temporary and interchangeable. Schedulers in these systems may take advantage of an idle CPU on another workstation in a network to balance the server workload. The scheduler would then require CPU allocation policies (i.e. to allocate the required work to an idle CPU) as well as node and server scheduling policies.

Further investigation into node-level scheduling policies is warranted. The additional policies that the simulator currently supports are described in Appendix A. The use of server runtime characteristics, such as request queue length, may be useful in improving performance. In addition to this research, an investigation into the performance of scheduling and multithreading in closed systems may be useful.

Another important area of research would be the calibration and validation of the client-server simulation model. The simulation model would require modification to include scheduling and message passing overheads. The measurement of a real in-use client-server system would provide the necessary data to calibrate the model. The model could then be calibrated using trace-based input from the measured client-server system. The effect of scheduling and multithreading on the newly calibrated model would be an important contribution to this area of research.

References

- [1] R.M. Adler, "Distributed Coordination Models for Client-Server Computing", *IEEE Computer*, Vol. 28, No. 4, April 1995, pp. 14-22.
- [2] S. Blumson, "Workload Characterization in a Large Distributed File System", Tech. Report 94-3, Center for Information Technology Integration, University of Michigan, 1994.
- [3] R.R. Bodnarchuk, R.B. Bunt, "A Synthetic Workload Model for a Distributed System File Server", *Proceedings of the ACM SIGMETRICS Conference 1991*, San Diego, California, May 21-24, 1991, pp 50-59.
- [4] M.J. Fontenot, "Software Congestion, Mobile Servers, and the Hyperbolic Model", *IEEE Transactions on Software Engineering*, Vol. 15, 1989, pp. 947-962.
- [5] W.C. Graham, S. Majumdar, "The Effect of Scheduling and Multithreading on the Performance of Client-Server Systems", Tech. Report SCE-95-16, Department of Systems and Computer Engineering, Carleton University, Ottawa, October 1995.
- [6] W.C. Graham, S. Majumdar, "The Performance of Scheduling and Multithreading on Client-Server Systems", submitted for publication to the Eighth IEEE Symposium on Parallel and Distributed Processing (SPDP'96) to be held in New Orleans, October 1996.

- [7] S.A. Heimlich, "Traffic Characterization of the NFSNET National Backbone", *Proceedings of the ACM SIGMETRICS Conference 1988*, Santa Fe, New Mexico, May 24-27, 1988, pp 257-258.
- [8] O.C. Ibe, H. Choi, K.S. Trivedi, "Performance Evaluation of Client-Server Systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 11, November 1993, pp 1217-1229.
- [9] R. Jain, S.A. Routhier, "Packet Trains - Measurements and a New Model for Computer Network Traffic", *IEEE Journal on Selected Areas in Communications*, SAC 4, September 1986, pp 986-995.
- [10] A.M. Khandker, P. Honeyman, T.J. Teorey, "Performance of DCE RPC", Tech. Report 95-2, Center for Information Technology Integration, University of Michigan, 1995.
- [11] A.M. Khandker, J.A. Rolia, T.J. Teorey, "Performance Modeling of the Distributed Computing Environment", Tech. Report 95-6, Center for Information Technology Integration, University of Michigan, 1995.
- [12] H. Kobayashi, "Modeling and Analysis: An Introduction to System Performance Evaluation Methodology", Addison-Wesley, 1981.
- [13] T.G. Lewis, "Where is Client/Server Software Headed?", *IEEE Computer*, Vol. 28, No. 4, April 1995, pp 49-55.
- [14] H.W. Lockhart Jr., "OSF DCE Guide to Developing Distributed Applications", McGraw-Hill, 1994.
- [15] S. Majumdar, T. Phillips, "Scheduling in Client-Server Systems" *Proceedings of Fourteenth Annual ACM PODC Conference*, Ottawa, Ontario, August 20-23, 1995, pp 262.

- [16] Majumdar, S., Phillips, T., "Performance of Scheduling Strategies for Client-Server Systems" Submitted for publication to the International Conference on Parallel and Distributed Systems (ICPADS'96), 1996.
- [17] D.A. Menasce, V.A.F. Almeida, L.W. Dowdy, "Capacity Planning and Performance Modeling from Mainframes to Client-Server Systems", Prentice Hall, 1994.
- [18] D. Mirchandani, P. Biswas, "Characterizing and Modeling Ethernet Performance of Distributed DECwindows Applications", *Proceedings of the ACM SIGMETRICS Conference 1988*, Santa Fe, New Mexico, May 24-27, 1988, pp 253-254.
- [19] J.E. Neilson, "PARASOL Users' Manual", School of Computer Science, Carleton University, Ottawa, 1995.
- [20] J.E. Neilson, C.M. Woodside, D.C. Petriu, S. Majumdar, "Software Bottlenecking in Client-Server Systems and Rendezvous Networks", *IEEE Trans. on Software Engineering*, Vol. 21, No. 9, September 1995, pp 776-782.
- [21] B.J. Nelson, A.D. Birrell, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984, pp 39-59.
- [22] Open Software Foundation, "Introduction to OSF DCE", Prentice-Hall, 1992.
- [23] Open Software Foundation, "DCE RPC Internals and Data Structures", Revision 1.0, Open Software Foundation, August 1993.
- [24] J.A. Rolia, M. Starkey, G. Boersma, "Modeling RPC Performance on the AS/400", *Proceedings of CASCON '93*, Toronto, Ontario, October 24-28, 1993, pp 677-690.
- [25] A. Silberschatz, P.B. Galvin, "Operating Systems Concepts (Fourth Edition)", Addison-Wesley, 1994.

- [26] A. Sinha, "Client-Server Computing" *Comm. of the ACM*, Vol. 35, July 1992, pp. 77-98.
- [27] R. Subramanian, "Workload Characterization of AFS File Servers", Tech. Report 92-6, Center for Information Technology Integration, University of Michigan, 1992.
- [28] Sun Microsystems, Inc. "SunOS 5.2 Guide to Multi-Thread Programming", May 1993.
- [29] A. Tanenbaum, "Modern Operating Systems", Prentice-Hall, 1992.
- [30] J. E. White, "A High-Level Framework for Network-Based Resource Sharing", *Proceedings of the National Computer Conference*, June 1976.
- [31] W. Willinger, M.S. Taqqu, W.E. Leland, D.V. Wilson, "Self Similarity in High-Speed Packet Traffic - Analysis and Modeling of Ethernet Traffic Measurements", *Statistical Science*, Vol. 10, No. 1, February 1995, pp 67-85.
- [32] C.M. Woodside, J.E. Neilson, D.C. Petriu, S. Majumdar, "The Stochastic Rendezvous Network Model for Client-Server-Like Distributed Software", *IEEE Trans. on Computers*, Vol. 44, January 1995, pp. 20-34.

Appendix A

Other Node Scheduling Policies

Overview

This Appendix contains a description of other node scheduling policies that have been implemented in the simulation model.

Longest Queue First (LQF)

Each server in a node has a request queue and the LQF node scheduler associates a higher priority to the server with largest request queue. Each thread in the ready to run queue has the request queue size attribute. The thread with the largest queue size is given the highest priority. Each server updates its queue size with the scheduler each time a request arrives and completes.

Shortest Queue First (SQF)

The SQF node scheduler works in a similar fashion to the LQF scheduler except it associates a higher priority to the server with the smallest queue size. The thread with the smallest queue size attribute is given the highest priority.

Active Server First (ASF)

The ASF policy applies to nodes that have active and pure servers on the same node. The ASF policy gives a higher priority to threads that belong to the active server. If no threads for an active server are ready to run then other servers are allowed to run.

Pure Server First (PSF)

The PSF policy also applies to nodes where active and pure servers share the same node. The PSF policy gives higher priority to threads that belong to a pure server. If no threads for a pure server are ready to run then other servers are allowed to run.

Shortest Total Service Time First (STSF)

The STSF policy associates a higher priority with the server with the least amount of work to perform. Each thread for a server has a service time attribute, each of these service times are summed to form the total service time for the server. The server with the smallest total service time is given the highest priority.

Appendix B

Additional Results of Single-Level Scheduling

Overview

This Appendix contains additional results from the experiments performed for single-level scheduling.

Effect of Relative Loading of Servers in the Two-Server System

Figure B.1: Performance of the multithreaded two server system of Figure 4.1 with non-preemptive scheduling. Server S_1 has 5 threads and mean service time at server S_1 is 5.0 and 1.0 at server S_2 . CV is 1.0.

Figure B.2: Performance of the multithreaded two server system of Figure 4.1 with non-preemptive scheduling. Server S_1 has 5 threads and mean service time at server S_1 is 1.0 and 5.0 at server S_2 . CV is 1.0.

Figure B.3: Performance of the multithreaded two server system of Figure 4.1 with preemptive scheduling. Server S_1 has 5 threads and mean service time at server S_1 is 5.0 and 1.0 at server S_2 . CV is 1.0.

Figure B.4: Performance of the multithreaded two server system of Figure 4.1 with preemptive scheduling. Server S_1 has 5 threads and mean service time at server S_1 is 1.0 and 5.0 at server S_2 . CV is 1.0.

Effect of Variability in Service Time in the Two-Server System

Figure B.5: Impact of variability in request service time for the active server S_1 in the client-server system of Figure 4.1. $\lambda = 0.3$ requests per second. Server S_1 is multithreaded with 5 threads and non-preemptive scheduling is used. Mean service time is 1.0.

Figure B.6: Impact of variability in request service time for the pure server S_2 in the client-server system of Figure 4.1. $\lambda = 0.3$ requests per second. Server S_1 is multithreaded with 5 threads and non-preemptive scheduling is used. Mean service time is 1.0.

Preemptive Scheduling in the Three-Server System

Figure B.7: The performance of the three server system of Figure 4.15. Servers are multithreaded with preemptive scheduling. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

Preemptive Scheduling in the Two Pure Server System

Figure B.8: The performance of the three server system of Figure 4.18.. Servers are multithreaded with preemptive scheduling. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

Appendix C

Additional Results of Node Scheduling Performance Experiments

Overview

This appendix contains additional results from the experiments in two-level scheduling not included in Chapter 5.

Comparison of FS and SL Node Scheduling for a System with Two Pure Servers Sharing One Node

Figure C.1: Comparison of the performance of the LPF policy with SL and FS node scheduling in client-server system of Figure 5.9. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

Figure C.2: Comparison of the performance of the SRF policy with SL and FS node scheduling in client-server system of Figure 5.9. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

Comparison of FS and SL Node Scheduling for a System with a Pure and Active Server Sharing the Same Node

Figure C.3: Comparison of the performance of the LPF server scheduling policy in conjunction with the SL and FS node scheduling policies in client-server system of Figure 5.5. Servers S_2 and S_3 share the same node and server S_1 resides on its own node. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

Figure C.4: Comparison of the performance of the SRF server scheduling policy in conjunction with the SL and FS node scheduling policies in client-server system of Figure 5.5. Servers S_2 and S_3 share the same node and server S_1 resides on its own node. Each server has 5 threads and mean service time is 1.0 and $CV = 1.0$.

Appendix D

Simulation Input File Format

Overview

This Appendix contains the input file format for the simulator. The file is in ASCII and is used as “standard input” for the simulator. For example, to run the simulator type the following command :

```
nice +10 hp_rpc_sim < test_file_1 > test_output &
```

File Format

```
<seed>
<simulation duration>
<simulation type>
<node scheduling policy>
<server scheduling policy>
<thread creation policy>
<number of clients>
<number of server nodes>
<number of servers>
  <mean client think time> -----\
  <mean client processing time> -----} -- for each client
  <number of calls to servers> -----\
    <server index> -----} for each
    <mean number of calls> -----} server called--<
  <node id for server> -----\
  <probability of small requests> -----}
  <mean service time> -----\
  <CV of service time> -----} -- for each server
  <number of executor threads> -----\
  <number of calls to other servers> -----}
```

```

<server index>      --\ for each
<mean number of calls> --\ server called --/

```

Field Descriptions

Seed : the seed is for the random number generator used in Parasol.

Simulation Duration : the duration of the simulation in seconds.

Simulation Type : the type of simulation, OPEN or CLOSED are the valid values for this field.

Node Scheduling Policy : the node scheduling policies supported are : FCFS (SL), FS, LQF, SQF, ALT, STSF, LTSF, ASF, and PSF.

Server Scheduling Policy : the server scheduling policies supported are : FCFS, LPF, ALPF, SJF, SJF/LPF, and RR.

Thread Creation Policy : the thread creation policies supported are STATIC and DYNAMIC.

Client Think Time : may be decimal number or to get hyperexponential inter-arrival times use : HYPER $x y z$ where x is the probability of small times, y is the mean time, and z is the CV.

Mean service time : may be decimal number or to get deterministic service times user : DET x , where x is there fixed service time. The probability and CV fields are ignored.

Node Index : is the node the server belongs to. Node number begins at zero. Servers that share the same node have the same node index. For example, say there are three servers, 0, 1, and 2. Server 0 and 1 share the same node 0 so there node index will be 0. Server 2 resides on its own node which would be 1.

Server Index : the first server in the input file has index zero, the second server has index one and so on.

Example File

This following file was used to generate one point on the graph of Figure 4.2 for the FCFS server scheduling policy.

```
4523234
2000000.0
OPEN
FCFS
FCFS
STATIC
1
2
2
2.78
1.0
2
0
1.0
1
1.0
0
0.99999
1.0
1.0
1
1
1
1.0
1
0.99999
1.0
1.0
1
0
```


END

2 1 / 1 1 / 9 6

FIN